UNIT 5

SEARCHING & SORTING TECHNIQUES

LEARNING OBJECTIVES

To learn and implement the various searching and sorting techniques, perform the asymptotic analysis over them for several cases.

5.1 SEARCHING

Searching is the process of finding a given value's index position in a list of values provided. It is the algorithmic process of finding a particular item in a collection of items given. It can be applied on an internal data structure or on external data structure.

5.1.1 Linear Search

ALGORITHM:

Step 1 : Select the first element of the list as the current element.

Step 2 : Compare the current element with the target search element. If matches, then go to step 5.

Step 3 : If there is a next element, then set current element to next element and go to Step 2.

Step 4 : Target search element not found. Go to Step 6.

Step 5 : Target search element found and return location.

Step 6 : Exit process.

PSEUDO CODE:

Read n

Flag=0

For 0 to n

    Read list a[]

Read target search element

For all elements in list

    If current list element = = target search element

        Flag=1

        Print current element index position

    If Flag= =0

        Print Element not found

EXAMPLE

For n=5, a=[12,16,18,14,11], target search element=14

For i	**If a[i]= =target search element**		**Flag**	**Output**
	Comparison	**Result**		
0	If 12==14	False		
1	If 16==14	False		
2	If 18==14	False		
3	If 14==14	True	1	Index=3
4	If 11==14	False		

The following Python Code intends to perform a Linear search over a list for the element 10.

Python Code:

```
def linearsearch(li, n, x):
    for i in range (0, n):
      if (li[i] == x):
         return i;
    return -1;

li = [ 2, 3, 4, 10, 40 ];
x = 10;
n = len(li);
result = linearsearch(li, n, x)
if(result == -1):
   print("Element is not present in list")
else:
   print("Element is present at index", result);
```

OUTPUT:

```
'Element is present at index', 3
```

The time complexity of above algorithm is O(n).

Linear search is rarely used practically because the running time is proportional to the position of element in the list. Whereas, other the algorithms such as the binary search and hash tables work significantly faster when compared to Linear search.

5.1.2 Binary Search

Search a sorted list of values by repeatedly dividing the search interval into two halves. Begin with an interval covering the whole list. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise narrow it to the upper half. Repeatedly check until the value is found or the interval is empty.

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 sub 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 right sub list of the middle element.

Step 8: Repeat the same process until the search element is found in the list or until sub list 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.

PSEUDO CODE

```
Procedure binary_search
        A ← sorted array
        n ← size of array
        x ← value to be searched
        Set lowerBound = 1
        Set upperBound = n

    while x not found
        if upperBound < lowerBound
            EXIT: x does not exist.
```

```
    set midPoint = lowerBound + ( upperBound - lowerBound ) / 2
    if A[midPoint] < x
        set lowerBound = midPoint + 1
    if A[midPoint] > x
        set upperBound = midPoint - 1
    if A[midPoint] = x
        EXIT: x found at location midPoint
  end while
  end procedure
```

PROGRAM

```
def binarySearch (li, l, r, x):
  if r >= l:
     mid = l + (r - l)//2
     if li[mid] == x:
        return mid
     elif li[mid] > x:
        return binarySearch(li, l, mid-1, x)
     else:
        return binarySearch(li, mid + 1, r, x)
  else:
      return -1

li = [ 22, 33, 44, 50, 70 ]
x = 50
result = binarySearch(li, 0, len(li)-1, x)
if result != -1:
   print ("Element is present at index % d" % result )
else:
   print ("Element is not present in list")
```

OUTPUT:

```
Element is present at index  3
```

The idea of binary search is to use the information that the array is sorted and reduce the time complexity to O(Log n).

EXAMPLE:

Consider the list

0	1	2	3	4	5	6	7	8	9
12	23	33	56	66	78	88	89	90	98

Target Search element: 89

Mid = 0+(9-1)//2 = 4

a[4] !=target search value == 66 != 89

12	23	33	56	66	78	88	89	90	98

but target search element is greater than the mid element == > 89>78

therefore consider the values after the middle element

12	23	33	56	66	78	88	89	90	98

Mid= 5+(9-1)//2 = 7

a[7] ==target search value ==> 89 == 89

12	23	33	56	66	78	88	89	90	98

Return index location 7

5.1.3 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.

Fibonacci Numbers are recursively defined as

F(n) = F(n-1) + F(n-2)

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

First few Fibinacci Numbers are 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...

ALGORITHM:

Let the searched element be x.

The idea is to first find the smallest Fibonacci number that is greater than or equal to the length of given list. Let the find Fibonacci number be fib (m'th Fibonacci number). (m-2)'th Fibonacci number is used as the index (If it is a valid index). Let (m-2)'th Fibonacci Number be i, li[i] with x is compared, if x is same, i is returned. Else if x is greater, we recur for sub list after i, else we recur for sub list before i.

Below is the complete algorithm

Let li[0..n-1] be the input array and element to be searched be x.

1. Find the smallest Fibonacci Number greater than or equal to n.
   (a) Let this number be fibM [m'th Fibonacci Number].
   (b) Let the two Fibonacci numbers preceding it be fibMm1 [(m-1)'th Fibonacci Number] and fibMm2 [(m-2)'th Fibonacci Number].
2. While the list has elements to be inspected:
   (a) Compare x with the last element of the range covered by fibMm2
   (b) **I**f x matches, return index
   (c) **Else If** x is less than the element, move the three Fibonacci variables two Fibonacci down, indicating elimination of approximately rear two-third of the remaining list elements.
   (d) **Else** x is greater than the element, move the three Fibonacci variables one Fibonacci down. Reset offset to index. Together these indicate elimination of approximately front one-third of the remaining list elements.
3. Since there might be a single element remaining for comparison, check if fibMm1 is 1. If Yes, compare x with that remaining element. If match, return index.

PROGRAM

```
from bisect import bisect_left
def fibMonaccianSearch(li, x, n):
    fibMMm2 = 0
    fibMMm1 = 1
    fibM = fibMMm2 + fibMMm1
    while (fibM < n):
        fibMMm2 = fibMMm1
        fibMMm1 = fibM
        fibM = fibMMm2 + fibMMm1
    offset = -1;
    while (fibM > 1):
        i = min(offset+fibMMm2, n-1)
        if (li[i] < x):
            fibM = fibMMm1
            fibMMm1 = fibMMm2
            fibMMm2 = fibM - fibMMm1
            offset = i
        elif (li[i] > x):
            fibM = fibMMm2
            fibMMm1 = fibMMm1 - fibMMm2
            fibMMm2 = fibM - fibMMm1
        else :
            return i
    if(fibMMm1 and li[offset+1] == x):
       return offset+1;
    return -1

#program begins
li = [11, 23, 33, 43, 46, 50, 67, 76, 85, 99, 101]
n = len(li)
x = 85
print("Found at index:", fibMonaccianSearch(li, x, n))
```

OUTPUT:

```
'Found at index:', 8
```

Illustration:

Let us understand the algorithm with the example given below:

i	1	2	3	4	5	6	7	8	9	10	11
li[i]	11	23	33	43	46	50	67	76	85	99	101

Illustration assumption:

Target search element x is 85.

Length of array n = 11

fibMMm2	**fibMMm1**	**fibM = fibMMm2 + fibMMm1**
0	1	1
While fibM < n		
1	1	2
1	2	3
2	3	5
3	5	8
5	8	13

Smallest Fibonacci number >= 11 is 13.

fibMm2 = 5,

fibMm1 = 8, and

fibM = 13. (**fibM = fibMMm2 + fibMMm1**)

i = min(offset + fibMm2, n)

fibMMm2	fibMMm1	fibM	offset	i = min(offset + fibMm2, n)	li[i]	Step to be performed
5	8	13	0	5	46	Move one down, reset offset
3	5	8	5	8	76	Move one down, reset offset
2	3	5	8	10	99	Move two down
1	1	2	8	9	85	Return i

5.2 SORTING

Data sorting is any process that involves arranging the data into some meaningful order to make it easier to understand, analyze or visualize. Data is typically sorted based on actual values, counts or percentages, in either ascending or descending order, but can also be sorted based on the variable value labels.

5.2.1 INSERTION SORT

Insertion sort is a simple sorting algorithm that works the way we sort playing cards in our hands.

ALGORITHM

Step 1: Read a list of unsorted numbers

Step 2: Set a marker for the sorted section after the first number in the list

Step 3: Repeat steps 4 through 6 until the unsorted List of elements exist

Step 4: Select the first unsorted number

Step 5: Swap this number to the left until it arrives at the correct sorted position

Step 6: Advance the marker to the right one position

PROGRAM

```
def insertionSort(li):
    for i in range(1, len(li)):
        key = li[i]
        j = i-1
        while j >= 0 and key < li[j] :
                li[j + 1] = li[j]
                j -= 1
        li[j + 1] = key
# Program Begins Execution
li = [92, 13, 43, 5, 6, 12, 50]
insertionSort(li)
for i in range(len(li)):
    print (li[i])
```

OUTPUT:

```
5
6
12
13
43
50
92
```

PSEUDO CODE

INSERTION-SORT (A)

```
1    for j <- 2 to length[A]
2    do key  A[j]
3    Insert A[j] into the sorted sequence A[1 . . j - 1].
4    i <- j - 1
5    while i > 0 and A[i] > key
6    do A[i + 1] <- A[i]
7    i <- i - 1
8    A[i + 1] <- key
```

EXAMPLE:

Given unsorted list elements

i	0	1	2	3	4	5	6
a[i]	92	13	43	5	6	12	50

For i=1

92	13	43	5	6	12	50

For i=2

13	92	43	5	6	12	50

For i=3

13	43	92	5	6	12	50

For i=4

5	13	43	92	6	12	50

For i=5

5	6	13	43	92	12	50

For i=6

5	6	12	13	43	92	50

Final Sorted list

5	6	12	13	43	50	92

Time Complexity: $O(n^2)$

5.2.2 Heapsort

Heap sort is a comparison based sorting technique based on Binary Heap data structure. It is similar to selection sort where we first find the maximum element and place the maximum element at the end. Heap sort algorithm uses one of the tree concepts called **Heap Tree**.

In this sorting algorithm, **Max Heap** is used to arrange list of elements in Descending order and **Min Heap** to arrange list elements in Ascending order.

ALGORITHM (Increasing order)

Step 1 - Construct a **Binary Tree** with given list of Elements.

Step 2 - Transform the Binary Tree into **Min Heap**.

a) The value assigned to the last node is considered.

b) Compare the value of this child node with its parent.

c) If the value of the parent is greater than the child, then swap them.

d) Repeat step c & d until Heap property holds.

Step 3 - Delete the root element from Min Heap using **Heapify** method.

Step 4 - Append the deleted element into the Sorted list.

Step 5 - Repeat the same until Min Heap becomes empty.

Step 6 - Display the sorted list.

ALGORITHM (descending order)

Step 1 - Construct a **Binary Tree** with given list of Elements.

Step 2 - Transform the Binary Tree into Max Heap.

a) The value assigned to the last node is considered.

b) Compare the value of this child node with its parent.

c) If the value of the parent is lesser than the child, then swap them.

d) Repeat step c & d until Heap property holds.

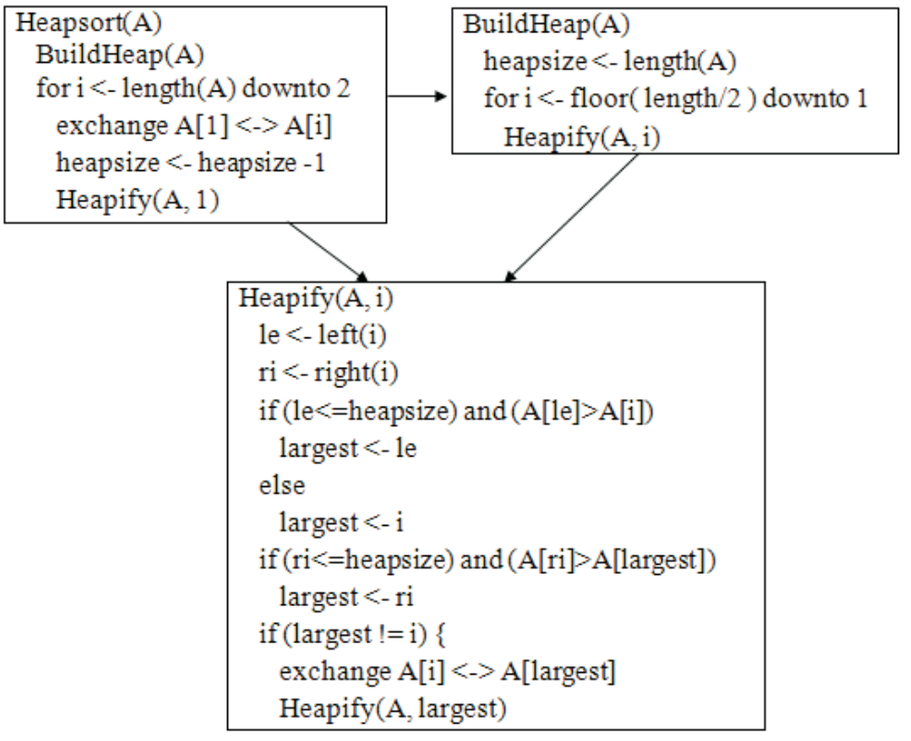
Step 3 - Delete the root element from Max Heap using Heapify method.

Step 4 - Append the deleted element into the Sorted list.

Step 5 - Repeat the same until Max Heap becomes empty.

Step 6 - Display the sorted list.

PSEUDO CODE



PROGRAM

```
def heapify(li, n, i):
    largest = i
    l = 2 * i + 1
    r = 2 * i + 2
    if l < n and li[i] < li[l]:
        largest = l
    if r < n and li[largest] < li[r]:
        largest = r
    if largest != i:
        li[i],li[largest] = li[largest],li[i]
        heapify(li, n, largest)
def heapSort(li):
    n = len(li)
    for i in range(n, -1, -1):
        heapify(li, n, i)
    for i in range(n-1, 0, -1):
        li[i], li[0] = li[0], li[i]
        heapify(li, i, 0)
li = [ 22, 11, 31, 5, 46, 7,2]
heapSort(li)
n = len(li)
print ("Sorted array is")
for i in range(n):
    print ("%d" %li[i])
```

OUTPUT:

```
Sorted array is
2
5
7
11
22
31
46
```

Time Complexity:

Time complexity of heapify is O(Log n).

Time complexity of create and BuildHeap() is O(n) and

overall time complexity of Heap Sort is O(n Log n).

5.2.3 BUBBLE SORT

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order.

ALGORITHM (Ascending order)

Step 1: Read N elements in the list to be sorted

Step 2: For N-1 times repeatedly perform

(a) Initialize the first element(index = 0)as the current element, compare the current element with the next element of the list.

(b) If the current element is greater than the next element of the array, swap them.

(c) Else If the current element is less than the next element, move to the next element.

Step 3: Return the sorted list of values

ALGORITHM (Descending order)

Step 1: Read N elements in the list to be sorted

Step 2: For N-1 times repeatedly perform

(a) Initialize the first element(index = 0)as the current element, compare the current element with the next element of the list.

(b) If the current element is lesser than the next element of the array, swap them.

(c) Else If the current element is greater than the next element, move to the next element.

Step 3: Return the sorted list of values

PSEUDO CODE

```
bubbleSort(array)
  for i <- 1 to indexOfLastUnsortedElement-1
    if leftElement > rightElement
      swap leftElement and rightElement
end bubbleSort
```

Example:

Given list of elements to be sorted in ascending order	12	5	6	2	9	11	3

First Pass:

ACTION TAKEN	VALUES IN LIST						
Compare first two pairs of elements	12	5	6	2	9	11	3
Exchange and compare next two pairs of elements	5	12	6	2	9	11	3
Exchange and compare next two pairs of elements	5	6	12	2	9	11	3
Exchange and compare next two pairs of elements	5	6	2	12	9	11	3
Exchange and compare next two pairs of elements	5	6	2	9	12	11	3
Exchange and compare next two pairs of elements	5	6	2	9	11	12	3
Exchange and since there are no more elements to be compared move to **second pass**	5	6	2	9	11	3	12

Second Pass:

ACTION TAKEN	VALUES IN LIST						
Compare first two pairs of elements	5	6	2	9	11	3	12
Since the pair of values are in correct order. Compare next two pairs of elements	5	6	2	9	11	3	12
Exchange and compare next two pairs of elements	5	2	6	9	11	3	12
Since the pair of values are in correct order. Compare next two pairs of elements	5	2	6	9	11	3	12
Since the pair of values are in correct order. Compare next two pairs of elements	5	2	6	9	11	3	12
Exchange and compare next two pairs of elements	5	2	6	9	3	11	12
Since the pair of values are in correct order and no more elements to be compared move to **third pass**	5	2	6	9	3	11	12

Third Pass:

ACTION TAKEN	VALUES IN LIST						
Compare first two pairs of elements	5	2	6	9	3	11	12
Exchange and compare next two pairs of elements	2	5	6	9	3	11	12
Since the pair of values are in correct order. Compare next two pairs of elements	2	5	6	9	3	11	12
Since the pair of values are in correct order. Compare next two pairs of elements	2	5	6	9	3	11	12
Exchange and compare next two pairs of elements	2	5	6	3	9	11	12
Since the pair of values are in correct order. Compare next two pairs of elements	2	5	6	3	9	11	12
Since the pair of values are in correct order and no more elements to be compared move to **fourth pass**	2	5	6	3	9	11	12

Fourth Pass:

ACTION TAKEN	VALUES IN LIST						
Compare first two pairs of elements	2	5	6	3	9	11	12
Since the pair of values are in correct order. Compare next two pairs of elements	2	5	6	3	9	11	12
Since the pair of values are in correct order. Compare next two pairs of elements	2	5	6	3	9	11	12
Exchange and compare next two pairs of elements	2	5	3	6	9	11	12
Since the pair of values are in correct order. Compare next two pairs of elements	2	5	3	6	9	11	12
Since the pair of values are in correct order. Compare next two pairs of elements	2	5	3	6	9	11	12
Since the pair of values are in correct order and no more elements to be compared move to **fifth pass**	2	5	3	6	9	11	12

Fifth Pass:

ACTION TAKEN	VALUES IN LIST						
Compare first two pairs of elements	2	5	3	6	9	11	12
Since the pair of values are in correct order. Compare next two pairs of elements	2	5	3	6	9	11	12
Exchange and compare next two pairs of elements	2	3	5	6	9	11	12
Since the pair of values are in correct order. Compare next two pairs of elements	2	3	5	6	9	11	12
Since the pair of values are in correct order. Compare next two pairs of elements	2	3	5	6	9	11	12
Since the pair of values are in correct order. Compare next two pairs of elements	2	3	5	6	9	11	12
Since the pair of values are in correct order and no more elements to be compared move to **sixth pass**	2	3	5	6	9	11	12

Sixth Pass:

ACTION TAKEN	VALUES IN LIST						
Compare first two pairs of elements	2	3	5	6	9	11	12
Since the pair of values are in correct order. Compare next two pairs of elements	2	3	5	6	9	11	12
Since the pair of values are in correct order. Compare next two pairs of elements	2	3	5	6	9	11	12
Since the pair of values are in correct order. Compare next two pairs of elements	2	3	5	6	9	11	12
Since the pair of values are in correct order. Compare next two pairs of elements	2	3	5 6	9	11	12	
Since the pair of values are in correct order. Compare next two pairs of elements	2	3	5	6	9	11	12
FINAL SORTED LIST OF VALUES	2	3	5	6	9	11	12

N=7 elements are sorted in N-1 number of passes (Six Passes)

PROGRAM

```
def bubbleSort(LI):
    n = len(LI)
    for i in range(n):
        for j in range(0, n-i-1):
            if LI[j] > LI[j+1] :
                LI[j], LI[j+1] = LI[j+1], LI[j]
#PROGRAM BEGINS
LI = [54, 34, 15, 12, 25, 10, 40]
bubbleSort(LI)
print ("Sorted array is:")
for i in range(len(LI)):
    print ("%d" %LI[i])
```

OUTPUT:

```
Sorted array is:
10
12
15
25
34
40
54
```

Time Complexity

Worst and Average Case: $O(n^2)$

Best Case: O(n)

5.2.4 Quick Sort

Quick Sort follows Divide and Conquer algorithm. An element is picked up as pivot and partitions the given list of values around the selected pivot element.

There are various ways followed inorder to pick out a pivot element. They are

1. Always select the first element of the list as pivot.

2. Always select the last element of the list as pivot

3. Select any random element from the list as pivot.

4. Select the median element as pivot.

The key processing done in quickSort is partition(). The main target of partitions is, given a list of values and an element x of the list as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.

PSEUDO CODE:

algorithm quicksort(list, lower, higher) is

```
if lower < higher then
    p := partition(list, lower, higher)
    quicksort(list, lower, p)
    quicksort(list, p + 1, higher)
```

Algorithm partition (list, lower, higher) is

```
pivot := list [lower + (higher - lower) / 2]
i := lower - 1
j := higher + 1
loop forever
    do
        i := i + 1
    while list [i] < pivot
    do
        j := j - 1
    while list [j] > pivot
    if i >= j then
        return j
    swap list [i] with list [j]
```

ALGORITHM

Step 1: Consider an element as pivot from the list, which is the last index of list.

Step 2: Break down the list into two such that the current pivot element is positioned in the correct location. All the elements smaller than the pivot will be on the left side of the pivot and all the elements greater than the pivot will be on the right side of it.

Step 3: The left and the right partitions a repeated with the step 1 and 2

Step 4: Return the sorted list of values

```
PROGRAM
def partition(LI,low,high):
    i = ( low-1 )
    pivot = LI[high]
      for j in range(low , high):
          if   LI[j] < pivot:
                i = i+1
                LI[i],LI[j] = LI[j],LI[i]
    LI[i+1],LI[high] = LI[high],LI[i+1]
    return ( i+1 )

def quickSort(LI,low,high):
    if low < high:
          pi = partition(LI,low,high)
          quickSort(LI, low, pi-1)
          quickSort(LI, pi+1, high)

# PROGRAM BEGINS HERE
LI = [16, 7, 81, 19, 12, 5]
n = len(LI)
quickSort(LI,0,n-1)
print ("Sorted array is:")
for i in range(n):
    print ("%d" %LI[i])
```

OUTPUT:

Sorted array is:

5

7

12

16

19

81

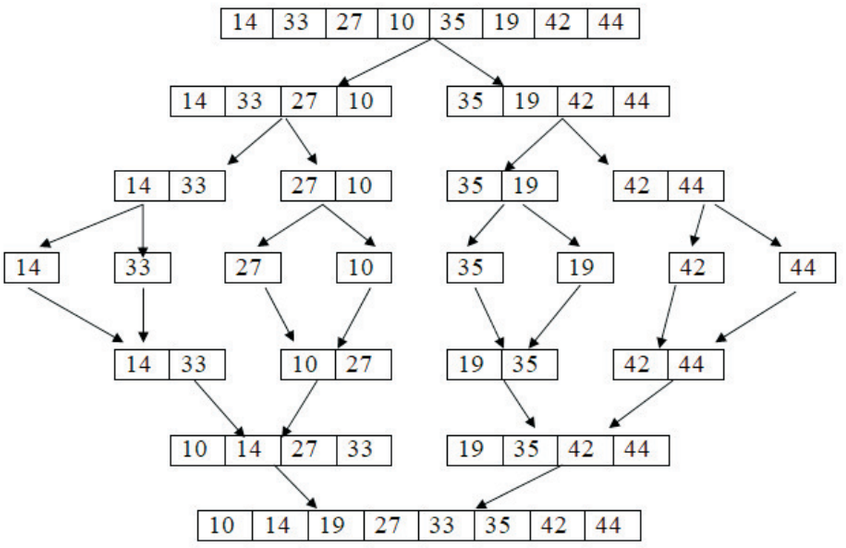
Time complexity:

worst case: $O(n^2)$

best and average case: O(n log n)

5.2.5 MERGE SORT

Merge Sort works under the principle of Divide and Conquer algorithm. It divides input list in two halves, Recursively until it reaches individual elements. The **merge() function** is used for merging the elements in a sorted manner.



PSEUDO CODE:

Procedure mergesort(var a as array)

If(n == 1) return a

Var l1 as array = a[0] ... a[n/2]

Var l2 as array = a [n/2+1] ... a[n]

l1 = mergesort(l1)

l2 = mergesort(l2)

return merge(l1, l2)

end procedure

procedure merge(var a as array, var b as array)

var c as array

while (a and b have elements)

if (a[0] > b[0])

add b[0] to the end of c

```
remove b[0] from b
else
add a[0] to the end of c
remove a[0] from a
end if
end while
while ( a has elements )
add a[0] to the end of c
remove a[0] from a
end while
while ( b has elements )
add b[0] to the end of c
remove b[0] from b
end while
return c
end procedure
```

ALGORITHM

Step 1: If it is only one element in the list it is already sorted, return.

Step 2: Divide the list recursively into two halves until it can no more be divided.

Step 3: Merge the smaller lists into new list in sorted order.

PROGRAM

```
def mergeSort(LI):
    if len(LI) >1:
        mid = len(LI)//2
        LEFT = LI[:mid]
        RIGHT = LI[mid:]

        mergeSort(LEFT)
        mergeSort(RIGHT)

        i = j = k = 0
        while i < len(LEFT) and j < len(RIGHT):
            if LEFT[i] < RIGHT[j]:
                LI[k] = LEFT[i]
                i+=1
            else:
                LI[k] = RIGHT[j]
                j+=1
            k+=1

        while i < len(LEFT):
            LI[k] = LEFT[i]
            i+=1
            k+=1

        while j < len(RIGHT):
            LI[k] = RIGHT[j]
            j+=1
            k+=1


def printList(LI):
    for i in range(len(LI)):
        print(LI[i],end=" ")
    print()
```

Time complexity:

worst case, best and average case: O(n log n).

```
# PROGRAM EXECUTION STARTS
if __name__ == '__main__':
    LI = [22, 41, 3, 15, 2, 10]
    print ("Given array is", end="\n")
    printList(LI)
    mergeSort(LI)
    print("Sorted array is: ", end="\n")
    printList(LI)
```

OUTPUT:

Given array is:
22 41 3 15 2 10
Sorted array is:
2 3 10 15 22 41

5.3 ANALYSIS OF SORTING TECHNIQUES

Sorting techniques are evaluated by its efficiency which can be measured by the following factors

* Time Complexity
* Space Complexity

Time complexity of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the length of the input. Similarly, **Space complexity** of an algorithm quantifies the amount of space or memory taken by an algorithm to run as a function of the length of the input.

5.3.1 Categories of Sorting Algorithms

There are two categories of sorting algorithms. They are

* Comparison based Sorting
* Non-Comparison based Sorting

In Comparison based sorting, elements of an array are compared with each other to find the sorted array. For example, Bubble Sort, Insertion Sort, Selection Sort, Merge Sort, Quick Sort and Heap Sort are Comparison based Sorting algorithms. In Non-Comparison based sorting, elements of array are not compared with each other to find the sorted array. For example, Radix Sort, Bucket Sort and Count Sort are categorized as Non-Comparison based Sorting algorithms.

Time and space complexity depends on lots of things like hardware, operating system, processors, etc. However, these factors are not considered while analyzing the algorithm. The execution time of an algorithm alone is considered. For example consider the following code for calculating time complexity.

```
int count = 0;

for (int i = 0;  i < N;  i++)

    for (int  j = 0; j < i;  j++)

        count++;
```

Let's see how many times count++ will run.

When i=0, it will run 0 time.

When i=1, it will run 1 time.

When i=2, it will run 2 times and so on.

Total number of times **count**++ will run is 0+1+2+...+ (N–1) = **N*(N-1) / 2**. So the time complexity will be $\mathbf{O(N^2)}$. Time and Space Complexity for various algorithms are given in below the table.

TIME AND SPACE COMPLEXITY

Algorithm	Best Time Complexity	Average Time Complexity	Worst Time Complexity	Worst Space Complexity
Linear Search	O(l)	O(n)	O(n)	O(l)
Binary Search	O(l)	O(n log n)	O(n log n)	O(l)
Bubble Sort	O(n)	$O(n^2)$	$O(n^2)$	O(l)
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(l)
Insertion Sort	O(n)	$O(n^2)$	$O(n^2)$	O(l)
Merge Sort	O(n log n)	O(n log n)	O(n log n)	O(n)
Quick Sort	O(n log n)	O(n log n)	O(n^2)	O(log n)
Heap Sort	O(n log n)	O(n log n)	O(n log n)	O(n)
Bucket Sort	O(n+k)	O(n+k)	$O(n^2)$	O(n)
Radix Sort	O(nk)	O(nk)	O(nk)	O(n+k)
Tim Sort	O(n)	O(n log n)	O(n log n)	O(n)
Shell Sort	O(n)	$O((n\log(n))^2)$	$O((n\log(n))^2)$	O(l)

Generally when the array is almost sorted, insertion sort can be preferred. When order of input is not known, merge sort is preferred as it has worst case time complexity of nlog n and it is stable as well. In case of sorted array, insertion and bubble sort gives complexity of n but quick sort gives complexity of n^2. Thus according to the efficiency, algorithms can be selected.

****