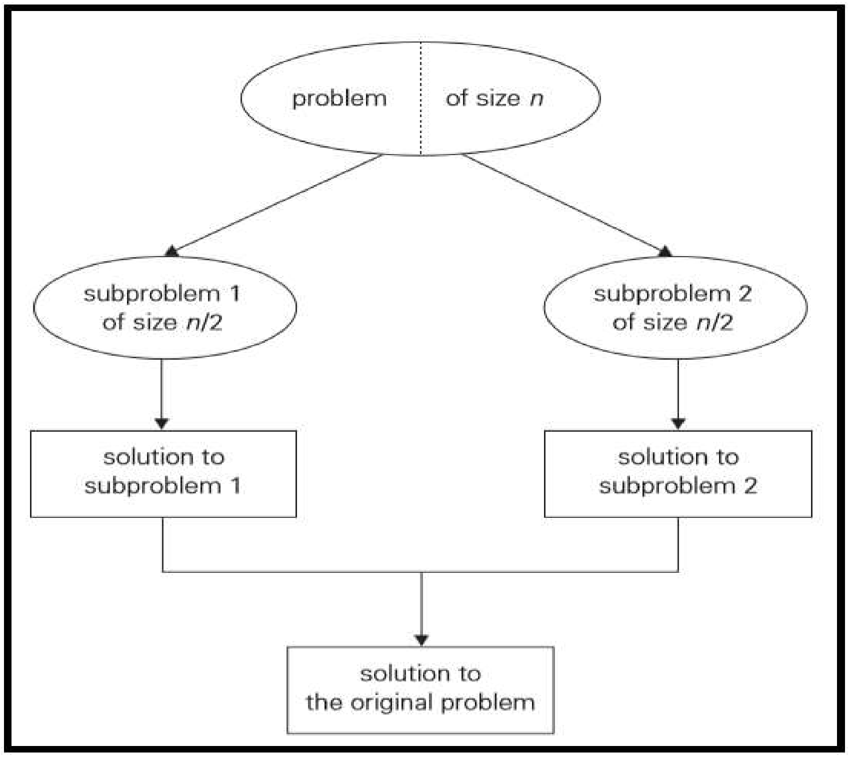
Week 4

Divide and conquer - Merge Sort, Quick Sort, Binary search.
Dynamic programming - Fibonacci sequence.
Backtracking – Concepts only
Greedy – Concepts only.

Divide & Conquer

Divide-and-conquer are one of the best-known general algorithm design techniques. Divide-and-conquer algorithms works as follows:

1. A problem is divided into several sub problems of the same type, ideally of about equal size.
2. The sub problems are solved.
3. Solutions to the sub problems are combined to get a solution to the original problem.

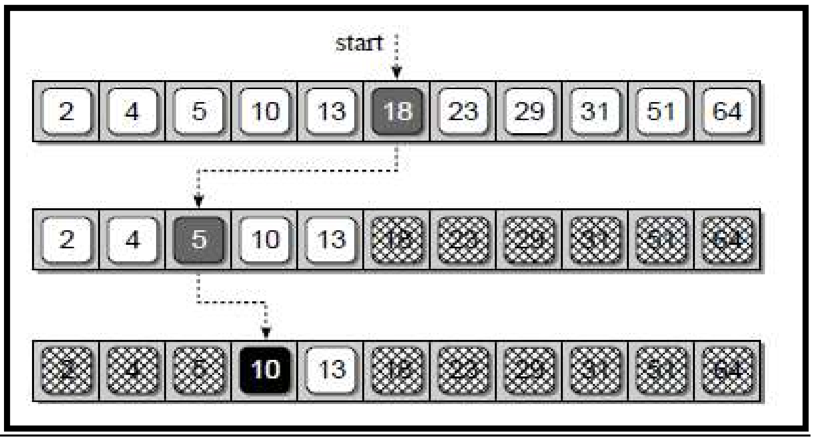

problem
of size n
subproblem 1 of size $n/2$
subproblem 2 of size $n/2$
solution to subproblem 1
solution to subproblem 2
solution to the original problem

Binary Search

- Binary search is an efficient algorithm for searching in a sorted array.
- It works by comparing a search key K with the array's middle element A[m].
- If they match, the algorithm stops; otherwise, the same operation is repeated recursively for the first half of the array if K <A[m], and for the second half if K >A[m]:

$$\underbrace{A[0]\dots A[m-1]}_{\substack{\text{search here if}\\ K<A[m]}}\quad \overset{\substack{K\\ \updownarrow}}{A[m]}\quad \underbrace{A[m+1]\dots A[n-1]}_{\substack{\text{search here if}\\ K>A[m]}}.$$

- **Here is an example: key = 10**



Python Function for Binary Search

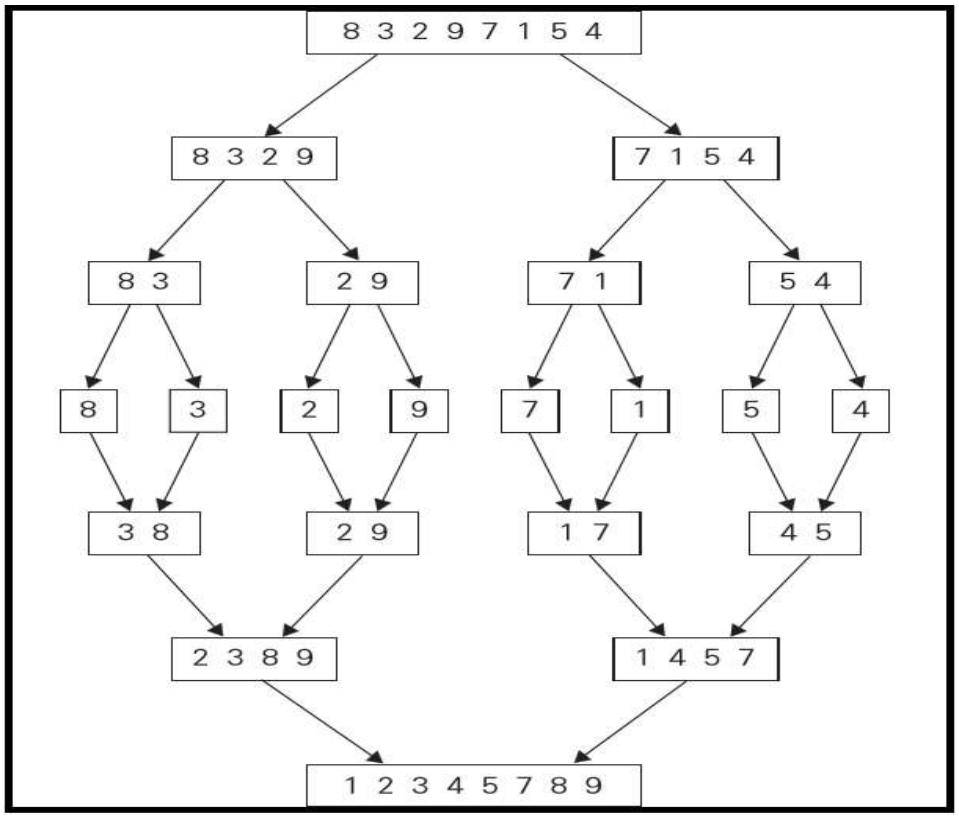
```
def binarySearch(a, key):

    start=0
    end=len(a)-1
    while start<=end:
        mid=(start+end)//2
        if key == a[mid]:
            return True
        elif key<a[mid]:
            end=mid-1
        else:
            start=mid+1
    return False
```

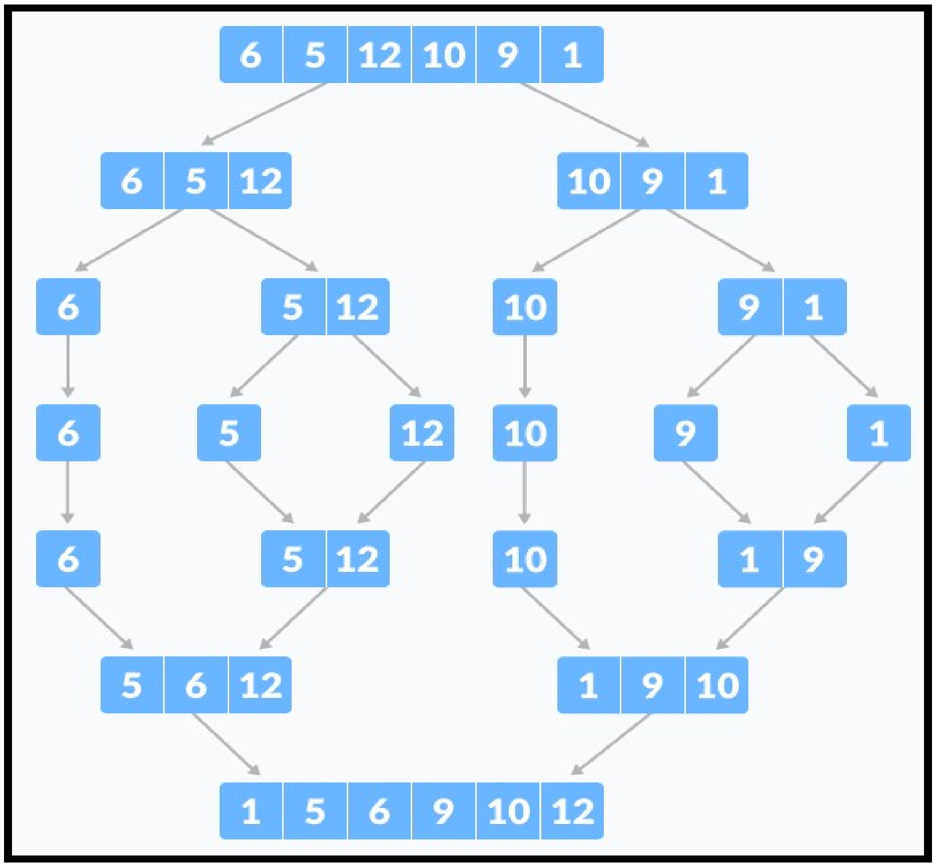
Merge Sort

- Merge Sort is a perfect example of the divide-and-conquer technique.
- It sorts a given List A[0..n − 1] by dividing it into two halves, sorting each of them separately, and then merging the two smaller sorted lists into a single sorted one.
- The merging of two sorted lists can be done as follows:
  - Two pointers (i & j) are used to point to the first elements of the lists being merged.
  - The elements pointed to are compared, and the smaller of them is added to a new list being constructed; after that, the index of the smaller element is incremented to point to its next element in the list it was copied from.
  - This operation is repeated until one of the two given lists is finished, and then the remaining elements of the other list are copied to the end of the new list.

- **Example 1:**



- **Example 2:**



Python Function for Merge Sort

```
def mergeSort(a):
  if len(a)<=1:
     return a
  else:
    mid=len(a)//2
    lefthalf=mergeSort(a[:mid])
    righthalf=mergeSort(a[mid:])

def mergeLists(A,B):
   i=0
   j=0
   newList = list()
   while i<len(A) and j <len(B):
      if A[i]< B[j]:
         newList.append(A[i])
         i=i+1
      else:
         newList.append(B[j])
         j=j+1

   while i < len(A):
      newList.append(A[i])
      i=i+1

   while j < len(B):
      newList.append(B[j])
      j=j+1
   return newList
```

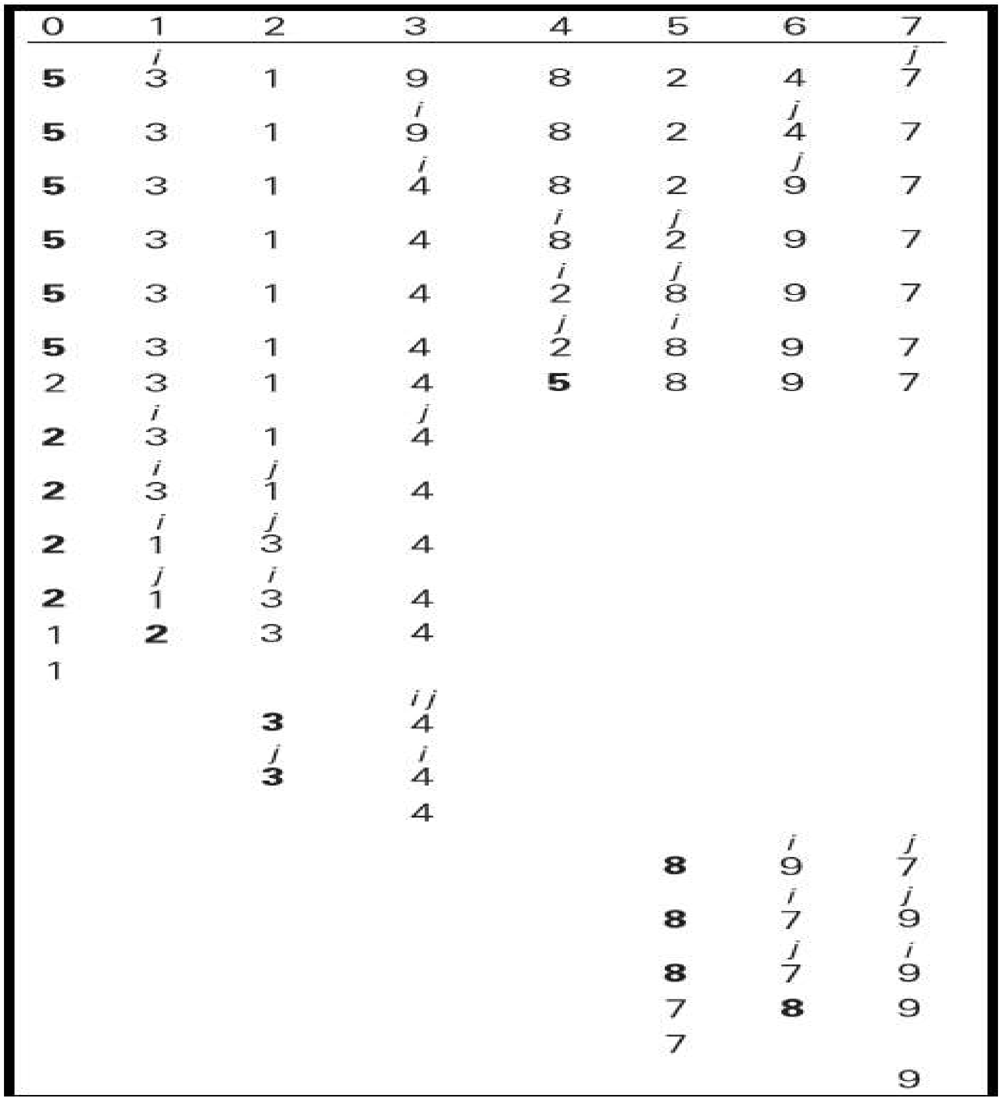
Quick Sort

➢ Quick Sort is the other important sorting algorithm that is based on the divide-and-conquer approach.

➢ **Quick Sort works as follows:**

- We start by selecting a pivot (first element as pivot).
- We will now scan the sub-array from both ends, comparing the sub-array's elements to the pivot.

- The left-to-right scan, denoted below by index pointer i, starts with the second element. This scan skips elements that are smaller than the pivot and stops upon encountering the first element greater than or equal to the pivot.
- The right-to-left scan, denoted below by index pointer j, starts with the last element of the sub-array. This scan skips over elements that are larger than the pivot and stops on encountering the first element smaller than or equal to the pivot.
- After both scans stop, three situations may arise, depending on whether or not the scanning indices have crossed.
  - If scanning indices i and j have not crossed, i.e.,i < j, we simply exchange A[i] and A[j] and continue the scans by incrementing i and decrementing j, respectively.
  - If the scanning indices have crossed over, i.e., i > j, we will have partitioned the sub-array after exchanging the pivot with A[j].
- **Example:**



0	1	2	3	4	5	6	7
5	$\overset{i}{3}$	1	9	8	2	4	$\overset{j}{7}$
5	3	1	$\overset{i}{9}$	8	2	$\overset{j}{4}$	7
5	3	1	$\overset{i}{4}$	8	2	$\overset{j}{9}$	7
5	3	1	4	$\overset{i}{8}$	$\overset{j}{2}$	9	7
5	3	1	4	$\overset{i}{2}$	$\overset{j}{8}$	9	7
5	3	1	4	$\overset{j}{2}$	$\overset{i}{8}$	9	7
2	3	1	4	**5**	8	9	7
2	$\overset{i}{3}$	1	$\overset{j}{4}$				
2	$\overset{i}{3}$	$\overset{j}{1}$	4				
2	$\overset{i}{1}$	$\overset{j}{3}$	4				
2	$\overset{j}{1}$	$\overset{i}{3}$	4				
1	**2**	3	4				
1							
		3	$\overset{ij}{4}$				
		$\overset{j}{\mathbf{3}}$	$\overset{i}{4}$				
			4				
					8	$\overset{i}{9}$	$\overset{j}{7}$
					8	$\overset{i}{7}$	$\overset{j}{9}$
					8	$\overset{j}{7}$	$\overset{i}{9}$
					7	**8**	9
					7		
							9

Python Function for Quick Sort

```
def quickSort(a,i,j):
    if i<j:
        s=partition(a,i,j)
        quickSort(a,i,s-1)
        quickSort(a,s+1,j)

def partition(a,l,r):
    pivot=l
    i=l+1
    j=r
    while(i<=j):
        while i<=j and a[i] <= a[pivot]:
            i=i+1

        while a[j] > a[pivot]:
            j=j-1

        if i < j:
            temp=a[i]
            a[i]=a[j]
            a[j]=temp

    temp=a[pivot]
    a[pivot]=a[j]
    a[j]=temp
    return j
```

Dynamic Programming

Dynamic Programming is an algorithm design technique, generally used for optimization problems.

- It breaks down the given problem into simpler sub-problems.
- It finds the optimal solution to these sub-problems.
- It stores the results of sub-problems.
- It reuses stored results.
- Finally, calculate the result of the given problem.

Fibonacci Sequence

- The Fibonacci sequence is a sequence of integer values in which the first two values are both 1 and each subsequent value is the sum of the two previous values.
- The first 11 terms of the sequence are:

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \ldots$$

Python Function for Fibonacci Sequence

```
def  Fib(n):
   f1=0
   f2=1
   a=list()
   i=2
   a.append(f1)
   a.append(f2)
   for i in range(2,n):
      f3=f1+f2
      a.append(f3)
      f1=f2
      f2=f3
   return a
```

Backtracking

- Backtracking is an algorithm design technique, works like brute force approach that tries out all the possible solutions and chooses the best solution among them.
- It is generally used for problems where there are possibilities of multiple solutions.
- It works as follows: If the current solution is not suitable, then eliminate that and backtrack (go back) and check for other solutions.

Greedy Method

- The Greedy technique suggests constructing a solution through a sequence of steps, each expanding a partially constructed solution obtained so far, until a complete solution to the problem is reached.
- On each step, the choice made must be:
  - ✓ Feasible solutions, i.e., it has to satisfy the problem's constraints.
  - ✓ Choose Locally optimal solution, i.e., best local choice among all feasible solutions.
  - ✓ Irrevocable, i.e., once made, it cannot be changed on subsequent steps of the algorithm.

Program #1: Python Program to Binary Search.

```
def binarySearch(a, key):
    start=0
    end=len(a)-1
    while start<=end:
        mid=(start+end)//2
        if key == a[mid]:
            return True
        elif key<a[mid]:
            end=mid-1
        else:
            start=mid+1
    return False

n=int(input("Enter Size of the List\n"))
a=list()
print("Enter List elements")
for i in range(n):
    num=int(input())
    a.append(num)
key=int(input("Enter Key element to be searched\n"))
print("\nSearch List: ", a)
print("Key:", key)
res=binarySearch(a,key)
if res==True:
    print("\nSuccessful Search")
else:
    print("\nUnsuccessful Search")
```

Output #1:

```
Enter Size of the List
5
Enter List elements
10
15
19
25
36
Enter Key element to be searched
```

Output #2:

```
Enter Size of the List
5
Enter List elements
10
15
19
25
36
Enter Key element to be searched
```

36 Search List: [10, 15, 19, 25, 36] Key: 36 Successful Search	36 Search List: [10, 15, 19, 25, 36] Key: 86 Unsuccessful Search

Program #2: Python Program to implement Merge Sort.

```
def mergeLists(A,B):
   i=0
   j=0
   newList = list()
   while i<len(A) and j <len(B):
      if A[i]< B[j]:
         newList.append(A[i])
         i=i+1
      else:
         newList.append(B[j])
         j=j+1

   while i < len(A):
      newList.append(A[i])
      i=i+1

   while j < len(B):
      newList.append(B[j])
      j=j+1
   return newList

def mergeSort(a):
  if len(a)<=1:
     return a
  else:
    mid=len(a)//2
    lefthalf=mergeSort(a[:mid])
    righthalf=mergeSort(a[mid:])

    newList=mergeLists(lefthalf,righthalf)
    return newList

n=int(input("Enter Size of the List\n"))
```

```
a=list()
print("Enter List elements")
for i in range(n):
    num=int(input())
    a.append(num)
print("\nUnsorted List")
for i in a:
    print(i,end="\t")
ns=mergeSort(a)
print("\n\nSorted List")
for i in ns:
    print(i,end="\t")
```

Output #1:

```
Enter Size of the List
5
Enter List elements
8
1
4
2
7

Unsorted List
8       1       4       2       7

Sorted List
1       2       4       7       8
```

Program #3: Python Program to implement Quick Sort.

```
def partition(a,l,r):
    pivot=l
    i=l+1
    j=r
    while(i<=j):
        while i<=j and a[i] <= a[pivot]:
            i=i+1

        while a[j] > a[pivot]:
            j=j-1

        if i < j:
            temp=a[i]
            a[i]=a[j]
            a[j]=temp

    temp=a[pivot]
    a[pivot]=a[j]
    a[j]=temp
    return j

def quickSort(a,i,j):
    if i<j:
        s=partition(a,i,j)
        quickSort(a,i,s-1)
        quickSort(a,s+1,j)


n=int(input("Enter Size of the List\n"))
a=list()
print("Enter List elements")
for i in range(n):
    num=int(input())
    a.append(num)
print("\nUnsorted List")
for i in a:
    print(i,end="\t")
quickSort(a,0,n-1)
print("\n\nSorted List")
for i in a:
    print(i,end="\t")
```

Output #1:

```
Enter Size of the List
5
Enter List elements
8
1
6
```

```
4
7

Unsorted List
8	1	6	4	7

Sorted List
1	4	6	7	8
```

Program #4: Python Program to implement Fibonaci Sequence.

```
def  Fib(n):
   f1=0
   f2=1
   a=list()
   i=2
   a.append(f1)
   a.append(f2)
   for i in range(2,n):
      f3=f1+f2
      a.append(f3)
      f1=f2
      f2=f3
   return a

n=int(input("Enter size of Fibonacci Sequence\n"))
fibList = Fib(n)
print("Fibonacci Sequence:")
for i in fibList:
   print(i,end="\t")
```

Output #1:

```
Enter size of Fibonacci Sequence
8
Fibonacci Sequence:
0	1	1	2	3	5	8	13
```

Activity #4

1. **Develop Python Program to implement Merge Sort to sort item in Descending order.**
2. **Develop Python Program to implement Quick Sort to sort item in Descending order.**
3. **Develop Python Program to generate Fibonacci Sequence of 10 numbers. First Two numbers should be 2 and 5.**
4. **Apply Merge sort and Quick sort on following list:**

   10, 8, 11, 4, 9, 15, 7, 14