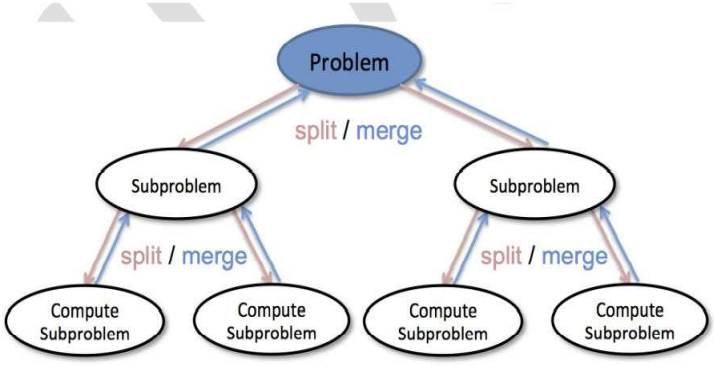
Divide and Conquer

- Divide and Conquer approach basically works on breaking the problem into sub problems that are similar to the original problem but smaller in size & simpler to solve.
- Once divided sub problems are solved recursively and then combine solutions of sub problems to create a solution to original problem.
- At each level of the recursion the divide and conquer approach follows three steps:
    - **Divide:** In this step whole problem is divided into several sub problems.
    - **Conquer:** The sub problems are conquered by solving them recursively, only if they are small enough to be solved, otherwise step1 is executed.
    - **Combine:** In this final step, the solution obtained by the sub problems are combined to create solution to the original problem.



- Examples: The specific computer algorithms are based on the Divide & Conquer approach:
    - Sorting algorithms such as Quick sort and Merge Sort
    - Binary Search
    - Binary Tree

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
- This is possible as the list is sorted and it is much quicker than linear search. Here we divide the given list and conquer by choosing the proper half of the list. We repeat this approach till we find the element or conclude about it's absence in the list.

- # implementation of Binary Search Algorithm using Iterative method

```
# binary search function definition
def binarySearch(array, key, low, high):
    while low <= high:
        mid = (low+high)//2
        if  key==array[mid]:
            return mid
        elif  key<array[mid] :
            high= mid -11
        else:
            low = mid +1
    return -1

# Code to call the binary search function
array = [3, 4, 5, 6, 7, 8, 9]
key=int(input("Enter the key to be searched:"))
result = binarySearch(array,key, 0,len(array)-1)
if result != -1:
    print("Element is present at %d index " %result)
else:
    print("Not found")
```

- # implementation of Binary Search Algorithm using Recursion

```
# binary search function definition
def binarySearch(array, key, low, high):
    if low <= high:
        mid = (low+high)//2
        if  key==array[mid]:
            return mid
        elif  key<array[mid] :
            binarySearch(array,key,low,mid-1)
        else:
            binarySearch(array,key,mid+1,high)
    return -1

# Code to call the binary search function
array = [3, 4, 5, 6, 7, 8, 9]
key=int(input("Enter the key to be searched:"))
result = binarySearch(array,key, 0,len(array)-1)
if result != -1:
    print("Element is present at %d index " %result)
else:
    print("Not found")
```

The advantages of binary search algorithm are-

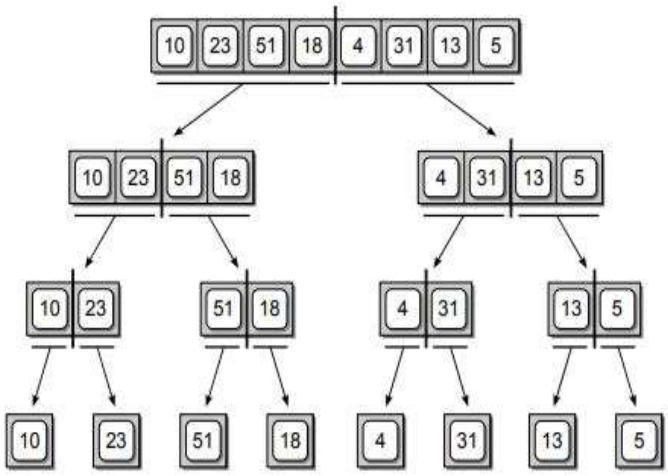
- It eliminates half of the list from further searching by using the result of each comparison.
- It indicates whether the element being searched is before or after the current position in the list.
- This information is used to narrow the search.
- For large lists of data, it works significantly better than linear search.

Disadvantages:

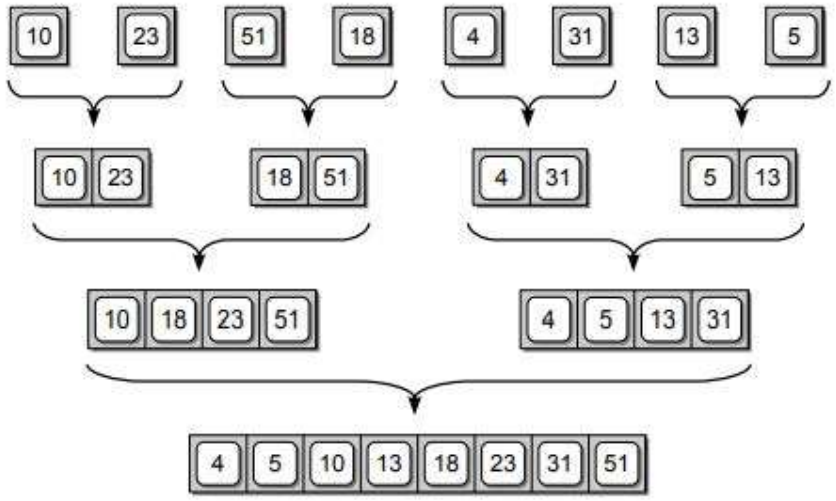
- Binary search can only be implemented on sorted data.

Merge Sort :

- The merge sort algorithm uses the divide and conquer strategy to sort the keys stored in a mutable sequence.
- The sequence of values is recursively divided into smaller and smaller subproblems until each value is contained within its own subsequences. The subsequences are then merged back together to create a sorted sequence.
- The algorithm starts by splitting the original list of values in the middle to create two sublists, each containing approximately the same number of values.
- The following figure shows Recursively splitting a list until each element is contained within its own list.



- After the list has been fully subdivided into individual sublists, the sublists are then merged back together, two at a time, to create a new sorted list.
- These sorted lists are themselves merged to create larger and larger lists until a single sortedlist has been constructed.
- During the merging phase, each pair of sorted sublists are merged to create a new sorted list containing all of the elements from both sublists.
- The following figure shows how the sublists are merged.



Implementation of Mergesort:

```
# Function definition for Mergesort
    def mergesort(list1):
        if len(list1)>1:
            mid=len(list1)//2 # Divide list into 2 halves
            left=list1[:mid]
            right=list1[mid:]
            mergesort(left)
            mergesort(right)
            i=j=k=0
             while i<len(left)andj<len(right):
                if left[i]<right[j]:
                     list1[k]=left[i]
                     i+=1
                else:
                   list1[k]=right[j]
                   j+=1
                k+=1
```

```
        while i<len(left):
          list1[k]=left[i]
          i+=1
          k+=1
        while j<len(right):
         list1[k]=right[j]
         j+=1
         k+=1
    return list1

 list1=[]
 n=int(input("Enter the size of  list"))
for i in range(n):
     list1.append(int(input("Enter the number")))
print("Before sorting: The list items are")
for i in range(len(list1)):
   print(list1[i],end=" ")
list1=mergesort(list1)
print()
for i in range(len(list1)):
   print(list1[i],end=" ")
```

Time Complexity:

The time complexity of mersort algorithm is O(nlogn)

Quick Sort

- The quick sort algorithm also uses the divide and conquer strategy. But unlike the merge sort, which splits the sequence of keys at the midpoint, the quicksort partitions the sequence by dividing it into two segments based on a selected pivot key
- The quick sort is a simple recursive algorithm that can be used to sort keys stored in either an array or list. Given the sequence, it performs the following steps:
    1. The first key is selected as the pivot, p. The pivot value is used to partition the sequence into two segments or subsequences, Left and Right, such that Left contains all keys less than the pivot element
    2. And Right contains all keys greater than or equal to p.
    3. The algorithm is then applied recursively to both Left and Right. The recursion continues until the base case is reached, which occurs when the list contains lesser than two keys.
    4. The two segments and the pivot value are merged to produce a sorted sequence.

Implementation:

```
# function definition to partition the array
def partition(array,start,end):
    pivot=array[start]
    low=start+1
    high=end
    while True:
      while low<=high and array[low]<pivot:
          low+=1
      while low<=highand array[high]>=pivot:
          high=high-1
       if low<=high:
        array[low],array[high]=array[high],array[low]
       else:
          break
    array[start],array[high]=array[high],array[start]
    return high

#Code for the Quicksort() which divides the array into two halves
def Quicksort(array,start,end):
    if start>=end:
       return
    p=partition(array,start,end)
    Quicksort(array,start,p-1)
    Quicksort(array,p+1,end)

#driver code to call Quicksort
function
array=[]
n=int(input("Enter the size of  list:"))
for i in range(n):
        array.append(int(input("Enter the %d number:" %i)))
print("Before sorting: The list items are")
for i in range(len(array)):
    print(array[i],end=" ")
Quicksort(array,0,len(array)-1)
print()
print("After sorting: The list items")
```

```
print(array)
```

The time complexity of Quicksort is O(nlogn)