EXPERIMENT NO. - 09

AIM: Edit/compile/run a program to implement quick sort/ merge sort/ bubble sort.

THEORY:

Python Program for QuickSort:

QuickSort is a **divide and conquer algorithm**. It picks an element as a pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

1. Always pick the first element as a pivot
2. Always pick the last element as a pivot
3. Pick a random element as a pivot
4. Pick median as a pivot

Here we will be picking the last element as a pivot. The key process in quickSort is partition(). Target of partitions is, given an array and an element 'x' of array as a pivot, put x at its correct position in a 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: Recursive QuickSort function

PROGRAM A) QUICK SORT ALGORITHM

```
# Python program for implementation of Quicksort Sort

# This implementation utilizes pivot as the last element in the nums list
# It has a pointer to keep track of the elements smaller than the pivot
# At the very end of partition() function, the pointer is swapped with
the pivot
# to come up with a "sorted" nums relative to the pivot



# Function to find the partition position
def partition(array, low, high):

    # choose the rightmost element as pivot
    pivot = array[high]

    # pointer for greater element
    i = low - 1

    # traverse through all elements
    # compare each element with pivot
    for j in range(low, high):
        if array[j] <= pivot:

            # If element smaller than pivot is found
            # swap it with the greater element pointed by i
            i = i + 1

            # Swapping element at i with element at j
            (array[i], array[j]) = (array[j], array[i])

    # Swap the pivot element with the greater element specified by i
    (array[i + 1], array[high]) = (array[high], array[i + 1])
    # Return the position from where partition is done
    return i + 1

# function to perform quicksort


def quickSort(array, low, high):
    if low < high:

        # Find pivot element such that
        # element smaller than pivot are on the left
        # element greater than pivot are on the right
        pi = partition(array, low, high)

        # Recursive call on the left of pivot
        quickSort(array, low, pi - 1)

        # Recursive call on the right of pivot
        quickSort(array, pi + 1, high)


data = [1, 7, 4, 1, 10, 9, -2]
print("Unsorted Array")
print(data)

size = len(data)

quickSort(data, 0, size - 1)

print('Sorted Array in Ascending Order:')
print(data)
```

OUTPUT:

```
Unsorted Array

[1, 7, 4, 1, 10, 9, -2]

Sorted Array in Ascending Order:

[-2, 1, 1, 4, 7, 9, 10]
```

***Merge sort** is defined as a sorting algorithm that works by dividing an array into smaller subarrays, sorting each subarray, and then merging thsubarrays back together to form the final sorted array.*
In simple terms, we can say that the process of merge sort is to divide the array into two halves, sort each half, and then merge the sorted halves back together. This process is repeated until the entire array is sorted.

Need for Merge Sort

One thing that you might wonder is what is the specialty of this algorithm. We already have a number of sorting algorithms then why do we need this algorithm? One of the main advantages of merge sort is that it has a time complexity of O(n log n), which means it can sort large arrays relatively quickly. It is also a stable sort, which means that the order of elements with equal values is preserved during the sort.

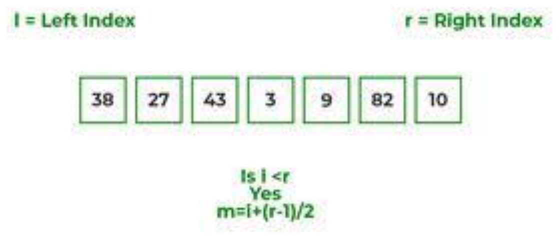
Merge sort is a popular choice for sorting large datasets because it is relatively efficient and easy to implement. It is often used in conjunction with other algorithms, such as quicksort, to improve the overall performance of a sorting routine.

Merge Sort Working Process:
Think of it as a recursive algorithm continuously splits the array in half until it cannot be further divided. This means that if the array becomes empty or has only one element left, the dividing will stop, i.e. it is the base case to stop the recursion. If the array has multiple elements, split the array into halves and recursively invoke the merge sort on each of the halves. Finally, when both halves are sorted, the merge operation is applied. Merge operation is the process of taking two smaller sorted arrays and combining them to eventually make a larger one.

Illustration:

To know the functioning of merge sort lets consider an array arr[] = {38, 27, 43, 3, 9, 82, 10}

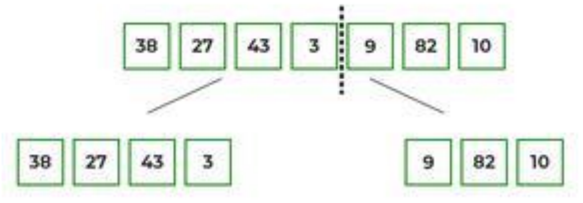
- *At first, check if the left index of array is less than the right index, if yes then calculate its mid point*



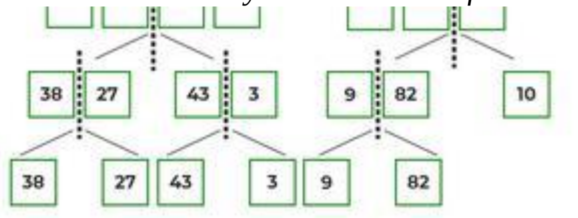
- *Now, as we already know that merge sort first divides the whole array iteratively into equal halves, unless the atomic values are achieved.*
- *Here, we see that an array of 7 items is divided into two arrays of size 4 and 3 respectively.*

l = Left Index
r = Right Index
38
27
43
3
9
82
10

- *Now, again find that is left index is less than the right index for both arrays, if found yes, then again calculate mid points for both the arrays.*

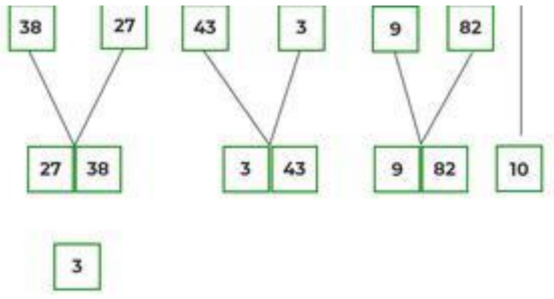


- *Now, further divide these two arrays into further halves, until the atomic units of the array is reached and further division is not possible.*

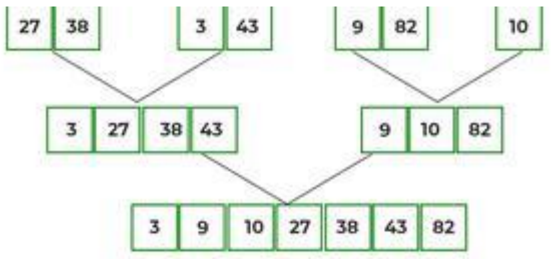


After dividing the array into smallest units merging starts, based on comparison of elements.

- *After dividing the array into smallest units, start merging the elements again based on comparison of size of elements*
- *Firstly, compare the element for each list and then combine them into another list in a sorted manner.*

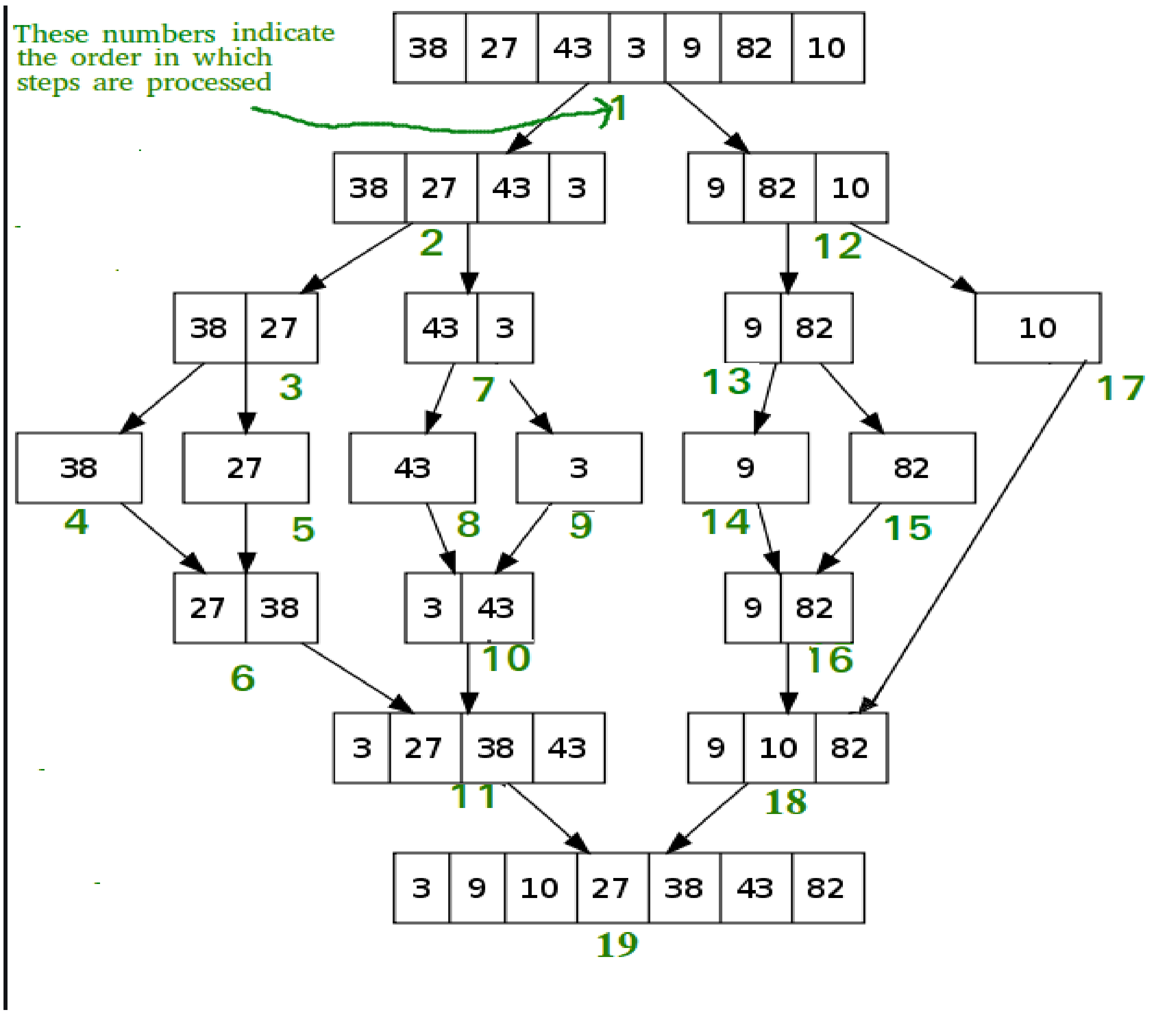


- *After the final merging, the list looks like this:*



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 into two halves till the size becomes 1. Once the size becomes 1, the merge processes come into action and start merging arrays back till the complete array is merged.



Recursive steps of merge sort

Algorithm:

```
step 1: start
step 2: declare array and left, right, mid variable
step 3: perform merge function.
    if left > right
        return
    mid= (left+right)/2
    mergesort(array, left, mid)
    mergesort(array, mid+1, right)
    merge(array, left, mid, right)
```

step 4: Stop

Follow the steps below to solve the problem:

```
MergeSort(arr[], l,  r)
If r > l
    • Find the middle point to divide the array into two halves:
        • middle m = l + (r – l)/2
    • Call mergeSort for first half:
        • Call mergeSort(arr, l, m)
    • Call mergeSort for second half:
        • Call mergeSort(arr, m + 1, r)
    • Merge the two halves sorted in steps 2 and 3:
        • Call merge(arr, l, m, r)
```

PROGRAM B) MERGE SORT ALGORITHM

```
# Python program for implementation of MergeSort
def mergeSort(arr):
    if len(arr) > 1:

         # Finding the mid of the array
        mid = len(arr)//2

        # Dividing the array elements
        L = arr[:mid]

        # into 2 halves
        R = arr[mid:]

        # Sorting the first half
        mergeSort(L)

        # Sorting the second half
        mergeSort(R)

        i = j = k = 0

        # Copy data to temp arrays L[] and R[]
        while i < len(L) and j < len(R):
            if L[i] <= R[j]:
                arr[k] = L[i]
                i += 1
            else:
                arr[k] = R[j]
                j += 1
            k += 1

        # Checking if any element was left
        while i < len(L):
            arr[k] = L[i]
            i += 1
            k += 1

        while j < len(R):
            arr[k] = R[j]
            j += 1
            k += 1

# Code to print the list


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



# Driver Code
if __name__ == '__main__':
    arr = [12, 11, 13, 5, 6, 7]
    print("Given array is", end="\n")
    printList(arr)
    mergeSort(arr)
    print("Sorted array is: ", end="\n")
    printList(arr)

# This code is contributed by Karan Korpe
```

OUTPUT:

```
Given array is

12 11 13 5 6 7

Sorted array is

5 6 7 11 12 13
```

Bubble Sort Algorithm

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order. This algorithm is not suitable for large data sets as its average and worst-case time complexity is quite high.

How does Bubble Sort Work?

Input: *arr[] = {6, 3, 0, 5}*

First Pass:

- *Bubble sort starts with very first two elements, comparing them to check which one is greater.*
    - *(6 3 0 5) –> (**3 6** 0 5), Here, algorithm compares the first two elements, and swaps since 6 > 3.*
    - *(3 **6 0** 5) –> (3 **0 6** 5), Swap since 6 > 0*
    - *(3 0 **6 5**) –> (3 0 **5 6**), Swap since 6 > 5*

Second Pass:

- *Now, during second iteration it should look like this:*
    - *(**3 0** 5 6) –> (**0 3** 5 6), Swap since 3 > 0*
    - *(0 **3 5** 6) –> (0 **3 5** 6), No change as 5 > 3*

Third Pass:

- *Now, the array is already sorted, but our algorithm does not know if it is completed.*

- *The algorithm needs one **whole** pass without **any** swap to know it is sorted.*
    - *(**0 3** 5 6) –> (**0 3** 5 6), No change as 3 > 0*

Array is now sorted and no more pass will happen.

Follow the below steps to solve the problem:

- Run a nested for loop to traverse the input array using two variables **i** and **j**, such that 0 ≤ i < n-1 and 0 ≤ j < n-i-1
- If **arr[j]** is greater than **arr[j+1]** then swap these adjacent elements, else move on
- Print the sorted array

Below is the implementation of the above approach:

PROGRAM C) BUBBLE SORT ALGORITHM

```
# Python program for implementation of Bubble Sort


def bubbleSort(arr):

    n = len(arr)

    # optimize code, so if the array is already sorted, it doesn't need

    # to go through the entire process

    swapped = False

    # Traverse through all array elements

    for i in range(n-1):

        # range(n) also work but outer loop will

        # repeat one time more than needed.

        # Last i elements are already in place

        for j in range(0, n-i-1):



            # traverse the array from 0 to n-i-1
            # Swap if the element found is greater

            # than the next element

            if arr[j] > arr[j + 1]:

                swapped = True

                arr[j], arr[j + 1] = arr[j + 1], arr[j]



        if not swapped:

            # if we haven't needed to make a single swap, we

            # can just exit the main loop.

            return




# Driver code to test above

arr = [64, 34, 25, 12, 22, 11, 90]



bubbleSort(arr)



print("Sorted array is:")

for i in range(len(arr)):

    print("% d" % arr[i], end=" ")
```

# This code is modified by Karan Korpe.

OUTPUT:

```
Sorted array:

11 12 22 25 34 64 90
```

Conclusion: Hence, we have successfully studied about program for implementing quick sort/ merge sort/ bubble sort.