AIM:

Implementing and analysing merge and quick sort in C

Theory:

Merge Sort

Merge sort is a divide-and-conquer algorithm that sorts an array by dividing it into two halves, recursively sorting each half, and then merging the two sorted halves into a single sorted array. The merge step involves comparing elements from the two sorted sub-arrays and placing them in order into a new array. This process is repeated until all elements are merged back into a single sorted array.

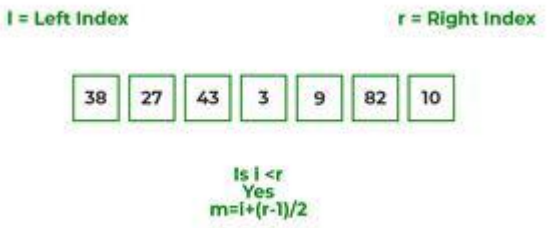
The time complexity of merge sort is O(nlogn) in the worst, average, and best cases. This is because the algorithm recursively divides the array into two halves, and then merges the sorted halves, with each merge operation taking O(n) time. While merge sort has a higher space complexity compared to other sorting algorithms, it is considered a stable sorting algorithm and is often used in external sorting algorithms where data is too large to fit in memory.

Algorithm

Example

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

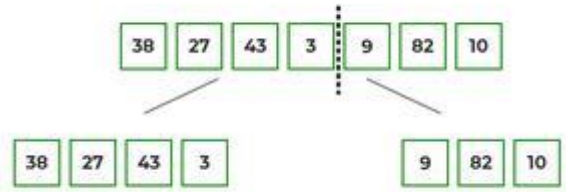
At first, check if the left index of array is less than the right index, if yes then calculate its mid point



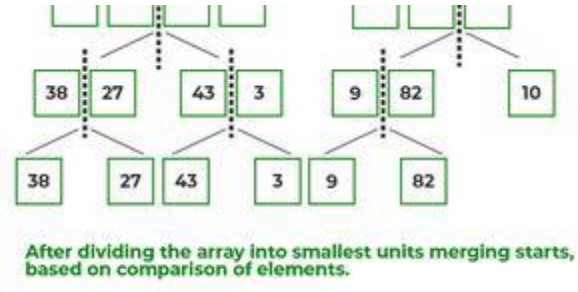
Now, as we already know that merge sort first divides the whole array iteratively into equal halves, unless the atomic values are achieved.

Here, we see that an array of 7 items is divided into two arrays of size 4 and 3 respectively.

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.

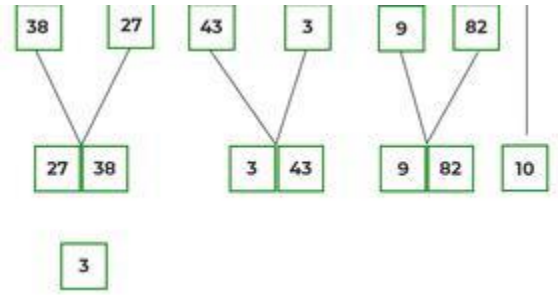


Now, further divide these two arrays into further halves, until the atomic units of the array is reached and further division is not possible.

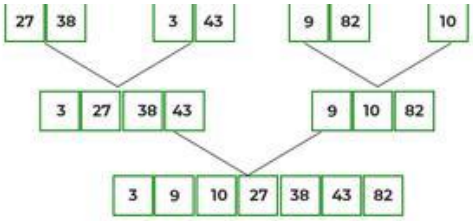


After dividing the array into smallest units, start merging the elements again based on comparison of size of elements

Firstly, compare the element for each list and then combine them into another list in a sorted manner.

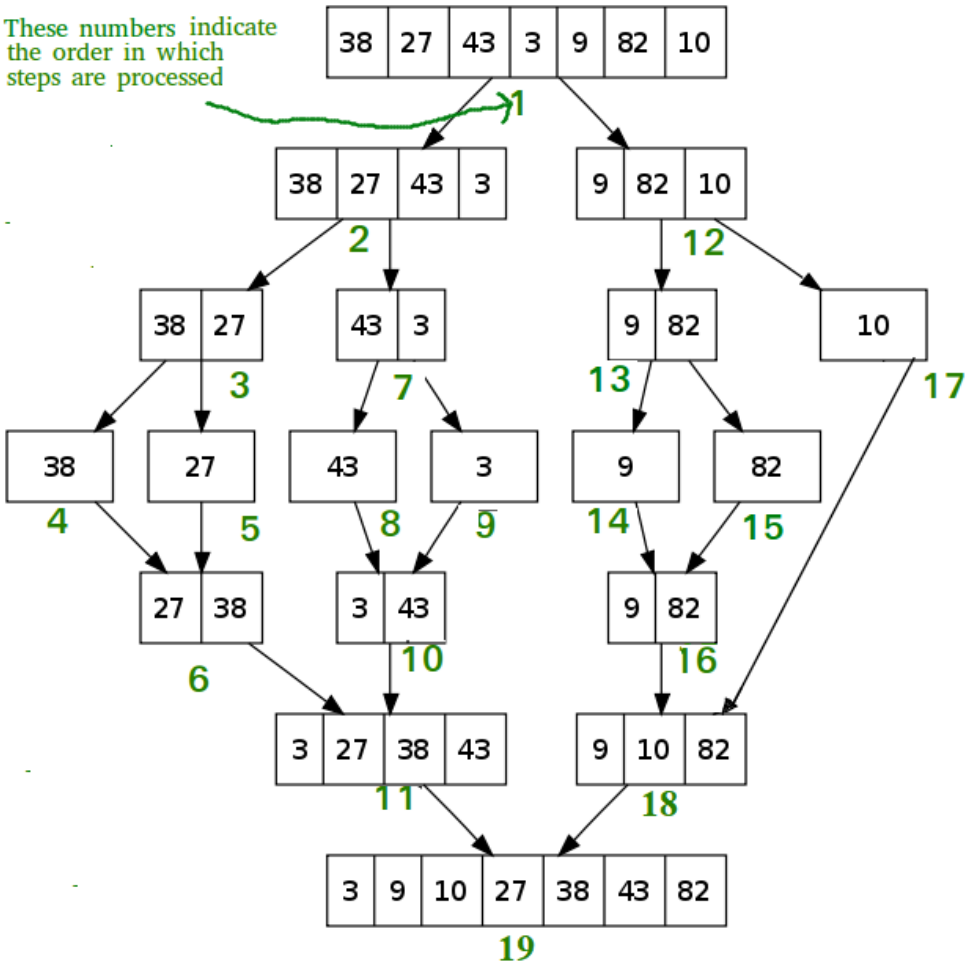


After the final merging, the list looks like this:



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.



Quick Sort

Quick sort is a comparison-based sorting algorithm that follows the divide-and-conquer approach. The algorithm works by partitioning the array into two parts around a pivot element, such that all elements on the left of the pivot are smaller than the pivot, and all elements on the right are larger. This is done recursively for each subarray, until the array is sorted.

The choice of the pivot element is crucial for the efficiency of the algorithm. Ideally, the pivot element should be the median of the subarray. However, finding the median is expensive and impractical. A commonly used approach is to choose the last element of the subarray as the pivot, but this can lead to worst-case performance if the input array is already sorted.

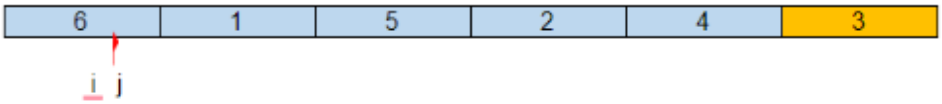
Quick sort has an average-case time complexity of O(nlogn), making it one of the fastest sorting algorithms in practice. However, the worst-case time complexity is O(n^2), which occurs when the pivot element is either the largest or smallest element of the subarray. To avoid this, several variants of quick sort have been developed, such as randomized quick sort, which randomly chooses the pivot element, and three-way quick sort, which handles arrays with many duplicate elements more efficiently.

Overall, quick sort is a versatile and efficient algorithm that is widely used in practice for sorting large datasets.

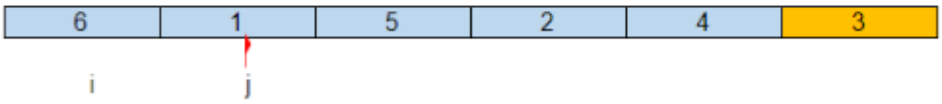
Example

Suppose the initial input array is:

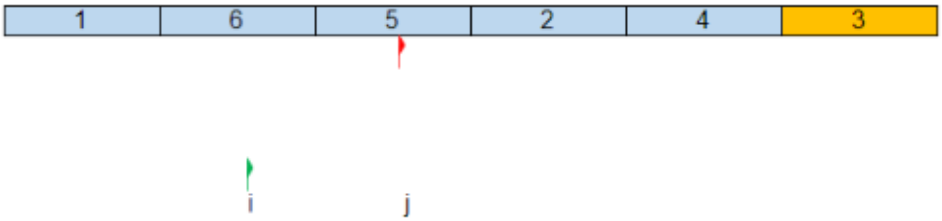
Here, 3 is chosen our pivot element and *partition()* function works as follows:



so a[j] is not <= pivot, we continue



now a[j] <= pivot so we swap a[i] and a[j], then increment i



similarly all other elements are traversed:

Now after j loop is over, we swap a[i] with the pivot

So after our first partition, the pivot element is at its correct place in the sorted array, with all its smaller elements to the left and greater to the right.

Now *QuickSort()* will be called recursively for the subarrays {1,2} and {6,4,5} and continue till the array is sorted.

Algorithm:

Merge Sort:

```
MergeSort(A,low,high)
        Mid=(low+high)/2
        MergeSort(A,low,mid)
        MergeSort(A,mid+1,high)
        Merge (A,low,mid,high)
Merge(A,low,mid,high)
        i=low
        j=mid+1
        k=low
        while(i<=mid && j<=high)
                if(A[i]<=A[j])
                        B[k]=A[i]
                        i++
                else
                        B[k]=A[j]
                        j++
                k++
        while(j<=high)
                B[k]=A[j]
                j++
                k++
        while(j<=high)
                B[k]=A[i]
                i++
                k++
        for i=low to high
                A[i] = B[i]
```

Quick Sort:

```
QuickSort(A,low,high)
        If(low<high)
                P = Partition(A,low,high)
                Quicksort(A,low,P-1)
                Quicksort(A,P+1,high)
Partition(A,low,high)
        pivot = A[high]
        i=low
        j=low
        while(i<high)
                if(A[i]<pivot)
                        swap(A[i],A[j])
                        j++
                i++
        swap(A[j],A[high])
```

Code:

Merge Sort

```
#include<stdio.h>
#include<time.h>

void merge(int a[],int low,int mid,int high)
{
  int b[10000];
  int i=low;
  int j=mid+1;
  int k=low;
  while(i<=mid && j<=high)
  {
```

```
    if (a[i]<a[j])
    {
        b[k]=a[i];
        i++;
    }
    else
    {
        b[k]=a[j];
        j++;
    }
    k++;
}
if(i>mid)
{
    while(j<=high)
    {
        b[k]=a[j];
        j++;
        k++;
    }
}
if(j>high)
{
    while(i<=mid)
    {
        b[k]=a[i];
        i++;
        k++;
    }
}
```

```
  for(i=low;i<=high;i++)
  {
     a[i]=b[i];
  }
}
void merge_sort(int a[], int low,int high)
{
  if(low<high)
  {    int mid;
     mid=(low+high)/2;
     merge_sort(a,low,mid);
     merge_sort(a,mid+1,high);
     merge(a,low,mid,high);

  }

}

void main ()
{
   clock_t start,end;
   double time1;
   start=clock();
   int a[10000],low,high,n;
   printf("Enter number of elements");
   scanf("%d",&n);
   for(int i=0;i<n;i++)
   {
      // printf("Enter value");
      // scanf("%d",&a[i]);
```

```
        a[i]=rand();
    }

    low=0;
    high=n-1;
    merge_sort(a,low,high);
    // for(int i=0;i<n;i++)
    // {
    //     printf("%d\t",a[i]);
    // }
    end=clock();
    time1=(double)(end-start);
    printf("\n time taken is %f",time1);

}
```

Quick Sort

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = (low - 1);
```

```
    for (int j = low; j <= high - 1; j++) {
        if (arr[j] < pivot) {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

void quick_sort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quick_sort(arr, low, pi - 1);
        quick_sort(arr, pi + 1, high);
    }
}

int main() {
    srand(time(NULL));
    int arr[1000],n;
    printf("Enter number of elements: ");
    scanf("%d",&n);
    for (int i = 0; i < n; i++) {
        arr[i] = rand() % 100;
    }

    // printf("Unsorted array:\n");
    // for (int i = 0; i < 1000; i++) {
    //     printf("%d ", arr[i]);
```

```
    // }
    // printf("\n");

    clock_t start = clock(); // Start time measurement
    quick_sort(arr, 0, n-1);
    clock_t end = clock(); // End time measurement
    double elapsed_time = ((double) (end - start)) / CLOCKS_PER_SEC;
    // printf("Sorted array:\n");
    // for (int i = 0; i < 1000; i++) {
    //     printf("%d ", arr[i]);
    // }
    printf("\n");
    printf("Time taken: %f seconds\n", elapsed_time);
    return 0;
}
```

Output:

Merge Sort

```
main.c: In function 'main':
main.c:72:14: warning: implicit declaration of function 'rand' [-Wimplicit-function-declaration]
   72 |         a[i]=rand();
      |              ^~~~
Enter number of elements1000

 time taken is 239.000000

...Program finished with exit code 0
Press ENTER to exit console.
```

Quick Sort

```
Enter number of elements: 1000

Time taken: 0.000098 seconds


...Program finished with exit code 0
Press ENTER to exit console.
```

Conclusion:

In summary, merge sort is a stable, comparison-based algorithm with time complexity O(nlogn) and is ideal for large datasets. Quick sort is faster for small to medium-sized datasets, has average time complexity O(nlogn), but is not stable and can have performance issues with certain data distributions.