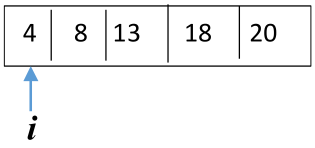
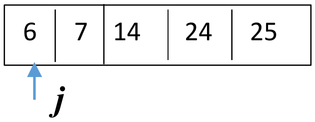
Merge Sort

Merging Operation:

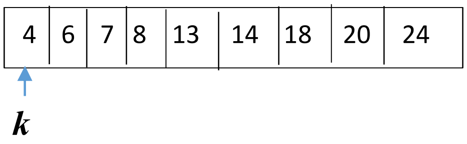
Given 2 sorted arrays P[ ] and Q[ ] we have to create a third sorted array R[ ] such that the R = P ∪ Q.



P[ ]



Q[ ]



R[ ]

Here we can see both P[ ] and Q[ ] are sorted array being indexed by *i* and *j*. We have to create a third sorted array R[ ] such that R = P ∪ Q. R is indexed by *k*. We compare P[*i*] with Q[*j*] and put the minimum of these two element in R[*k*] and increment the index *k*. If the element is copied from P[ ] to R[ ] we increment *i* otherwise the element is copied from Q [ ] to R[ ] and we increment *j*. Thus we keep on incrementing index pointers *i* and *j* in alternate fashion till we exhaust one of the 2 arrays namely P[ ] or Q[ ]. Lastly we copy the remaining elements of the unfinished array of P[ ] or Q[ ] in the array R[ ].

If the size of the arrays P[ ] and Q[ ] are *m* and *n* respectively then the time to perform merge operation is O(*m*+*n*) since the index pointer *k* fills up *m*+*n* elements in the array R[ ] without backtracking.

```
#include<stdio.h>
#define MAX 1000

void merge(int P[], int Q[], int R[], int m, int n)
{
       int i=0; int j=0; int k=0;

       while((i<m)&&(j<n))
       {
             if(P[i] < Q[j]) {R[k]=P[i]; i++; k++;}
             else {R[k] = Q[j]; j++; k++;}

       }

      while(i<m)
      { R[k] = P[i];
        i++; k++;
      }

      while(j<n)
      { R[k] = Q[j];
        j++; k++;
      }
}

void mergesort(int A[], int low, int high)
{
       int mid;
       int i, j, k, l, m, n;
       int P[MAX], Q[MAX], R[MAX];
       if (low < high)
       {
            mid = (low+high)/2;
            mergesort(A, low, mid);
            mergesort(A, mid+1, high);
            i=low; m=0;
            while(i<=mid) {
                 P[m] = A[i]; i++; m++;
             }
             j=mid+1; n=0;
             while(j<=high){
                 Q[n]= A[j]; j++; n++;
             }
            merge(P, Q, R, m, n);
            l=low; k=0;
            while(l<=high){
                 A[l]=R[k]; l++; k++;
             }

       }
}

int main()

{
      int A[MAX]; int i,  size;

      char c;

      printf("Enter the number of elements of the array\n");

      scanf("%d", &size);
```

```
      printf("Enter the elements of the array\n");

      for (i=0; i<size; i++)
      {
            scanf("%d", &A[i]);

      }

      mergesort(A,0,size-1);

      for (i=0; i<size; i++)
      {
            printf("%d\n", A[i]);

      }

printf("Enter character\n");

c=getchar();

return(0);

}
```

Merge Sort

Merge sort splits the array A[*low*..*high*] to be sorted in two halves A[*low*..*mid*] and A[*mid*+1..*high*] where *mid* = (*low*+*high*)/2 and recursively invokes mergesort function on these two halves. Then it invokes the merging routine to mege already sorted two arrays A[*low*..*mid*] and A[*mid*+1..*high*] to the original array A[*low*..*high*].

This merge-sort function is invoked from the driver or main function with *low* =0 and *high* = *size* where *size* is the number of elements in the array A[ ].

Time complexity Analysis:

If T(n) is the time complexity of the program where *size* = n then we can write

T(n) = 2T(n/2) + $c.n$ since we have to spend $c.n$ time for the merging operation on 2 sub-arrays of size n/2 and 2T(n/2) term comes for 2 recursive invocations. Thus T(n) $\in$ O($n$log$n$).