Bubble sort

- **Bubble sort**, sometimes referred as **sinking sort.**
- It is a simple sorting algorithm that works by repeatedly stepping through the list to be sorted, comparing each pair of adjacent items and swapping them if they are in the wrong order.
- The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted.
- The algorithm gets its name from the way smaller elements "bubble" to the top of the list.
- As it only uses comparisons to operate on elements, it is a comparison sort.
- Although the algorithm is simple, it is too slow for practical use, even compared to insertion sort.

Algorithm

```
for i ← 1 to n do
        for j ← 1 to n-i do
                If Array[j] > Array[j+1] then       /* For decreasing order use < */
                        temp ← Array[j]
                        Array[j] ← A [j+1]
                        Array[j+1] ← temp
```

Program

```
#include <stdio.h>
void main()
{
        int array[100], n, i, j, temp;
        printf("Enter number of elements\n");
        scanf("%d", &n);
        printf("Enter %d integers\n", n);
        for (i = 0; I < n; i++)
        {
                scanf("%d", &array[i]);
        }
        for (i = 0 ;i< ( n - 1 );i++)
        {
                for (j = 0 ; j< n - c - 1; j++)
                {
                        if (array[j] > array[j+1])              /* For decreasing order use < */
                        {
```

```
                        temp      = array[j];
                        array[j]  = array[j+1];
                        array[j+1] = temp;
                    }
                }
        }
        printf("Sorted list in ascending order:\n");
        for (i = 0 ;i< n ;i++ )
        {
                printf("%d\n", array[i]);
        }
        getch();
}
```

Example

Consider an array A of 5 element

A[0]	45
A[1]	34
A[2]	56
A[3]	23
A[4]	12

Pass-1: The comparisons for pass-1 are as follows.

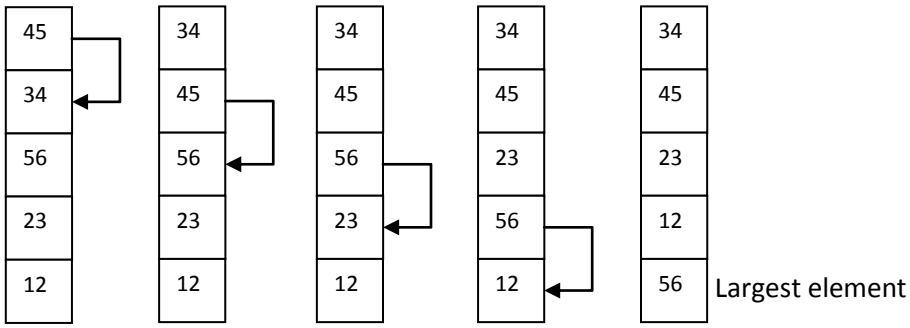
Compare A[0] and A[1]. Since 45>34, interchange them.

Compare A[1] and A[2]. Since 45<56, no interchange.

Compare A[2] and A[3]. Since 56>23, interchange them.

Compare A[3] and A[4]. Since 56>12 interchange them.

At the end of first pass the largest element of the array, 56, is bubbled up to the last position in the array as shown.

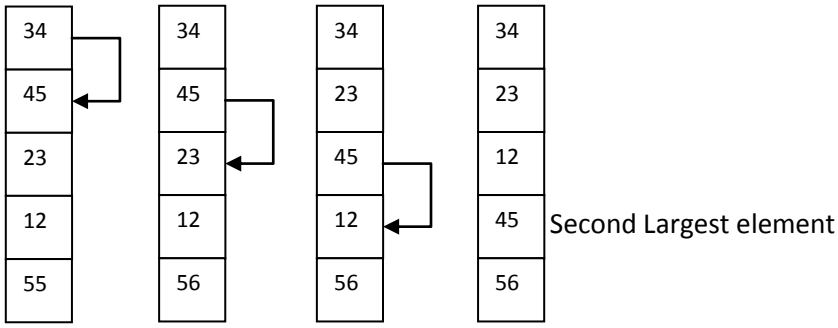


Pass-2: The comparisons for pass-2 are as follows.

Compare A[0] and A[1]. Since 34<45, no interchange.

Compare A[1] and A[2]. Since 45>23, interchange them.

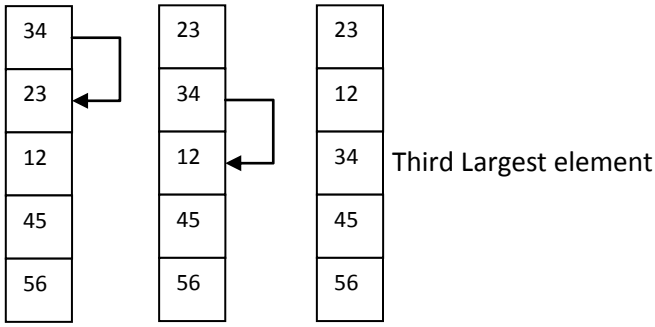
Compare A[2] and A[3]. Since 45>12, interchange them.



Pass-3: The comparisons for pass-3 are as follows.

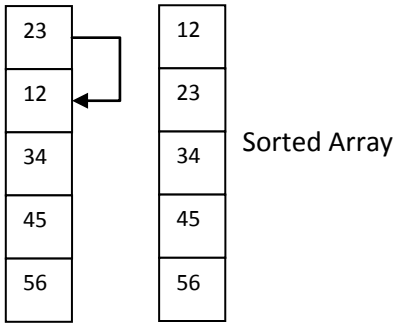
Compare A[0] and A[1]. Since 34>23, interchange them.

Compare A[1] and A[2]. Since 34>12, interchange them.



Pass-4: The comparisons for pass-4 are as follows.

Compare A[0] and A[1]. Since 23>12, interchange them.



Selection Sort

- The idea of algorithm is quite simple.
- Array is imaginary divided into two parts - sorted one and unsorted one.
- At the beginning, sorted part is empty, while unsorted one contains whole array.
- At every step, algorithm finds minimal element in the unsorted part and adds it to the end of the sorted one.
- When unsorted part becomes empty, algorithm stops.

Algorithm

SELECTION_SORT (A)

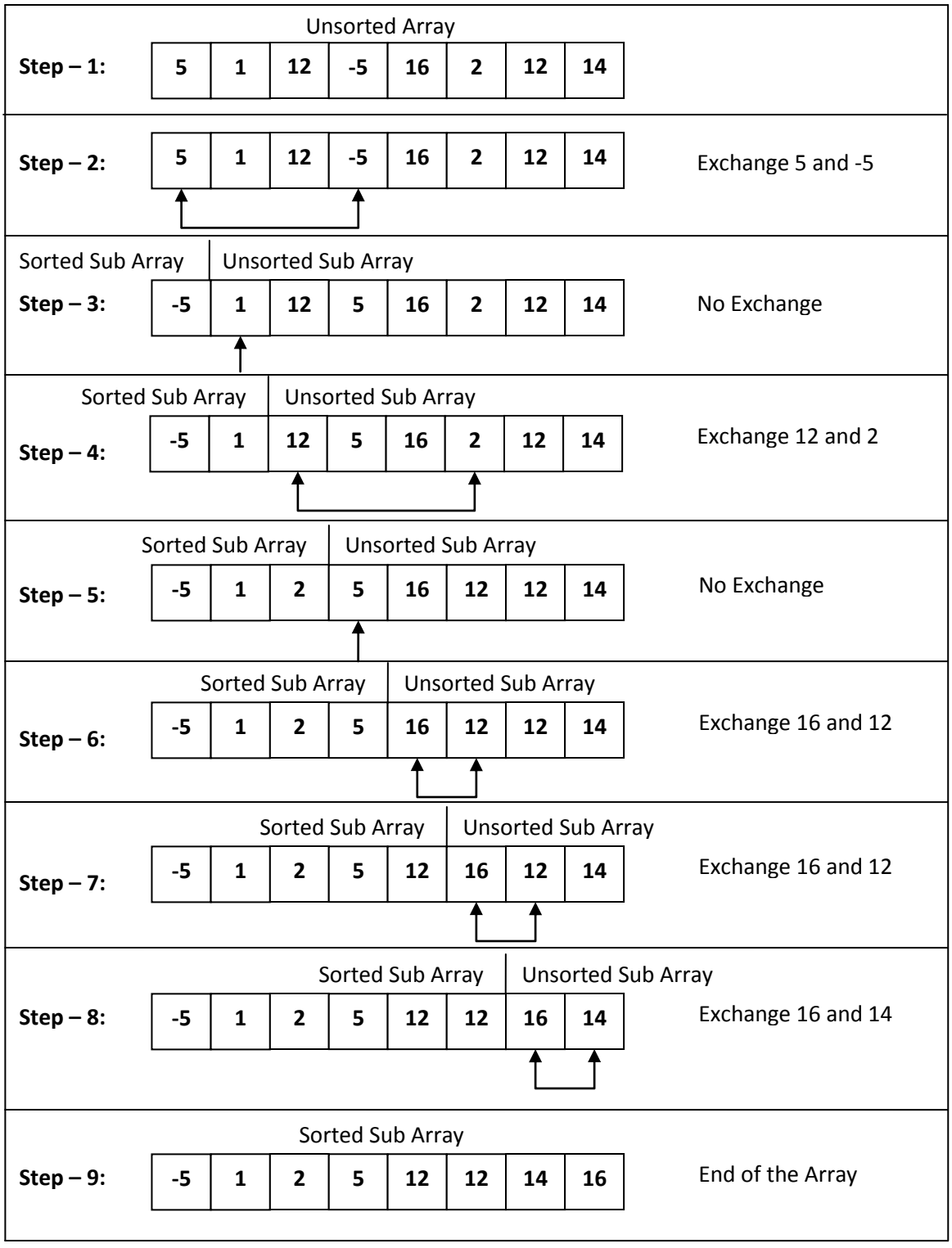
```
for i ← 1 to n-1 do
        min ← i;
        for j ← i + 1 to n do
                If A[j] < A[i] then
                        min ← j
        If min!=i then
                temp ← A[i]
                A[i] ← A [min]
                 A[min] ← temp
```

Program

```
#include <stdio.h>
void main()
{
        int array[100], n, i, j, min, temp;
        printf("Enter number of elements\n");
        scanf("%d", &n);
        printf("Enter %d integers\n", n);
        for ( i = 0 ; i < n ; i++ )
        {
                scanf("%d", &array[i]);
        }
        for ( i = 0 ; i < ( n - 1 ) ; i++ )
        {
                min = i;
                for ( j = i + 1 ; j < n ; j++ )
                {
                        if ( array[min] > array[j] )
                        min = j;
                }
```

```
            if ( min != i )
            {
                  temp = array[i];
                  array[i] = array[min];
                  array[min] = temp;
            }
      }
      printf("Sorted list in ascending order:\n");
      for ( i = 0 ; i < n ; i++ )
      {
            printf("%d\n", array[i]);
      }
      getch();
}
```

Example


Unsorted Array
Step – 1:
5 1 12 -5 16 2 12 14
Step – 2:
5 1 12 -5 16 2 12 14
Exchange 5 and -5
Sorted Sub Array
Unsorted Sub Array
Step – 3:
-5 1 12 5 16 2 12 14
No Exchange
Sorted Sub Array
Unsorted Sub Array
Step – 4:
-5 1 12 5 16 2 12 14
Exchange 12 and 2
Sorted Sub Array
Unsorted Sub Array
Step – 5:
-5 1 2 5 16 12 12 14
No Exchange
Sorted Sub Array
Unsorted Sub Array
Step – 6:
-5 1 2 5 16 12 12 14
Exchange 16 and 12
Sorted Sub Array
Unsorted Sub Array
Step – 7:
-5 1 2 5 12 16 12 14
Exchange 16 and 12
Sorted Sub Array
Unsorted Sub Array
Step – 8:
-5 1 2 5 12 12 16 14
Exchange 16 and 14
Sorted Sub Array
Step – 9:
-5 1 2 5 12 12 14 16
End of the Array

Quick Sort

- Quicksort is the currently fastest known sorting algorithm and is often the best practical choice for sorting, as its average expected running time is O(n log(n)).
- Pick an element, called a pivot, from the array.
- Reorder the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the partition operation.
- Recursively apply the above steps to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values.
- Quicksort, like merge sort, is a divide-and-conquer recursive algorithm.
- The basic divide-and-conquer process for sorting a sub array A[$i..j$] is summarized in the following three easy steps:
  - **Divide:** Partition T[$i..j$] Into two sub arrays T[$i..l$-1] and T[l+1… j] such that each element of T[$i..l$-1] is less than or equal to T[l], which is, in turn, less than or equal to each element of T[l+1… j]. Compute the index l as part of this partitioning procedure
  - **Conquer:** Sort the two sub arrays T[$i..l$-1] and T[l+1… j] by recursive calls to quicksort.
  - **Combine:** Since the sub arrays are sorted in place, no work is needed to combing them: the entire array T[$i..j$] is now sorted.

Algorithm

Procedure *pivot* (T [$i\ldots j$]; ***var*** l)
{Permutes the elements in array T [$i\ldots j$] and returns a value l such that, at the end, $i<=l<=j$, T[k] <=P for all $i \le k < l$, T[l] =P, and T[k] > P for all $l < k \le j$, where P is the initial value T[i]}
P ← T[i]
$K \leftarrow i$; $l \leftarrow j$+1
Repeat $k \leftarrow k$+1 **until** T[k] > P
Repeat $l \leftarrow l$-1 **until** T[l] ≤ P
While $k < l$ **do**
    Swap T[k] and T[l]
    Repeat $k \leftarrow k$+1 **until** T[k] > P
    Repeat $l \leftarrow l$-1 **until** T[l] ≤ P
Swap T[i] and T[l]

Procedure *quicksort* (T [$i\ldots j$])
{Sorts sub array T [$i\ldots j$] into non decreasing order}
if $j - i$ is sufficiently small **then** insert (T[$i,\ldots,j$])
else
 pivot (T[$i,\ldots,j$],l)
 quicksort (T[$i,\ldots, l$ - 1])
 quicksort (T[l+1,…,j]

Program

```
#include<stdio.h>
void quicksort(int [10],int,int);
int partition(int [10],int, int);
void main()
{
        int x[20],size,i;

        printf("Enter size of the array: ");
        scanf("%d",&size);

        printf("Enter %d elements: ",size);
        for(i=0;i<size;i++)
        {
                scanf("%d",&x[i]);
        }

        quicksort(x,0,size-1);

        printf("Sorted elements: ");
        for(i=0;i<size;i++)
        {
                printf(" %d",x[i]);
        }
        getch();
}

void quicksort(int x[10],int first,int last)
{
        Int mid;
        if(first<last)
        {
                mid= partition(int x,int first,int last)
                quicksort(x,first,mid-1);
                quicksort(x,mid+1,last);
        }
}
int partition(int x[10],int p,int r)
{
        int value, i, j, temp;
```

```
        value=x[r];
        i=p-1;
        for(j=p;j<=r-1;j++)
        {
                If(x[j] ≤value)
                {
                        i=i+1;
                        temp=x[i];
                        x[i]=x[j];
                        x[j]=temp;
                }
        }
        temp=x[i+1];
        x[i]=x[r];
        x[r]=temp;

        Return (i+1);
}
```

Example

Sort given array using Quick Sort:

2	8	7	1	3	5	6	4

(a)

i	P , j						r
2	8	7	1	3	5	6	4

(b)

P , i	j						r
2	8	7	1	3	5	6	4

(c)

P , i		j					r
2	8	7	1	3	5	6	4

(d)

P , i			j				r
2	8	7	1	3	5	6	4

Exchange 8 and 1

(e)

P	i			j			r
2	1	7	8	3	5	6	4

Exchange 7 and 3

(f)

P		i			j		r
2	1	3	8	7	5	6	4

(g)

P		i				j	r
2	1	3	8	7	5	6	4

(h)

P		i					r
2	1	3	8	7	5	6	4

Exchange 8 and 4

(i)

P		i					r
2	1	3	4	7	5	6	8

2	1	3

Left Sub Array

4

7	5	6	8

Right Sub Array

Apply same method for left and right sub array finally we will get sorted

Merge Sort

- The merge sort algorithm is based on the classical divide-and-conquer paradigm. It operates as follows:
  - **DIVIDE:** Partition the n-element sequence to be sorted into two subsequences of n/2 elements each.
  - **CONQUER:** Sort the two subsequences recursively using the merge sort.
  - **COMBINE:** Merge the two sorted subsequences of size n/2 each to produce the sorted sequence consisting of n elements.
- Note that recursion "**bottoms out**" when the sequence to be sorted is of unit length.
- Since every sequence of length 1 is in sorted order, no further recursive call is necessary.
- The key operation of the merge sort algorithm is the merging of the two sorted sub sequences in the "combine step".
- To perform the merging, we use an auxiliary procedure Merge (A,p,q,r), where A is an array and p,q and r are indices numbering elements of the array such that procedure assumes that the sub arrays A[p..q] and A[q+1...r] are in sorted order.
- It merges them to form a single sorted sub array that replaces the current sub array A[p..r]. Thus finally, we obtain the sorted array A[1..n], which is the solution.

Algorithm

MERGE (*A,p,q,r*)

```
n1 = q  -p + 1
n2 = r – q
let L[1…n1+1] and R[1…n2+1] be new arrays
for i = 1 to n1
        L[i] = A[p+i-1]
for j = 1 to n2
        R[j] = A[q+j]
L[n1+1] = infinite
R[n2+1]= infinite
i=1
j=1
for k = p to r
        if L[i] ≤ R[j]
                A[k]=L[i]
                i = i +1
        else A[k] = R[j]
                j = j + 1
```

MERGE SORT (*A,p,r*)

```
if p < r
        then q<-- [ (p + r) / 2 ]
        MERGE SORT(A,p,q)
        MERGER SORT(A,q + 1,r)
        MERGE(A,p,q,r)
```

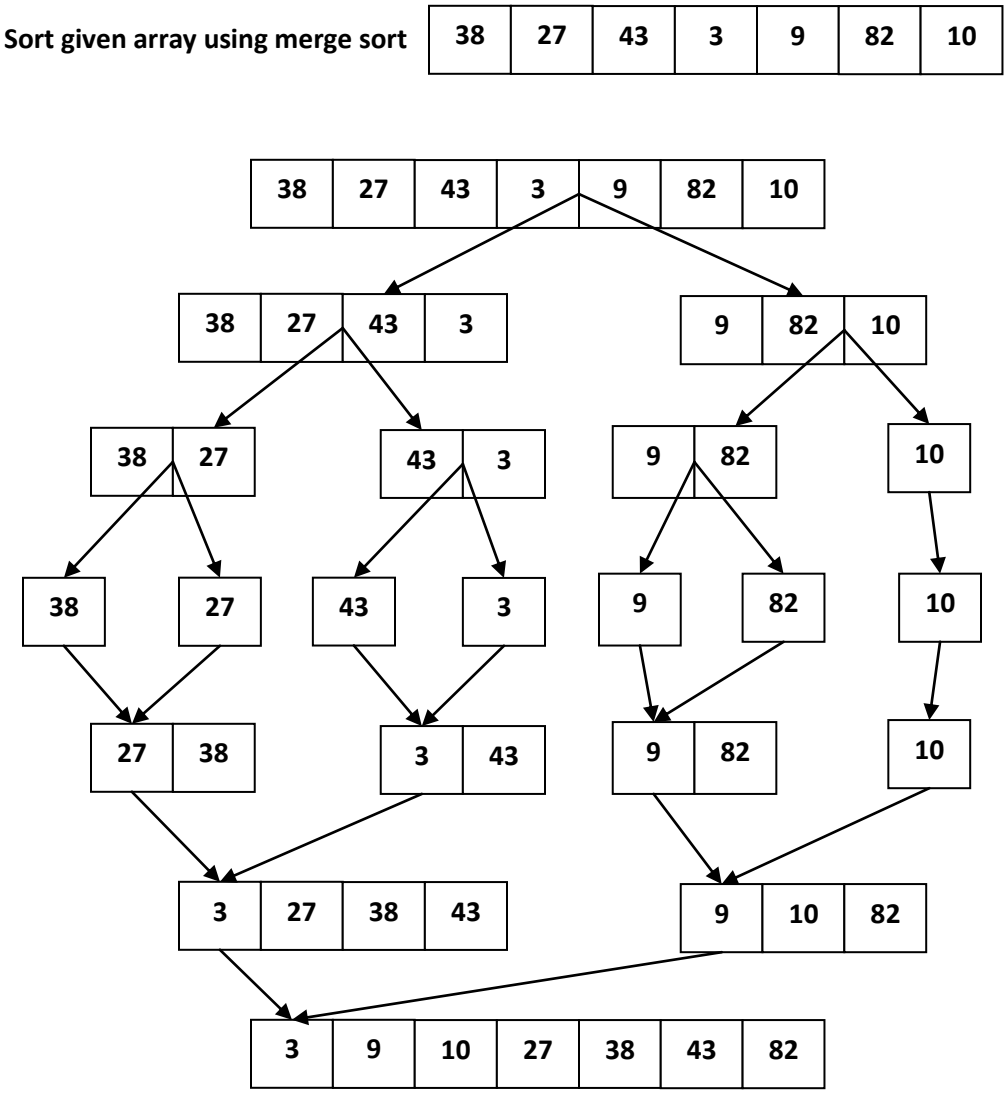
Program

```
#include<stdio.h>
void mergesort(int [20],int,int);
int merge(int [20],int, int, int);
void main()
{
        int x[20],size,i;

        printf("Enter size of the array: ");
        scanf("%d",&size);

        printf("Enter %d elements: ",size);
        for(i=0;i<size;i++)
        {
                scanf("%d",&x[i]);
        }

        mergesort(x,0,size-1);

        printf("Sorted elements: ");
        for(i=0;i<size;i++)
        {
                printf(" %d",x[i]);
        }
        getch();
}

void mergesort(int x[20],int p,int r)
{
        Int q;
        if(p<r)
```

```
        {
                q=(p+r)/2;
                mergesort(x,p,q);
                mergesort(x,q+1,r);
                merge(x,p,q,r)

        }
}
int merge(int x[20],int p,int q,int r)
{
        Int n1,n2,L[20],R[20],i,j,k;
        n1 = q  -p + 1;
        n2 = r – q;
        for(i=1; i<=n1;i++)
        {
                L[i]=x[p+i-1];
        }
        for(j=1; i<=n2;j++)
        {
                R[j]=x[q+j];
        }
        L[n1+1]=NULL;
        L[n2+1]=NULL;
        I=1;
        J=1;
        For(k=p;k<=r;k++)
        {
                If(L[i]<=R[j])
                {
                        X[k]=L[i];
                        I++;
                }
                Else
                {
                        x[k] = R[j];
                        j++;
                }
        }
}
```

Example

Sort given array using merge sort



Linear/Sequential Search

- In computer science, linear search or sequential search is a method for finding a particular value in a list that consists of checking every one of its elements, one at a time and in sequence, until the desired one is found.
- Linear search is the simplest search algorithm.
- It is a special case of brute-force search. Its worst case cost is proportional to the number of elements in the list.

Algorithm

```
# Input: Array A, integer key
# Output: first index of key in A,
# or -1 if not found

Algorith: Linear_Search
for i = 0 to last index of A:
        if A[i] equals key:
                return i
return -1
```

Program

```
#include <stdio.h>
void main()
{
        int array[100], key, i, n;

        printf("Enter the number of elements in array\n");
        scanf("%d",&n);

        printf("Enter %d integer(s)\n", n);

        for (i = 0; i < n; i++)
        {
                printf("Array[%d]=", i);
                scanf("%d", &array[i]);
        }

        printf("Enter the number to search\n");
        scanf("%d", &key);
```

```
        for (i = 0; i < n; i++)
        {
                if (array[i] == key)    /* if required element found */
                {
                        printf("%d is present at location %d.\n", key, i+1);
                        break;
                }
         }
        if (i == n)
        {
                printf("%d is not present in array.\n", search);
        }
        getch();
}
```

Example

Search for 1 in given array:

2	9	3	1	8

Comparing value of i^{th} index with element to be search one by one until we get seache element or end of the array

(a)

2	9	3	1	8

i

(b)

2	9	3	1	8

i

(c)

2	9	3	1	8

i

(d)

2	9	3	1	8

i

Element found at i^{th} index

Binary Search

- If we have an array that is sorted, we can use a much more efficient algorithm called a Binary Search.
- In binary search each time we divide array into two equal half and compare middle element with search element.
- If middle element is equal to search element then we got that element and return that index otherwise if middle element is less than search element we look right part of array and if middle element is greater than search element we look left part of array.

Algorithm

```
# Input: Sorted Array A, integer key
# Output: first index of key in A, or -1 if not found

Algorith: Binary_Search (A, left, right)
while left <= right
        middle = index halfway between left, right
        if D[middle] matches key
                return middle
        else if key less than A[middle]
                right = middle -1
        else
                left = middle + 1
return -1
```

Program

```
#include <stdio.h>
void main()
{
        int i, first, last, middle, n, key, array[100];

        printf("Enter number of elements\n");
        scanf("%d",&n);
        printf("Enter %d integers in sorted order\n", n);
        for ( i = 0 ; i < n ; i++ )
        {
                scanf("%d",&array[i]);
        }
        printf("Enter value to find\n");
        scanf("%d",&key);
```

```
        first = 0;
        last = n - 1;
        middle = (first+last)/2;

         while( first <= last )
        {
                if (array[middle] == key)
                {
                        printf("%d found at location %d.\n", key, middle+1);
                        break;
                }
                else if ( array[middle]>key )
                 {
                        Last=middle - 1;
                }
                else
                        first = middle + 1;

                middle = (first + last)/2;
         }
        if ( first > last )
        {
                printf("Not found! %d is not present in the list.\n", key);
        }
        getch();
}
```

Example

Find 6 in {-1, 5, 6, 18, 19, 25, 46, 78, 102, 114}.

Step 1 --> (middle element is 19 > 6): Search in left part

-1 5 6 18 **19** 25 46 78 102 114

Step 2 --> (middle element is 5 < 6): Search in Right part

-1 **5** 6 18

Step 3 --> (middle element is 6 == 6): Element Found

6 18