

---

# Chapter Table of Contents

## Chapter 2.2

### Sorting Techniques

Aim.....	97
Instructional Objectives.....	97
Learning Outcomes.....	97
2.2.1 Introduction.....	98
2.2.2 Basics of Sorting .....	99
Self-assessment Questions.....	105
2.2.3 Sorting Techniques .....	106
(i) The Bubble Sort .....	106
(ii) Insertion Sort.....	110
(iii) Selection Sort.....	113
(iv) Merge Sort.....	117
(v) Quick Sort.....	123
Self-assessment Questions.....	132
Summary .....	134
Terminal Questions.....	135
Answer Keys.....	135
Activity.....	136
Bibliography.....	137
e-References .....	137
External Resources .....	137
Video Links .....	137

---





## **Aim**

To educate the students in searching and sorting techniques



## **Instructional Objectives**

After completing this chapter, you should be able to:

- Explain the need of sorting
- Demonstrate bubble and insertion sort algorithms with example
- Discuss the time and space complexities of merge and quick sort algorithms



## **Learning Outcomes**

At the end of this chapter, you are expected to:

- Calculate the complexities of all sorting algorithms
- Identify the efficient algorithm
- Outline the steps to sort the unsorted numbers using quick sort

---

## 2.2.1 Introduction

Sorting is a technique to store the data in sorted order. Data can be sorted either in ascending or descending order, which can be numerical, lexicographical, or any user-defined order. The term Sorting is related to Searching of data. Here the data considered consists of only integers, but it may be anything like string or records.

In our real life, we need to search many things, like a particular record in database, students' marks in the result database, a particular person's telephone number, any students name in the list etc. The sorting process will arrange the data in a particular sequence making it easier to search whenever needed. Thus data searching can be optimized to a great extent by using sorting techniques.

Every single record to be sorted will contain one key based on which the record will be sorted. **For example**, suppose we have a record of students, every such record will have data like Roll number, name and percentage.

In the above record we can sort the record in ascending and descending order based on key i.e. Roll number. If we wish to search a student with roll no. 54, we don't need to search the complete record but we will simply search between the Students with roll no. 50 to 60, thus saving a lot of time.

**Some of the examples of sorting in real life scenarios are as followings:**

1. **Telephone Directory:** A Telephone directory keeps telephone numbers of people sorted based on their names. So that names can be searched very easily.
2. **Dictionary:** A Dictionary contains words in alphabetical order so that searching of any word becomes easy.

**Before studying any sorting algorithms, it is necessary to know about the 2 main operations involved.**

1. **Comparison:** Two values need to be compared with each other depending upon the sorting criteria. ,
2. **Exchange or Swapping:** When two values are compared with each other, if required they need to be exchanged with each other.

---

Sorting algorithm helps in arranging the elements in a particular order. Efficient sorting algorithm is important to optimize the use of other algorithms (such as search and merge algorithms) which require sorted lists to work correctly.

**More importantly, the output must satisfy two conditions:**

1. The output is in non-decreasing order (each element is not smaller than the previous element according to the desired total order).

*For example*, consider the following set of elements to be sorted: 45, 76, 2, 56, 89, 4

As per the first condition, the output of a sorting algorithm must be in non-decreasing order *i.e.*, 2, 4, 45, 56, 76, 89

2. The output is a permutation, or reordering, of the input.

The output should be always the reordering of the same elements to be sorted. You cannot add or delete or replace any element from the set.

As per this second condition the sorted list will be: 2, 4, 45, 56, 76, 89

Right from the beginning, the sorting problem has attracted a great deal of research. This is perhaps due to the complexity of solving the problem efficiently despite its simple, familiar statement.

Although many consider it a solved problem, new useful sorting algorithms are still being invented *For example*, library sort was first published in 2004). The Sorting algorithms are prevalent in introductory computer science classes. Here the abundance of algorithms for the problem provides a gentle introduction to a variety of core algorithm concepts. They are big O notation, data structures, and the divide and conquer algorithms, randomized algorithms, best, worst and average case analysis, time-space trade-offs, and lower bounds.

In this chapter, we will look at basics of sorting and sorting techniques like bubble sort, selection sort, insertion sort, merge sort and quick sort in detail.

## 2.2.2 Basics of Sorting

Data can be sorted either in ascending (increasing) or in decreasing (decreasing) order. If the order is not mentioned then it is assumed to be ascending order. In this chapter sorting is

---

done by ascending order. These algorithms can be made to work for descending order also by making simple modifications.

## Sort Stability

Sort stability comes into the picture if the key on which the data is being sorted is not unique for each record. I.e. two or more records have identical keys. **For example**, consider a list of records where each record contains the name and age of a person. Consider name as sort key and sort all the records according to the names as shown in table 2.1.1.

*Table 2.1.1: Unsorted List*

Name	Age
Vineet	25
Amit	37
Deepa	67
Shriya	45
Deepa	20
Kiran	18
Deepa	56

*Table 2.1.2: Sorted-Unstable List*

Name	Age
Amit	37
Deepa	56
Deepa	67
Deepa	20
Kiran	18
Shriya	45
Vineet	25

---

Table 2.1.3: Sorted Unstable List

Name	Age
Amit	37
Deepa	20
Deepa	56
Deepa	67
Kiran	18
Shriya	45
Vineet	25

Table 2.1.4: Sorted-Stable List

Name	Age
Amit	37
Deepa	67
Deepa	20
Deepa	56
Kiran	18
Shriya	45
Vineet	25

Any sorting algorithm would place (Amit, 37) in 1st position, (Kiran, 18) in 5th position, (Shriya, 45) in 6th position and (Vineet, 25) in 7th position. There are identical keys (names), which are (Deepa, 67), (Deepa, 20) and (Deepa, 56) and any sorting algorithm would place them in adjacent locations i.e. 2nd, 3rd, and 4th locations but not necessarily in same relative order.

A sorting algorithm is said to be stable if it maintains the relative order of the duplicate keys in the sorted output. *i.e.*, if the keys are equal then their relative order in the sorted output is

---

the same. ***For example,*** the records  $R_i$  and  $R_j$  have equal keys and if the record  $R_j$  precedes record  $R_i$  in the input data then  $R_i$  should precede  $R_j$  in the sorted output data also if the sort is stable. If the sort is not stable then  $R_i$  and  $R_j$  may be in any order in the sorted output. So in an unstable sort the duplicate keys may occur in any order in the sorted output.

## Sort efficiency

Sorting is an important and frequent operation in many applications. So the aim is not only to get sorted data but to get it in the most efficient manner. Therefore many algorithms have been developed for sorting and to decide which one to use when we need to compare them using some parameters.

**Choice is made using these three parameters:**

1. **Coding time:** Coding time is the time taken to write the program for implementing a particular sorting algorithm. Coding time depends upon the algorithm you are choosing for sorting.

***For example,*** simple sorting programs like bubble sort will require less coding time whereas heap sorting will consume more Coding time.

2. **Space requirement:** It is the space required to store the executable program, constants, variables etc.

***For example,*** below program demonstrates how to find the space requirement of a sorting program. It uses size command to display the space required for text, data etc.

```
/*program to find space requirement */
#include <stdio.h>
#include <time.h>
int main(int argc, char *argv[])
{
    time_t start, stop;
    clock_t ticks; long count;
    time(&start);
    int array[4]={3, 67, 2, 64}, c, d, temp;
    for( c=0; c<(4-1); c++)
    {
        for(d=0; d<4-c-1; d++)
        {
            if(array[d]>array[d+1])
            {
                temp= array[d];
```



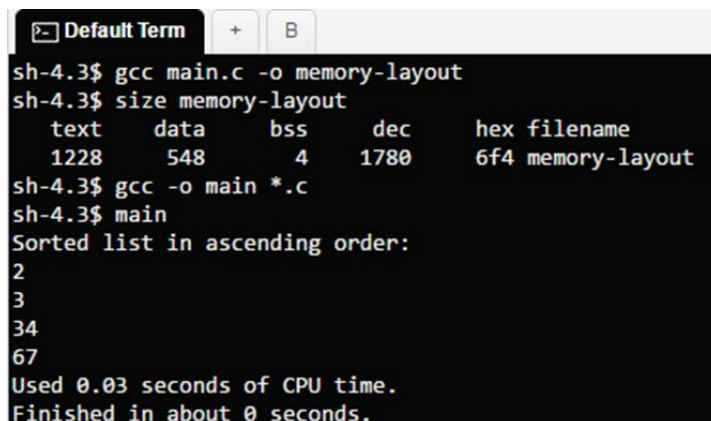
```

        array[d]=array[d+1];
        array[d+1]=temp;
    }
}
}
printf("Sorted list is ascending order:\n");
for(c=0;c<4;c++)
    printf("%d\n", array[c]);
int i=0;

while(i<50000)
{
    i++;
    ticks = clock();
}
time(&stop);
printf("Used %.2f seconds of CPU time. \n",
(double)ticks/CLOCKS_PER_SEC);
printf("Finished in about %.0f seconds. \n", difftime(stop, start));
return 0;
}

```

### Output:



```

sh-4.3$ gcc main.c -o memory-layout
sh-4.3$ size memory-layout
   text    data     bss     dec     hex filename
   1228     548        4    1780     6f4 memory-layout
sh-4.3$ gcc -o main *.c
sh-4.3$ main
Sorted list in ascending order:
2
3
34
67
Used 0.03 seconds of CPU time.
Finished in about 0 seconds.

```

3. **Run time or execution time:** It is the time taken to successfully execute a sorting algorithm to obtain a sorted list of elements.

*For example,* the below program demonstrates how to calculate total execution time of a sorting program. It uses header file <time.h> and calculates the execution time of a sorting program.

```

/*program to find execution time */
#include <stdio.h>
#include <time.h>
int main(int argc, char *argv[])
{

```

---

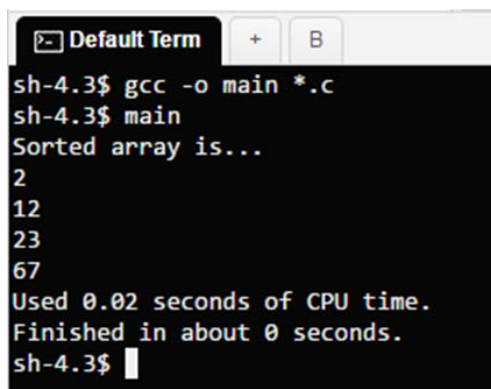
```

time_t start, stop;
clock_t ticks; long count;
time(&start);
int array[4]={2, 67, 12, 23}, i, j, temp;
for( i=0; i<4; i++)
{
    for(j=0; j<4-i-1; j++)
    {
        if(array[j]>array[j+1])
        {
            temp= array[j];
            array[j]=array[j+1];
            array[j+1]=temp;
        }
    }
}
printf("Sorted array is:\n");
for(i=0;i<4;i++)
    printf("%d\n", array[i]);
int k=0;

while(k<50000)
{
    k++;
    ticks = clock();
}
time(&stop);
printf("Used %0.2f seconds of CPU time. \n",
(double)ticks/CLOCKS_PER_SEC);
printf("Finished in about %.0f seconds. \n", difftime(stop, start));
return 0;
}

```

### Output:



```

sh-4.3$ gcc -o main *.c
sh-4.3$ main
Sorted array is...
2
12
23
67
Used 0.02 seconds of CPU time.
Finished in about 0 seconds.
sh-4.3$

```

If data is in small quantity and sorting is needed only during a few occasions, then any simple sorting technique can be used. This is because in these cases, a simple or a less efficient

---

---

technique would behave at par with the complex techniques which are developed to minimize run time and space requirements. So it is pointless to search and apply complex algorithm.

Running time can be defined as the total time taken by the sorting program to run to completion. Hence, running time is one of the most important factors in implementation of algorithms. If the amount of data to be sorted is in large quantity, then it is crucial to minimize runtime by choosing an efficient runtime technique.

The 2 basic operations in sorting are comparison and moving records. The record moves or any other operations are generally a constant factor of number of comparisons. Moreover the record moves can be considerably reduced so that run time is measured by considering only the comparisons. Calculating the exact number of comparisons may not be always possible so an approximation is given by big=O notation. Thus the run time efficiency of each algorithm is expressed as O notation. The efficiency of most of the sorting algorithm is between  $O(n \log n)$  and  $O(n^2)$ .



## Self-assessment Questions

- 1) The technique used for arranging data elements in a specific order is called as \_\_\_\_\_.
  - a) Arranging
  - b) Filtering
  - c) Sorting
  - d) Distributing
- 2) The Time required to Complete the execution of a sorting program is called as \_\_\_\_\_.
  - a) Coding Time
  - b) Average Time
  - c) Running Time
  - d) Total Time
- 3) A sorting technique is called stable if it \_\_\_\_\_.
  - a) Takes  $O(n \log n)$  times
  - b) Maintains the relative order of occurrence of non-distinct elements
  - c) Uses divide-and-conquer paradigm
  - d) Takes  $O(n)$  space

---

## 2.2.3 Sorting Techniques

Sorting techniques depends on two important parameters. The first parameter is the execution time of program, which means time taken for execution of program. The second parameter is the space, which means space or memory taken by the program. The algorithm that you choose must be more efficient in terms of execution time and space usage.

There are many techniques for sorting. *For example*, Bubble sort, Selection sort, merge sort etc. The choice of sorting algorithm depends upon the particular situation.

### In-place sorting and Not-in-place sorting

Sorting algorithms may require some extra space for comparison of elements and temporary storage of few data elements.

The sorting algorithms which does not require any extra space for sorting, and usually the sorting happens within array is called as in-place sorting. This is called in-place sorting. Bubble sort is an example of **in-place sorting**. Many other in-place sorting algorithms include selection sort, insertion sort, heap sort, and Shell sort.

But in some sorting algorithms, the program requires space which is more than or equal to the elements being sorted. Sorting which uses equal or more space for temporary storage is called not-in-place sorting. They sometimes require arrays to be separated and sorted. Merge-sort is an example of **not-in-place sorting**.

To understand the more complex and efficient first sorting algorithms, it is important to understand the simpler, but slower algorithms. This topic deals with bubble sort, insertion sort; and selection sort, merge sort and quick sort. Any of these sorting algorithms are good enough for most small tasks.

### (i) The Bubble Sort

Bubble Sort is an algorithm which is used to sort  $N$  elements that are given in a memory. *For example*, an Array with  $N$  number of elements. Bubble Sort compares the entire element one by one and sort them based on their values.

The bubble sort makes multiple passes through a list. It compares adjacent items and exchanges those that are out of order. Each pass through the list places the next largest value in its proper place. In essence, each item “bubbles” up to the location where it belongs.

Sorting takes place by stepping through all the data items one-by-one in pairs and comparing adjacent data items and swapping each pair that is out of order.

Fig 2.2.2 shows the first pass of a bubble sort. The shaded items are being compared to see if they are out of order. If there are  $n$  items in the list, then there are  $n-1$  pairs of items that need to be compared on the first pass. It is important to note that once the largest value in the list is part of a pair, it will continually be moved along until the pass is complete.

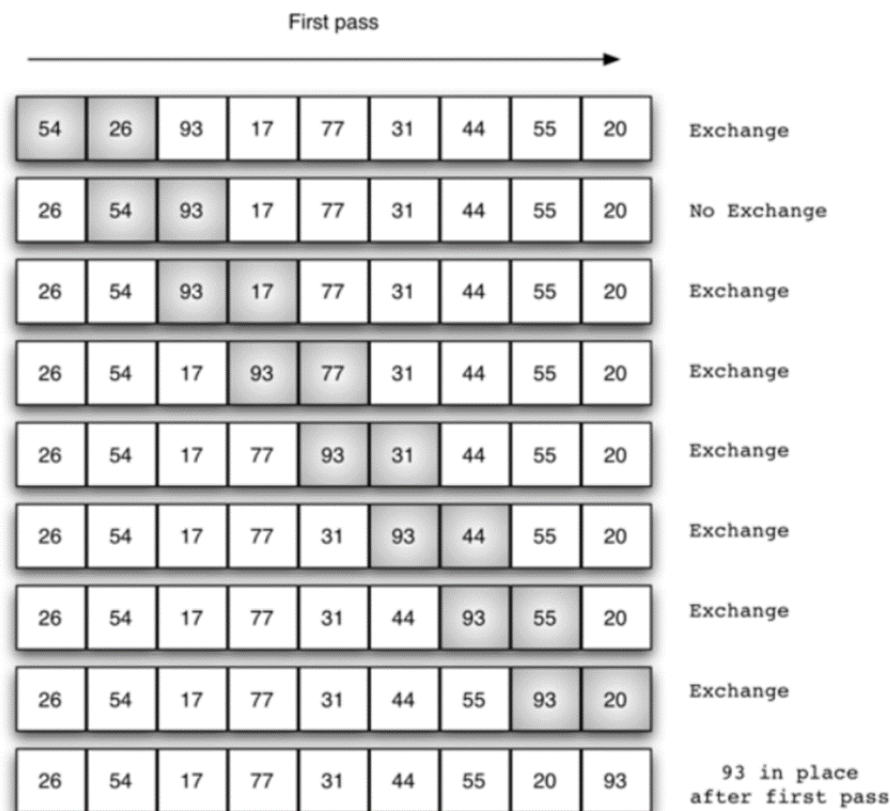


Figure 2.2.2: First pass of Bubble sort

At the start of the second pass as shown in the below Figure 2.2.3, the largest value is now in place. There are  $n-1$  items left to sort, meaning that there will be  $n-2$  pairs. Since each pass places the next largest value in place, the total number of passes necessary will be  $n-1$ . After completing the  $n-1$  passes, the smallest item must be in the correct position with no further processing required.

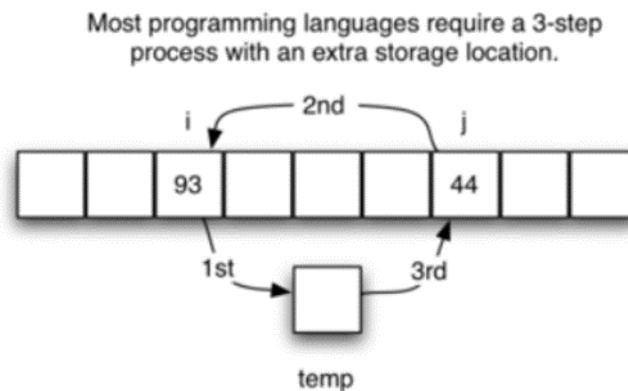
The exchange operation is sometimes called a “swap”. Typically, swapping two elements in a list requires a temporary storage location (an additional memory location).

---

**A code fragment such as:**

```
temp = alist[i]
alist[i] = alist[j]
alist[j] = temp
```

will exchange the  $i^{\text{th}}$  and  $j^{\text{th}}$  items in the list. Without the temporary storage, one of the values would be overwritten.



*Figure 2.2.3: Second pass of Bubble sort*

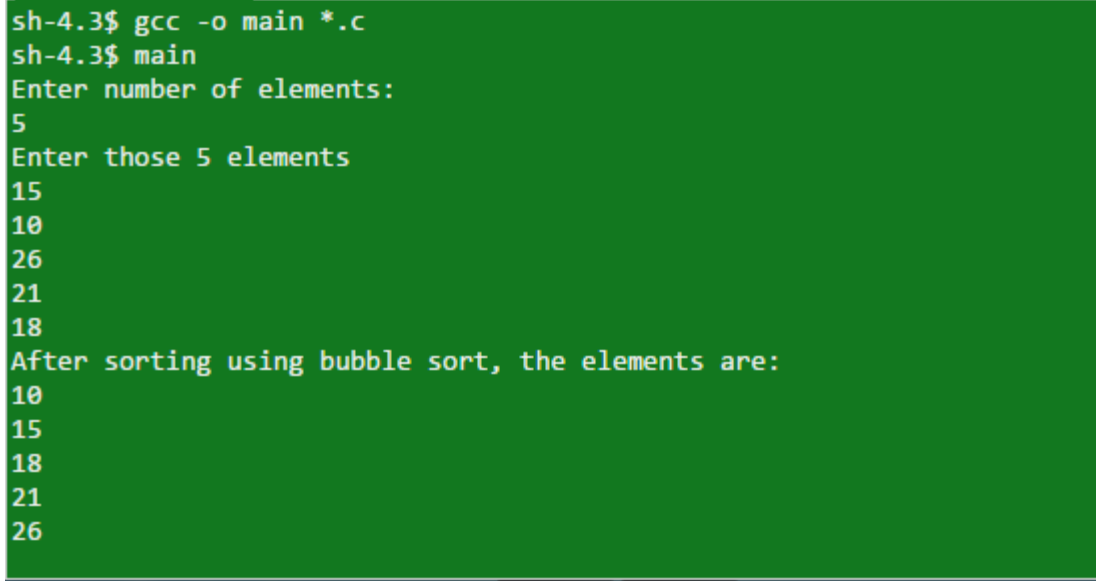
Below is the code to implement Bubble sort.

```
/* Implementation of Bubble sort Algorithm */
#include <stdio.h>
int main()
{
    int arr[300], n, i, j, swap;
    printf("Enter number of elements:\n");
    scanf("%d",&n);
    printf("Enter those %d elements\n", n);
    for(i =0; i < n; i++)
        scanf("%d",&arr[i]);
    for(i =0; i < ( n -1);i++)
    {
        for(j =0; j < n - i -1; j++)
        {
            if(arr[j]> arr[j+1])
            {
                swap= arr[j];
                arr[j]= arr[j+1];
                arr[j+1]= swap;
            }
        }
    }
}
```

---

```
    }
    printf("After sorting using bubble sort, the elements are:\n");
    for( i =0; i < n ;i++)
        printf("%d\n", arr[i]);
    return 0;
}
```

### Output:



```
sh-4.3$ gcc -o main *.c
sh-4.3$ main
Enter number of elements:
5
Enter those 5 elements
15
10
26
21
18
After sorting using bubble sort, the elements are:
10
15
18
21
26
```

## Complexity Analysis of Bubble Sorting

### Worst case Time Complexity:

Bubble sort algorithm will sort the array on  $n$  elements as given below.

1<sup>st</sup> Pass:  $n-1$  Comparisons and  $n-1$  swaps

2<sup>nd</sup> Pass:  $n-2$  Comparisons and  $n-2$  swaps

....

$(n-1)^{\text{th}}$  Pass: 1 comparison and 1 swap.

All together:  $c ((n-1) + (n-2) + \dots + 1)$ , where  $c$  is the time required to do one comparison, one swap.

*i.e.*,  $(n-1)+(n-2)+(n-3)+\dots+3+2+1$

Sum of the above series =  $n(n-1)/2$

---

$$\text{Sum} = O(n^2)$$

Hence the worst time complexity of Bubble Sort is  $O(n^2)$ .

### **Space complexity:**

Bubble Sort has Space complexity of  $O(1)$ , because only one additional memory space is required for **temp** variable.

### **Best-case Time Complexity:**

Best Case Time Complexity is  $O(n)$ , when the given list of elements is already sorted.

A bubble sort is considered as the most inefficient sorting method. Bubble sort algorithm exchanges elements before the final location of element is known, thus utilizing more time in these exchange operations. However, as the bubble sort algorithm makes passes through the entire unsorted portion of the list of elements.

## **(ii) Insertion Sort**

Consider a contiguous list. In this case, it is necessary to move entries in the list to make room for the insertion. To find the position where the insertion is to be made, we must search. One method for performing ordered insertion into a contiguous list is first to do a binary search to find the correct location, and then move the entries as required and insert the new entry.

Since so much time is needed to move entries no matter how the search is done, it turns out in many cases to be just as fast to use sequential search as binary search. By doing sequential search from the end of the list, the search and the movement of entries can be combined in a single loop, thereby reducing the overhead required in the function.

**Following are some of the important characteristics of Insertion Sort.**

- It has one of the simplest implementation
- It is efficient for smaller data sets, but very inefficient for larger lists.
- Insertion Sort is adaptive, that means it reduces its total number of steps if given a partially sorted list, hence it increases its efficiency.
- It is better than Selection Sort and Bubble Sort algorithms.



- Its space complexity is less, like Bubble Sorting. Insertion sort also requires a single additional memory space.
- It is Stable, as it does not change the relative order of elements with equal keys.

The procedure for insertion sort of elements when elements are equal is demonstrated in the figure 2.2.4

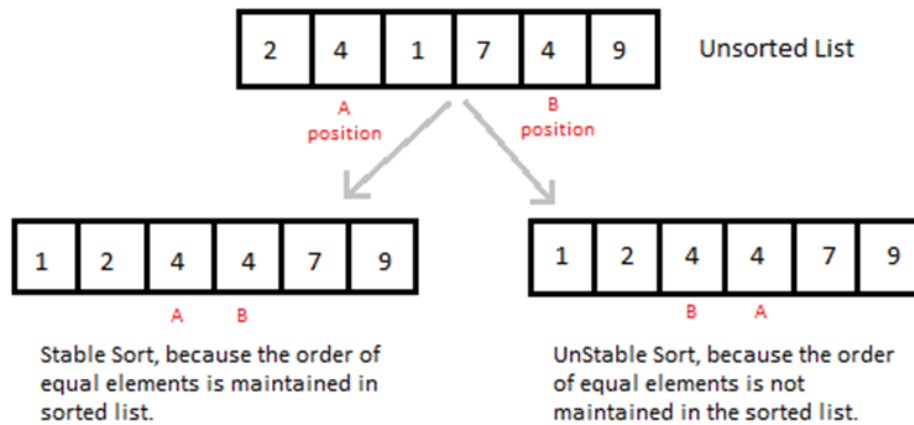


Figure 2.2.4: Insertion sort with equal elements

The working of insertion sort algorithm with an example is depicted in figure 2.2.5

### How Insertion Sorting Works

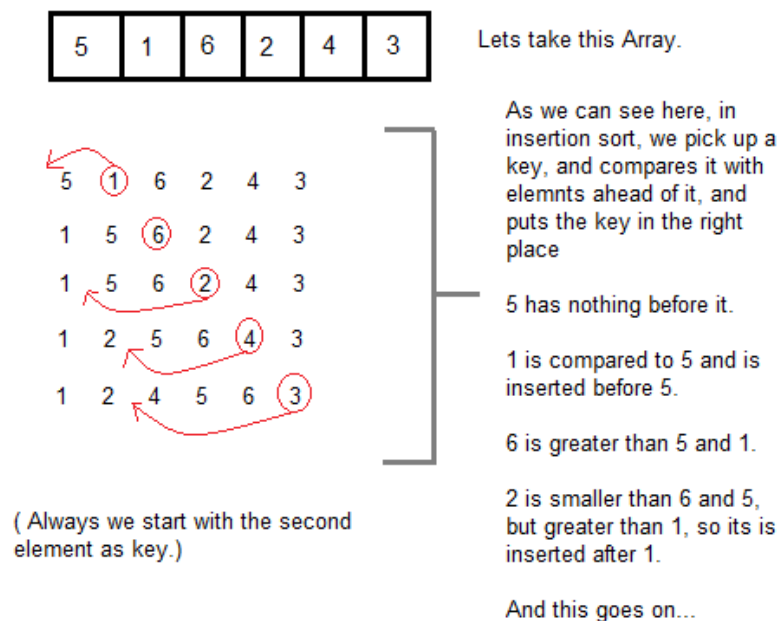


Figure 2.2.5: Working of Insertion Sort Algorithm

---

## Pseudocode:

```
void insertionSort(int arr[], int length) {
    int i, j, tmp;
    for (i = 1; i < length; i++) {
        j = i;
        while (j > 0 && arr[j - 1] > arr[j]) {
            tmp = arr[j];
            arr[j] = arr[j - 1];
            arr[j - 1] = tmp;
            j--;
        }
    }
}
```

## Program:

```
/* Implementation of Insertion sort Algorithm */
#include <stdio.h>
int main()
{
    int arr[500], n, i, j, temp;
    printf("Enter the total number of elements:");
    scanf("%d", &n);
    printf("Enter those %d Elements : \n", n);
    for(i=0; i<n; i++)
    {
        scanf("%d",&arr[i]);
    }
    for(i=1; i<n; i++)
    {
        temp=arr[i];
        j=i-1;
        while((temp<arr[j]) && (j>=0))
        {
            arr[j+1]=arr[j];
            j--;
        }
        arr[j+1]=temp;
    }
    printf("After sorting using insertion sort, the elements are: \n");
    for(i=0; i<n; i++)
    {
        printf("%d\n",arr[i]);
    }
    return 0;
}
```

---

---

## Output:

```
sh-4.3$ gcc -o main *.c
sh-4.3$ main
Enter the total number of elements:5

Enter those 5 Elements: 24
16
20
42
36

After sorting using insertion sort, the elements are:
16
20
24
36
42
```

## Complexity Analysis of Insertion Sort

The analysis is is same as bubble sorting.

Worst Case Time Complexity:  $O(n^2)$

Best Case Time Complexity:  $O(n)$

Average Time Complexity:  $O(n^2)$

Space Complexity:  $O(1)$

### (iii) Selection Sort

Insertion sort has one major disadvantage. Even after most entries have been sorted properly into the first part of the list, the insertion of a later entry may require that many of them be moved. All the moves made by insertion sort are moves of only one position at a time. Thus to move an entry 20 positions up the list requires 20 separate moves. If the entries are small, perhaps a key alone, or if the entries are in linked storage, then the many moves may not require excessive time. In case the entries are very large, records containing hundreds of components like personnel files or student transcripts, and the records must be kept in contiguous storage. In such cases it would be far more efficient if, when it is necessary to move an entry, it could be moved immediately to its final position. Selection sort method accomplishes this goal.

---

## How Selection Sorting Works

Consider an array of  $n$  elements. Selection sort algorithm starts by comparing first two elements of that array and swap the elements if required

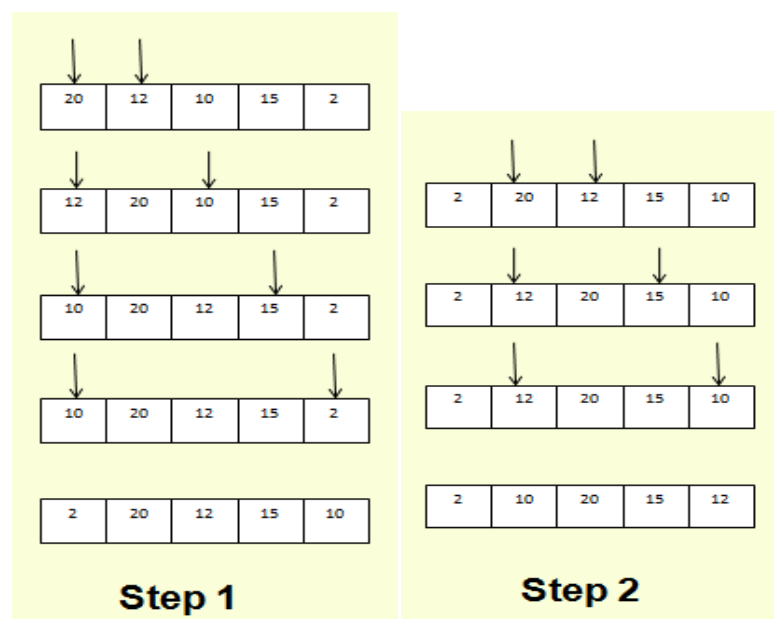
*For example*, if you want to sort the elements in ascending order and if the first element is greater than second element, then it will swap the elements but, if the first element is smaller than second element, it will leave the elements as it is.

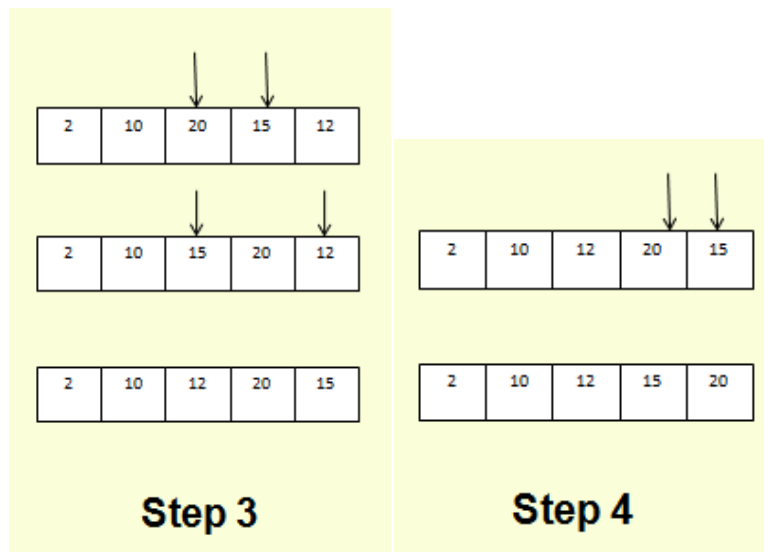
Then, again first element and third element are compared and swapped if required. This process will continue until first and last element of that array is compared, thus completing the first pass of selection sort.

In this algorithm, after the first pass the required element will be already placed at its final position. Hence during the second pass through this algorithm, it starts from second element of array and repeats the procedure  $n-1$  times.

For sorting in ascending order, smallest element will be at first and in case of sorting in descending order; largest element will be at first. Similarly, this process will continue until all elements in array are sorted.

Below figure 2.2.6 demonstrates how selection sort algorithm works:





*Figure 2.2.6: Working of selection sort with example*

In the first pass, 2 is found to be the smallest. Hence, it is placed in the first position. In the second pass 10 is found to be the smallest and placed at 2nd position and so on until the full list is sorted.

### Sorting using Selection Sort Algorithm

```
void selectionSort(int a[], int size)
{
    int i, j, min, temp;
    for(i=0; i< size-1; i++)
    {
        min = i;    //setting min as i
        for(j=i+1; j < size; j++)
        {
            if(a[j] < a[min])    //if element at j is less than element at min position
            {
                min = j;    //then set min as j
            }
        }
        temp = a[i];
        a[i] = a[min];
        a[min] = temp;
    }
}
```

---

## Selection sort algorithm implementation in c

```
/* Implementation of Selection sort Algorithm */
#include<stdio.h>
int main()
{
    int arr[200],i,j,n,t,min,pos;
    printf("Enter the total number of elements:");
    scanf("%d",&n);
    printf("Enter those %d elements:\n", n);
    for(i=0; i<n; i++)
        scanf("%d",&arr[i]);
    for(i=0; i<n-1; i++)
    {
        min=arr[i];
        pos=i;
        for(j=i+1; j<n; j++)
        {
            if(min>arr[j])        //Compare values
            {
                min=arr[j];
                pos=j;
            }
        }
        t=arr[i];        //Swap the values
        arr[i]=arr[pos];
        arr[pos]=t;
    }
    printf("\n After sorting using selection sort, the elements are::\n");
    for(i=0; i<n; i++)
        printf("%d \n",arr[i]);
    return 0;
}
```

### Output:

```
sh-4.3$ gcc -o main *.c
sh-4.3$ main
Enter the total number of elements:
5
Enter those 5 elements:
17
10
23
36
19

After sorting using selection sort, the elements are::
10
17
19
23
36
```

---

## Complexity Analysis of Selection Sorting

Worst Case Time Complexity:  $O(n^2)$

Best Case Time Complexity:  $O(n^2)$

Average Time Complexity:  $O(n^2)$

Space Complexity:  $O(1)$



### Did you Know?

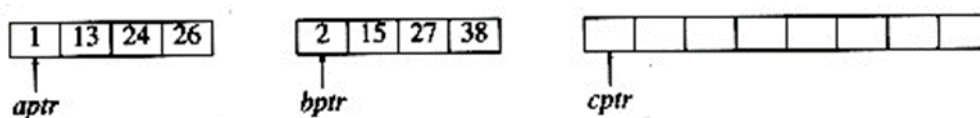
The worst case time complexity of bubble sort, selection sort and insertion sort is  $n^2$ .

## (iv) Merge Sort

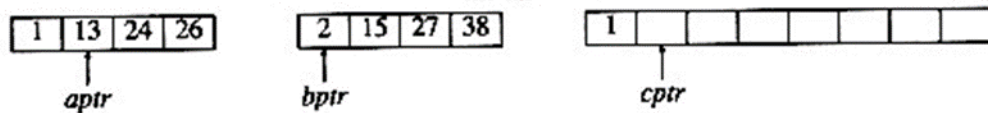
Merge sort is a fine example of a recursive algorithm.

The fundamental operation in this algorithm is merging two sorted lists. This can be done in one pass through the input, if the output is put in a third list because the lists are sorted. Merge sort is a sorting technique based on divide and conquer technique. Merge sort first divides the array into equal halves and then combines them in a sorted manner.

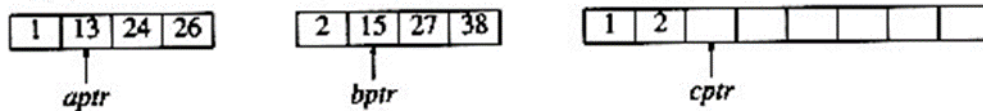
The basic merging algorithm takes two input arrays A and B, an output array C, and three counters, *aptr*, *bptr*, and *cptr*, which are initially set to the beginning of their respective arrays. The smaller of  $A[aptr]$  and  $B[bptr]$  is copied to the next entry in C, and the appropriate counters are advanced. When either input list is exhausted, the remainder of the other list is copied to C. An example of how the merge routine works is provided for the following input.



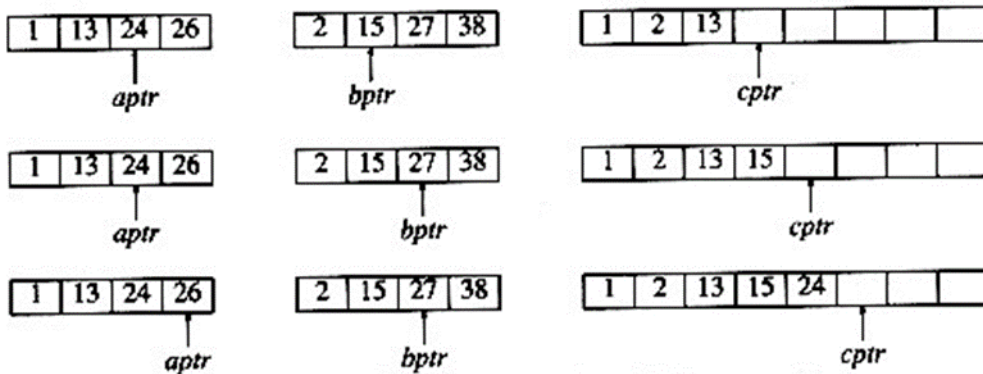
If the array A contains 1, 13, 24, 26, and b contains 2, 15, 27, 38, then the algorithm proceeds as follows: First, a comparison is done between 1 and 2. 1 is added to C, and then 13 and 2 are compared.



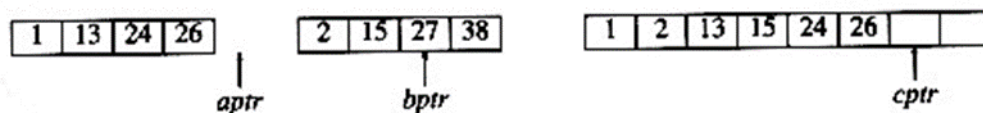
2 is added to C, and then 13 and 15 are compared.



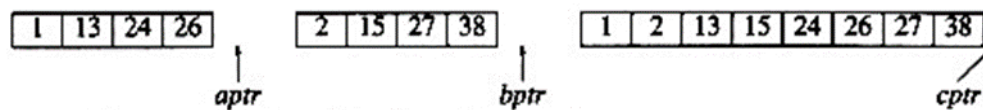
13 is added to C, and then 24 and 15 are compared. This proceeds until 26 and 27 are compared.



26 is added to C, and the A array is exhausted.



The remainder of the B array is then copied to C.



The time to merge two sorted lists is clearly linear, because at most  $n - 1$  comparisons are made, where  $n$  is the total number of elements. Note that every comparison adds an element to  $c$ , except the last comparison, which adds at least two.

The merge sort algorithm is therefore easy to describe. If  $n = 1$ , there is only one element to sort, and the answer is at hand. Otherwise, recursively merge sort the first half and the second



---

half. This gives two sorted halves, which can then be merged together using the merging algorithm described above. For instance, to sort the eight-element array 24, 13, 26, 1, 2, 27, 38, 15, recursively sort the first four and last four elements, obtaining 1, 13, 24, 26, 2, 15, 27, 38. Then merge the two halves as above, obtaining the final list 1, 2, 13, 15, 24, 26, 27, 38. This algorithm is a classic divide-and-conquer strategy. The problem is divided into smaller problems and solved recursively. The conquering phase consists of patching together the answers. Divide-and-conquer is a very powerful use of recursion that will be seen many times.

## Algorithm

Merge sort keeps on dividing the list into equal halves until it can no more be divided. By definition, if it is only one element in the list, it is sorted. Then merge sort combines smaller sorted lists keeping the new list sorted too.

**Step 1** – if it is only one element in the list, it is already sorted, return.

**Step 2** – divide the list recursively into two halves until it can no more be divided.

**Step 3** – merge the smaller lists into new list in sorted order.

## Pseudocode

We shall now see the pseudocodes for merge-sort functions. As our algorithms points out two main functions – divide & merge. Merge sort works with recursion and we shall see our implementation in the same way.

```
procedure mergesort(var a as array )
  if( n ==1) return a

  var l1 as array = a[0]... a[n/2]
  var l2 as array = a[n/2+1]... a[n]

  l1 =mergesort( l1 )
  l2 =mergesort( l2 )

  return merge( l1, l2 )
end procedure

procedure merge(var a asarray,var b as array )

  var c as array

  while( a and b have elements )
    if( a[0]> b[0])
```

---

```
add b[0] to the end of c
remove b[0]from b
else
add a[0] to the end of c
remove a[0]from a
endif
endwhile
```

```
while( a has elements )
add a[0] to the end of c
remove a[0]from a
endwhile
```

```
while( b has elements )
add b[0] to the end of c
remove b[0]from b
endwhile
```

```
return c
```

```
end procedure
```

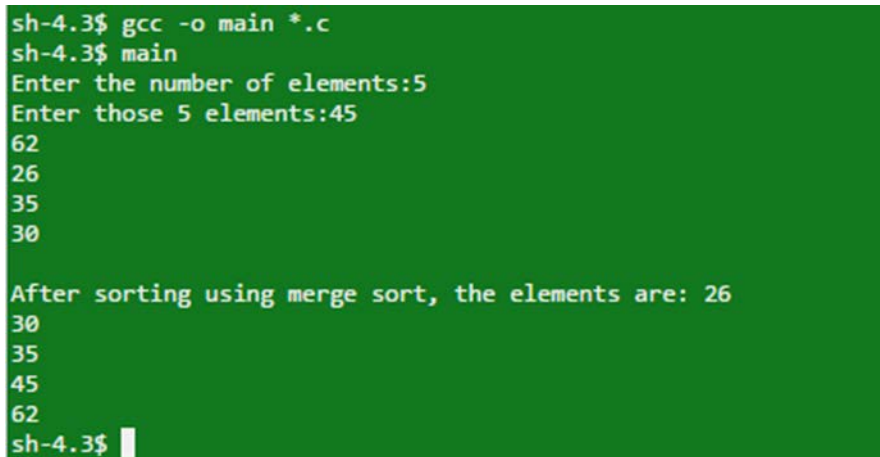
```
/* Implementation of Merge sort Algorithm */
#include<stdio.h>
int arr[20],i,n,b[20];
void merge(int arr[],int low,int m ,int high)
{
    int h,i,j,k;
    h=low;
    i=low;
    j=m+1;
    while(h<=m && j<=high)
    {
        if(arr[h]<=arr[j])
            b[i]=arr[h++];
        else
            b[i]=arr[j++];
        i++;
    }
    if( h > m)
        for(k=j;k<=high;k++)
            b[i++]=arr[k];
    else
        for(k=h;k<=m;k++)
            b[i++]=arr[k];
    for(k=low;k<=high;k++)
    {
        arr[k]=b[k];
    }
}
```

---

```
void mergesort(int arr[],int i,int j)
{
    int m;
    if(i<j)
    {
        m=(i+j)/2;
        mergesort(arr,i,m);
        mergesort(arr,m+1,j);
        merge(arr,i,m,j);
    }
}
int main()
{
    printf("\nEnter the number of elements:");
    scanf("%d",&n);
    printf("Enter those %d elements:", n);
    for(i=0; i<n; i++)
        scanf("%d",&arr[i]);
    mergesort(arr,0,n-1);
    printf("\nAfter sorting using merge sort, the elements are: ");
    for(i=0;i<n;i++)
        printf("%d\n", arr[i]);
    return 0;
}
```

## Output:

The output of the program should be as follows:



```
sh-4.3$ gcc -o main *.c
sh-4.3$ main
Enter the number of elements:5
Enter those 5 elements:45
62
26
35
30

After sorting using merge sort, the elements are: 26
30
35
45
62
sh-4.3$
```

## Analysis of Merge Sort

Merge sort algorithm uses a divide and conquer strategy. This is a recursive algorithm that continuously divides the list of elements into 2 parts.

---

**Case 1:** If the List is empty or if it has only single item, then the list is already sorted. This will be considered as the Best Case.

**Case 2:** If the List contains N number of elements, the algorithm will divide the list into half and perform merge sorting individually on both halves. Once both parts are sorted, a merge operation is performed to combine the already sorted smaller parts.

### **Worst Case Time Complexity:**

In worst case, in every step a comparison is required. This is because in every merge step, one value will remain in the opposing list. Hence, merge sort algorithm must continue comparing the elements in the opposing lists.

### **The complexity of worst-case Merge Sort is:**

$$T(N) = 2T(N/2) + N-1 \quad \text{Equation 1}$$

Where  $T(N)$  is the total number of comparisons between the elements in a list and  $N$  refers to the total number of elements in a list.

$2T(N/2)$  shows that merge sort is performed on two halves of the list during the divide stage and

$N-1$  represents the total comparisons in the merge stage.

This merge sort procedure is recursive. Hence it will include substitutions also.

$$T(N) = 2[2T(N/4) + N/2-1] + N-1 \quad \text{Equation 2}$$

$$T(N) = 4[2T(N/8) + N/4-1] + 2N-1 \quad \text{Equation 3}$$

These equations represent those substitutions required during recursions,

So we have substituted  $T(N)$  value into  $2T(N/2)$  of equation 1 to obtain equation 4

$$T(N) = 8T(N/8) + N + N + N - 4 - 2 - 1 \quad \text{Equation 4}$$

This is during the 3rd recursive call.

Let us consider a value  $k$  representing the depth of recursion.

Recursion stops when the list will contain only one element. In general we get,

$$T(N) = 2^k T(N/2^k) + kN - (2^k - 1) \quad \text{Equation 5}$$

This procedure of dividing will continue until list contains a single element. And we know that a list with a single element is already sorted.

---

$T(1)=0$	Equation 6
----------	------------

$2^k = N$	Equation 7
-----------	------------

$k=\log_2 N$	Equation 8
--------------	------------

$T(N) = N \log_2 N - N + 1$	Equation 9
-----------------------------	------------

Hence, the worst time complexity of Merge sort algorithm is  $O(N \log(N))$

### **Best Case Time Complexity:**

It is when the largest element of one sorted part is smaller than the first element of its opposing part, for every merge step that occurs. Only one element from the opposing list is compared thus reducing the number of comparisons in each merge step to  $N/2$ .

Hence best case time complexity is also  $O(N \log(N))$  because the merging is always linear. Same follows for the Average case time complexity.

## **(v) Quick Sort**

Even though the time complexity of merge sort algorithm is  $O(n \log n)$ , it is not desirable to use as it consumes more space. It needs more space to merge the array partitions. Quick sort is one of the fastest sorting algorithms.

The quick sort algorithm also uses divide and conquer rule to sort the elements without using additional storage.

A quick sort algorithm first selects a value, which is called the pivot value. This algorithm will partition all elements based on whether they are smaller than or greater than the pivot element.

**Thus we get two partitions:** One partition having elements larger than the pivot element and another partition having elements smaller than the pivot element. The selected pivot element ends up in its final sorted position.

Thus, the elements to the right and left of the pivot element can be sorted successfully. Hence, we can again implement a recursive algorithm to sort the elements using divide and conquer approach. All the partitioned array elements remain in the same array hence saving the space where they can be combined together.

---

For sorting the elements we have to use a recursive function. We have to pass both the partitions of array along with the pivot element to this function as parameters.

Our prior sorting functions, however, have no parameters, so for consistency of notation we do the recursion in a function `recursive_quick_sort` that is invoked by the method `quick_sort`, which has no parameters.

Quick Sort, as the name suggests, sorts any list very quickly. Quick sort is not stable search, but it is very fast and requires very less additional space. It is based on the rule of Divide and Conquer (also called partition-exchange sort).

**This algorithm divides the list into three main parts:**

1. Elements less than the Pivot element
2. Pivot element
3. Elements greater than the pivot element

In the list of elements, mentioned in below example, we have taken 25 as pivot. So after the first pass, the list will be changed like this.

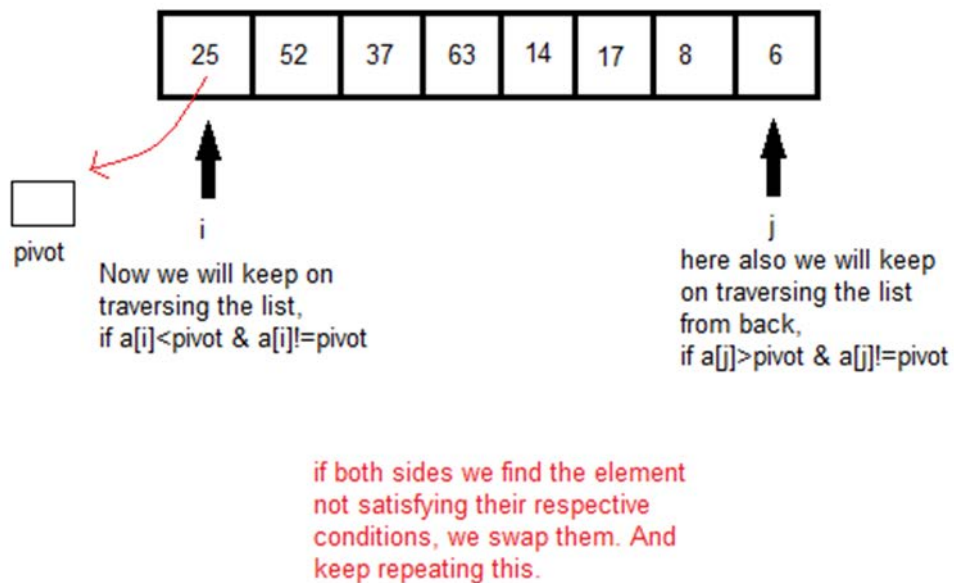
6 8 17 14 25 63 37 52

Hence, after the first pass, pivot will be set at its position, with all the elements smaller to it on its left and all the elements larger than it on the right. Now 6 8 17 14 and 63 37 52 are considered as two separate lists, and same logic is applied on them, and we keep doing this until the complete list is sorted.

The working of Quick sort algorithm is shown in figure 2.2.7.

---

## How Quick Sorting Works



### DIVIDE AND CONQUER - QUICK SORT

Figure 2.2.7: Divide and Conquer-Quick Sort

## QuickSort Pivot Algorithm

Based on our understanding of partitioning in quicksort, we should now try to write an algorithm for it here.

- Step 1** - Choose the highest index value has pivot
- Step 2** - Take two variables to point left and right of the list excluding pivot
- Step 3** - left points to the low index
- Step 4** - right points to the high
- Step 5** - while value at left is less than pivot move right
- Step 6** - while value at right is greater than pivot move left
- Step 7** - if both step 5 and step 6 does not match swap left and right
- Step 8** - if  $\text{left} \geq \text{right}$ , the point where they met is new pivot

---

## QuickSort Pivot Pseudocode

The pseudocode for the above algorithm can be derived as –

```
function partitionFunc(left, right, pivot)
    leftPointer = left - 1
    rightPointer = right

    while true do
        while A[++leftPointer] < pivot do
            //do-nothing
        endwhile

        while rightPointer > 0 && A[--rightPointer] > pivot do
            //do-nothing
        endwhile

        if leftPointer >= rightPointer
            break
        else
            swap leftPointer, rightPointer
        endif
    endwhile

    swap leftPointer, right
    return leftPointer
endfunction
```

## QuickSort Algorithm

Using pivot algorithm recursively we end-up with smaller possible partitions. Each partition then processed for quick sort. We define recursive algorithm for quicksort as below –

- Step 1** – Make the right-most index value pivot
- Step 2** – partition the array using pivot value
- Step 3** – quicksort left partition recursively
- Step 4** – quicksort right partition recursively



---

## QuickSortPseudocode

To get more into it, let see the pseudocode for quick sort algorithm –

```
procedure quickSort(left, right)

    if right-left <=0
        return
    else
        pivot= A[right]
        partition=partitionFunc(left, right, pivot)
        quickSort(left,partition-1)
        quickSort(partition+1,right)
    endif

end procedure
```

## Sorting using Quick Sort Algorithm

/\* a[] is the array, p is starting index, that is 0,  
and r is the last index of array. \*/

```
void quicksort(int a[], int p, int r)
{
    if(p < r)
    {
        int q;
        q = partition(a, p, r);
        quicksort(a, p, q);
        quicksort(a, q+1, r);
    }
}

int partition(int a[], int p, int r)
{
    int i, j, pivot, temp;
    pivot = a[p];
    i = p;
    j = r;
    while(1)
    {
        while(a[i] < pivot && a[i] != pivot)
            i++;
        while(a[j] > pivot && a[j] != pivot)
            j--;
```

---

```

if(i< j)
{
temp = a[i];
a[i] = a[j];
a[j] = temp;
}
else
{
return j;
}
}
}

```

```

/* Implementation of Quick sort Algorithm */
#include<stdio.h>
#include<stdbool.h>
#define MAX 7
int intArray[MAX]={ 14,6,23,12,76,49,57};

void printline(int count)
{
    int i;
    for(i=0;i <count-1;i++)
    {
        printf("=");
    }
    printf("=\n");
}

void display()
{
    int i;
    printf("[");

    // navigate through all items
    for(i=0;i<MAX;i++)
    {
        printf("%d ",intArray[i]);
    }
    printf("]\n");
}

void swap(int num1,int num2)
{
    int temp =intArray[num1];
    intArray[num1]=intArray[num2];
    intArray[num2]= temp;
}

int partition(int left,int right,int pivot)
{

```

---

---

```

    int leftPointer= left -1;
    int rightPointer= right;

while(true)
{

    while(intArray[++leftPointer]< pivot)
    {
        //do nothing
    }

    while(rightPointer>0&&intArray[--rightPointer]> pivot)
    {
        //do nothing
    }

    if(leftPointer>=rightPointer)
    {
        break;
    }
    else
    {
        printf(" item swapped :%d,%d\n",
            intArray[leftPointer],intArray[rightPointer]);
        swap(leftPointer,rightPointer);
    }

}

printf(" pivot swapped :%d,%d\n",intArray[leftPointer],intArray[right]);
swap(leftPointer,right);
printf("Updated Array: ");
display();
return leftPointer;
}

void quickSort(int left,int right)
{
    if(right-left <=0)
    {
        return;
    }
    else
    {
        int pivot =intArray[right];
        int partitionPoint= partition(left, right, pivot);
        quickSort(left,partitionPoint-1);
        quickSort(partitionPoint+1,right);
    }
}

int main()

```

---

---

```
{
    printf("\nBefore Sorting: ");
    display();
    printline(50);
    quickSort(0,MAX-1);
    printf("\nAfter sorting using quick sort, the elements are: ");
    display();
    printline(50);
    return 0;
}
```

### Output:

```
sh-4.3$ gcc -o main *.c
sh-4.3$ main

Before Sorting: [14 6 23 12 76 49 57 ]
=====
item swapped :76,49
pivot swapped :76,57
Updated Array: [14 6 23 12 49 57 76 ]
pivot swapped :49,49
Updated Array: [14 6 23 12 49 57 76 ]
item swapped :14,6
pivot swapped :14,12
Updated Array: [6 12 23 14 49 57 76 ]
pivot swapped :23,14
Updated Array: [6 12 14 23 49 57 76 ]

After sorting using quick sort, the elements are: [6 12 14 23 49 57 76 ]
```

### Complexity Analysis of Quick Sort

Worst Case Time Complexity:  $O(n^2)$

Best Case Time Complexity:  $O(n \log n)$

Average Time Complexity:  $O(n \log n)$

Space Complexity:  $O(n \log n)$

- Space required by quick sort is very less, only  $O(n \log n)$  additional space is required.
- Quick sort is not a stable sorting technique, so it might change the occurrence of two similar elements in the list while sorting.

### Analysis of Quick sort

To analyse the running time of Quick Sort, we use the same approach as we did for Merge Sort (and is common for many recursive algorithms, unless they are completely obvious).

---

Let  $T(n)$  represent the worst-case running time of the Quick Sort algorithm on an array of size  $n$ . To get a hold of  $T(n)$ , we look at the algorithm line by line. The call to partition takes time  $\Theta(n)$ , because it runs one linear scan through the array, plus some constant time. Then, have two recursive calls to Quick Sort.

$$\text{Let } k = m - 1 - l$$

Denote the size of the left subarray. Then, the first recursive call takes time  $T(k)$ , because it is a call on an array of size  $k$ . The second recursive call will take time  $T(n - 1 - k)$ , because the size of the right subarray is  $n - 1 - k$ . Therefore, the total running time of Quick Sort satisfies the recurrence.

$$T(n) = \Theta(n) + T(k) + T(n - 1 - k),$$

$$T(1) = \Theta(1).$$

This is quite a bit messier-looking than the recurrence for Merge Sort, and has no idea about  $k$ , solving this recurrence problem isn't feasible. Certainly can work around and explore different possible values of  $k$ .

1. For  $k = n/2$ , the recurrence becomes much simpler:  $T(n) = \Theta(n) + T(n/2) + T(n/2 - 1)$ , which — as we discussed in the context of Merge Sort — we can simplify to  $T(n) = \Theta(n) + 2T(n/2)$ . That's exactly the recurrence that is already solved for Merge Sort, and thus the running time of Quick Sort would be  $\Theta(n \log n)$ .
2. At the other extreme is  $k = 0$  (or, similarly,  $k = n - 1$ ). Then, results only that  $T(n) = \Theta(n) + T(0) + T(n - 1)$ , and since  $T(0) = \Theta(1)$ , this recurrence becomes  $T(n) = \Theta(n) + T(n - 1)$ . This recurrence unrolls as  $T(n) = \Theta(n) + \Theta(n - 1) + \Theta(n - 2) + \dots + \Theta(1)$ , so

$$T(n) = \Theta\left(\sum_{i=1}^n i\right) = \Theta(n^2).$$

The running time for  $k = 0$  or  $k = n - 1$  is thus just as bad as for the simple algorithms, and in fact, for  $k = 0$ , Quick Sort is essentially the same as Selection Sort. Of course, this quadratic running time would not be a problem if only the cases  $k = 0$  and  $k = n - 1$  did not appear in practice. But in fact, they do: with the pivot choice we implemented, these cases will happen whenever the array is already sorted (increasingly or decreasingly), which should actually be an easy case. They will also happen if the array is nearly sorted.

---

---

This is quite likely in practice, for instance, because the array may have been sorted, and then just messed up a little with some new insertions.



### Did you Know?

Quicksort (sometimes called partition-exchange sort) is an efficient sorting algorithm, serving as a systematic method for placing the elements of an array in order. Developed by Tony Hoare in 1959, with his work published in 1961, it is still a commonly used algorithm for sorting. When implemented well, it can be about two or three times faster than its main competitors, merge sort and heapsort.



### Self-assessment Questions

- 4) Which of the following is an example of not in-place sorting algorithm?
  - a) Bubble Sort
  - b) Merge Sort
  - c) Selection Sort
  - d) Heap Sort
  
- 5) Sorting Algorithm that does not require any extra space for sorting is known as \_\_\_\_\_.
  - a) In-Place Sorting
  - b) Out-Place Sorting
  - c) Not in-Place Sorting
  - d) Not Out-Place Sorting
  
- 6) Which of the following is not a stable sorting algorithm?
  - a) Insertion sort
  - b) Selection sort
  - c) Bubble sort
  - d) Merge sort
  
- 7) Running merge sort on an array of size  $n$  which is already sorted is,
  - a)  $O(n \log n)$
  - b)  $O(n)$
  - c)  $O(n^2)$
  - d)  $O(n^3)$

- 
- 8) Merge sort uses,
- a) Divide-and-conquer
  - b) Backtracking
  - c) Heuristic approach
  - d) Greedy approach
- 9) For merging two sorted lists of size  $m$  and  $n$  into sorted list of size  $m+n$ , we require comparisons of:
- a)  $O(m)$
  - b)  $O(n)$
  - c)  $O(m+n)$
  - d)  $O(\log m + \log n)$
- 10) Quick sort is also known as \_\_\_\_\_.
- a) Merge sort
  - b) Tree sort
  - c) Shell sort
  - d) Partition and exchange sort



## Summary

- Bubble sort is a simple sorting algorithm. It compares the first two elements, and if the first is greater than the second, then it swaps them. It continues doing this for each pair of adjacent elements to the end of the data set, repeating until no swaps have occurred on the last pass.
- Selection sort is an in-place comparison sort. It has  $O(n^2)$  complexity, making it inefficient on large lists, and generally performs worse than the similar insertion sort.
- Insertion sort is a simple sorting algorithm that is relatively efficient for small lists and mostly sorted lists, and often is used as part of more sophisticated algorithms. It works by taking elements from the list one by one and inserting them in their correct position into a new sorted list.
- Merge sort takes advantage of the ease of merging already sorted lists into a new sorted list. It starts by comparing every two elements (*i.e.*, 1 with 2, then 3 with 4...) and swapping them if the first should come after the second. It then merges each of the resulting lists.
- Quicksort is a divide and conquer algorithm which relies on a partition operation: to partition an array an element called a pivot is selected. All elements smaller than the pivot is moved before it and all greater elements are moved after it.





## Terminal Questions

1. Explain different types of Sorting Algorithms.
2. Write down the procedure for Bubble sort.
3. Explain the sorting technique based on divide and conquer policy and find its time complexity.
4. Explain merge sort algorithm and find its time complexity.



## Answer Keys

Self-assessment Questions	
Question No.	Answer
1	c
2	c
3	a
4	b
5	a
6	b
7	a
8	a
9	c
10	d



## Activity

**1. Activity Type: Offline**

**Duration: 10 Minutes**

**Description:**

1. Divide the class into 5 groups.
2. Assign an algorithm and list of numbers to each group
3. Students should sort the list using assigned algorithm.

---

## Bibliography



### e-Reference

- pages.cs.wisc.edu, (2016). *Computer Sciences User Pages*. Retrieved on 19 April 2016, from <http://pages.cs.wisc.edu/~bobh/367/SORTING.html>



### External Resources

- Kruse, R. (2006). *Data Structures and program designing using 'C' (2nd ed.)*. Pearson Education.
- Srivastava, S. K., & Srivastava, D. (2004). *Data Structures Through C in Depth (2nd ed.)*. BPB Publications.
- Weiss, M. A. (2001). *Data Structures and Algorithm Analysis in C (2nd ed.)*. Pearson Education



### Video Links

Topic	Link
Introduction to Basics of sorting techniques	<a href="https://www.youtube.com/watch?v=pkFqlG0Hds">https://www.youtube.com/watch?v=pkFqlG0Hds</a>
Selection Sort	<a href="https://www.youtube.com/watch?v=LeNbr2ftWIo">https://www.youtube.com/watch?v=LeNbr2ftWIo</a>
Merge Sort	<a href="https://www.youtube.com/watch?v=TzeBrDU-JaY">https://www.youtube.com/watch?v=TzeBrDU-JaY</a>



**Notes:**

