Experiment -3

**Aim:**

Implement different types of sorting techniques on a given list

(i) Bubble sort

(ii) Insertion sort

(iii) Selection sort

(iv) Merge sort

(v) Quick Sort

**Description:**

Sorting is any process of arranging items according to a certain sequence using a sorting algorithm. A sorting algorithm is an algorithm that puts elements of a list in a certain order. The most-used orders are numerical order and lexicographical order, such order is called collating sequence. We can distinguish two types of sorting:

1. If the number of objects is small enough to fits into the main memory, sorting is called *internal sorting*.
2. If the number of objects is so large that some of them reside on external storage during the sort, it is called *external sorting*.

**1. Bubble Sort:**

Bubble sort uses the concept of sorting by exchange. The algorithm works by comparing each item in the list with the item next to it, and swapping them if required. In other words, the largest element has bubbled to the top of the array. The algorithm repeats this process until it makes a pass all the way through the list without swapping any items

**Algorithm:**

procedure BUBBLE\_SORT(L, n)

/\* L[0:n-1] is a linear unordered list of data elements to be sorted in accending order. \*/

for i = 0  to n-1 do /\* n-1 passes \*/

for j = 0 to n-i do

if  ( L[j] > L[j+1] )

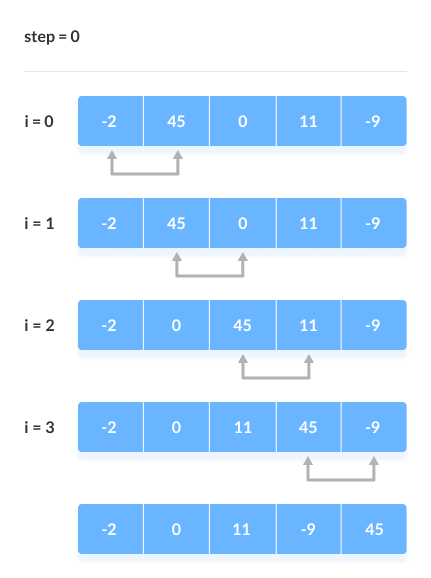
swap ( L[j], L[j+1] ); /\* swap pair wise elements \*/

end

end /\* the next largest element “bubbles” to the last position\*/

end BUBBLE\_SORT.

**1. First Iteration (Compare and Swap)**

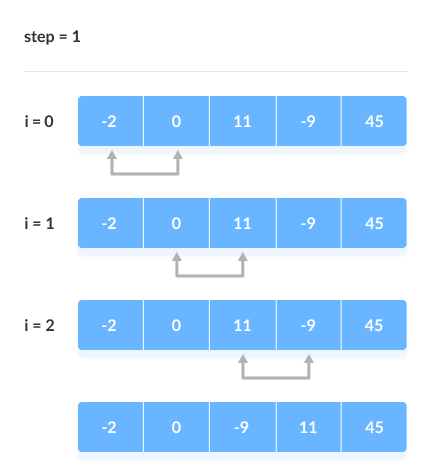
1. Starting from the first index, compare the first and the second elements.
2. If the first element is greater than the second element, they are swapped.
3. Now, compare the second and the third elements. Swap them if they are not in order.
4. The above process goes on until the last element.

Compare the Adjacent Elements

**2. Remaining Iteration**

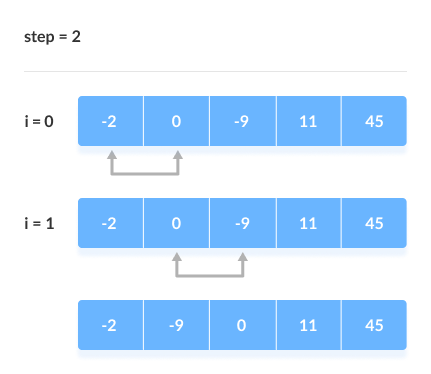
The same process goes on for the remaining iterations.

After each iteration, the largest element among the unsorted elements is placed at the end.



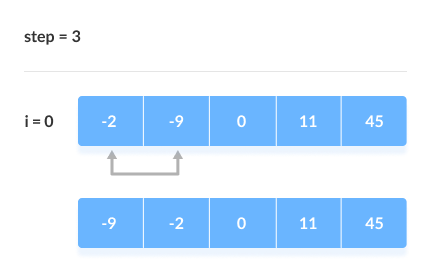
Put the largest element at the end

In each iteration, the comparison takes place up to the last unsorted element.



Compare the adjacent elements

The array is sorted when all the unsorted elements are placed at their correct positions.



The array is sorted if all elements are kept in the right order

**Complexity Analysis of Bubble Sort:**

Time Complexity: O(N2)

Auxiliary Space: O(1)

**Advantages of Bubble Sort:**

* Bubble sort is easy to understand and implement.
* It does not require any additional memory space.
* It is a stable sorting algorithm, meaning that elements with the same key value maintain their relative order in the sorted output.

**Disadvantages of Bubble Sort:**

* Bubble sort has a time complexity of O(N2) which makes it very slow for large data sets.
* Bubble sort is a comparison-based sorting algorithm, which means that it requires a comparison operator to determine the relative order of elements in the input data set. It can limit the efficiency of the algorithm in certain cases.

**Applications:**

Due to its simplicity, bubble sort is often used to introduce the concept of a sorting algorithm. In computer graphics, it is popular for its capability to detect a tiny error (like a swap of just two elements) in almost-sorted arrays and fix it with just linear complexity (2n).

**Perform bubble sort on integers**

**Program:**

/\*Bubble Sort\*/

#include<stdio.h>

int main(){

    int n;

    int arr[200];

    int temp;

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

    scanf("%d",&n);

    printf("Enter the elements:\n");

    for(int i=0;i<n;i++){

        scanf("%d",&arr[i]);

    }

    for(int i=1;i<n;i++){

        for(int j=0;j<n-i;j++){

            if(arr[j]>arr[j+1]){

             temp=arr[j];

             arr[j]=arr[j+1];

             arr[j+1]=temp;

            }

        }

    }

    printf("After sorting:\n");

    for(int i=0;i<n;i++){

        printf("%d\t",arr[i]);

    }

    return 0;

}

**Output:**

Enter the number of elements:

5

Enter the elements:

4

3

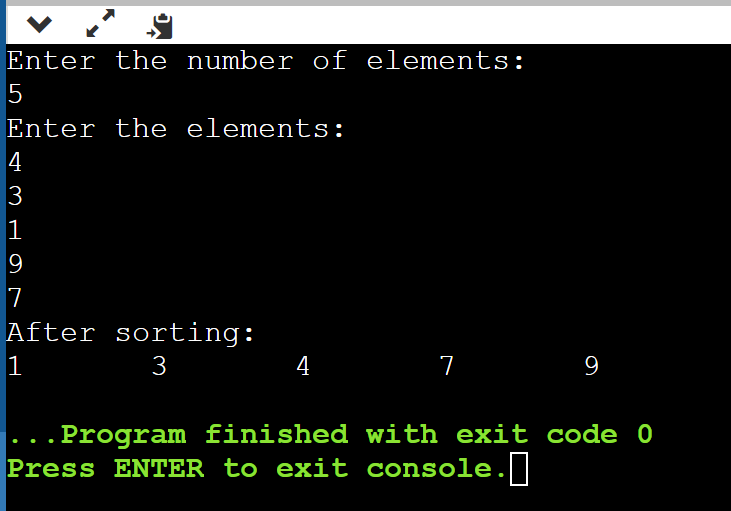
1

9

7

After sorting:

1 3 4 7 9



Enter the number of elements:

5

Enter the elements:

0

-1

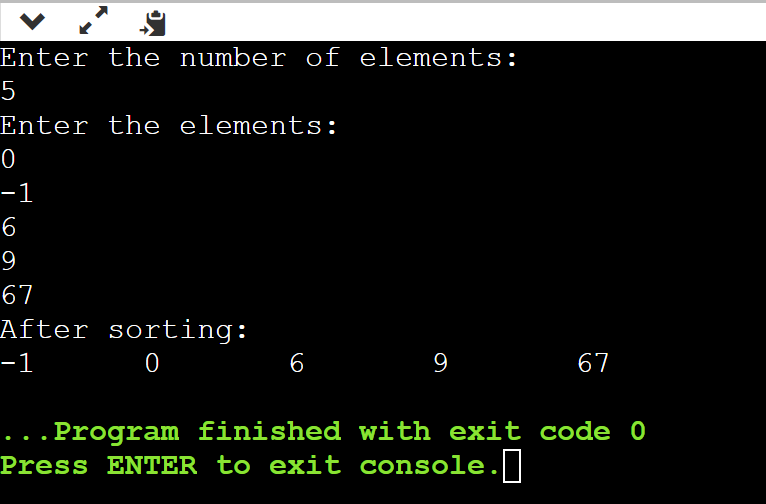
6

9

67

After sorting:

-1 0 6 9 67



**2. Insertion Sort:**

Insertion sort uses the concept of sorting by insertion. The algorithm works by inserting each element into its sorted position in the array as we read it. In this way, we can keep the array in sorted form at all times.

**Algorithm:**

procedure INSERTION\_SORT(L, n)

/\* L[0:n-1] is a linear unordered list of data elements to be sorted in accending order. \*/

for i = 2 to n do /\* n-1 passes\*/

key=L[i]; /\* key is the key to be inserted and position is its location in the unordered list \*/

position = i;

/\* compare key with its sorted sublist of predecessors for insertion at the appropriate position \*/

while (position > 1)  and (L[position-1] > key) do

L[position] = L[position – 1];

position = position – 1;

L[position] = key;

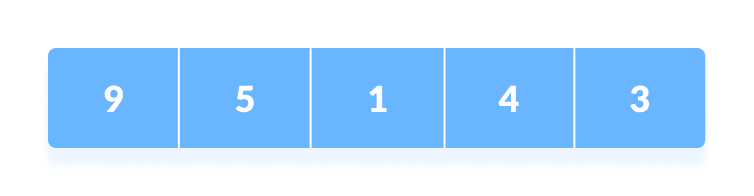
end

end

end INSERTION\_SORT.

**Example:**

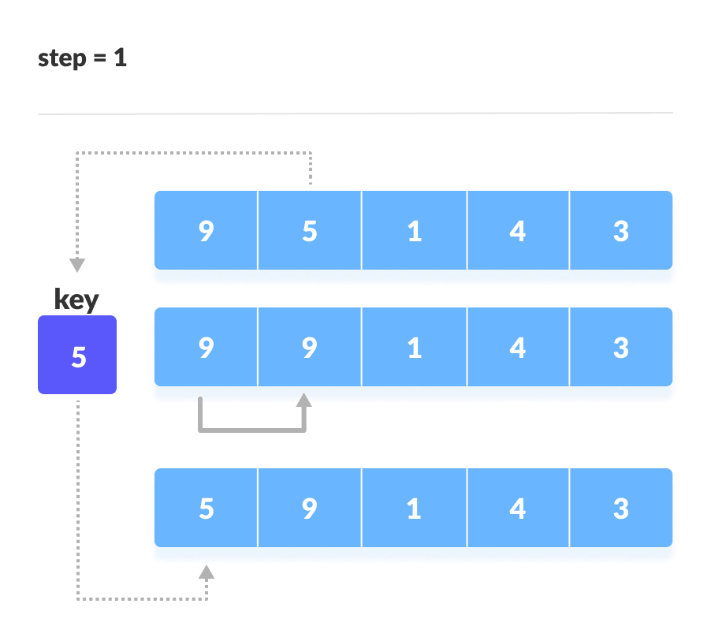
Suppose we need to sort the following array.



Initial array

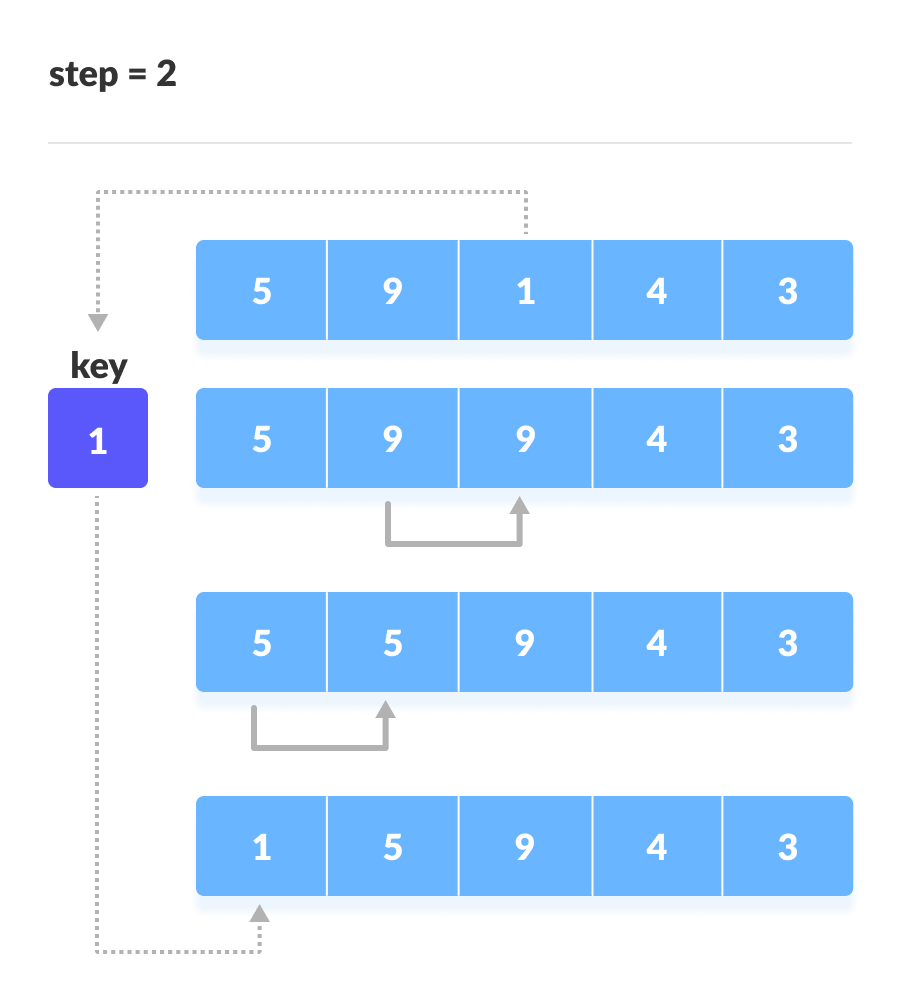
1. The first element in the array is assumed to be sorted. Take the second element and store it separately in key.  
     
   Compare key with the first element. If the first element is greater than key,

then key is placed in front of the first element.



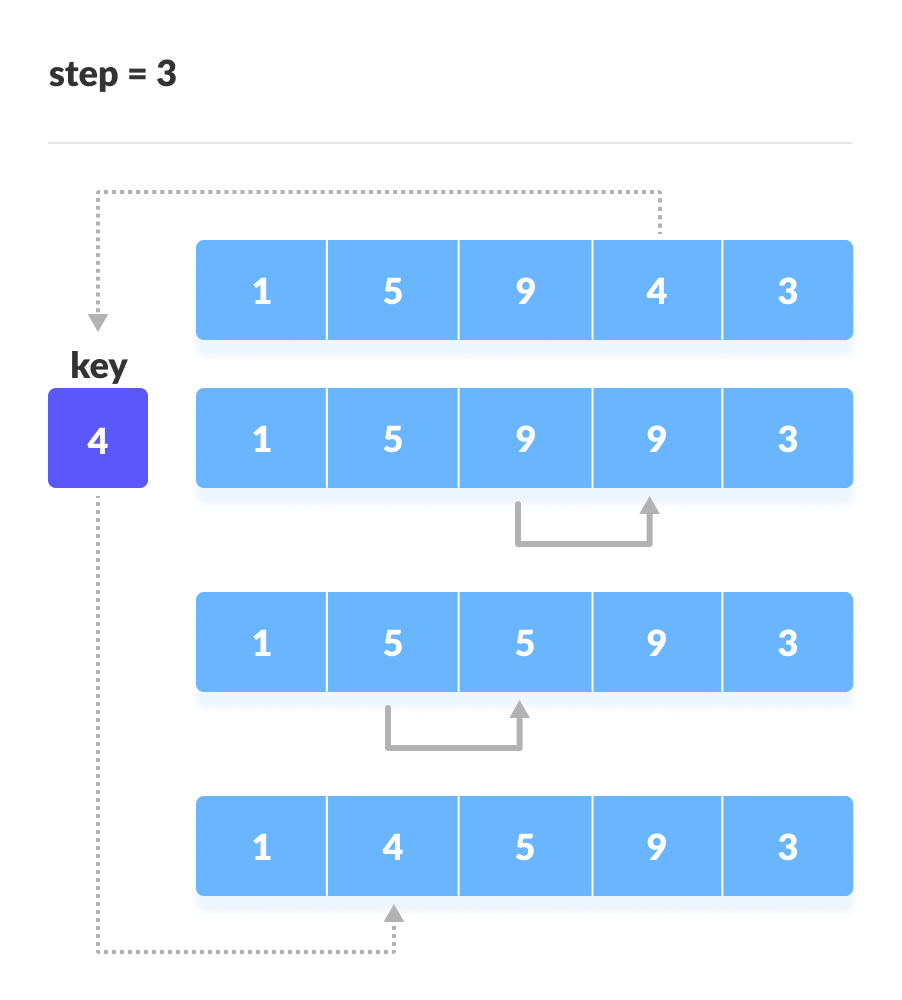
If the first element is greater than key, then key is placed in front of the first element.

1. Now, the first two elements are sorted.  
     
   Take the third element and compare it with the elements on the left of it. Placed it just behind the element smaller than it. If there is no element smaller than it, then place it at the beginning of the arr

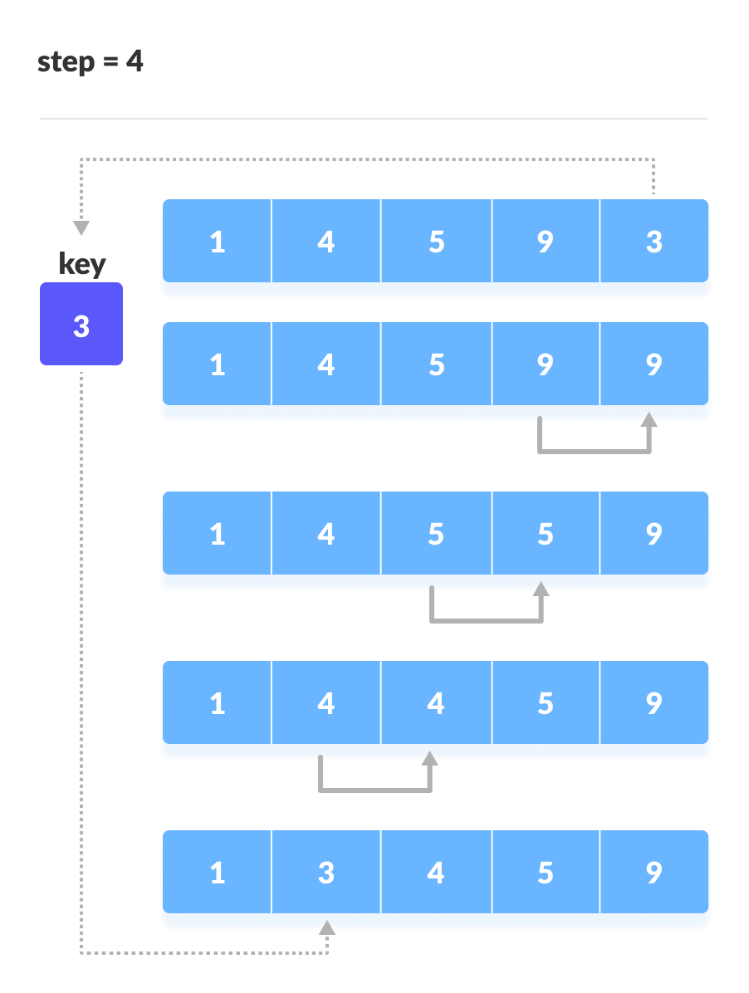


Place 1 at the beginning

1. Similarly, place every unsorted element at its correct position.



Place 4 behind 1



Place 3 behind 1 and the array is sorted

**Complexity Analysis of Insertion Sort:**

**Time Complexity of Insertion Sort**

* The worst-case time complexity of the Insertion sort is O(N^2)
* The average case time complexity of the Insertion sort is O(N^2)
* The time complexity of the best case is O(N).

**Space Complexity of Insertion Sort**

* The auxiliary space complexity of Insertion Sort is O(1)

**Characteristics of Insertion Sort:**

* This algorithm is one of the simplest algorithms with a simple implementation
* Basically, Insertion sort is efficient for small data values
* Insertion sort is adaptive in nature, i.e. it is appropriate for data sets that are already partially sorted.

**Perform insertion sort on integers**

**Program:**

/\*Insertion sort\*/

#include<stdio.h>

int main(){

    int n;

    int arr[200];

    int temp;

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

    scanf("%d",&n);

    printf("Enter the elements:\n");

    for(int i=0;i<n;i++){

        scanf("%d",&arr[i]);

    }

    for(int i=1;i<n;i++){

        int key = arr[i];

        int j=i-1;

        for(;j >= 0 && arr[j] > key;j--)

            arr[j+1] = arr[j];

        arr[j+1] = key;

    }

    printf("After sorting:\n");

    for(int i=0;i<n;i++){

        printf("%d\t",arr[i]);

    }

    return 0;

}

**Output:**

**Enter the number of elements:**

**5**

**Enter the elements:**

**4**

**1**

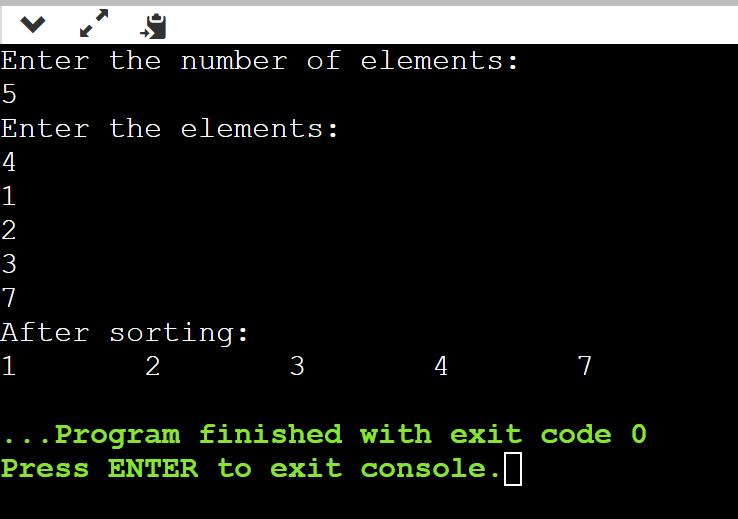
**2**

**3**

**7**

**After sorting:**

**1 2 3 4 7**

****

**Enter the number of elements:**

**5**

**Enter the elements:**

**1**

**-9**

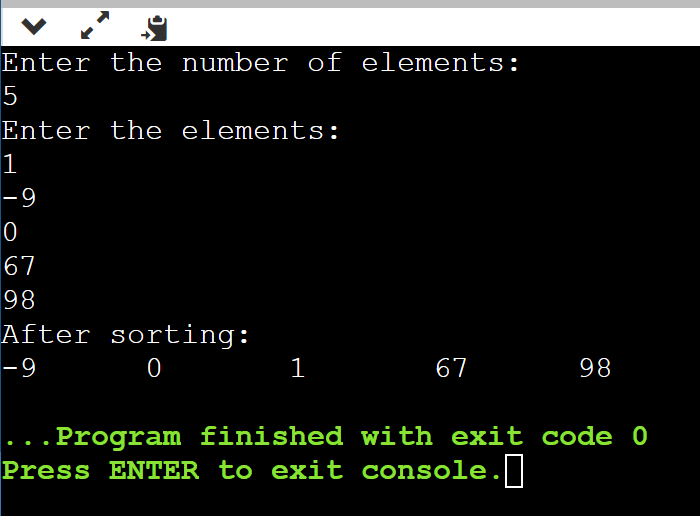
**0**

**67**

**98**

**After sorting:**

**-9 0 1 67 98**

****

**3. Selection Sort:**

Selection sort uses the concept of sorting by selection. The algorithm works by selecting the smallest unsorted item and then swapping it with the item in the next position to be filled.

The selection sort works as follows: you look through the entire array for the smallest element, once you find it you swap it (the smallest element) with the first element of the array. Then you look for the smallest element in the remaining array (an array without the first element) and swap it with the second element. Then you look for the smallest element in the remaining array (an array without first and second elements) and swap it with the third element, and so on.

**Algorithm:**

procedure SELECTION\_SORT(L, n)

/\* L[0:n-1] is a linear unordered list of data elements to be sorted in accending order. \*/

for i = 1 to n-1 do /\* n-1 passes \*/

minimum\_index = FIND\_MINIMUM(L, i, n);

/\* find minimum element of the list L[i:n] and store the position index of the element in minimum\_index \*/

swap ( L[i], L[minimum\_index]);

end

end SELECTION\_SORT.

**Example:**

Now, let's see the working of the Selection sort Algorithm.

To understand the working of the Selection sort algorithm, let's take an unsorted array. It will be easier to understand the Selection sort via an example.

Let the elements of array are -

selection Sort Algorithm

Now, for the first position in the sorted array, the entire array is to be scanned sequentially.

At present, **12** is stored at the first position, after searching the entire array, it is found that **8** is the smallest value.

selection Sort Algorithm

So, swap 12 with 8. After the first iteration, 8 will appear at the first position in the sorted array.

selection Sort Algorithm

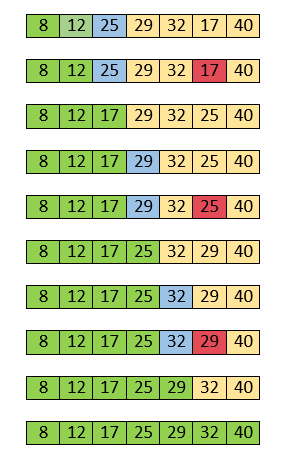
For the second position, where 29 is stored presently, we again sequentially scan the rest of the items of unsorted array. After scanning, we find that 12 is the second lowest element in the array that should be appeared at second position.

selection Sort Algorithm

Now, swap 29 with 12. After the second iteration, 12 will appear at the second position in the sorted array. So, after two iterations, the two smallest values are placed at the beginning in a sorted way.

selection Sort Algorithm

The same process is applied to the rest of the array elements. Now, we are showing a pictorial representation of the entire sorting process.



Now, the array is completely sorted.

**Complexity Analysis of Selection Sort:**

**Time Complexity:** The time complexity of Selection Sort is O(N2) as there are two nested loops:

* One loop to select an element of Array one by one = O(N)
* Another loop to compare that element with every other Array element = O(N)
* Therefore overall complexity = O(N) \* O(N) = O(N\*N) = O(N2)

**Auxiliary Space:** O(1) as the only extra memory used is for temporary variables while swapping two values in Array. The selection sort never makes more than O(N) swaps and can be useful when memory writing is costly.

**Advantages of Selection Sort Algorithm**

* Simple and easy to understand.
* Works well with small datasets.

**Disadvantages of the Selection Sort Algorithm**

* Selection sort has a time complexity of O(n^2) in the worst and average case.
* Does not work well on large datasets.
* Does not preserve the relative order of items with equal keys which means it is not stable.

**Perform selection sort on integers**

**Program:**

/\*Selection sort\*/

#include<stdio.h>

int main(){

    int n;

    int arr[200];

    int temp;

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

    scanf("%d",&n);

    printf("Enter the elements:\n");

    for(int i=0;i<n;i++){

        scanf("%d",&arr[i]);

    }

    for(int i=0;i<n;i++){

        int min=i;

        for(int j=i+1;j<n;j++){

            if(arr[j]<arr[min])min=j;

        }

        arr[i]=arr[i]+arr[min]-(arr[min]=arr[i]);

    }

    printf("After sorting:\n");

    for(int i=0;i<n;i++){

        printf("%d\t",arr[i]);

    }

    return 0;

}

**Output:**

**Enter the number of elements:**

**5**

**Enter the elements:**

**4**

**7**

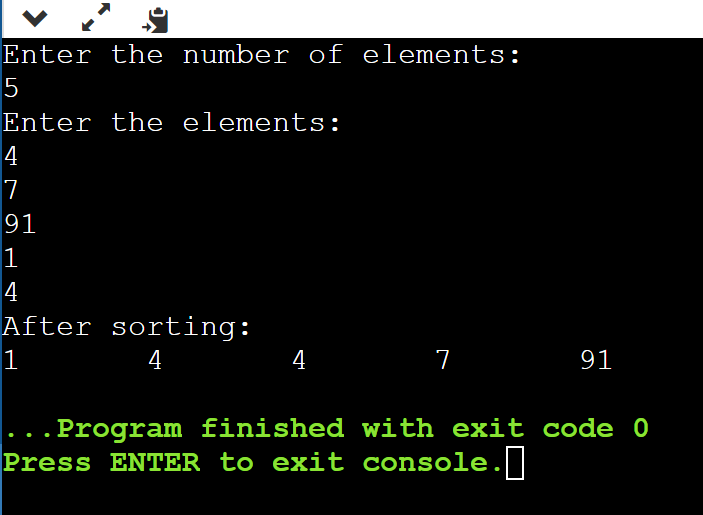
**91**

**1**

**4**

**After sorting:**

**1 4 4 7 91**

****

**Enter the number of elements:**

**5**

**Enter the elements:**

**56**

**-98**

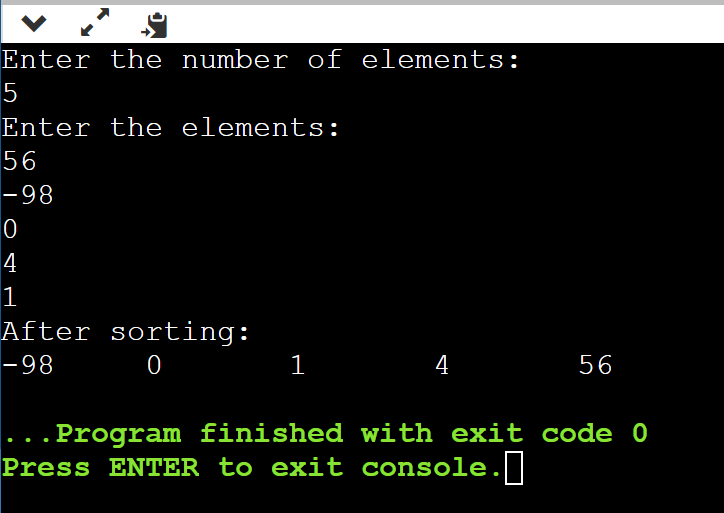
**0**

**4**

**1**

**After sorting:**

**-98 0 1 4 56**

****

**4. Merge Sort:**

Merge sort uses the concept of sorting by merging. Merge-sort is based on the divide-and-conquer paradigm. It involves the following three steps:

1. Divide the array into two (or more) subarrays using recurssion
2. Sort each subarray
3. Merge them into one.

**Algorithm:**

procedure MERGE\_SORT(a, first, last)

/\*a[first:last] is the unsorted list of elements to be merge sorted. The call to the procedure to sort the list a[1:n] would be MERGE\_SORT(a,1,n) \*/

if (first < last) then

{

mid=(first + last)/2; /\* divide the list into two sublists\*/

MERGE\_SORT(a, first, mid); /\* merge sort the sublist a[first,mid]\*/

MERGE\_SORT(a, mid+1,last);  /\* merge sort the sublist a[mid+1,last]\*/

MERGE(a, first, mid, last); /\* merge the two sublists a[first,mid] and a[mid+1,last] \*/

end MERGE\_SORT.

procedure MERGE (x, first, mid, last)

/\* x[first:mid] and x[mid+1:last] are ordered lists of data elements to be merged into a single ordered list x[first:last] \*/

first1=first;

last1=mid;

first2=mid + 1;

last2=last; /\* set the beginning and the ending indexes of the two lists into the appropriate variables\*/

i = first; /\* i is the index variable for the temporary output list temp\*/

/\* begin pair wise comparisons of elements from the two lists\*/

while (first1<=last1) and (first2<=last2) do

case

: x [first1] < x[first2] : { temp[i]= x[first1];

first1=first1 + 1;

i = i + 1;

  }

: x [first1] > x[first2] : { temp[i]= x[first2];

first2 = first2 + 1;

i = i + 1;

  }

: x [first1] = x[first2] : { temp[i]= x[first1];

temp[i + 1] = x[first2];

first1 = first1 + 1;

first2 = first2 + 1;

i = i + 2;

  }

end /\*end case\*/

end /\* end while\*/

/\* the first list gets exhausted\*/

while(first2 <= last2) do

temp[i]= x[first2];

first2 = first2 + 1;

i = i + 1;

end

/\* the second list gets exhausted\*/

while (first1<= last1) do

temp[i]= x[first1];

first1 = first1 + 1;

i = i + 1;

end

/\* copy list temp to list x\*/

for j = first to last do

x[j] = templ[j];

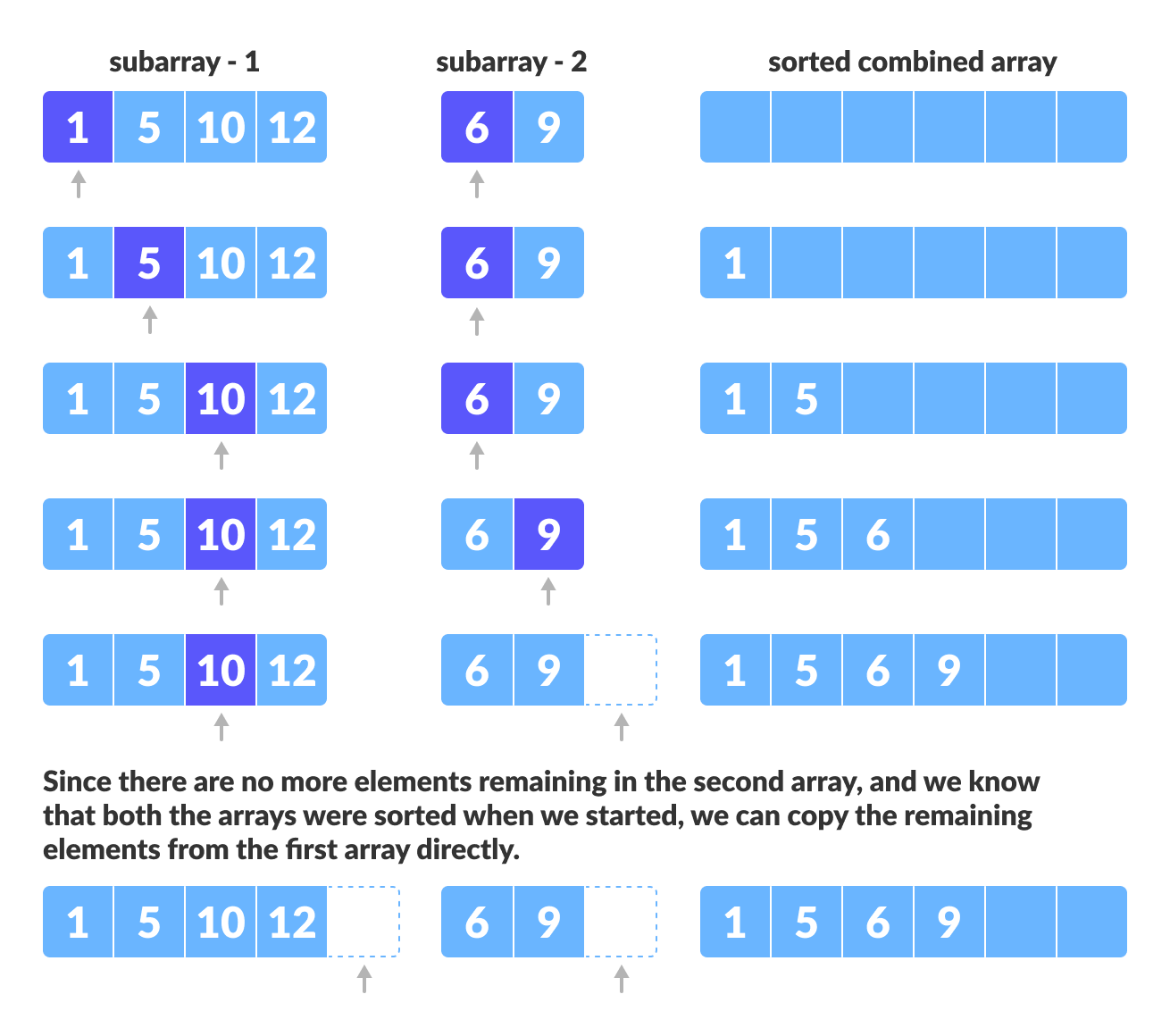
end

end MERGE.

**Example:**



**Merge step:**



**Applications of Merge Sort:**

* **Sorting large datasets:** Merge sort is particularly well-suited for sorting large datasets due to its guaranteed worst-case time complexity of O(n log n).
* **External sorting:** Merge sort is commonly used in external sorting, where the data to be sorted is too large to fit into memory.
* **Custom sorting:** Merge sort can be adapted to handle different input distributions, such as partially sorted, nearly sorted, or completely unsorted data.

**Advantages of Merge Sort:**

* **Stability:** Merge sort is a stable sorting algorithm, which means it maintains the relative order of equal elements in the input array.
* **Guaranteed worst-case performance:** Merge sort has a worst-case time complexity of O(N logN), which means it performs well even on large datasets.
* **Parallelizable**: Merge sort is a naturally parallelizable algorithm, which means it can be easily parallelized to take advantage of multiple processors or threads.

**Drawbacks of Merge Sort:**

* **Space complexity:** Merge sort requires additional memory to store the merged sub-arrays during the sorting process.
* **Not in-place:** Merge sort is not an in-place sorting algorithm, which means it requires additional memory to store the sorted data. This can be a disadvantage in applications where memory usage is a concern.
* **Not always optimal for small datasets:** For small datasets, Merge sort has a higher time complexity than some other sorting algorithms, such as insertion sort. This can result in slower performance for very small datasets.

**Time complexity:**

* **Best Case Complexity -** It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of merge sort is **O(n\*logn)**.
* **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of merge sort is **O(n\*logn)**.
* **Worst Case Complexity -** It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of merge sort is **O(n\*logn)**.

**Space complexity:**

* The space complexity of merge sort is O(n). It is because, in merge sort, an extra variable is required for swapping.

**Perform merge sort on integers**

**Program:**

/\*Merge sort\*/

#include<stdio.h>

void mergesort(int low,int high,int a[]){

    if(low<high){

    int mid=(low+high)/2;

    mergesort(low,mid,a);

    mergesort(mid+1,high,a);

    merge(low,mid,high,a);

    }

}

void merge(int low,int mid,int high,int a[]){

    int b[200];

    int h=low;

    int i=low;

    int j=mid+1;

    int k;

    while(h<=mid && j<=high){

        if(a[h]<a[j]){

            b[i]=a[h];

            h=h+1;

        }else{

            b[i]=a[j];

            j=j+1;

        }

        i=i+1;

    }

    if(h>mid){

        for(k=j;k<=high;k++){

            b[i]=a[k];

            i++;

        }

    }else{

        for(k=h;k<=mid;k++){

            b[i]=a[k];

            i++;

        }

    }

    for(k=low;k<=high;k++){

        a[k]=b[k];

    }

}

int main(){

   int n,a[200];

   printf("Enter the number:\n");

   scanf("%d",&n);

   printf("Enter the elements:");

   for(int i=0;i<n;i++){

    scanf("%d", &a[i]);

    }

    mergesort(0,n-1,a);

    printf("The elements are:");

    for(int i=0;i<n;i++){

    printf("%d \t", a[i]);

    }

}

**Output:**

**Enter the number:**

**5**

**Enter the elements:4**

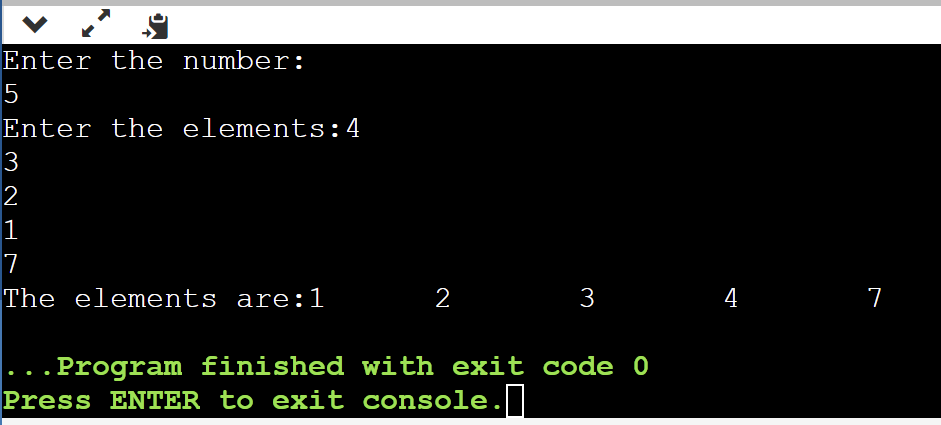
**3**

**2**

**1**

**7**

**The elements are:1 2 3 4 7**

****

**Enter the number:**

**5**

**Enter the elements:0**

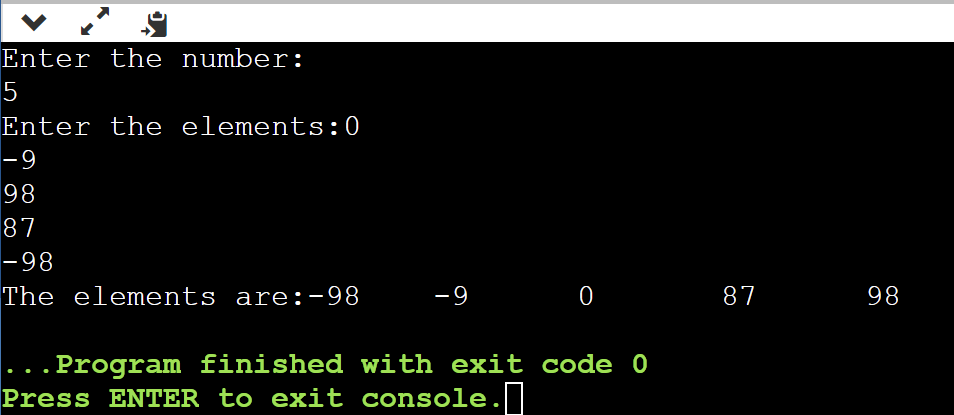
**-9**

**98**

**87**

**-98**

**The elements are:-98 -9 0 87 98**

****

**5. Quick Sort:**

Quick sort works on the principle sorting by exchange or transposition. Quick sort is a divide and conquer algorithm.  Quick sort first divides a large array into two smaller sub-arrays: the low elements and the high elements. Quick sort can then recursively sort the sub-arrays. The steps are:

* 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.

**Algorithm:**

Procedure QUICK\_SORT(L, first, last )

/\* L[first:last] is the unordered list of elements to be quick sorted. The call to the procedure to sort the list L[1:n] would be QUICK\_SORT(L, 1, n)\*/

if (first < last) then

{ PARTITION(L, first, last, loc) ; /\* partition the list into two sublists at

                                                                     loc\*/

QUICK\_SORT(L, first, loc-1 ); /\* quick sort the sublist L[first,loc-1]\*/

QUICK\_SORT(L, loc+1, last ): /\* quick sort the sublist L[loc+1, last]\*/

}

end OUICK\_SORT.

procedure PARTITION (L, first, last, loc )

/\* L[first:last] is the list to be partitioned. loc is the position where the pivot element finally settles down\*/

left = first;

right = last+1;

pivot\_elt = L[first];       /\* set the pivot element to the first element in list L\*/

while (left < right) do

repeat

left = left+1; /\* pivot element moves left to right\*/

until    L[left] >= pivot\_elt;

repeat

right = right -1; /\* pivot element moves right to left\*/

until    L[right] <= pivot\_elt;

if (left < right) then

swap(L[left], L[right]); /\*arrows face each other\*/

end

loc = right

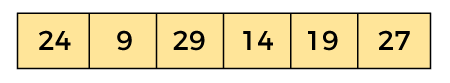
/\* arrows have crossed each other – exchange pivot element L[first] with L[right] \*/

end PARTITION.

Now, let's see the working of the Quicksort Algorithm.

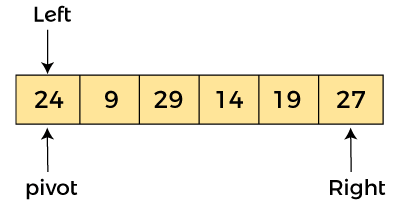
To understand the working of quick sort, let's take an unsorted array. It will make the concept more clear and understandable.

Let the elements of array are -

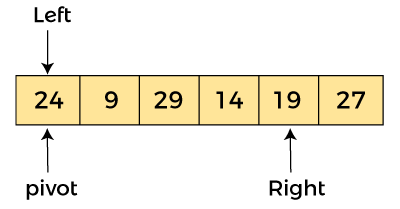


In the given array, we consider the leftmost element as pivot. So, in this case, a[left] = 24, a[right] = 27 and a[pivot] = 24.

Since, pivot is at left, so algorithm starts from right and move towards left.

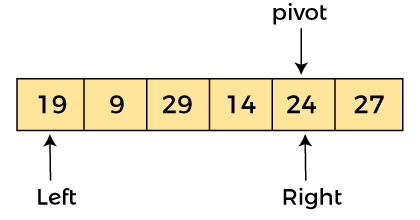


Now, a[pivot] < a[right], so algorithm moves forward one position towards left, i.e. -



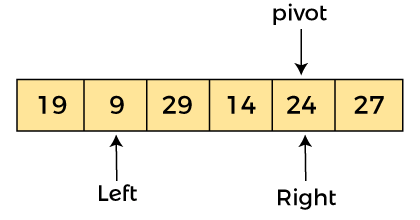
Now, a[left] = 24, a[right] = 19, and a[pivot] = 24.

Because, a[pivot] > a[right], so, algorithm will swap a[pivot] with a[right], and pivot moves to right, as -

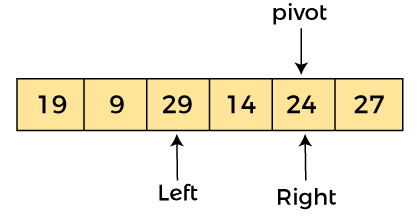


Now, a[left] = 19, a[right] = 24, and a[pivot] = 24. Since, pivot is at right, so algorithm starts from left and moves to right.

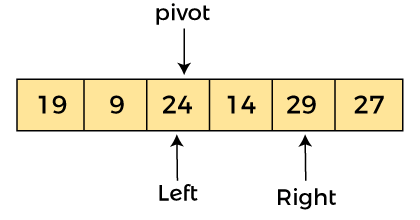
As a[pivot] > a[left], so algorithm moves one position to right as -



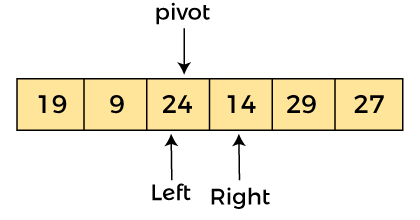
Now, a[left] = 9, a[right] = 24, and a[pivot] = 24. As a[pivot] > a[left], so algorithm moves one position to right as -



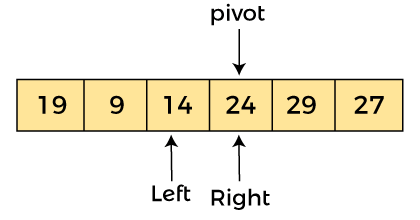
Now, a[left] = 29, a[right] = 24, and a[pivot] = 24. As a[pivot] < a[left], so, swap a[pivot] and a[left], now pivot is at left, i.e. -



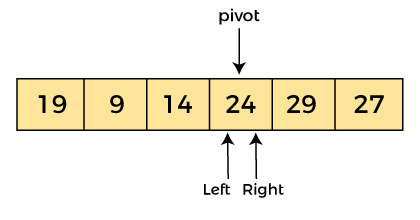
Since, pivot is at left, so algorithm starts from right, and move to left. Now, a[left] = 24, a[right] = 29, and a[pivot] = 24. As a[pivot] < a[right], so algorithm moves one position to left, as -



Now, a[pivot] = 24, a[left] = 24, and a[right] = 14. As a[pivot] > a[right], so, swap a[pivot] and a[right], now pivot is at right, i.e. -



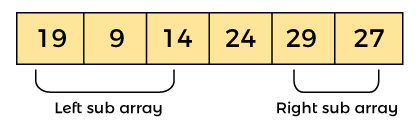
Now, a[pivot] = 24, a[left] = 14, and a[right] = 24. Pivot is at right, so the algorithm starts from left and move to right.



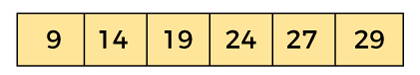
Now, a[pivot] = 24, a[left] = 24, and a[right] = 24. So, pivot, left and right are pointing the same element. It represents the termination of procedure.

Element 24, which is the pivot element is placed at its exact position.

Elements that are right side of element 24 are greater than it, and the elements that are left side of element 24 are smaller than it.



Now, in a similar manner, quick sort algorithm is separately applied to the left and right sub-arrays. After sorting gets done, the array will be -



**Advantages of Quick Sort:**

* It is a divide-and-conquer algorithm that makes it easier to solve problems.
* It is efficient on large data sets.
* It has a low overhead, as it only requires a small amount of memory to function.

**Disadvantages of Quick Sort:**

* It has a worst-case time complexity of O(N2), which occurs when the pivot is chosen poorly.
* It is not a good choice for small data sets.
* It is not a stable sort, meaning that if two elements have the same key, their relative order will not be preserved in the sorted output in case of quick sort, because here we are swapping elements according to the pivot’s position (without considering their original positions).

**Time complexity:**

* **Best Case Complexity -** In Quicksort, the best-case occurs when the pivot element is the middle element or near to the middle element. The best-case time complexity of quicksort is **O(n\*logn)**.
* **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of quicksort is **O(n\*logn)**.
* **Worst Case Complexity -** In quick sort, worst case occurs when the pivot element is either greatest or smallest element. Suppose, if the pivot element is always the last element of the array, the worst case would occur when the given array is sorted already in ascending or descending order. The worst-case time complexity of quicksort is **O(n2)**.

**Space complexity:**

* The space complexity of quicksort is O(n\*logn).

**Perform quick sort o integrers**

**Program:**

/\*Quicksort\*/

#include<stdio.h>

void quicksort(int a[],int p,int q){

    if(p<q){

        int j=partition(a,p,q);

        quicksort(a,p,j-1);

        quicksort(a,j+1,q);

    }

}

int partition(int a[],int low,int high){

       int pivot= a[low];

       int i=low;

       int j=high;

       int temp;

       while(i<j)

        {

                 while (a[i]<=pivot)

                          i++;

                 while(a[j]>pivot)

                          j--;

                 if(i<j){

                        temp=a[i];

                        a[i]=a[j];

                        a[j]=temp;

                     }

         }

         a[low]=a[j];

         a[j]=pivot;

         return j;

}

int main(){

     int n,a[200];

   printf("Enter the number:\n");

   scanf("%d",&n);

   printf("Enter the elements:");

   for(int i=0;i<n;i++){

    scanf("%d", &a[i]);

    }

    quicksort(a,0,n-1);

    printf("The elements are:");

    for(int i=0;i<n;i++){

    printf("%d \t", a[i]);

    }

    return 0;}

**Output:**

**Enter the number:**

**5**

**Enter the elements:1**

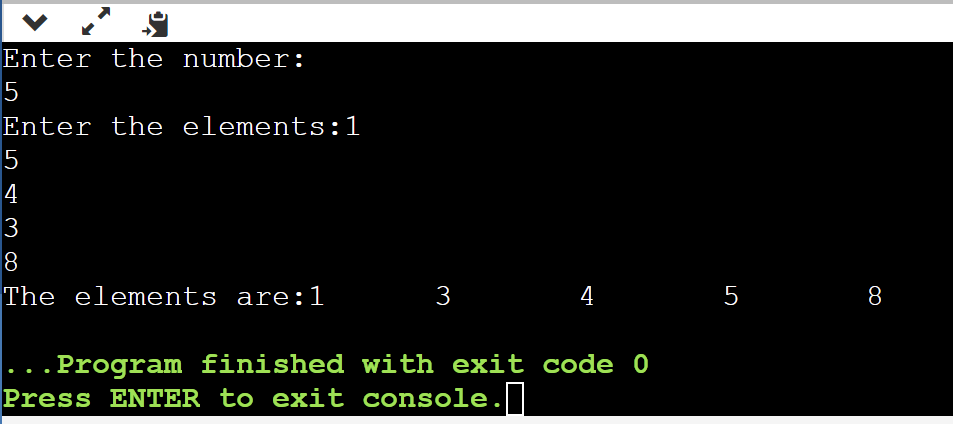
**5**

**4**

**3**

**8**

**The elements are:1 3 4 5 8**

****

**Enter the number:**

**5**

**Enter the elements:-9**

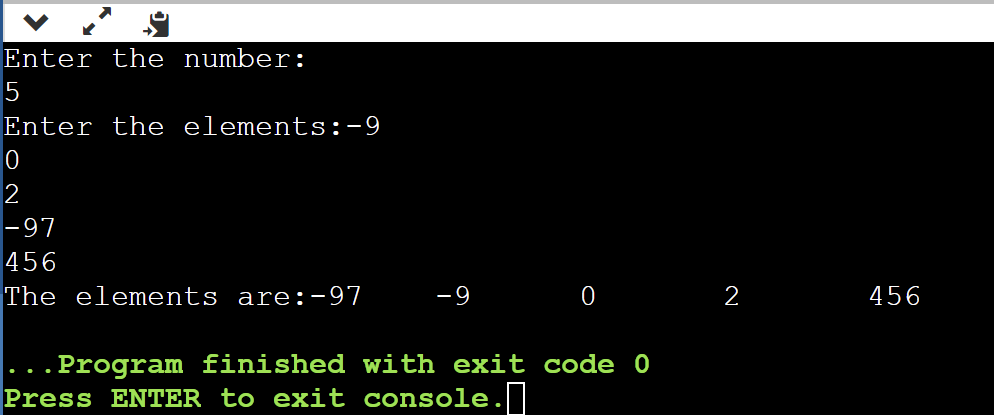
**0**

**2**

**-97**

**456**

**The elements are:-97 -9 0 2 456**

****

**Additional problems:**

**Big sorting**

**Program:**

#include <stdio.h> #include <string.h> #include <stdlib.h>

int stringAsInt(const void \*pLeft, const void \*pRight)

{

const char \*left = \*(const char \*\*)pLeft; const char \*right = \*(const char \*\*)pRight; int leftLen = (int)strlen(left);

int rightLen = (int)strlen(right);

if (leftLen != rightLen)

{

return leftLen - rightLen;

}

else

{

return strcmp(left, right);

}

}

int main()

{

int n; scanf("%d", &n);

char buffer[1000000 + 1];

char \*\*a = malloc(sizeof(char \*) \* (size\_t)n);

for (int i = 0; i < n; i++)

{

scanf("%1000000s", buffer);

a[i] = malloc(sizeof(char) \* (strlen(buffer) + 1)); strcpy(a[i], buffer);

}

qsort(a, (size\_t)n, sizeof(a[0]), stringAsInt);

for (int i = 0; i < n; i++)

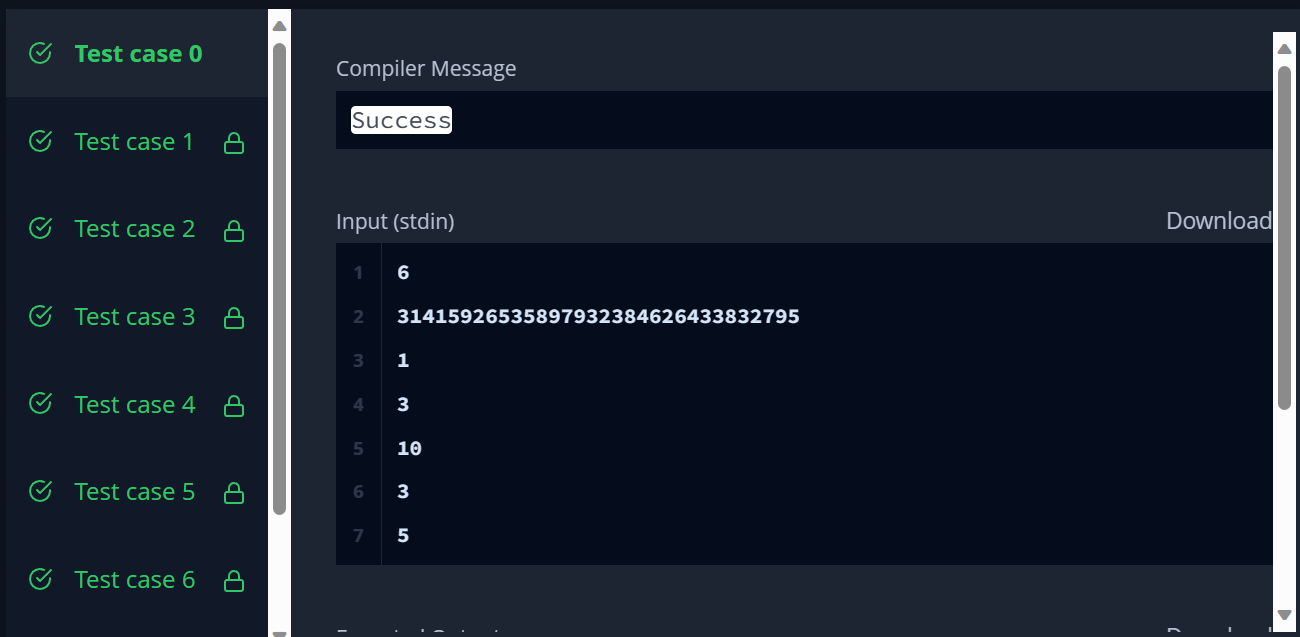
{

printf("%s\n", a[i]); free(a[i]);

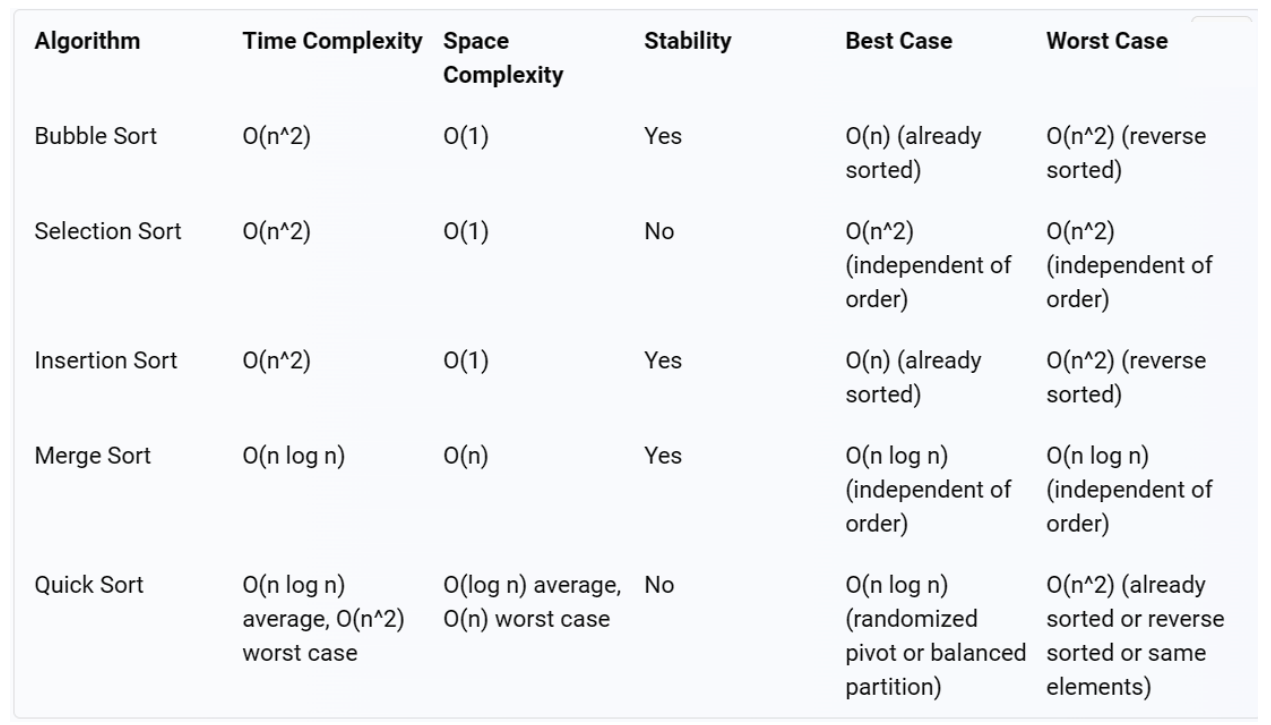
}

free(a); return 0;

}

**Output:**

**Comparison between all 5 sorting algorithms:**

****

**Viva questions:**

**1. What is a sorting algorithm?**

**Answer:** A sorting algorithm is a method or process used to arrange a list of items or data elements in a specific order, usually in ascending or descending order.

**2. Why do we need sorting algorithms?**

**Answer:** Sorting algorithms are essential for organizing data, making it easier to search for and retrieve information efficiently. They are used in various applications such as databases, search engines, and data analysis.

**3. What is the time complexity of a sorting algorithm, and why is it important?**

**Answer:** The time complexity of a sorting algorithm describes how its runtime grows with the size of the input data. It's crucial because it helps us analyze and compare the efficiency of different sorting algorithms, allowing us to choose the most suitable one for a given problem.

**4. Can you explain the difference between internal and external sorting?**

**Answer:** Internal sorting refers to sorting data that can fit entirely in the computer's memory (RAM). External sorting is used when the data is too large to fit in memory and must be sorted in smaller chunks, typically using disk-based storage.

**5. What is the difference between stable and unstable sorting algorithms?**

**Answer:** A stable sorting algorithm maintains the relative order of equal elements in the sorted output, while an unstable sorting algorithm may change the relative order of equal elements.

**6. Give an example of a stable sorting algorithm.**

**Answer:** Merge Sort is an example of a stable sorting algorithm.

**7. Explain the concept of comparison-based sorting algorithms.**

**Answer:** Comparison-based sorting algorithms sort elements by comparing pairs of elements and making decisions based on the outcomes of these comparisons. The most common comparison operation is less than (<).

**8. What is the time complexity of the bubble sort algorithm?**

**Answer:** The average and worst-case time complexity of the bubble sort algorithm is O(n^2), where 'n' is the number of elements to be sorted.

**9. Describe the quicksort algorithm and its time complexity.**

**Answer:** Quicksort is a divide-and-conquer sorting algorithm that selects a pivot element and partitions the array into two subarrays: one with elements less than the pivot and one with elements greater than the pivot. The time complexity of quicksort is O(n log n) on average, but it can degrade to O(n^2) in the worst case.