

**Module IV: Sorting:** Bubble Sort, Selection Sort, and Insertion Sort. **Searching:** Sequential searching, binary searching. **Hashing-** hash table, types of hash functions, Collision Resolution Techniques-linear probing, quadratic probing, double hashing, chaining.

### **Linear search /Sequential search**

Linear search is a very simple search algorithm. In this type of search, a sequential search is made over all items one by one. Every item is checked and if a match is found then that particular item is returned otherwise search continues till the end of the data collection. **Algorithm**

Linear Search ( Array A, Value x)

Step 1: Set i to 1

Step 2: if  $i > n$  then go to step 7

Step 3: if  $A[i] = x$  then go to step 6

Step 4: Set i to  $i + 1$

Step 5: Go to Step 2

Step 6: Print Element x Found at index i and go to step 8

Step 7: Print element not found

Step 8: Exit.

### **Pseudocode**

```
Procedure linear_search (list,
value)
  for each item in the list
    if match item == value
      return the item's location
    end if
  end for
end procedure
```

### **Binary search**

- Binary search searches a particular item by comparing the middle most item of the collection.
- If a match occurs then the index of the item is returned.
- If the middle item is greater than the item then the item is searched in the sub-array to the right of the middle item otherwise the item is searched in the sub-array to the left of the middle item.
- This process continues on the sub-array as well until the size of the subarray reduces to zero.

### **How binary search works?**


For a binary search to work, it is mandatory for the target array to be sorted. We shall learn the process of binary search with a pictorial example. The below given is our sorted array and assume that we need to search the location of value 31 using binary search.

10	14	19	26	27	31	33	35	42	44
0	1	2	3	4	5	6	7	8	9

First, we shall determine the half of the array by using this formula –

$$\text{mid} = \text{low} + (\text{high} - \text{low}) / 2$$

Here it is,  $0 + (9 - 0) / 2 = 4$  (integer value of 4.5). So 4 is the mid of array.



10	14	19	26	27	31	33	35	42	44
0	1	2	3	4	5	6	7	8	9

Now we compare the value stored at location 4, with the value being searched i.e. 31. We find that value at location 4 is 27, which is not a match. Because value is greater than 27 and we have a sorted array so we also know that target value must be in upper portion of the array.

10	14	19	26	27	31	33	35	42	44
0	1	2	3	4	5	6	7	8	9

We change our low to mid + 1 and find the new mid value again.

$$\text{low} = \text{mid} + 1$$

$$\text{mid} = \text{low} + (\text{high} - \text{low}) / 2$$

Our new mid is 7 now. We compare the value stored at location 7 with our target value 31.

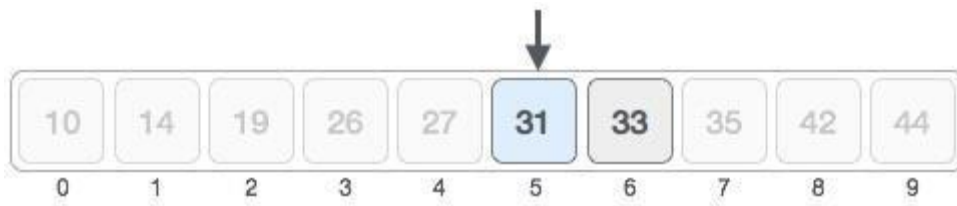


10	14	19	26	27	31	33	35	42	44
0	1	2	3	4	5	6	7	8	9

The value stored at location 7 is not a match, rather it is less than what we are looking for. So the value must be in lower part from this location.

10	14	19	26	27	31	33	35	42	44
0	1	2	3	4	5	6	7	8	9

So we calculate the mid again. This time it is 5.



We compare the value stored at location 5 with our target value. We find that it is a match.



We conclude that the target value 31 is stored at location 5.

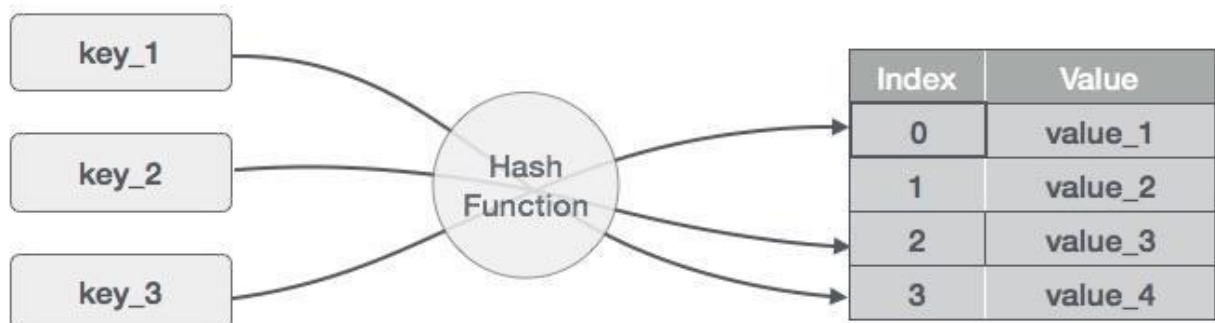
Binary search halves the searchable items and thus reduces the count of comparisons to be made to very less numbers.

## Hashing

Hashing is a technique to convert a range of key values into a range of indexes of an array.

We're using modulo operator to get a range of key values.

Consider an example of hashtable of size 20, and following items are to be stored. Items are in (key,value) format.



- (1,20)
- (2,70)
- (42,80)
- (4,25)
- (12,44)
- (14,32)
- (17,11)
- (13,78)
- (37,98)

S.n.	Key	Hash	Array Index
1	1	$1 \% 20 = 1$	1
2	2	$2 \% 20 = 2$	2
3	42	$42 \% 20 = 2$	2
4	4	$4 \% 20 = 4$	4
5	12	$12 \% 20 = 12$	12
6	14	$14 \% 20 = 14$	14
7	17	$17 \% 20 = 17$	17
8	13	$13 \% 20 = 13$	13
9	37	$37 \% 20 = 17$	17

### Basic Operations

Following are basic primary operations of a hashtable which are following.

1. Search – search an element in a hashtable.
2. Insert – insert an element in a hashtable.
3. delete – delete an element from a hashtable.

### Dataltem

Define a data item having some data, and key based on which search is to be conducted in hashtable.

```
struct Dataltem
```

```
{
int
data;
int
key;
};
```

### Hash Method

Define a hashing method to compute the hash code of the key of the data item.

```
int hashCode(int key)
{
return key % SIZE;
}
```

## Collision Resolution Techniques

### Collision in hashing

1. In this, the hash function is used to compute the index of the array.
2. The hash value is used to store the key in the hash table, as an index.
3. The hash function can return the same hash value for two or more keys.
4. When two or more keys are given the same hash value, it is called a **collision**. To handle this collision, we use collision resolution techniques.

### Collision resolution techniques

There are two types of collision resolution techniques.

1. Separate chaining (open hashing)
2. Open addressing (closed hashing)

### Separate chaining

In this technique, a linked list is created from the slot in which collision has occurred, after which the new key is inserted into the linked list. This linked list of slots looks like a chain, so it is called **separate chaining**. It is used more when we do not know how many keys to insert or delete.

### Time complexity

1. Its worst-case complexity for searching is  $O(n)$ .
2. Its worst-case complexity for deletion is  $O(n)$ .

### Advantages of separate chaining

1. It is easy to implement.
2. The hash table never fills full, so we can add more elements to the chain.
3. It is less sensitive to the function of the hashing.

### Disadvantages of separate chaining

1. In this, cache performance of chaining is not good.
2. The memory wastage is too much in this method.
3. It requires more space for element links.

## **Bubble Sort / Exchange sort**

- Bubble sort is a simple sorting algorithm.
- This sorting algorithm is comparison based algorithm in which each pair of adjacent elements is compared and elements are swapped if they are not in order.
- This algorithm is not suitable for large data sets as its average and worst case complexity are of  $O(n^2)$  where  $n$  are no. of items.

How bubble sort works?

We take an unsorted array for our example. Bubble sort take  $O(n^2)$  time so we're keeping short and precise.



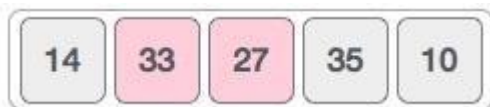
Bubble sort starts with very first two elements, comparing them to check which one is greater.



In this case, value 33 is greater than 14, so it is already in sorted locations. Next, we compare 33 with 27.



We find that 27 is smaller than 33 and these two values must be swapped.



The new array should look like this –



Next we compare 33 and 35. We find that both are in already sorted positions.



Then we move to next two values, 35 and 10.



We know that 10 is smaller than 35. Hence they are not sorted.



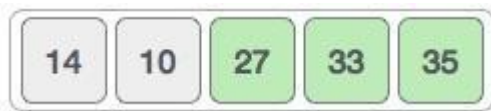
We swap these values. We find that we reach at the end of the array. After one iteration the array should look like this –



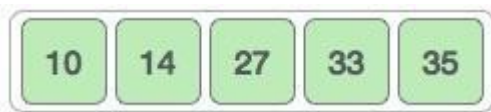
To be precise, we are now showing that how array should look like after each iteration. After second iteration, it should look like this –



Notice that after each iteration, at least one value moves at the end.



And when there's no swap required, bubble sort learns that array is completely sorted.



Now we should look into some practical aspects of bubble sort.

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

#define MAX 10
int list[MAX] =
{1,8,4,6,0,3,5,2,7,9};

void
display(){
int i;

printf("[");

    // navigate through all
items    for(i = 0; i < MAX;
i++){          printf("%d
",list[i]);
```

```

    }

    printf("]\n");
} void
bubbleSort() {
    int temp;
    int i,j;

    bool swapped = false;

    // loop through all numbers
    for(i = 0; i < MAX-1; i++) {
        swapped = false;

        // loop through numbers falling ahead        for(j = 0; j <
MAX-1-i; j++) {            printf("        Items compared: [ %d, %d ]
", list[j],list[j+1]);

            // check if next number is lesser than current no
            //    swap the numbers.
            //    (Bubble up the highest number)

            if(list[j] >
list[j+1]) {                temp
= list[j];                list[j]
= list[j+1];
list[j+1] = temp;

```



```

        swapped = true;                printf(" => swapped [%d,
%d]\n",list[j],list[j+1]);
    }else {                            printf(" =>
not swapped\n");
    }

    }

    // if no number was swapped that means
//  array is sorted now, break the loop.
if(!swapped) {                        break;
    }                                printf("Iteration
%d#: ",(i+1));                        display();
    }

}  main(){    printf("Input
Array: ");    display();
printf("\n");

    bubbleSort();
printf("\nOutput Array: ");
display();
}

```

## Output

```
Input Array: [1 8 4 6 0 3 5 2 7 9 ]
```

```
    Items compared: [ 1, 8 ] => not swapped
    Items compared: [ 8, 4 ] => swapped [4, 8]
Items compared: [ 8, 6 ] => swapped [6, 8]
Items compared: [ 8, 0 ] => swapped [0, 8]
Items compared: [ 8, 3 ] => swapped [3, 8]
Items compared: [ 8, 5 ] => swapped [5, 8]
Items compared: [ 8, 2 ] => swapped [2, 8]
    Items compared: [ 8, 7 ] => swapped [7, 8]
    Items compared: [ 8, 9 ] => not swapped Iteration
1#: [1 4 6 0 3 5 2 7 8 9 ]
    Items compared: [ 1, 4 ] => not swapped
    Items compared: [ 4, 6 ] => not swapped
    Items compared: [ 6, 0 ] => swapped [0, 6]
Items compared: [ 6, 3 ] => swapped [3, 6]
Items compared: [ 6, 5 ] => swapped [5, 6]
    Items compared: [ 6, 2 ] => swapped [2, 6]
    Items compared: [ 6, 7 ] => not swapped
    Items compared: [ 7, 8 ] => not swapped Iteration
2#: [1 4 0 3 5 2 6 7 8 9 ]
    Items compared: [ 1, 4 ] => not swapped
    Items compared: [ 4, 0 ] => swapped [0, 4]
    Items compared: [ 4, 3 ] => swapped [3, 4]
Items compared: [ 4, 5 ] => not swapped
    Items compared: [ 5, 2 ] => swapped [2, 5]
    Items compared: [ 5, 6 ] => not swapped
    Items compared: [ 6, 7 ] => not swapped
Iteration 3#: [1 0 3 4 2 5 6 7 8 9 ]
    Items compared: [ 1, 0 ] => swapped [0, 1]
    Items compared: [ 1, 3 ] => not swapped
Items compared: [ 3, 4 ] => not swapped
    Items compared: [ 4, 2 ] => swapped [2, 4]
    Items compared: [ 4, 5 ] => not swapped
    Items compared: [ 5, 6 ] => not swapped Iteration
4#: [0 1 3 2 4 5 6 7 8 9 ]
    Items compared: [ 0, 1 ] => not swapped
Items compared: [ 1, 3 ] => not swapped
    Items compared: [ 3, 2 ] => swapped [2, 3]
    Items compared: [ 3, 4 ] => not swapped
Items compared: [ 4, 5 ] => not swapped
Iteration 5#: [0 1 2 3 4 5 6 7 8 9 ]
    Items compared: [ 0, 1 ] => not swapped
Items compared: [ 1, 2 ] => not swapped
```

Items compared: [ 2, 3 ] => not swapped

Items compared: [ 3, 4 ] => not swapped

Output Array: [ 0 1 2 3 4 5 6 7 8 9 ]

## Insertion sort

- This is a in-place comparison based sorting algorithm.
- Here, a sub-list is maintained which is always sorted. For example, the lower part of an array is maintained to be sorted. A element which is to be 'insert'ed in this sorted sub-list, has to find its appropriate place and insert it there. Hence the name **insertion sort**.
- The array is searched sequentially and unsorted items are moved and inserted into sorted sub-list (in the same array). This algorithm is not suitable for large data sets as its average and worst case complexity are of  $O(n^2)$  where n are no. of items.

### How insertion sort works?

We take an unsorted array for our example.



Insertion sort compares the first two elements.



It finds that both 14 and 33 are already in ascending order. For now, 14 is in sorted sub-list.



Insertion sort moves ahead and compares 33 with 27.



And finds that 33 is not in correct position.



It swaps 33 with 27. Also it checks with all the elements of sorted sublist. Here we see that sorted sub-list has only one element 14 and 27 is greater than 14. Hence sorted sub-list remain sorted after swapping.



By now we have 14 and 27 in the sorted sublist. Next it compares 33 with 10,.



These values are not in sorted order.



So we swap them.



But swapping makes 27 and 10 unsorted.



So we swap them too.



Again we find 14 and 10 in unsorted order.



And we swap them. By the end of third iteration we have a sorted sublist of 4 items.



This process goes until all the unsorted values are covered in sorted sublist. And now we shall see some programming aspects of insertion sort.

## **Algorithm**

Now we have a bigger picture of how this sorting technique works, so we can derive simple steps by which we can achieve insertion sort.

Step 1 – If it is the first element, it is already sorted. return 1;

Step 2 – Pick next element

Step 3 – Compare with all elements in the sorted sub-list

Step 4 – Shift all the elements in the sorted sub-list that is greater than the value to be sorted

Step 5 – Insert the value

Step 6 – Repeat until list is sorted

## Implementation in C

This is a in-place comparison based sorting algorithm. Here, a sub-list is maintained which is always sorted. For example, the lower part of an array is maintained to be sorted. A element which is to be 'insert'ed in this sorted sublist, has to find its appropriate place and insert it there. Hence the name insertion sort.

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

#define MAX 7
int intArray[MAX] =
{4,6,3,2,1,9,7};

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
insertionSort()
{    int
valueToInsert;
int
holePosition;
int i;

    // loop through all numbers
for(i = 1; i < MAX; i++)
{

    // select a value to be inserted.
    valueToInsert = intArray[i];

    // select the hole position where number is to be inserted
holePosition = i;

    // check if previous no. is larger than value to be inserted
while (holePosition > 0 && intArray[holePosition-1] > valueToInsert){
intArray[holePosition] = intArray[holePosition-1];
holePosition--;        printf(" item moved : %d\n" ,
intArray[holePosition]);
    }
if(holePosition != i)
{

    printf(" item inserted : %d, at position : %d\n" , valueToInsert,holePosition);

```

```

        // insert the number at hole
    position      intArray[holePosition] =
valueToInsert;
    }
    printf("Iteration %d#:",i);
    display();

}
} main() {
    printf("Input Array:
");    display();
    printline(50);
    insertionSort();
    printf("Output Array:
");    display();
    printline(50);
}

```

If we compile and run the above program then it would produce following result –

# Output

```
Input Array: [4, 6, 3, 2, 1, 9, 7]
=====
iteration 1#: [4, 6, 3, 2, 1, 9, 7]  item moved :6
item moved :4
    item inserted :3, at position :0
iteration 2#: [3, 4, 6, 2, 1, 9, 7]
item moved :6  item moved :4  item
moved :3
    item inserted :2, at position :0
iteration 3#: [2, 3, 4, 6, 1, 9, 7]
item moved :6  item moved :4  item
moved :3  item moved :2
    item inserted :1, at position :0
iteration 4#: [1, 2, 3, 4, 6, 9, 7]
iteration 5#: [1, 2, 3, 4, 6, 9, 7]
item moved :9  item moved :6
    item inserted :7, at position :4
iteration 6#: [1, 2, 3, 4, 7, 6, 9] Output
Array: [1, 2, 3, 4, 7, 6, 9]
=====
```

## Selection sort

- Selection sort is a simple sorting algorithm.
- This sorting algorithm is a in-place comparison based algorithm in which the list is divided into two parts, sorted part at left end and unsorted part at right end. Initially sorted part is empty and unsorted part is entire list.



- Smallest element is selected from the unsorted array and swapped with the leftmost element and that element becomes part of sorted array. This process continues moving unsorted array boundary by one element to the right.
- This algorithm is not suitable for large data sets as its average and worst case complexity are of  $O(n^2)$  where n are no. of items.

## How selection sort works?

We take the below depicted array for our example.



For the first position in the sorted list, the whole list is scanned sequentially. The first position where 14 is stored presently, we search the whole list and find that 10 is the lowest value.



So we replace 14 with 10. After one iteration 10, which happens to be the minimum value in the list, appears in the first position of sorted list.



For the second position, where 33 is residing, we start scanning the rest of the list in linear manner.



We find that 14 is the second lowest value in the list and it should appear at the second place. We swap these values.



After two iterations, two least values are positioned at the the beginning in the sorted manner.



The same process is applied on the rest of the items in the array. We shall see an pictorial depiction of entire sorting process –

## Algorithm for selection sort

Step 1 – Set MIN to location 0

Step 2 – Search the minimum element in the list

Step 3 – Swap with value at location MIN

Step 4 – Increment MIN to point to next element

Step 5 – Repeat until list is sorted

Selection sort is a simple sorting algorithm. This sorting algorithm is a inplace comparison based algorithm in which the list is divided into two parts, sorted part at left end and unsorted part at right end. Initially sorted part is empty and unsorted part is entire list.

## Implementation in C

```
#include <stdio.h>
#include <stdbool.h>
#define MAX 7

int intArray[MAX] = {4,6,3,2,1,9,7}; 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
selectionSort(){
int
indexMin,i,j;

    // loop through all numbers
for(i = 0; i < MAX-1; i++){

    // set current element as minimum
indexMin = i;
```

```

        // check the element to be minimum
for(j = i+1;j<MAX;j++){
if(intArray[j] < intArray[indexMin]){
indexMin = j;
    }
    if(indexMin != i){           printf("Items swapped: [ %d, %d ]\n" ,
intArray[i], intArray[indexMin]);

        // swap the numbers      int
temp = intArray[indexMin];
intArray[indexMin] = intArray[i];
intArray[i] = temp;
    }
    printf("Iteration
%d#:",(i+1));    display();
    }
} main(){
printf("Input Array: ");
display();    println(50);
selectionSort();
printf("Output Array: ");
display();    println(50);
}

```

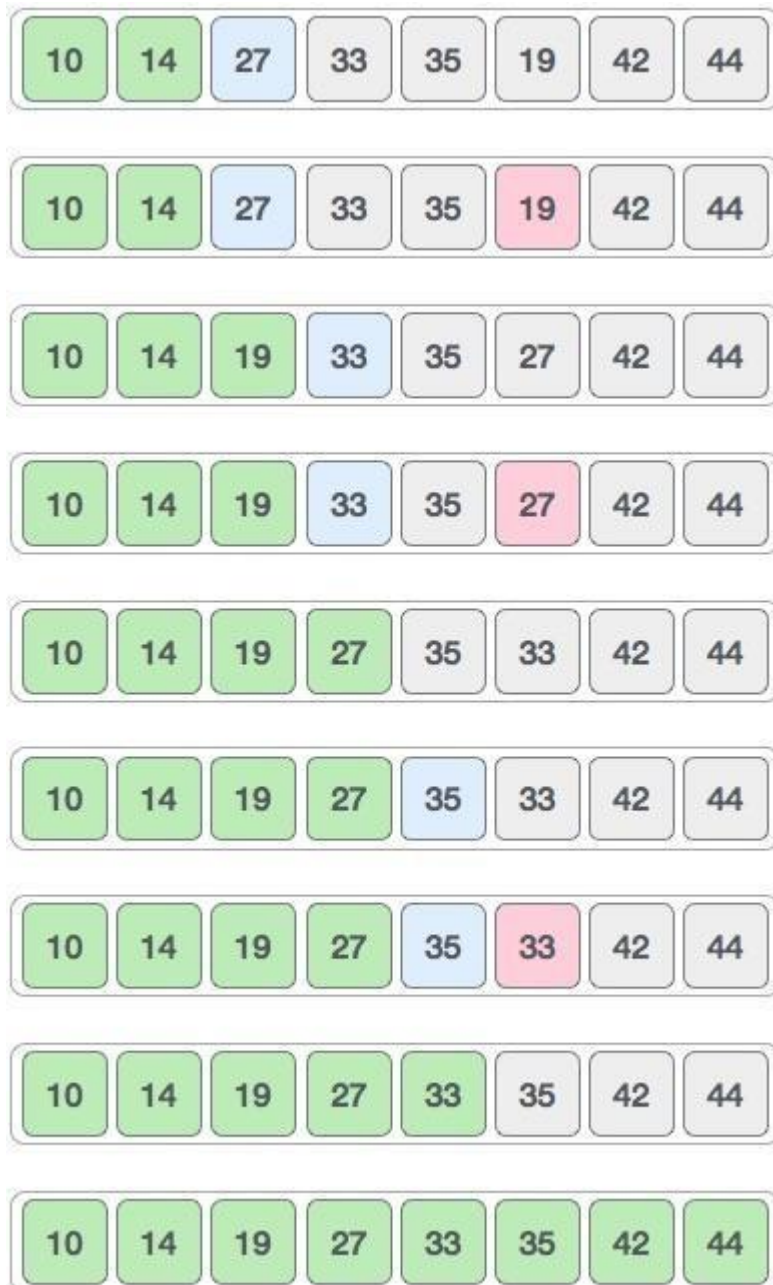
If we compile and run the above program then it would produce following result –

## Output

```
Input Array: [4, 6, 3, 2, 1, 9, 7]
```

```
=====
====      Items swapped: [ 4, 1 ] iteration
```

```
1#: [1, 6, 3, 2, 4, 9, 7]      Items swapped:
[ 6, 2 ] iteration 2#: [1, 2, 3, 6, 4, 9, 7]
iteration 3#: [1, 2, 3, 6, 4, 9, 7]      Items
swapped: [ 6, 4 ] iteration 4#: [1, 2, 3, 4,
6, 9, 7] iteration 5#: [1, 2, 3, 4, 6, 9, 7]
Items swapped: [ 9, 7 ] iteration 6#: [1, 2,
3, 4, 6, 7, 9] Output Array: [1, 2, 3, 4, 6,
7, 9]
=====
=====
```



Now, we should learn some programming aspects of selection sort.

## Algorithm

Step 1 – Set MIN to location 0

Step 2 – Search the minimum element in the list

Step 3 – Swap with value at location MIN

Step 4 – Increment MIN to point to next element

Step 5 – Repeat until list is sorted

Selection sort is a simple sorting algorithm. This sorting algorithm is a inplace comparison based algorithm in which the list is divided into two

parts, sorted part at left end and unsorted part at right end. Initially sorted part is empty and unsorted part is entire list.

## Implementation in C

```
#include <stdio.h>
#include <stdbool.h>
#define MAX 7
int intArray[MAX] =
{4,6,3,2,1,9,7};

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
selectionSort(){
    int
indexMin,i,j;
```

```

        // loop through all numbers
for(i = 0; i < MAX-1; i++){

        // set current element as minimum
indexMin = i;

        // check the element to be minimum
for(j = i+1; j<MAX; j++){
if(intArray[j] < intArray[indexMin]){
indexMin = j;
        }
        } if(indexMin != i){           printf("Items swapped: [ %d, %d
]\n" , intArray[i], intArray[indexMin]);

        // swap the numbers
int temp = intArray[indexMin];
intArray[indexMin] = intArray[i];
intArray[i] = temp;
        }
        printf("Iteration
%d#:",(i+1));           display();
        }
}

```

If we compile and run the above program then it would produce following result –

## Output

Input Array: [4, 6, 3, 2, 1, 9, 7]

```

=====
====      Items swapped: [ 4, 1 ] iteration
1#: [1, 6, 3, 2, 4, 9, 7]      Items swapped:

```



```
[ 6, 2 ] iteration 2#: [1, 2, 3, 6, 4, 9, 7]
iteration 3#: [1, 2, 3, 6, 4, 9, 7]      Items
swapped: [ 6, 4 ] iteration 4#: [1, 2, 3, 4,
6, 9, 7] iteration 5#: [1, 2, 3, 4, 6, 9, 7]
Items swapped: [ 9, 7 ] iteration 6#: [1, 2,
3, 4, 6, 7, 9] Output Array: [1, 2, 3, 4, 6,
7, 9]

=====
=====
```