CHAPTER TWO

Simple Sorting and Searching Algorithms

1. Simple Searching Algorithms

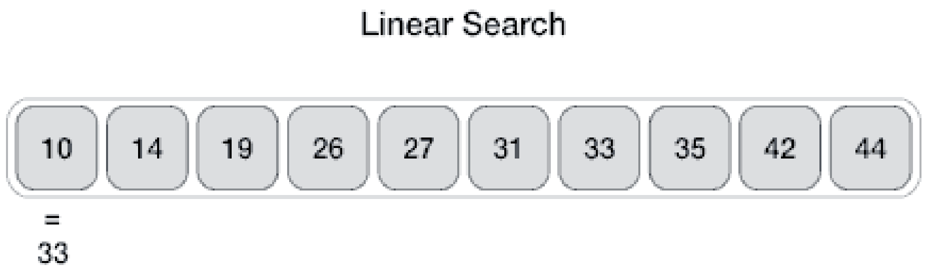
- **Searching is a process of finding an element in a list of items or determining that the item is not in the list.**
- **The process of finding the location of an element within the data structure is called Searching.**
  - **Locate an element x in a list of distinct elements a1,a2,...an or determine that it is not in the list.**
  - **The solution to this search problem is the location of the item in the list that equals x and is 0 if x is not in the list.**
- **There are two simple Searching algorithms:**

  A. **Linear (Sequential) Search**

  B. **Binary Search**

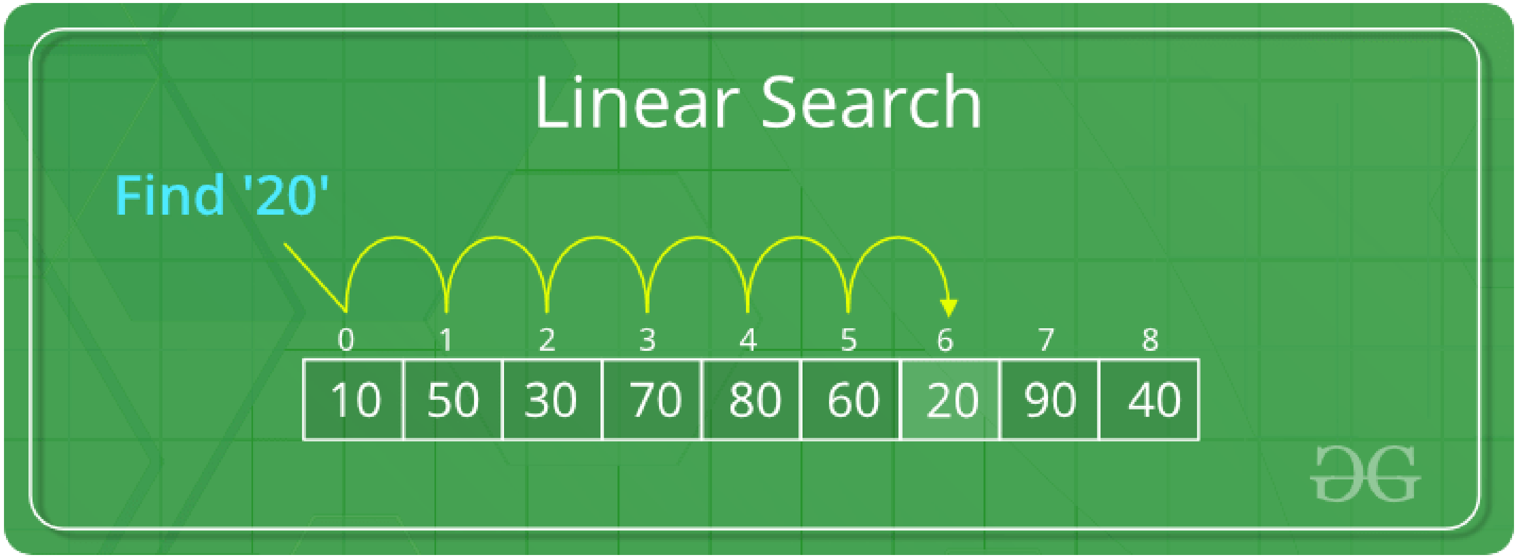
A. Linear Searching

- Linear search is **a very simple** search algorithm.
- This algorithm can be implemented on **the unsorted list**.
- 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 the search continues till the end of the data collection.
  - It compares the element to be searched with all the elements in an array, **if the match is found**, then **it returns the index** of the element else it **returns -1**.



A. Linear Searching: How it works

- In a linear search, we start with top (beginning) of the list and compare the element at top with the key.
- If we have a match, the search terminates, and the index number is returned.
- If not, we go on the next element in the list.
- If we reach the end of the list without finding a match, we return -1.



A. Linear Searching: 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**

A. Linear Searching: Pseudocode

```
procedure linear_search (list, value)
  for each item in the list
    if match item == value
      return the item's location
    else
      return the item is not found
    end if
  end for
end procedure
```

A. Linear Searching: Implementation

```
int linearSearch(int a[], int n, int val)
{
 for (int i = 0; i < n; i++)
   {
     if (a[i] == val)
     return i;
   }
    return -1;
}
```

A. Linear Searching: Complexity Analysis

- **In Linear search,**
  - **the best case occurs when the element we are looking is located at the first position of the array.**
  - **the worst case occurs when the element we are looking is present at the end of the array or not present in the given array.**

Case	Time Complexity
Best Case	O(1)
Average Case	O(n)
Worst Case	O(n)

- **However, the time complexity of linear search is O(n) because every element in the array is compared only once.**

B. Binary Searching

- **A Binary search algorithm is the simplest algorithm that searches the element very quickly.**
- **This search algorithm works on the principle of divide and conquer approach.**
- **It is used to search the element from the sorted list.**
  - **The elements must be stored in sequential order or the sorted manner to implement the binary algorithm.**
  - **Binary search cannot be implemented if the elements are stored in a random manner.**
- **It is used to find the middle element of the list.**

B. Binary Searching: How it works?

- **In a binary search, we look for the key in the middle of the list. *If we get a match, the search is over.***
- **If the key is greater than the element in the middle of the list, we make the top (upper) half the list to search.**
- **If the key is smaller, we make the bottom (lower) half the list to search.**
- **Repeat the above three steps until one element remains.**
- **If this element matches return the index of the element, else return -1 index. (-1 shows that the key is not in the list).**

B. Binary Searching: Algorithm

Binary_Search(a, lower_bound, upper_bound, val)

Step 1: set beg = lower_bound, end = upper_bound, pos = - 1

Step 2: repeat steps 3 and 4 while beg <=end

Step 3: set mid = (beg + end)/2

Step 4: if a[mid] = val

set pos = mid

print pos

go to step 6

else if a[mid] > val

set end = mid - 1

else

set beg = mid + 1

[end of if]

[end of loop]

Step 5: if pos = -1

print "value is not present in the array"

[end of if]

Step 6: exit

- 'a' is the given array
- 'lower_bound' is the index of the first array element
- 'upper_bound' is the index of the last array element
- 'val' is the value to search

B. Binary Searching: Example 1

- **Let the elements of array are**

0	1	2	3	4	5	6	7	8
10	12	24	29	39	40	51	56	69

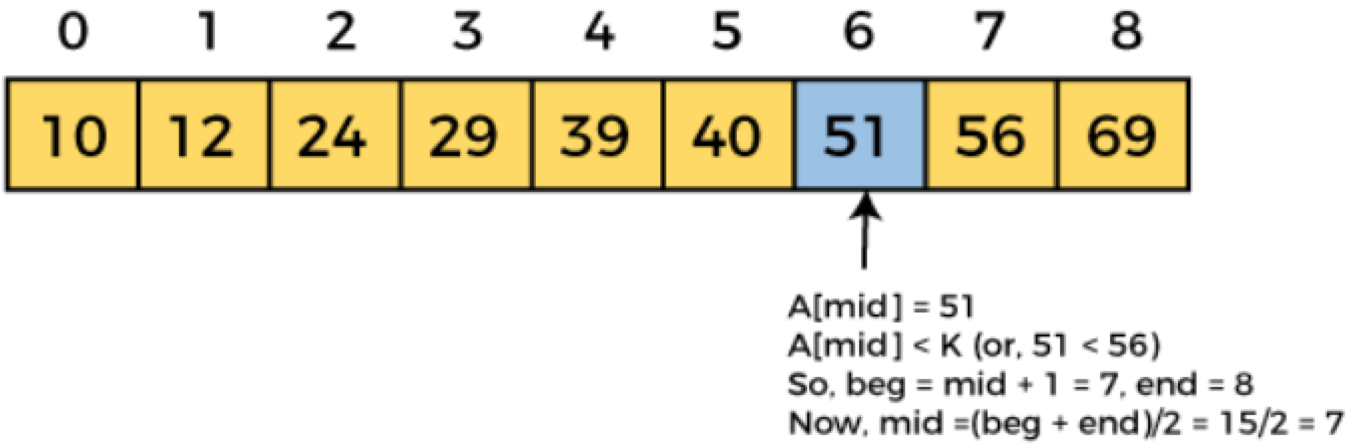
- **Let the element to search is, K = 56**
- **Calculate the mid of the array :- mid = (beg + end)/2**

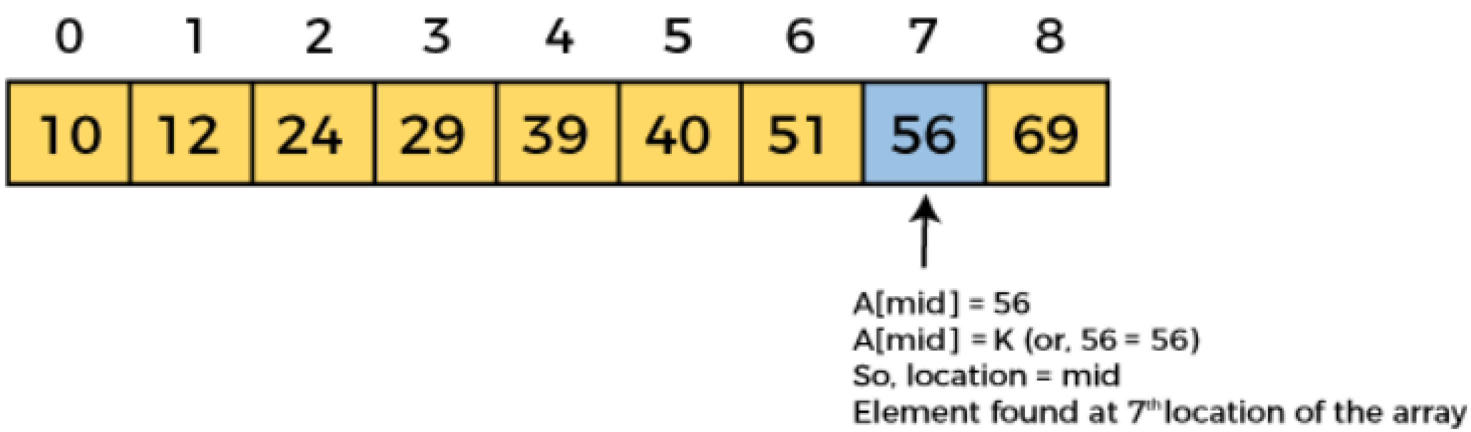
  mid = (0 + 8)/2 = 4. So, 4 is the mid of the array.

0	1	2	3	4	5	6	7	8
10	12	24	29	39	40	51	56	69

A[mid] = 39
A[mid] < K (or,39 < 56)
So, beg = mid + 1 = 5, end = 8
Now, mid =(beg + end)/2 = 13/2 = 6

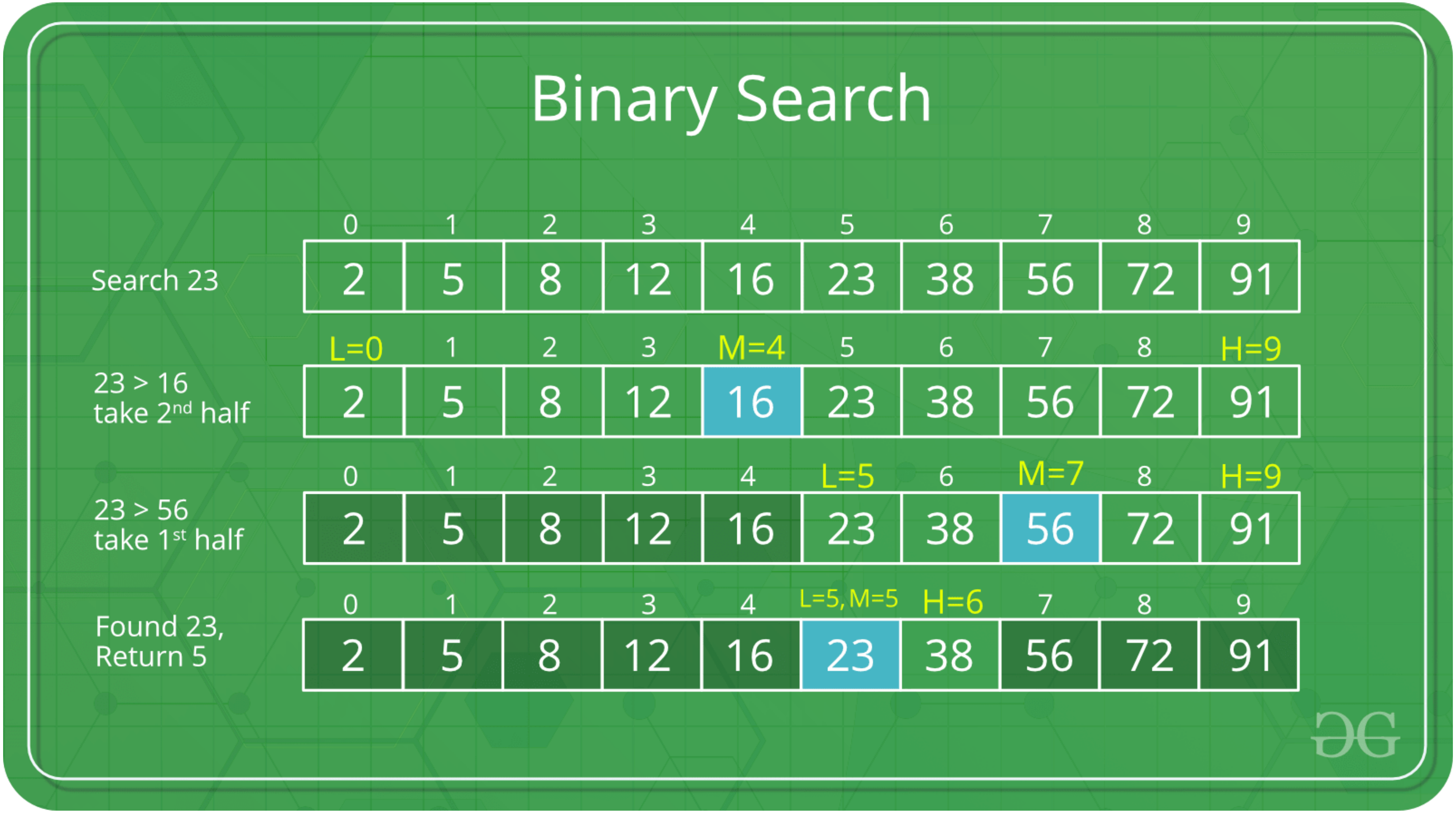
B. Binary Searching: Example 1





- **Now, the element to search is found.**
- **So algorithm will return the index of the element matched.**

B. Binary Searching: Example 2



B. Binary Searching: Implementation

```
int binarySearch(int a[], int beg, int end, int val)   {
   int mid;
   if(end >= beg) {
      mid = (beg + end)/2;
/* if the item to be searched is present at middle */
      if(a[mid] == val) {
         return mid;
      }
/* if the item to be searched is smaller than middle, then it can only be in left subarray */
      else if(a[mid] < val) {
         return binarySearch(a, mid+1, end, val);
      }
 /* if the item to be searched is greater than middle, then it can only be in right subarray */
   else {
         return binarySearch(a, beg, mid-1, val);
      }
   }
   return -1;
}
```

B. Binary Searching: Complexity Analysis

- **In Binary search,**
  - **the best case occurs when the element to search is found in first comparison,**
    - ***i.e., when the first middle element itself is the element to be searched.***
  - **the worst case occurs, when we must keep reducing the search space till it has only one element.**

Case	Time Complexity
Best Case	O(1)
Average Case	O(logn)
Worst Case	O(logn)

- **Therefore, the time complexity of binary search algorithm is O(logn).**

2. Simple Sorting Algorithms

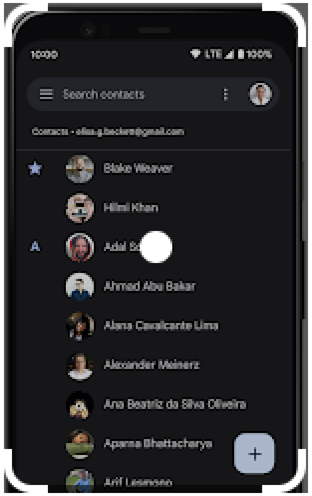
- **Sorting algorithms are used to rearrange the elements in an array or a given data structure either in an ascending or descending order.**
- **It is a process of reordering a list of items in either increasing or decreasing order.**
- **The comparison operator decides the new order of the elements.**

Why do we need a sorting algorithm?

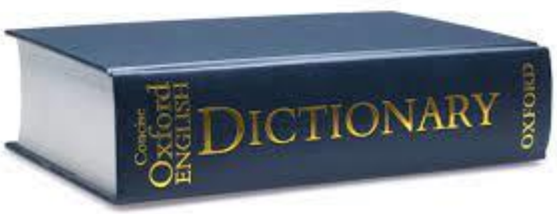
- Sorting is the most important operation performed by computers.
- Sorting is the first step in more complex algorithms.
- An efficient sorting algorithm is required for optimizing the efficiency of other algorithms like binary search algorithm.
  - a binary search algorithm requires an array to be sorted in a particular order, mainly in ascending order.
- It produces information in a sorted order, which is a human-readable format.
- Searching a particular element in a sorted list is faster than the unsorted list.

Example: Sorting in real-life scenarios

- Telephone Directory
  - The telephone directory stores the telephone numbers of people sorted by their names, so that the names can be searched easily.



- Dictionary
  - The dictionary stores words in an alphabetical order so that searching of any word becomes easy.



Properties of sorting algorithms

A. **In-place Sorting and Not in-place Sorting**
B. **Stable and Not Stable Sorting**
C. **Adaptive and Non-Adaptive Sorting**

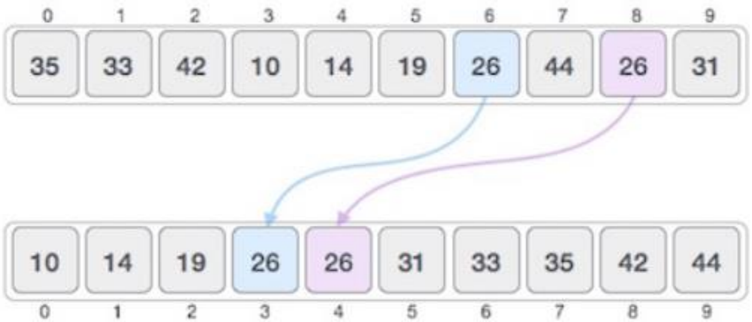
A. In-place Sorting vs Not-in-place Sorting

- **Sorting algorithms may require some extra space for comparison and temporary storage of few data elements.**
- **In-place Sorting:**
  - **These algorithms do not require any extra space and sorting is said to happen in-place, or for example, within the array itself.**
  - **Example: Bubble sort algorithm**

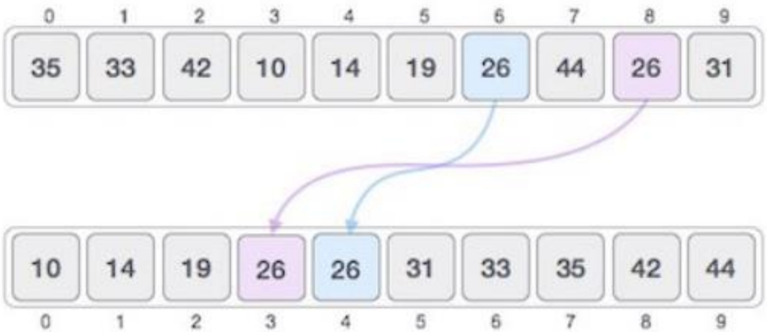
- **Not in-place Sorting:**
  - **These algorithms require extra space which is more than or equal to the elements being sorted.**
  - **Example: Merge-sort algorithm**

B. Stable vs Not Stable Sorting

- **Stable Sorting:**
  - **If a sorting algorithm, after sorting the contents, does not change the sequence of similar content in which they appear, it is called stable sorting.**



- **Not-Stable Sorting:**
  - **If a sorting algorithm, after sorting the contents, changes the sequence of similar content in which they appear, it is called unstable sorting.**



C. Adaptive vs Non-Adaptive Sorting

- **Adaptive Sorting:**
  - **A sorting algorithm is said to be adaptive, if it takes advantage of already 'sorted' elements in the list that is to be sorted.**
    - **i.e., while sorting if the source list has some element already sorted, adaptive algorithms will take this into account and will not try to re-order them.**

- **Non-Adaptive Sorting:**
  - **A non-adaptive algorithm is one which does not take into account the elements which are already sorted.**
  - **They try to force every single element to be re-ordered to confirm their sortedness.**

Simple Sorting Algorithms

- **Simple sorting algorithms include:**

  A. **Bubble Sorting**

  B. **Selection Sorting**

  C. **Insertion Sorting**

A. Bubble Sort

- **It is a simple sorting algorithm that compares two adjacent elements and swaps them if they are not in the intended order.**
  - **It works on the repeatedly swapping of adjacent elements if they are not in the intended order.**
- **It is not suitable for large data sets as its average and worst-case complexity are of $O(n^2)$, where n is a number of items.**
- **Bubble sort is majorly used where -**
  - **complexity does not matter**
  - **simple and short code is preferred**

A. Bubble Sort: How it works?

1. **Compare each element (except the last one) with its neighbor to the right.**
   - **If they are out of order, swap them**
   - **This puts the largest element at the very end**
   - **The last element is now in the correct and final place**
2. **Compare each element (except the last two) with its neighbor to the right.**
   - **If they are out of order, swap them**
   - **This puts the second largest element before last**
   - **The last two elements are now in their correct and final places**
3. **Continue as above until you have no unsorted elements on the left.**

A. Bubble Sort: Algorithm

- **In the algorithm given below, suppose arr is an array of n elements. The assumed swap function in the algorithm will swap the values of given array elements.**

```
begin BubbleSort(arr)
  for all array elements
    if arr[i] > arr[i+1]
      swap(arr[i], arr[i+1])
    end if
  end for
  return arr
end BubbleSort
```

A. Bubble Sort: Example 1

- **Let the elements of array are -**

13	32	26	35	10

First Iteration:

- **Sorting will start from the initial two elements. Let compare them to check which is greater.**

13	32	26	35	10

- **Here, 32 is greater than 13 (32 > 13), so it is already sorted. Now, compare 32 with 26.**

13	32	26	35	10

- **Here, 26 is smaller than 36. So, swapping is required. After swapping new array will look like –**

13	26	32	35	10

- **Now, compare 32 and 35.**

13	26	32	35	10

A. Bubble Sort: Example 1

- **Here, 35 is greater than 32. So, there is no swapping required as they are already sorted.**
- **Now, the comparison will be in between 35 and 10.**

13	26	32	35	10

- **Here, 10 is smaller than 35 that are not sorted. So, swapping is required. Now, we reach at the end of the array. After first pass, the array will be –**

13	26	32	10	35

- **Now, move to the second iteration.**

A. Bubble Sort: Example 1

Second Iteration:

- **The same process will be followed for second iteration.**

13	26	32	10	35

13	26	32	10	35

13	26	32	10	35

- **Here, 10 is smaller than 32. So, swapping is required. After swapping, the array will be -**

13	26	10	32	35

13	26	10	32	35

- **Now, move to the third iteration.**

A. Bubble Sort: Example 1

Third Iteration:

- **The same process will be followed for third iteration.**

13	26	10	32	35

13	26	10	32	35

- **Here, 10 is smaller than 26. So, swapping is required. After swapping, the array will be -**

13	10	26	32	35

13	10	26	32	35

13	10	26	32	35

- **Now, move to the fourth iteration.**

A. Bubble Sort: Example 1

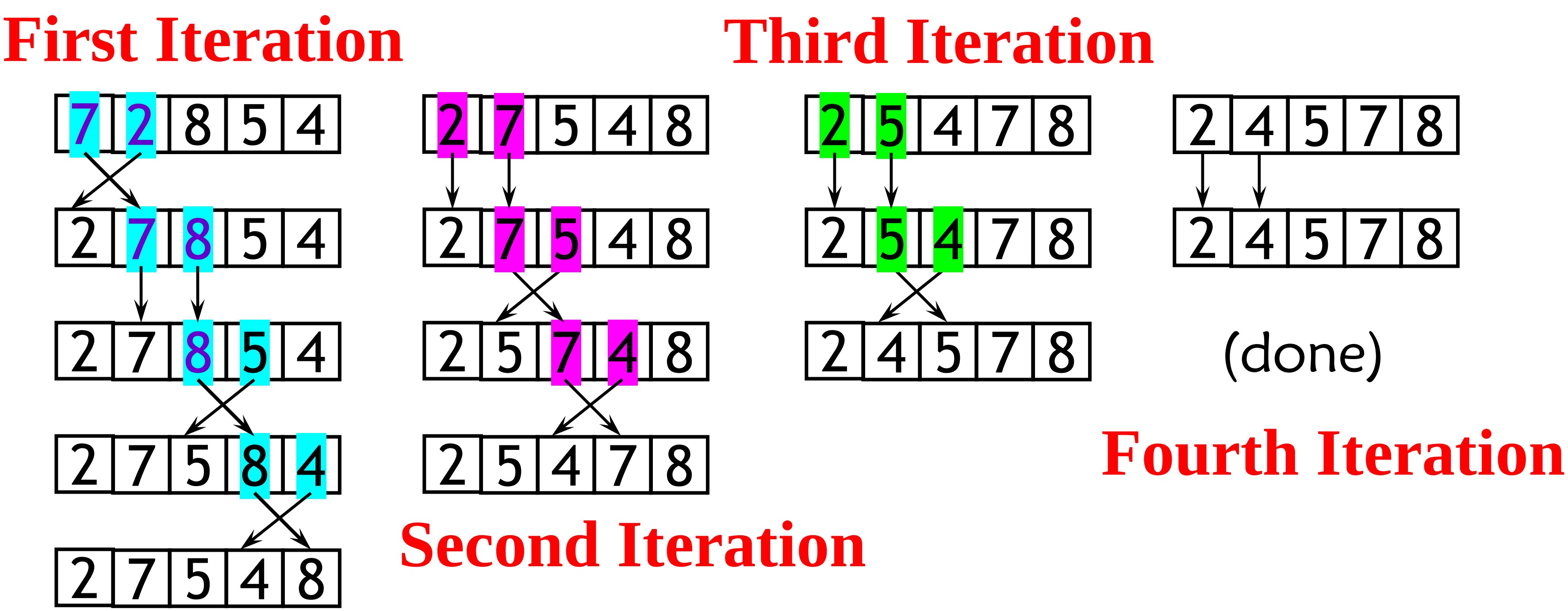
Fourth Iteration:

- **Similarly, after the fourth iteration, the array will be -**

10	13	26	32	35

- **Hence, there is no swapping required, so the array is completely sorted.**

A. Bubble Sort: Example 2



A. Bubble Sort: Implementation

```
void bubble(int a[], int n) {
int i, j, temp;
  for(i = 0; i < n; i++)
   {
     for(j = i+1; j < n; j++)
       {
           if(a[j] < a[i])
           {
               temp = a[i];
               a[i] = a[j];
               a[j] = temp;
           }
       }
   }
}
```

```
int main()
{
    int i, j,temp;
    int a[5] = {45, 1, 32, 13, 26};
    int n = sizeof(a)/sizeof(a[0]);
    cout<<"Before sorting:- \n";
    print(a, n);
    bubble(a, n);
    cout<<"\nAfter sorting:- \n";
    print(a, n);
return 0;
}
```

A. Bubble Sort: Complexity Analysis

- In bubble sort algorithm,
  - the best case time complexity occurs when there is no sorting required
    - i.e. the array is already sorted.
  - the worst case time complexity occurs when the array elements are required to be sorted in reverse order.

Case	Time Complexity
Best Case	$O(n)$
Average Case	$O(n^2)$
Worst Case	$O(n^2)$

- Therefore, the time complexity of a bubble sort algorithm is $O(n^2)$.

B. Selection Sort

- **Selection sort is a sorting algorithm that selects the smallest element from an unsorted list in each iteration and places that element at the beginning of the unsorted list.**
  - **i.e., the smallest value among the unsorted elements of the array is selected in every pass and inserted to its appropriate position into the array.**
- **It is the simplest algorithm.**
- **It is an in-place comparison sorting algorithm.**

B. Selection Sort: How it works?

- **In this algorithm, the array is divided into two parts: sorted part and unsorted part.**
  - **Initially, the sorted part of the array is empty, and unsorted part is the given array.**
  - **Sorted part is placed at the left, while the unsorted part is placed at the right.**
- **In selection sort, the first smallest element is selected from the unsorted array and placed at the first position.**
- **After that second smallest element is selected and placed in the second position.**
- **The process continues until the array is entirely sorted.**

B. Selection Sort: Algorithm

```
SELECTION SORT(arr, n)
    Step 1: Repeat Steps 2 and 3 for i = 0 to n-1
    Step 2: CALL SMALLEST(arr, i, n, pos)
    Step 3: SWAP arr[i] with arr[pos]
    [END OF LOOP]
    Step 4: EXIT
```

```
SMALLEST (arr, i, n, pos)
    Step 1: [INITIALIZE] SET SMALL = arr[i]
    Step 2: [INITIALIZE] SET pos = i
    Step 3: Repeat for j = i+1 to n
    if (SMALL > arr[j])
        SET SMALL = arr[j]
    SET pos = j
    [END OF if]
    [END OF LOOP]
    Step 4: RETURN pos
```

B. Selection Sort: Example 1

- Selection sort is generally used when
  - A small array is to be sorted
  - Swapping cost doesn't matter
  - It is compulsory to check all elements

Example 1:

- Let the elements of array are –

12	29	25	8	32	17	40

- 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.

12	29	25	8	32	17	40

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

8	29	25	12	32	17	40

B. Selection Sort: Example 1

- 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.

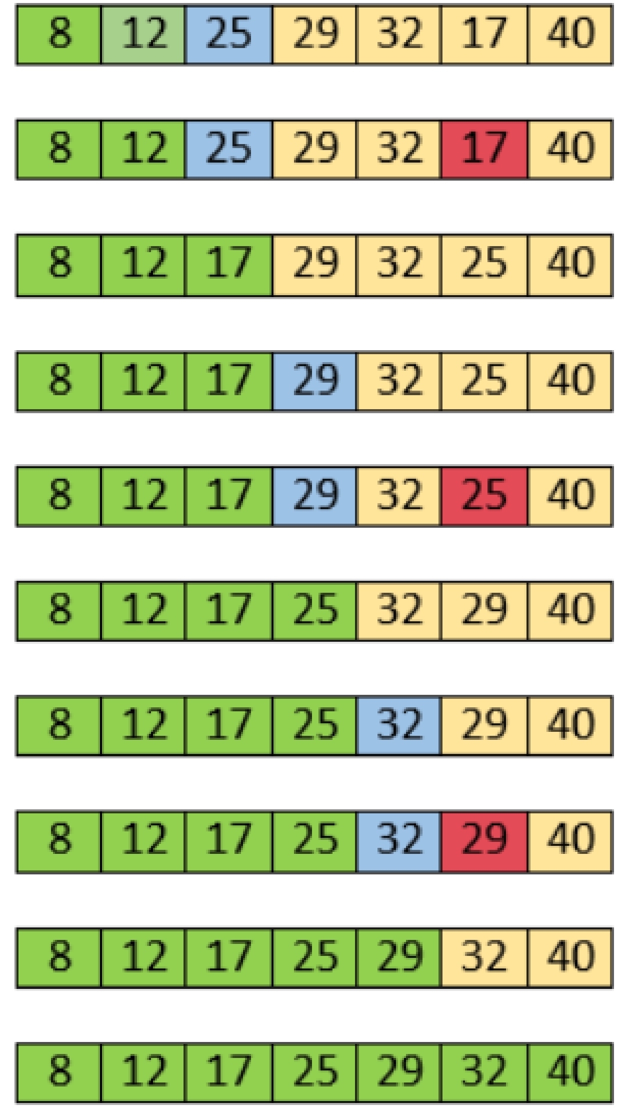
8	29	25	12	32	17	40

- 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.

8	12	25	29	32	17	40

B. Selection Sort: Example 1

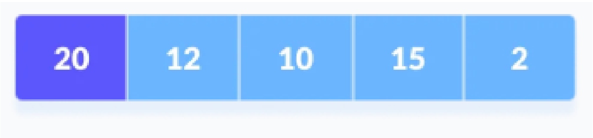
- **The same process is applied to the rest of the array elements.**
- **Now, we are showing a pictorial representation of the entire sorting process.**



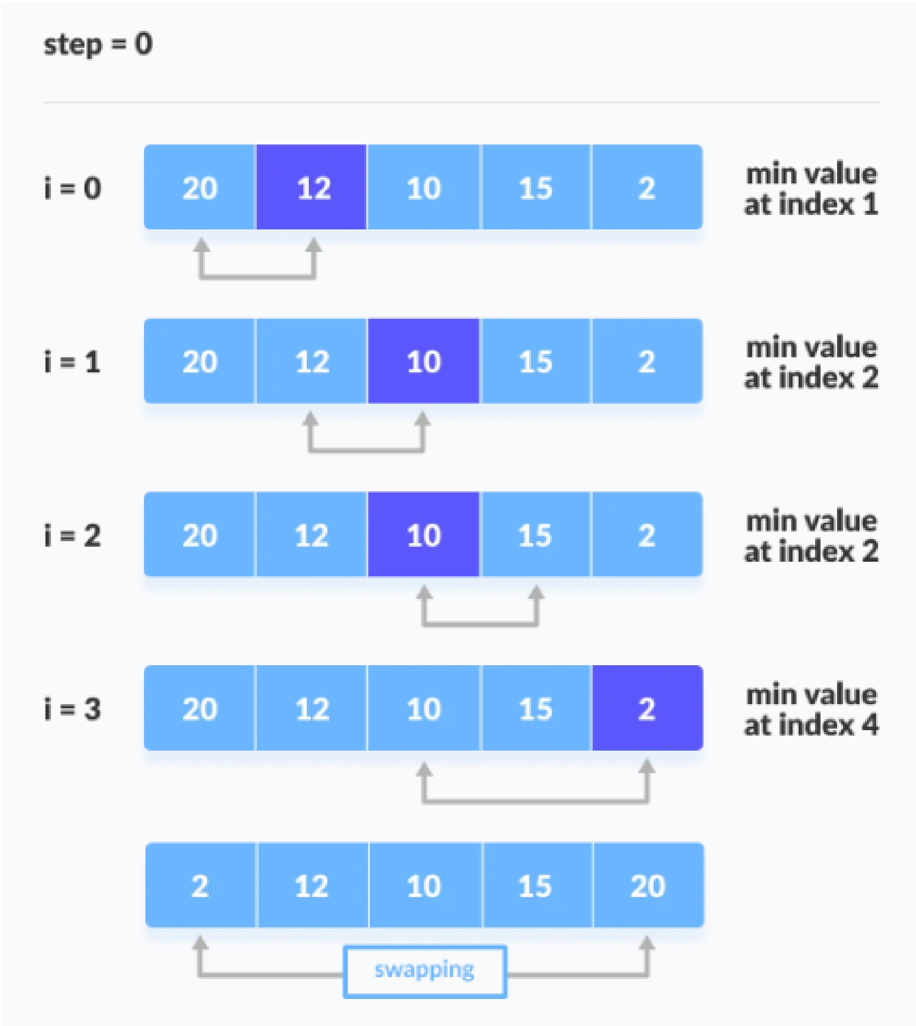
- **Now, the array is completely sorted.**

B. Selection Sort: Example 2

- Let the elements of array are –

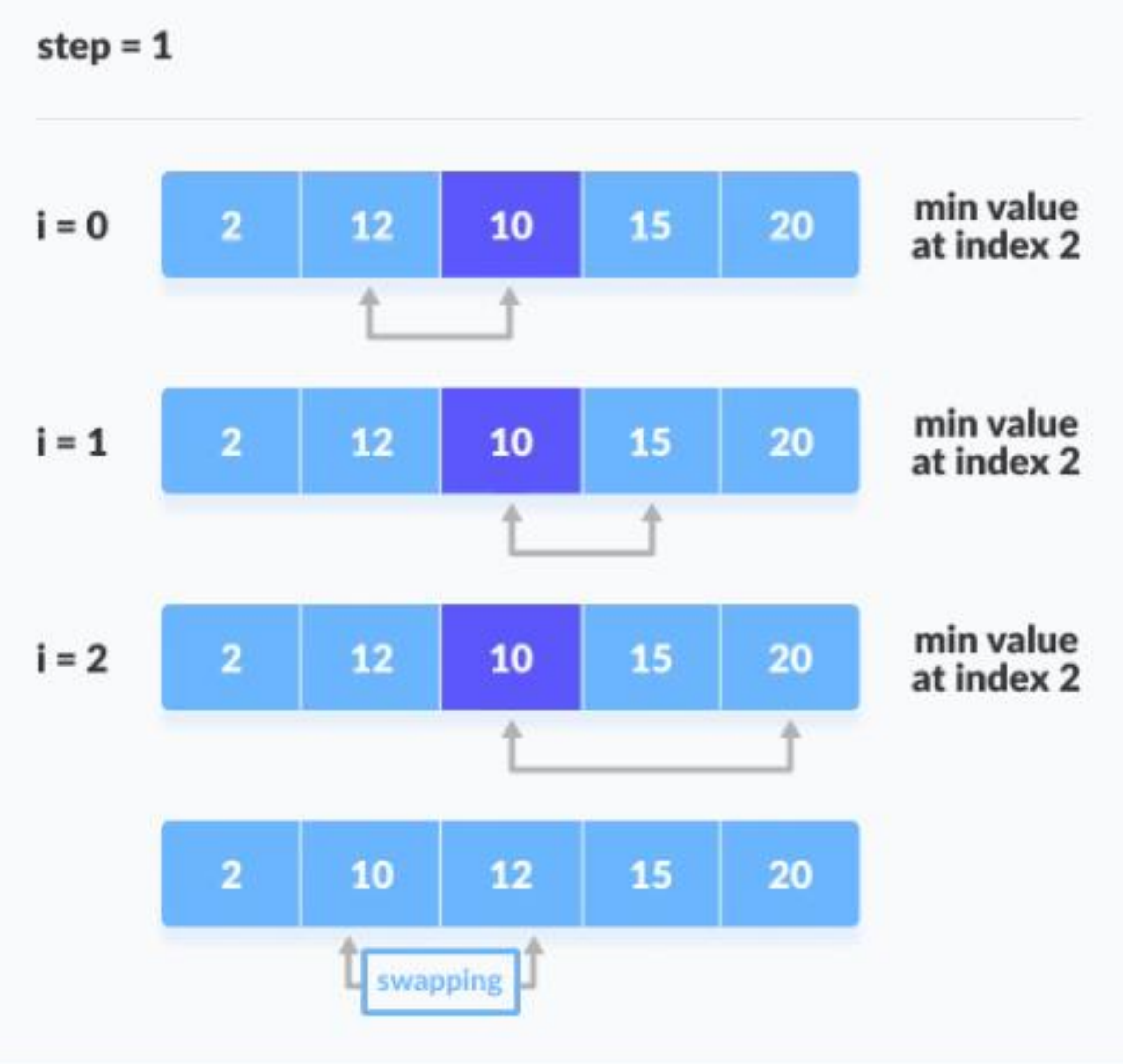


Select first element as minimum



The first iteration

B. Selection Sort: Example 2



The second iteration

B. Selection Sort: Example 2

The third iteration

The fourth iteration

B. Selection Sort: Implementation

```
void selection(int arr[], int n)
{
   int i, j, small;
   for (i = 0; i < n-1; i++)// One by one move boundary of unsorted subarray
   {
      small = i; //minimum element in unsorted array

      for (j = i+1; j < n; j++)
      if (arr[j] < arr[small])
         small = j;
// Swap the minimum element with the first element
   int temp = arr[small];
   arr[small] = arr[i];
   arr[i] = temp;
   }
}
```

```
int main()
{
   int a[] = { 80, 10, 29, 11, 8, 30, 15 };
   int n = sizeof(a) / sizeof(a[0]);
   cout<< "before sorted:- "<<endl;
   printArr(a, n);
   selection(a, n);
   cout<< "after sorted"<<endl;
   printArr(a, n);

   return 0;
}
```

B. Selection Sort: Complexity Analysis

- **In Selection sort algorithm,**
  - **the best case time complexity occurs when there is no sorting required,**
    - **i.e. the array is already sorted.**
  - **the worst case time complexity 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.**

Case	Time Complexity
Best Case	$O(n^2)$
Average Case	$O(n^2)$
Worst Case	$O(n^2)$

- **Overall algorithm complexity is $O(n^2)$**

C. Insertion Sort

- Insertion sort is a sorting algorithm that places an unsorted element at its suitable place in each iteration.
- The idea behind the insertion sort is that first take one element, iterate it through the sorted array.
- It is simple to use.
- It is not appropriate for large data sets as the time complexity of insertion sort in the average case and worst case is $O(n^2)$, where n is the number of items.
- Insertion sort is less efficient than the other sorting algorithms like heap sort, quick sort, merge sort, etc.

C. Insertion Sort: How it works?

- Insertion sort algorithm somewhat resembles Selection Sort and Bubble sort.
- Array is imaginary divided into two parts - sorted and unsorted.
- At the beginning, sorted part contains first element of the array and unsorted one contains the rest.
- At every step, algorithm takes first element in the unsorted part and inserts it to the right place of the sorted one.
- When unsorted part becomes empty, algorithm stops.
- It is reasonable to use binary search algorithm to find a proper place for insertion.
- Insertion sort works by inserting item into its proper place in the list.

C. Insertion Sort: How it works?

- **Insertion sort works similarly as we sort cards in our hand in a card game.**
  - **We assume that the first card is already sorted then, we select an unsorted card.**
  - **If the unsorted card is greater than the card in hand, it is placed on the right otherwise, to the left.**
  - **In the same way, other unsorted cards are taken and put in their right place.**
  - **This process is repeated until all the cards are in the correct sequence.**
- **A similar approach is used by insertion sort.**
- **Insertion sort is over twice as fast as the bubble sort and is just as easy to implement as the selection sort.**

C. Insertion Sort: Algorithm

Step 1 - If the element is the first element, assume that it is already sorted. Return 1.

Step 2 - Pick the next element and store it separately in a key.

Step 3 - Now, compare the key with all elements in the sorted array.

Step 4 - If the element in the sorted array is smaller than the current element, then move to the next element. Else, shift greater elements in the array towards the right.

Step 5 - Insert the value.

Step 6 - Repeat until the array is sorted.

C. Insertion Sort: Example 1

- Let the elements of array are –

12	31	25	8	32	17

- Initially, the first two elements are compared in insertion sort.

12	31	25	8	32	17

- Here, 31 is greater than 12.
  - That means both elements are already in ascending order.
  - So, for now, 12 is stored in a sorted sub-array.

12	31	25	8	32	17

C. Insertion Sort: Example 1

- **Now, two elements in the sorted array are 12 and 25. Move forward to the next elements that are 31 and 8.**

12	25	31	8	32	17

12	25	31	8	32	17

- **Both 31 and 8 are not sorted. So, swap them.**

12	25	8	31	32	17

- **After swapping, elements 25 and 8 are unsorted.**

12	25	8	31	32	17

- **So, swap them.**

12	8	25	31	32	17

C. Insertion Sort: Example 1

- **Now, elements 12 and 8 are unsorted.**

12	8	25	31	32	17

- **So, swap them too.**

8	12	25	31	32	17

- **Now, the sorted array has three items that are 8, 12 and 25. Move to the next items that are 31 and 32.**

8	12	25	31	32	17

- **Hence, they are already sorted. Now, the sorted array includes 8, 12, 25 and 31.**

8	12	25	31	32	17

C. Insertion Sort: Example 1

- **Move to the next elements that are 32 and 17.**

8	12	25	31	32	17

- **17 is smaller than 32. So, swap them.**

8	12	25	31	17	32

8	12	25	31	17	32

- **Swapping makes 31 and 17 unsorted. So, swap them too.**

8	12	25	17	31	32

8	12	25	17	31	32

- **Now, swapping makes 25 and 17 unsorted. So, perform swapping again.**

8	12	17	25	31	32

- **Now, the array is completely sorted.**

C. Insertion Sort: Implementation

```
void insert(int a[], int n) /* function to sort an array with insertion sort */
{
   int i, j, temp;
   for (i = 1; i < n; i++) {
      temp = a[i];
      j = i - 1;

      while(j>=0 && temp <= a[j])  /* Move the elements greater than temp to
one position ahead from their current position*/
      {
         a[j+1] = a[j];
         j = j-1;
      }
      a[j+1] = temp;
   }
}
```

```
int main() {
   int a[] = { 89, 45, 35, 8, 12, 2 };
   int n = sizeof(a) / sizeof(a[0]);
   cout<<"Before sorting: "<<endl;
   printArr(a, n);
   insert(a, n);
   cout<<"\nAfter sorting: "<<endl;
   printArr(a, n);
     return 0;
}
```

C. Insertion Sort: Complexity Analysis

- **In Insertion sort algorithm,**
  - **the best case time complexity occurs when there is no sorting required,**
    - **i.e. the array is already sorted.**
  - **the worst case time complexity 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.**

Case	Time Complexity
Best Case	$O(n)$
Average Case	$O(n^2)$
Worst Case	$O(n^2)$

- **Overall algorithm complexity is $O(n^2)$**

Thank You

Question?