SREENIVASA INSTITUTE OF TECHNOLOGY AND MANAGEMENT STUDIES
(Autonomous)

Subject: Data Structures
I B.Tech, II Sem
Unit 1 Question bank

Short Answer Questions:

1. Define data structure.
Ans. Data structures are the fundamental building blocks of computer programming. They define how data is organized, stored, and manipulated within a program. Understanding data structures is very important for developing efficient and effective algorithms. It is a way of arranging data on a computer so that it can be accessed and updated efficiently. It is also used for processing, retrieving, and storing data. Examples include arrays, linked lists, stacks, and trees.

2. What are the different types of data structures?
Ans. There are two different types of Data Structures: 1. Linear Data Structure, 2. Non-Linear Data Structure.
Linear Data Structure: Elements are arranged in one dimension, also known as linear dimension. Example: lists, stack, queue, etc.
Non-Linear Data Structure: Elements are arranged in one-many, many-one and many-many dimensions. Example: tree, graph, table, etc.

3. Differentiate linear and nonlinear data structures.
Ans. Linear Data Structure:
Data structure where data elements are arranged sequentially or linearly where each and every element is attached to its previous and next adjacent is called a **linear data structure**. In linear data structure, single level is involved. Therefore, we can traverse all the elements in single run only. Linear data structures are easy to implement because computer memory is arranged in a linear way. Its examples are array, stack, queue, linked list, etc.
Non-linear Data Structure:
Data structures where data elements are not arranged sequentially or linearly are called **non-linear data structures**. In a non-linear data structure, single level is not involved. Therefore, we can't traverse all the elements in single run only. Non-linear data structures are not easy to implement in comparison to linear data structure. It utilizes computer memory efficiently in comparison to a linear data structure. Its examples are trees and graphs.

4. List the various operations that can be performed on data structure.
Ans. Various operations that can be performed on the data structure are
- Create
- Insertion of element
- Deletion of element
- Searching for the desired element
- Sorting the elements in the data structure
- Reversing the list of elements
- Copying the elements
- Merging two data structures.

5. Describe abstract data type with example.
Ans. An Abstract Data Type (ADT) is a conceptual model that defines a set of operations and behaviors for a data structure, without specifying how these operations are implemented or how data is organized in memory. The definition of ADT only mentions what operations are to be performed but not how these operations will be implemented. It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations. It is called "abstract" because it provides an implementation-independent view. Examples of ADTs: 1. Stacks: A linear data structure that follows the LIFO (Last In, First Out) principle. Stacks are commonly implemented using arrays or linked lists.
2. Queues: A linear data structure that allows data to be accessed from both ends, the front and the rear. Queues are a natural component of many simulation models of real processes. Other Examples: List, Map, Set, Table, Tree, and Vector.

6. List out the areas in which data structures are applied extensively.

Ans. Data structures are applied extensively in a wide array of computer science areas, including but not limited to compiler design, operating systems, database management, artificial intelligence, and more.

1. Compiler Design:
   Data structures like parse trees, symbol tables, and abstract syntax trees are vital for efficiently translating source code into executable form.
2. Operating Systems:
   Data structures such as queues, stacks, and priority queues are used for managing processes, memory, and scheduling tasks within an operating system.
3. Database Management Systems:
   Trees (like B-trees), hash tables, and various indexing techniques are crucial for efficient data storage, retrieval, and manipulation within database systems.
4. Algorithms:
   Efficient data structures are essential for designing and implementing fast algorithms, such as those for searching, sorting, and graph traversal.
5. Artificial Intelligence (AI):
   Data structures like decision trees, graphs, and knowledge bases play a vital role in AI algorithms, enabling machines to learn, make decisions, and solve complex problems.

6. What are the advantages of an ADT?

Ans. Advantages of ADT

- Encapsulation: ADTs help encapsulate data and operations into a single unit, making managing and modifying the data structure easier.
- Abstraction: You can work with ADTs without knowing the implementation details, which can simplify programming and reduce errors.
- Data Structure Independence: ADTs can be implemented using different data structures, to make it easier to adapt to changing needs and requirements.
- Information Hiding: ADTs protect data integrity by controlling access and preventing unauthorized modifications.
- Modularity: ADTs can be combined with other ADTs to form more complex data structures increasing flexibility and modularity in programming.

7. What is time complexity of an algorithm?

Ans. Time Complexity: The time complexity of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the length of the input. Note that the time to run is a function of the length of the input and not the actual execution time of the machine on which the algorithm is running on.

The valid algorithm takes a finite amount of time for execution. The time required by the algorithm to solve given problem is called ***time complexity*** of the algorithm. Time complexity is very useful measure in algorithm analysis.

It is the time needed for the completion of an algorithm. To estimate the time complexity, we need to consider the time required for each fundamental instruction and the number of times the instruction is executed.

8. What is space complexity of an algorithm?

Ans. Space complexity is a measure of the amount of memory an algorithm uses, expressed in terms of the size of the input. It refers to the amount of memory storage required to execute the algorithm and solve a problem. A low space complexity means that an algorithm requires relatively little memory to solve the problem, while a high space complexity means that it requires a large amount of memory, potentially leading to slow performance or memory limitations.

9. Why is searching required? What are the different searching techniques used?

Ans. Searching is required to locate specific data within a collection, and efficient techniques like linear search, binary search, and hashing are crucial for fast and reliable retrieval. Searching algorithms are essential for solving various problems, including finding information in databases, searching for specific items in a list, and locating relevant web pages.

10. What are the drawbacks of using static memory allocation, and how does dynamic memory allocation overcome those limitations?

Ans. Drawbacks of Static Memory Allocation:

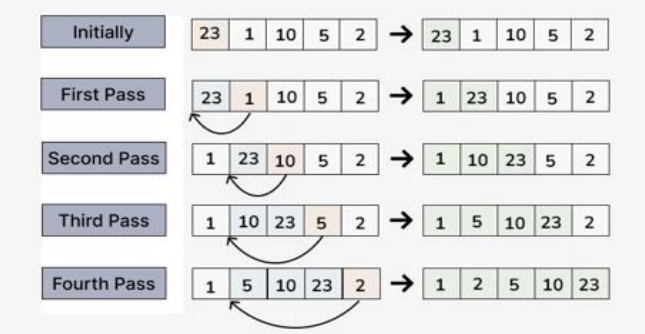
Fixed Size: In static memory allocation, the amount of memory a variable or data structure occupies is

determined at compile time and cannot be changed during runtime.
Potential for Wastage: If you allocate more memory than needed, you waste space. Conversely, if you allocate too little, you'll run into problems when your program needs more memory.
Lack of Flexibility: Static allocation is not suitable for applications where memory requirements can vary during program execution.
Compile-Time Overhead: The compiler needs to know the size of all data structures upfront, which can make the compilation process slower and less flexible.
Dynamic Memory Allocation Overcomes These Limitations:
Runtime Allocation: Dynamic memory allocation allows you to allocate memory during program execution, meaning you can request memory as needed.
Flexibility: You can allocate and deallocate memory as your program's requirements change, making dynamic allocation suitable for applications with variable memory needs.
Efficient Memory Usage: You can allocate only the amount of memory you need, reducing wastage.
Data Structures like Linked Lists: Dynamic memory allocation is essential for implementing data structures like linked lists, where the size of the data structure can change during runtime.
Dynamic memory allocation is typically managed using functions like malloc(), calloc(), realloc() and free().

11. What is the process of insertion sort, and how is it used to sort a collection of data elements in ascending or descending order?

Ans. Insertion sort is a simple sorting algorithm that works by iteratively inserting each element of an unsorted list into its correct position in a sorted portion of the list. It is like sorting playing cards in your hands. You split the cards into two groups: the sorted cards and the unsorted cards. Then, you pick a card from the unsorted group and put it in the right place in the sorted group.

- We start with second element of the array as first element in the array is assumed to be sorted.
- Compare second element with the first element and check if the second element is smaller then swap them.
- Move to the third element and compare it with the first two elements and put at its correct position
- Repeat until the entire array is sorted.



12. What is the process of linear search in programming, and how is it used to search for a specific element in a collection of data?

Ans. Linear search in programming is a straightforward algorithm that sequentially checks each element in a collection (like an array or list) until a match for the target element is found or the end of the collection is reached.

Search Process:

- The search starts at the beginning of the collection.
- It compares each element in the collection with the target element you are searching for.
- If a match is found, the search stops, and the index (position) of the matched element is returned.
- If the end of the collection is reached without finding a match, the search returns a value indicating that the target element was not found (e.g., -1).

13. What is the process of binary search in programming, and how is it used to search for a specific element in a collection of data?

Ans. Binary search is an efficient algorithm for finding an element in a sorted collection (like an array) by repeatedly dividing the search interval in half, comparing the target value with the middle element, and narrowing the search space until the element is found or the interval is empty.

explanation:

1. Sorted Collection: Binary search requires the data to be sorted in ascending or descending order.
2. Identify the Middle Element: Determine the middle element of the current search interval (initially the entire array).
3. Compare: Compare the target value with the middle element:

If the target value matches the middle element: The search is successful, and the index of the middle element is returned.

If the target value is less than the middle element: The search continues in the left half of the array (excluding the middle element).

If the target value is greater than the middle element: The search continues in the right half of the array (excluding the middle element).

4. Repeat: Repeat steps 2 and 3 with the new search interval until the target value is found or the search interval becomes empty.
5. Not Found: If the search interval becomes empty without finding the target value, the element is not present in the collection, and a "not found" indicator (e.g., -1) is returned.

14. What is the process of selection sort, and how is it used to sort a collection of data elements in ascending or descending order?

Ans. Selection sort is a simple comparison-based sorting algorithm that sorts a collection of data elements by repeatedly finding the minimum element (for ascending order) or maximum element (for descending order) from the unsorted part and swapping it with the first element of the unsorted part.

explanation:

- Divide and Conquer: Selection sort conceptually divides the input list into two parts: a sorted portion and an unsorted portion.
- Find the Minimum/Maximum: In each iteration, it finds the minimum (ascending) or maximum (descending) element within the unsorted portion.
- Swap: This minimum/maximum element is then swapped with the first element of the unsorted portion, effectively extending the sorted portion by one element.
- Repeat: This process is repeated until the entire list is sorted.

15. What is the process of bubble sort, and how is it used to sort a collection of data elements in ascending or descending order?

Ans. Bubble sort is a simple sorting algorithm that works by repeatedly stepping through the list, comparing adjacent elements and swapping them if they are in the wrong order, until the list is sorted. This process can be used to sort data elements in either ascending or descending order.

Here's a more detailed explanation:

Process of Bubble Sort:

Comparison: The algorithm starts by comparing the first two elements in the list.

Swapping (if needed): If the elements are in the wrong order (e.g., larger element before smaller element for ascending order), they are swapped.

Iteration: The algorithm then moves to the next pair of adjacent elements and repeats the comparison and swapping process.

Pass: This process continues until the end of the list, completing one "pass".

Repeat: The algorithm then repeats the passes until no more swaps are needed, indicating that the list is sorted.

Ascending/Descending Order: The direction of the comparison (swapping if the left element is greater than the right for ascending or the reverse for descending) determines whether the list is sorted in ascending or descending order.

16. What do you mean by sorting? Mention the types of sorting.

Ans. Sorting is a fundamental operation in computer science that involves arranging a set of data in a specific order, such as numerical or alphabetical order.

This ordered arrangement can be either ascending (smallest to largest) or descending (largest to smallest). Sorting is crucial for many algorithms and data structures, as it simplifies searching and other operations.

Types of Sorting Algorithms:
Comparison-based sorting: These algorithms compare elements to determine their relative order.
Bubble Sort: Iteratively compares adjacent elements and swaps them if they are in the wrong order.
Selection Sort: Finds the minimum (or maximum) element and places it at the beginning of the sorted portion of the list.
Insertion Sort: Iteratively inserts each element into its correct position within the sorted portion of the list.
Merge Sort: Divides the list into smaller sublists, sorts them, and then merges them back together.
Quick Sort: Selects a pivot element and partitions the list around it, recursively sorting the partitions.

17. What do you mean by searching? Mention the types of searching.

Ans. Searching is the process of finding a given value position in a list of values. It decides whether a search key is present in the data or not. It is the algorithmic process of finding a particular item in a collection of items. It can be done on internal data structure or on external data structure.
Types of Searching Algorithms:
Linear Search (or Sequential Search):It sequentially checks each element in the data structure, starting from the beginning, until the target element is found or the end is reached. It is best for small, unsorted datasets.
Binary Search: It repeatedly divides the search interval in half, comparing the target element with the middle element of the interval. It requires the data to be sorted, and is much faster than linear search for large, sorted datasets.

18. Differentiate Linear Search and Binary search.

Linear Search	Binary Search
In linear search input data need not to be in sorted.	In binary search input data need to be in sorted order.
It is also called sequential search.	It is also called half-interval search.
The time complexity of linear search O(n).	The time complexity of binary search O(log n).
Multidimensional array can be used.	Only single dimensional array is used.
Linear search performs equality comparisons	Binary search performs ordering comparisons
It is less complex.	It is more complex.
It is very slow process.	It is very fast process.

19. Differentiate bubble sort, and selection sort.

S.No.	Bubble Sort	Selection Sort
1.	Bubble sort is a simple sorting algorithm which continuously moves through the list and compares the adjacent pairs for proper sorting of the elements.	Selection sort is a sorting algorithm which takes either smallest value (ascending order) or largest value (descending order) in the list and place it at the proper position in the list.
2.	Bubble sort compares the adjacent elements and move accordingly.	Selection sort selects the smallest element from the unsorted list and moves it at the next position of the sorted list.
3.	Bubble sort performs a large number of swaps or moves to sort the list.	Selection sort performs comparatively less number of swaps or moves to sort the list.
4.	Bubble sort is relatively slower.	Selection sort is faster as compared to bubble sort.
5.	The efficiency of the bubble sort is less.	The efficiency of the selection sort is high.
6.	Bubble sort performs sorting of an array by exchanging elements.	Selection sort performs sorting of a list by the selection of element.
7.	Time complexity: O(n^2) in the worst and average cases, O(n) in the best case (when the input array is already sorted) Space complexity: O(1)	Time complexity: O(n^2) in all cases (worst, average, and best) Space complexity: O(1)

20. Differentiate bubble sort, and insertion sort.

Bubble sort	Insertion sort
Bubble sort compares adjacent elements and swaps them if they are in the wrong order. This process is repeated until no more swaps are needed, indicating that the	Insertion sort builds a sorted array one element at a time. It iterates through the array, picking up an element and inserting it into its correct position within the sorted

	list is sorted. Time Complexity: Best Case: O(n) (already sorted array) Average Case: $O(n^2)$ Worst Case: $O(n^2)$ (reverse sorted array) Space Complexity: O(1) (in-place sorting) Stability: Stable (preserves the relative order of equal elements) Efficiency: Generally less efficient than insertion sort, especially for larger datasets.	portion. Time Complexity: Best Case: O(n) (already sorted array) Average Case: $O(n^2)$ Worst Case: $O(n^2)$ (reverse sorted array) Space Complexity: O(1) (in-place sorting) Stability: Stable (preserves the relative order of equal elements) Efficiency: Generally more efficient than bubble sort for smaller datasets and nearly sorted datasets.	

Long Answers Questions

1. Write C program to perform searching operation using linear and binary search.

Linear Search:

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

  printf("Enter number of elements in array\n");
  scanf("%d", &n);
  printf("Enter %d integer(s)\n", n);
  for (c = 0; c < n; c++)
    scanf("%d", &array[c]);
  printf("Enter a number to search\n");
  scanf("%d", &search);
  for (c = 0; c < n; c++)
  {
    if (array[c] == search)    /* If required element is found */
    {
      printf("%d is present at location %d.\n", search, c+1);
      break;
    }
  }
  if (c == n)
    printf("%d isn't present in the array.\n", search);
  return 0;
}
```

Output:

```
Enter number of elements in array
5
Enter 5 integer(s)
11
36
68
94
54
Enter a number to search
94
94 is present at location 4.
```

Binary Search:

```
// Binary Search in C

#include <stdio.h>
int binarySearch(int array[], int x, int low, int high) {
  // Repeat until the pointers low and high meet each other
  while (low <= high) {
    int mid = low + (high - low) / 2;
    if (x == array[mid])
      return mid;
    if (x > array[mid])
      low = mid + 1;
    else
      high = mid - 1;
  }

  return -1;
}

int main(void) {
  int array[] = {3, 4, 5, 6, 7, 8, 9};
  int n = sizeof(array) / sizeof(array[0]);
  int x = 4;
  int result = binarySearch(array, x, 0, n - 1);
  if (result == -1)
    printf("Not found");
  else
    printf("Element is found at index %d", result);
  return 0;
}
```

Output:
Element is found at index 1

2. What is meant by time complexity and space complexity? How do we analyze the time complexity and space complexity of linear data structures?

Ans. Linear Data Structures

Linear Data structure are those data structures in which data is stored sequentially, and each element is connected to the previous or next element so that they can be accessed in a single run. Some examples of linear data structures are arrays, stacks, queues, etc.

Let us discuss their time and space complexity.

Time Complexity

Time complexity can be understood as a concept that constitutes the quantification of time taken by an algorithm or code snippet to execute. The time complexity is also a measure of the efficiency, such that the lesser the time is taken by the algorithm, the more its efficiency will be.

We will be discussing only the worst-case time complexity in this article.

Space Complexity

Space Complexity can be understood as the amount of memory space occupied by a code snippet or algorithm. It is one of the two measures of the efficiency of an algorithm. The lesser the space it takes, the more efficient it is.

Now let's discuss some linear data structures.

Array

A Data Structure in which a similar type of data is stored at contiguous memory locations is called an array. Array is the most frequently used linear data structure in computer science and due to its connectivity with the previous and next element, it became the inspiration for data structures like linked lists, queues, etc.

Queue

A queue can be defined as a linear data structure, which is simply a collection of entries that are tracked in order,

such that the addition of entries happens at one end of the queue, while the removal of entries takes place from the other end. Its order is also known as First In First Out (FIFO).

Stack

A stack is a linear data structure, following a particular order in which operations can be performed. Its order is also known as LIFO(Last In First Out). Stacks are implemented using arrays and linked lists. Let us discuss its efficiency in terms of time and space complexity.

Linked List

A linked list can be understood as a data structure that consists of nodes to store data. Each node consists of a data field and a pointer to the next node. The various elements in a linked list are linked together using pointers. In Linked List, unlike arrays, elements are not stored at contiguous memory locations but rather at different memory locations.

Data Structure	Insert	Delete	Access	Search
Array	O(N)	O(N)	O(1)	O(N)
String	O(N)	O(N)	O(1)	O(N)
Queue	O(1)	O(1)	O(N)	O(N)
Stack	O(1)	O(1)	O(N)	O(N)
Linked List	O(1), if inserted on head O(N), elsewhere	O(1), if deleted on head O(N), elsewhere	O(N)	O(N)

Where 'N' is the size of the respective data structure.

Space Complexity of Linear Data Structures:

Space complexity for each operation in a linked list linear data structures is **O(1)**, as no extra space is required for any operation.

3. Differentiate Linear and Nonlinear data structures

Factor	Linear Data Structure	Non-Linear Data Structure
Data Element Arrangement	In a linear data structure, data elements are sequentially connected, allowing users to traverse all elements in one run.	In a non-linear data structure, data elements are hierarchically connected, appearing on multiple levels.
Implementation Complexity	Linear data structures are relatively easier to implement.	Non-linear data structures require a higher level of understanding and are more complex to implement.
Levels	All data elements in a linear data structure exist on a single level.	Data elements in a non-linear data structure span multiple levels.
Traversal	A linear data structure can be traversed in a single run.	More complex, requiring specialized algorithms like depth-first search or breadth-first search
Memory Utilization	Linear data structures do not efficiently utilize memory.	Non-linear data structures are more memory-friendly.
Time Complexity	The time complexity of a linear data structure is directly proportional to its size, increasing as input size increases.	The time complexity of a non-linear data structure often remains constant, irrespective of its input size.
Applications	Linear data structures are ideal for application software development.	Non-linear data structures are commonly used in image processing and Artificial Intelligence.
Examples	Linked List, Queue, Stack, Array.	Tree, Graph, Hash Map.

4. Write a program to reverse an array.

Ans.

```
#include <stdio.h>

void reverseArray(int arr[], int size) {
   int temp[size]; // Temporary array
   int j = 0;

   // Copy elements in reverse order
   for (int i = size - 1; i >= 0; i--) {
      temp[j++] = arr[i];
   }

   // Copy reversed elements back to the original array
   for (int i = 0; i < size; i++) {
      arr[i] = temp[i];
   }
}

int main() {
   int arr[] = {1, 2, 3, 4, 5};
   int size = sizeof(arr) / sizeof(arr[0]);

   // Display the original array
   printf("Original array: ");
   for (int i = 0; i < size; i++) {
      printf("%d ", arr[i]);
   }
   printf("");

   // Reverse the array
   reverseArray(arr, size);

   // Display the reversed array
   printf("Reversed array: ");
   for (int i = 0; i < size; i++) {
      printf("%d ", arr[i]);
   }
   printf("");
   return 0;
}
```

Output:

Original array: 1 2 3 4 5

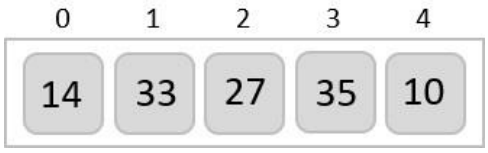
Reversed array: 5 4 3 2 1

5. What is sorting? Explain and write an algorithm for bubble sorting and trace the algorithm with an example.

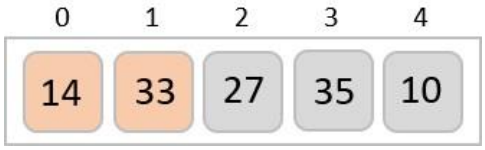
In computer science, a sorting algorithm is an algorithm that puts elements of a list into an order. The most frequently used orders are numerical order and lexicographical order, and either ascending or descending. Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the 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** is the number of items.
Bubble Sort works by repeatedly exchanging adjacent elements, if necessary. When no exchanges are required, the file is sorted.
Algorithm:
We assume **list** is an array of **n** elements.
Step 1 − Check if the first element in the input array is greater than the next element in the array.
Step 2 − If it is greater, swap the two elements; otherwise move the pointer forward in the array.
Step 3 − Repeat Step 2 until we reach the end of the array.
Step 4 − Check if the elements are sorted; if not, repeat the same process (Step 1 to Step 3) from the last element of the array to the first.
Step 5 − The final output achieved is the sorted array.

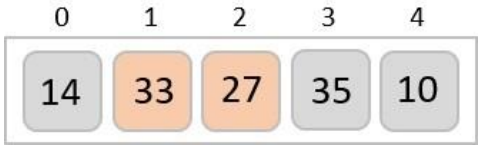
Example
We take an unsorted array for our example. Bubble sort takes O(n2) time so we're keeping it short and precise.



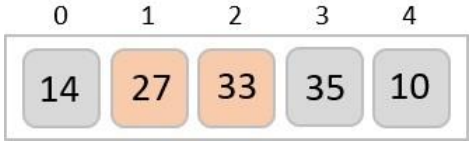
Bubble sort starts with very first two elements, comparing them to check which one is greater.



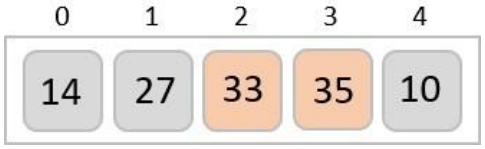
In this case, value 33 is greater than 14, so it is already in sorted locations. Next, we compare 33 with 27.



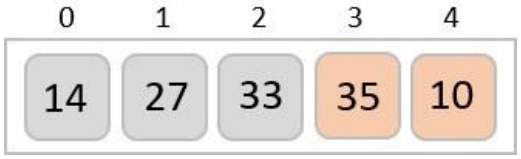
We find that 27 is smaller than 33 and these two values must be swapped.



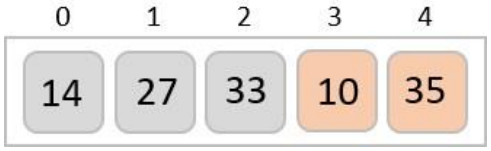
Next we compare 33 and 35. We find that both are in already sorted positions.



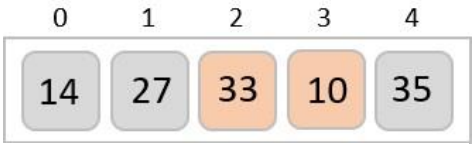
Then we move to the next two values, 35 and 10.

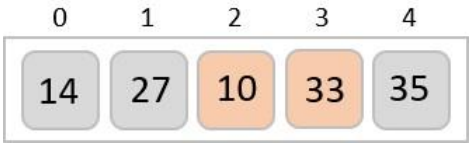


We know then that 10 is smaller 35. Hence they are not sorted. We swap these values. We find that we have reached the end of the array. After one iteration, the array should look like this −

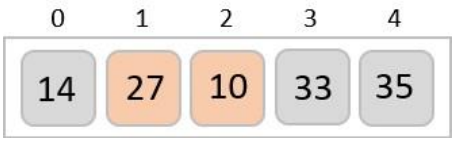


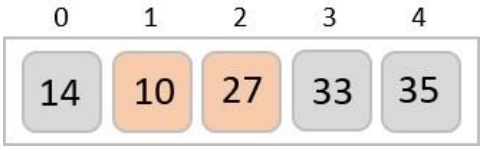
To be precise, we are now showing how an array should look like after each iteration. After the second iteration, it should look like this −

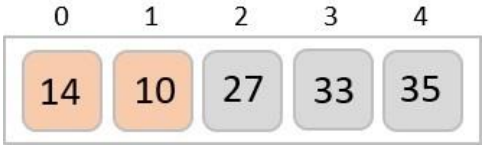


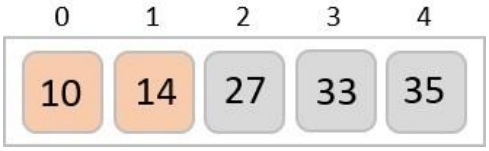


Notice that after each iteration, at least one value moves at the end.

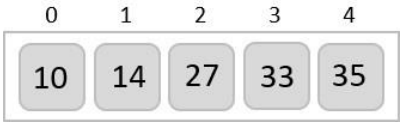








And when there's no swap required, bubble sort learns that an array is completely sorted.



6. Describe the Insertion Sort algorithm, trace the steps for sorting the following list, and derive the time complexity.

Insertion sort is a very simple method to sort numbers in an ascending or descending order. This method follows the incremental method. It can be compared with the technique how cards are sorted at the time of playing a game.

This is an 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. An element which is to be 'inserted' in this sorted sub-list, has to find its appropriate place and then it has to be inserted there. Hence the name, **insertion sort**.

The array is searched sequentially and unsorted items are moved and inserted into the 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** is the number of items.

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.

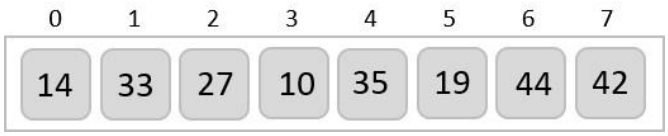
Step 1 − If it is the first element, it is already sorted. return 1;

Step 2 − Pick next element

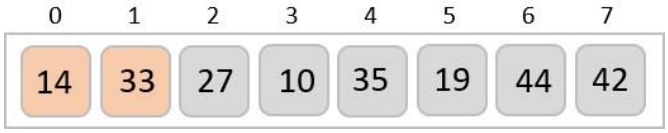
Step 3 − Compare with all elements in the sorted sub-list
Step 4 − Shift all the elements in the sorted sub-list that is greater than the value to be sorted
Step 5 − Insert the value
Step 6 − Repeat until list is sorted

Analysis
Run time of this algorithm is very much dependent on the given input.
If the given numbers are sorted, this algorithm runs in **O(n)** time. If the given numbers are in reverse order, the algorithm runs in **O(n^2)** time.

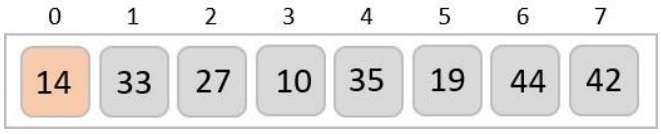
Example
We take an unsorted array for our example.



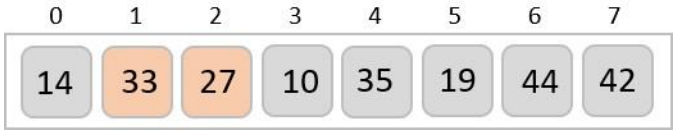
Insertion sort compares the first two elements.



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

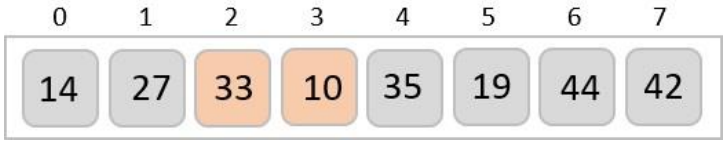


Insertion sort moves ahead and compares 33 with 27.



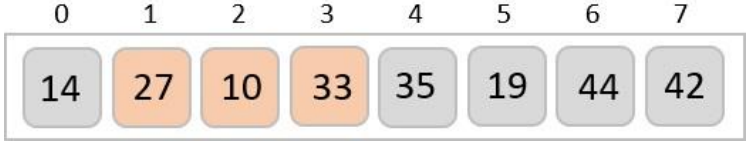
And finds that 33 is not in the correct position. It swaps 33 with 27. It also checks with all the elements of sorted sub-list. Here we see that the sorted sub-list has only one element 14, and 27 is greater than 14. Hence, the sorted sub-list remains sorted after swapping.

By now we have 14 and 27 in the sorted sub-list. Next, it compares 33 with 10. These values are not in a sorted order.

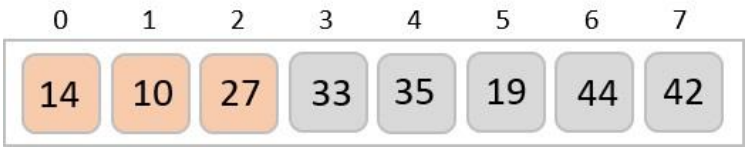


So they are swapped.

However, swapping makes 27 and 10 unsorted.

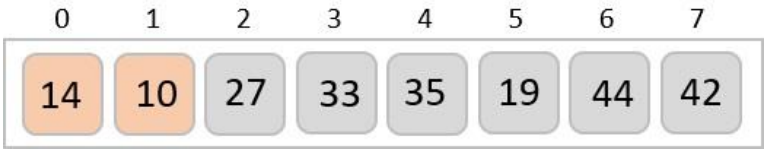


Hence, we swap them too.

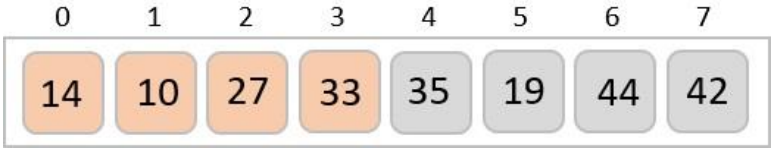


Again we find 14 and 10 in an unsorted order.

We swap them again.



By the end of third iteration, we have a sorted sub-list of 4 items.



This process goes on until all the unsorted values are covered in a sorted sub-list. Now we shall see some programming aspects of insertion sort.

7. Describe the selection Sort algorithm, trace the steps for sorting the following list, and derive the time complexity.

Selection sort is a simple sorting algorithm. This sorting algorithm, like insertion sort, is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is the entire list.
The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array. This process continues moving unsorted array boundaries by one element to the right.
This algorithm is not suitable for large data sets as its average and worst case complexities are of **$O(n^2)$**, where **n** is the number of items.

Selection Sort Algorithm
This type of sorting is called Selection Sort as it works by repeatedly sorting elements. That is: we first find the smallest value in the array and exchange it with the element in the first position, then find the second smallest element and exchange it with the element in the second position, and we continue the process in this way until the entire array is sorted.

1. Set MIN to location 0.

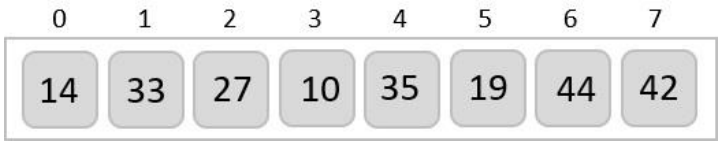
2. Search the minimum element in the list.

3. Swap with value at location MIN.

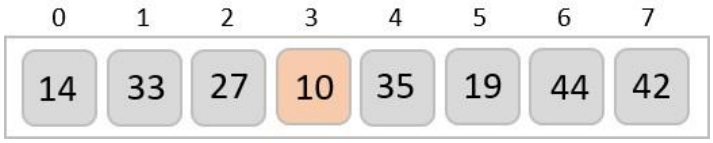
4. Increment MIN to point to next element.

5. Repeat until the list is sorted.

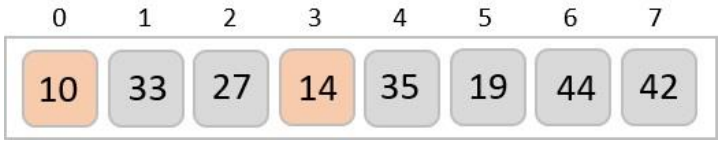
Example
Consider the following depicted array as an example.



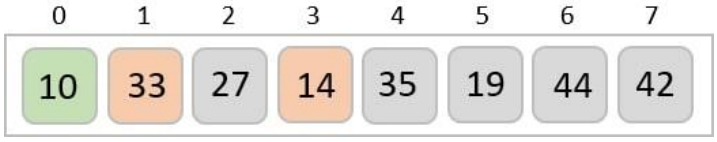
For the first position in the sorted list, the whole list is scanned sequentially. The first position where 14 is stored presently, we search the whole list and find that 10 is the lowest value.



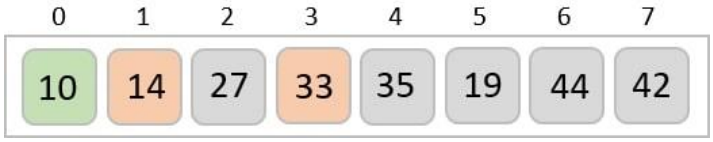
So we replace 14 with 10. After one iteration 10, which happens to be the minimum value in the list, appears in the first position of the sorted list.



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

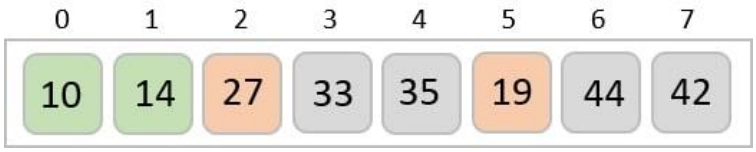


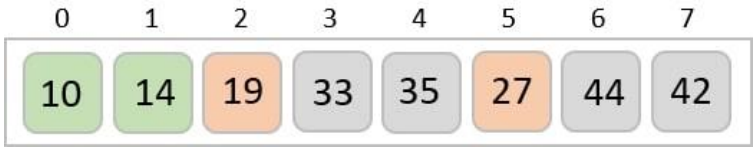
We find that 14 is the second lowest value in the list and it should appear at the second place. We swap these values.

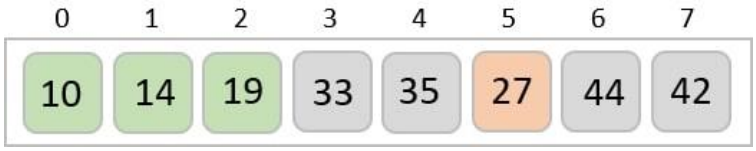


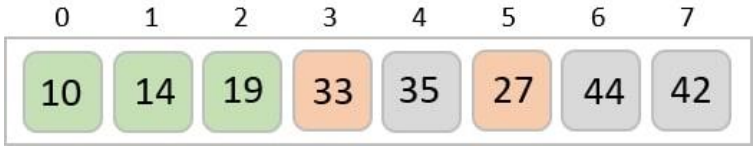
After two iterations, two least values are positioned at the beginning in a sorted manner.

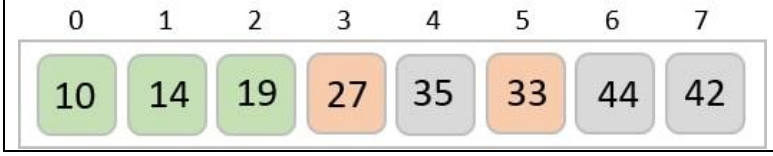
The same process is applied to the rest of the items in the array −

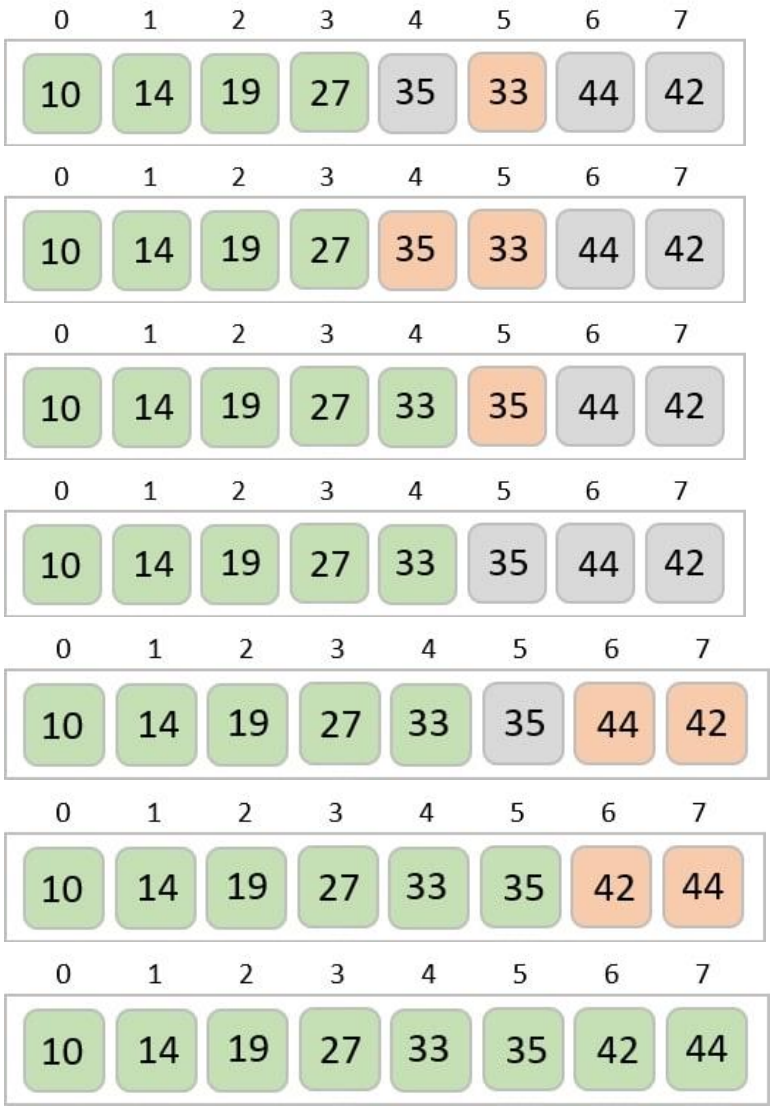












8. Write C program to perform bubble sort.

```
// Bubble sort in C

#include <stdio.h>

// perform the bubble sort
void bubbleSort(int array[], int size) {

  // loop to access each array element
  for (int step = 0; step < size - 1; ++step) {

    // loop to compare array elements
    for (int i = 0; i < size - step - 1; ++i) {

      // compare two adjacent elements
      // change > to < to sort in descending order
      if (array[i] > array[i + 1]) {

        // swapping occurs if elements
        // are not in the intended order
        int temp = array[i];
```

```
        array[i] = array[i + 1];
        array[i + 1] = temp;
      }
    }
  }
}

// print array
void printArray(int array[], int size) {
  for (int i = 0; i < size; ++i) {
    printf("%d  ", array[i]);
  }
  printf("\n");
}

int main() {
  int data[] = {-2, 45, 0, 11, -9};

  // find the array's length
  int size = sizeof(data) / sizeof(data[0]);

  bubbleSort(data, size);

  printf("Sorted Array in Ascending Order:\n");
  printArray(data, size);
}
```

Output:
Sorted Array in Ascending Order:
-9 -2 0 11 45

9. What is data structure? Explain its role in problem solving? Explain the different types of data structure with suitable examples.

Ans. A data structure is a method of organizing and storing data, enabling efficient access and manipulation. It plays a crucial role in problem-solving by optimizing data organization and retrieval, leading to faster and more efficient algorithms.

Role in Problem Solving:

Efficiency:

Data structures are designed to optimize operations like searching, insertion, deletion, and sorting, leading to faster and more efficient algorithms.

Organization:

They provide a systematic way to store and manage large amounts of data, making it easier to access and manipulate.

Algorithm Design:

The choice of data structure can significantly impact the performance and complexity of algorithms, making it a crucial aspect of problem-solving.

Code Reusability:

Well-designed data structures can be reused across different projects, saving time and effort in software development.

Types of Data Structures:

1. Linear Data Structures:

Arrays:

A collection of elements stored in contiguous memory locations, allowing for direct access to any element using an index.

Example: An array to store a list of student IDs: [101, 102, 103, 104].

Linked Lists:

A collection of nodes, where each node contains data and a pointer to the next node, allowing for dynamic allocation and insertion/deletion.

Example: A linked list to store a list of names: Node(John) -> Node(Mary) -> Node(Peter).

Stacks:

A linear data structure where elements are added and removed in a Last-In, First-Out (LIFO) manner.
Example: A stack to store the history of visited web pages in a browser.

Queues:

A linear data structure where elements are added and removed in a First-In, First-Out (FIFO) manner.
Example: A queue to manage a list of tasks to be processed.

2. Non-Linear Data Structures:

Trees:

A hierarchical data structure where elements are organized in a parent-child relationship.
Example: A file system directory structure.

Graphs:

A data structure that represents relationships between entities, consisting of nodes (vertices) and edges.
Example: A social network where users are nodes and connections are edges.

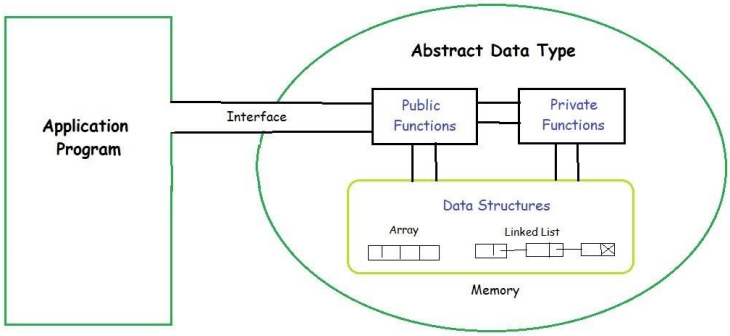
Hash Tables:

A data structure that uses a hash function to map keys to values, allowing for fast retrieval of data.
Example: A dictionary where words are keys and their meanings are values.

10. Explain about abstract data types with examples.

Ans. An Abstract Data Type (ADT) is a conceptual model that defines a set of operations and behaviors for a data structure, without specifying how these operations are implemented or how data is organized in memory. The definition of ADT only mentions what operations are to be performed but not how these operations will be implemented. It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations. It is called "abstract" because it provides an implementation-independent view.

The process of providing only the essentials and hiding the details is known as abstraction.



The above figure shows the ADT model. There are two types of models in the ADT model, i.e., the public function and the private function. The ADT model also contains the data structures that we are using in a program. In this model, first encapsulation is performed, i.e., all the data is wrapped in a single unit, i.e., ADT. Then, the abstraction is performed means showing the operations that can be performed on the data structure and what are the data structures that we are using in a program.

Examples of ADTs

Now, let's understand three common ADT's: List ADT, Stack ADT, and Queue ADT.

1. List ADT

The List ADT (Abstract Data Type) is a sequential collection of elements that supports a set of operations without specifying the internal implementation. It provides an ordered way to store, access, and modify data.

The List ADT need to store the required data in the sequence and should have the following operations**:**

- **get():** Return an element from the list at any given position.
- **insert():** Insert an element at any position in the list.
- **remove():** Remove the first occurrence of any element from a non-empty list.
- **removeAt():** Remove the element at a specified location from a non-empty list.
- **replace():** Replace an element at any position with another element.
- **size():** Return the number of elements in the list.
- **isEmpty():** Return true if the list is empty; otherwise, return false.
- **isFull():** Return true if the list is full; otherwise, return false.

2. Stack ADT

The Stack ADT is a linear data structure that follows the LIFO (Last In, First Out) principle. It allows elements to be added and removed only from one end, called the top of the stack.
In Stack ADT, the order of insertion and deletion should be according to the FILO or LIFO Principle. Elements are inserted and removed from the same end, called the top of the stack. It should also support the following operations:

- **push():** Insert an element at one end of the stack called the top.
- **pop():** Remove and return the element at the top of the stack, if it is not empty.
- **peek():** Return the element at the top of the stack without removing it, if the stack is not empty.
- **size():** Return the number of elements in the stack.
- **isEmpty():** Return true if the stack is empty; otherwise, return false.
- **isFull():** Return true if the stack is full; otherwise, return false.

3. Queue ADT
The Queue ADT is a linear data structure that follows the FIFO (First In, First Out) principle. It allows elements to be inserted at one end (rear) and removed from the other end (front).
The Queue ADT follows a design similar to the Stack ADT, but the order of insertion and deletion changes to FIFO. Elements are inserted at one end (called the rear) and removed from the other end (called the front). It should support the following operations:

- **enqueue():** Insert an element at the end of the queue.
- **dequeue():** Remove and return the first element of the queue, if the queue is not empty.
- **peek():** Return the element of the queue without removing it, if the queue is not empty.
- **size():** Return the number of elements in the queue.
- **isEmpty():** Return true if the queue is empty; otherwise, return false.

Features of ADT
Abstract data types (ADTs) are a way of encapsulating data and operations on that data into a single unit. Some of the key features of ADTs include:

- **Abstraction:** The user does not need to know the implementation of the data structure only essentials are provided.
- **Better Conceptualization:** ADT gives us a better conceptualization of the real world.
- **Robust:** The program is robust and has the ability to catch errors.
- **Encapsulation**: ADTs hide the internal details of the data and provide a public interface for users to interact with the data. This allows for easier maintenance and modification of the data structure.
- **Data Abstraction**: ADTs provide a level of abstraction from the implementation details of the data. Users only need to know the operations that can be performed on the data, not how those operations are implemented.
- **Data Structure Independence**: ADTs can be implemented using different data structures, such as arrays or linked lists, without affecting the functionality of the ADT.
- **Information Hiding:** ADTs can protect the integrity of the data by allowing access only to authorized users and operations. This helps prevent errors and misuse of the data.
- **Modularity**: ADTs can be combined with other ADTs to form larger, more complex data structures. This allows for greater flexibility and modularity in programming.

11. **Write c program for selection sort and insertion sort**.

Ans. C Program to Implement Selection Sort Algorithm

```
// C program for implementation of selection sort
#include <stdio.h>

void selectionSort(int arr[], int N) {

  // Start with the whole array as unsored and one by
   // one move boundary of unsorted subarray towards right
  for (int i = 0; i < N - 1; i++) {

    // Find the minimum element in unsorted array
    int min_idx = i;
    for (int j = i + 1; j < N; j++) {
```

```
            if (arr[j] < arr[min_idx]) {
                min_idx = j;
            }
        }

        // Swap the found minimum element with the first
        // element in the unsorted part
        int temp = arr[min_idx];
        arr[min_idx] = arr[i];
        arr[i] = temp;
    }
}

int main() {
    int arr[] = {64, 25, 12, 22, 11};
    int N = sizeof(arr) / sizeof(arr[0]);
    printf("Unsorted array: \n");
    for (int i = 0; i < N; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

     // Calling selection sort
    selectionSort(arr, N);

    printf("Sorted array: \n");
    for (int i = 0; i < N; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
    return 0;
}
```

Output

```
Unsorted array:

64 25 12 22 11

Sorted array:

11 12 22 25 64
```