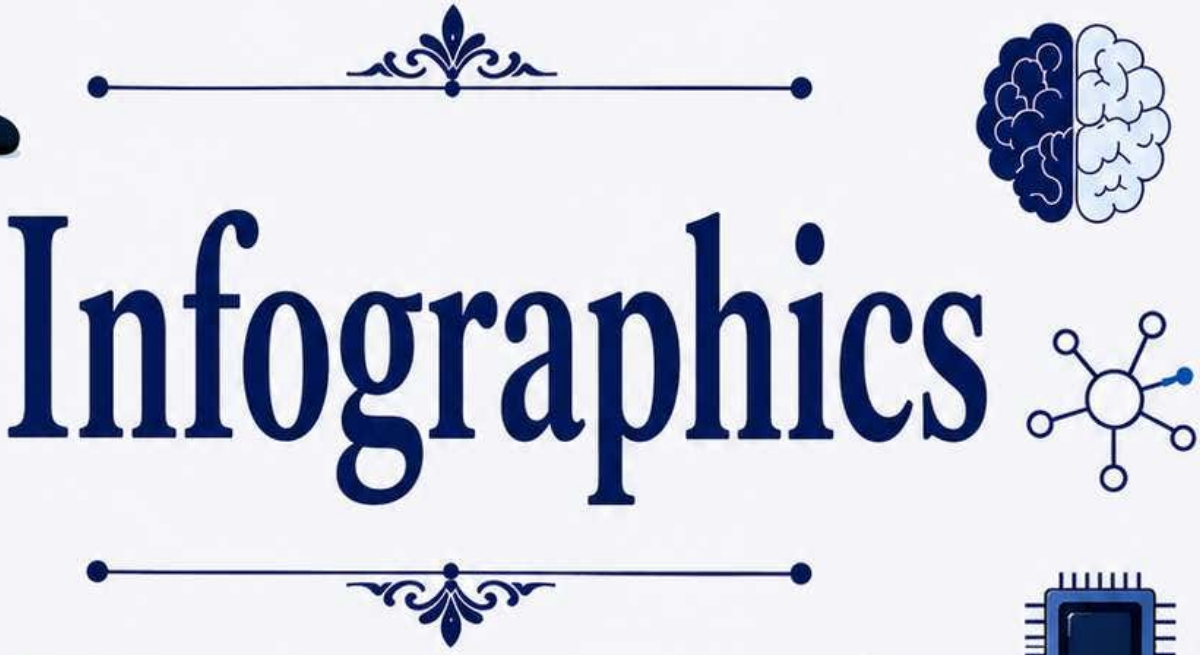
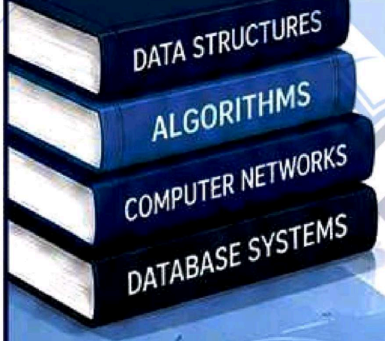
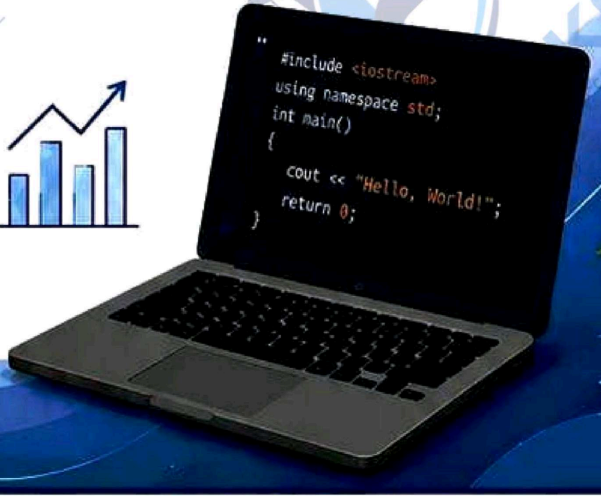
Infographics



Unit - IV


DATA STRUCTURES
ALGORITHMS
COMPUTER NETWORKS
DATABASE SYSTEMS


#include <iostream>
using namespace std;
int main()
{
cout << "Hello, World!";
return 0;
}

SEARCHING

DEFINITION AND CONCEPT

1. DEFINITION

Searching is the process of finding the location (position) of a specific element (key) in a collection of elements (data set). If the element is found, the algorithm returns its position, otherwise it indicates that the element is not present.

In Simple Words

Searching means looking for a particular item in a collection to know whether it exists or not and where it is located.

Visual Example

Index	0	1	2	3	4	5	6
Data	12	25	37	42	55	68	90

Search Key = 42

Key 42 is present at index 3 (position 4th).

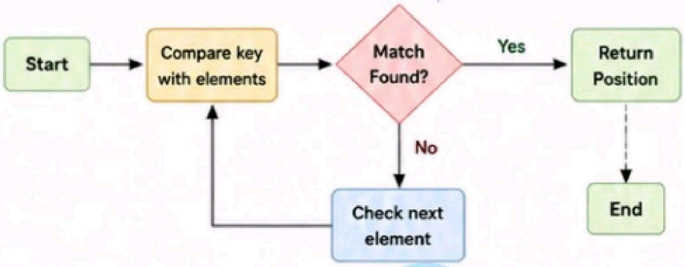
Search Key = 100

Key 100 is not present in the list.

2. CONCEPT

- We have a collection of elements stored in an array or other data structure.
- We compare the search key with the elements according to a strategy.
- If a match is found → return its position.
- If all elements are checked and no match is found → return **NOT FOUND**.

General Searching Process



3. TYPES OF SEARCHING

(A) Linear Search (Sequential Search)

Elements are checked one by one from the beginning until the key is found or the list ends.

Example:

Index	0	1	2	3	4	5	6
Data	12	25	37	42	55	68	90

Search Key = 37

Comparisons:

12 × → 25 × → 37 ✓ (Found at index 2)

Works on both sorted and unsorted data.

(B) Binary Search

Used on sorted data. The list is divided into halves. Key is compared with middle element.

Example (Sorted Data):

Index	0	1	2	3	4	5	6
Data	12	25	37	42	55	68	90

Search Key = 42

Steps:

1. Middle element = 42 (index 3) → Match Found ✓

Very efficient for large sorted lists.

4. IMPORTANT TERMS

Key	The element which is to be searched.
Position / Index	Location of the key in the data structure.
Success	Key is found in the given collection.
Failure	Key is not found in the given collection.
Efficiency	Measured by the number of comparisons and time taken.

5. EXAMPLE ILLUSTRATION

Array:

15	28	36	47	59	63	75

Linear Search
Search Key = 59

15 ×
28 ×
36 ×
47 ×
59 ✓ (Found at index 4)

Binary Search (Sorted)
Search Key = 59

1. Mid=47 → 59>47 → Right Half
2. Mid=63 → 59<63 → Left Half
3. Mid=59 → Match Found ✓

(Found at index 4)

6. TIME COMPLEXITY COMPARISON

Algorithm	Best Case	Average Case	Worst Case	Space Complexity	Applicable To
Linear Search	O(1)	O(n)	O(n)	O(1)	Sorted & Unsorted
Binary Search	O(1)	O(log n)	O(log n)	O(1)	Sorted Only

7. KEY TAKEAWAYS

- Searching helps to quickly locate a desired element.
- Linear Search is simple but slower for large data.
- Binary Search is very fast but requires sorted data.
- Choosing the right search method improves the efficiency of programs.

SEARCHING TECHNIQUES
LINEAR SEARCH

Linear Search (Sequential Search) is the simplest searching technique in which each element of the list is **checked one by one** from the beginning until the desired element (key) is found or the list ends.

1. DEFINITION

Linear Search is a searching technique that compares the search key with each element of the list sequentially from the first element to the last element. If the key matches an element, its position (index) is returned, otherwise the search is unsuccessful.

In Simple Words

We start from the first element and check every element one by one until we find the key or reach the end of the list.

2. CONCEPT

- Compare the search key with the first element.
- If match is found → return the index.
- If not matched → move to the next element.
- Repeat the process until key is found or we reach the end of the list.
- If key is not found → return **NOT FOUND**.

3. HOW LINEAR SEARCH WORKS

Index	0	1	2	3	4	5	6	7
Array	25	17	31	44	58	13	99	27

Search Key = 44

Step 1:	**25**	17	31	44	58	13	99	27	25 ≠ 44 → Move Next
Step 2:	25	**17**	31	44	58	13	99	27	17 ≠ 44 → Move Next
Step 3:	25	17	**31**	44	58	13	99	27	31 ≠ 44 → Move Next
Step 4:	25	17	31	**44**	58	13	99	27	44 = 44 → Found at index 3

4. ALGORITHM

1. Start
2. Read array A[0...n-1] and search key K
3. Set i = 0
4. While i < n
   - If A[i] == K
     → Return i (index of key)
   - Else
     → i = i + 1
5. If loop ends and key not found
   → Return NOT FOUND
6. Stop

5. TRACE TABLE EXAMPLE

Array: A = [25, 17, 31, 44, 58, 13, 99, 27] **Key = 44**

Step	i (Index)	A[i]	Comparison A[i] == Key	Action	Result
1	0	25	25 == 44 (False)	Move to next	Continue
2	1	17	17 == 44 (False)	Move to next	Continue
3	2	31	31 == 44 (False)	Move to next	Continue
4	3	44	44 == 44 (True)	Key Found	Return index 3

(Search Successful at index 3)

6. PSEUDO CODE

```
LinearSearch(A, n, K)
1. for i = 0 to n-1
2.     if A[i] == K
3.         return i
4. return NOT FOUND
End LinearSearch
```

7. C / C++ CODE

```c
int linearSearch(int A[], int n, int key)
{
    for (int i = 0; i < n; i++)
    {
        if (A[i] == key)
            return i;  // key found
    }
    return -1; // key not found
}
```

Example Run:

A = [25, 17, 31, 44, 58, 13, 99, 27]

Key = 44

Output: Key found at index 3

Key = 100

Output: Key not found (-1)

8. TIME COMPLEXITY

Best Case	:	O(1) (Key is at first position)
Average Case	:	O(n)
Worst Case	:	O(n) (Key is at last or not present)
Space Complexity	:	O(1)

9. ADVANTAGES

- Very simple to understand
- Works on both sorted and unsorted data
- No extra space required

10. DISADVANTAGES

- Slow for large data
- Not efficient for large lists
- Many comparisons may be needed

11. APPLICATIONS

- Small lists
- Unsorted data
- Searching in files
- Checking presence of data

KEY TAKEAWAY

Linear Search is easy to implement and useful for small or unsorted data, but it requires up to n comparisons in the worst case.

BINARY SEARCH

Searching Technique for Sorted Data

Binary Search is an efficient searching technique used on **sorted** (ascending or descending) data by repeatedly dividing the search interval in half.

1. DEFINITION

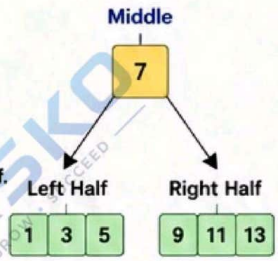
Binary Search compares the search key with the middle element of the sorted list. If they are not equal, the half in which the key cannot lie is eliminated and the search continues on the other half, again taking the middle element to compare, and so on until the key is found or the list is empty.

In Simple Words

We repeatedly divide the list into two halves, check the middle element, and continue the search in the appropriate half.

2. CONCEPT

- Data must be sorted.
- Compare the key with the middle element.
- If key = middle element → return its index.
- If key < middle element → search in left half.
- If key > middle element → search in right half.
- Repeat the process until key is found or the search interval becomes empty.
- If not found → return NOT FOUND.



3. HOW BINARY SEARCH WORKS (Example)

Search Key = 11

Sorted Array (Index)

	0	1	2	3	4	5	6
Array	2	4	6	8	10	12	14

Step 1	low = 0, high = 6 mid = (0+6)/2 = 3	2 4 6 **8** 10 12 14	8 < 11 Search in right half
Step 2	low = 4, high = 6 mid = (4+6)/2 = 5	2 4 6 8 10 **12** 14	12 > 11 Search in left half
Step 3	low = 4, high = 4 mid = (4+4)/2 = 4	2 4 6 **10** 10 12 14	10 < 11 Search in right half
Step 4	low = 5, high = 4 (low > high)		Key not found

4. ALGORITHM

1. Start
2. Set low = 0, high = n - 1
3. While (low ≤ high)
   - mid = (low + high) / 2
   - If A[mid] == key → return mid
   - Else if key < A[mid] → high = mid - 1
     Else → low = mid + 1
4. If loop ends → return NOT FOUND.
5. Stop

Note

This works only on sorted data.

5. TRACE TABLE EXAMPLE

Array: A = [2, 4, 6, 8, 10, 12, 14] Key = 11

Step	low	high	mid	A[mid]	Comparison	Action
1	0	6	3	8	8 < 11	low = mid + 1 (4)
2	4	6	5	12	12 > 11	high = mid - 1 (4)
3	4	4	4	10	10 < 11	low = mid + 1 (5)
4	5	4	-	-	low > high	Key not found

Result: Key 11 is not present in the array.

6. PSEUDO CODE

```
BinarySearch(A, n, key)
1. low ← 0
2. high ← n - 1
3. while (low ≤ high)
     mid ← (low + high) / 2
     if (A[mid] == key)
          return mid
     else if (key < A[mid])
          high ← mid - 1
     else
          low ← mid + 1
4. return NOT FOUND
End BinarySearch
```

7. C / C++ CODE

```
int binarySearch(int A[], int n, int key)
{
    int low = 0, high = n - 1, mid;
    while (low <= high)
    {
        mid = (low + high) / 2;
        if (A[mid] == key)
            return mid;
        else if (key < A[mid])
            high = mid - 1;
        else
            low = mid + 1;
    }
    return -1;  // Not Found
}
```

8. EXAMPLE RUN (Key = 11)

A = [2, 4, 6, 8, 10, 12, 14]

The steps are same as in the trace table.

Output: Key 11 is not found (-1)

9. TIME COMPLEXITY

Best Case	: O(1)
Average Case	: O(log n)
Worst Case	: O(log n)
Space Complexity	: O(1) (iterative) O(log n) (recursive)

10. ADVANTAGES

- Very fast for large datasets.
- Requires only O(log n) comparisons.
- Easy to implement (iterative).
- Works efficiently on sorted data.

11. DISADVANTAGES

- Data must be sorted.
- Insertion/Deletion in sorted array is costly.
- Not suitable for unsorted data.

12. APPLICATIONS

- Searching in large sorted lists.
- Used in databases and files.
- Dictionary lookups.
- Routing and network algorithms.
- Many real-world systems.

KEY TAKEAWAY

Binary Search is much faster than Linear Search and is ideal for large, sorted datasets, reducing the search space by half in each step.

INDEXED SEQUENTIAL SEARCH

A searching technique that combines the advantages of **Sequential Search** and **Binary Search**.

1. DEFINITION

Indexed Sequential Search is a searching technique for ordered data that maintains an index (a secondary list) with key values and their addresses (or positions).

It first performs a binary search on the index to find the block where the key may exist, and then performs a sequential search within that block.

In Simple Words

We first search in the index using binary search to find the appropriate block, then search sequentially in that block.

2. CONCEPT

- Data is sorted and divided into blocks.
- An index is created for the first element of each block along with its address.
- Binary search is applied on the index.
- Once the block is located, linear search is applied within that block.
- If key is found → return its position.
- If key is not found → return **NOT FOUND.**

Index Table (Primary Index)

Key	Address (Block No.)
10	0
30	1
50	2
70	3

Search in appropriate block sequentially

3. DATA STRUCTURE EXAMPLE

Assume Block Size = 4

Main Data (Ordered)

Index :	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Data :	10	12	15	18	30	31	35	40	50	55	60	65	70	72	78	85
	Block 0				Block 1				Block 2				Block 3			

Index Table

Key (First element)	10	30	50	70
Address (Block No.)	0	1	2	3

4. HOW IT WORKS (Example)

Search Key = 60

Step 1: Binary search on Index Table

Index Table Keys: [10, 30, 50, 70]

60 > 50 and 60 < 70 → Possible in Block 3 (Address = 3)

Step 2: Sequential search in Block 3

Block 3 Elements: [70, 72, 78, 85]

60 is not in this block.

Hence, Key 60 is NOT FOUND.

5. ALGORITHM

1. Create an index table for the ordered data (first key of each block and its address).
2. Perform binary search on the index table for the given key.
3. If exact match is found in index → return its position.
4. Otherwise, get the address of the block where the key may exist.
5. Apply sequential search within that block.
6. If key is found → return its position.
7. If not found in the block → return NOT FOUND.
8. If key is not in index range → return NOT FOUND.
9. Stop.

Note

Block size (b) is predefined.

Data must be sorted.

6. TRACE TABLE EXAMPLE (Key = 60)

Index Table: [10, 30, 50, 70] — **Block Size = 4**

Step	low	high	mid	Index[mid]	Comparison	Result
1	0	3	1	30	60 > 30	low = mid + 1 (2)
2	2	3	2	50	60 > 50	low = mid + 1 (3)
3	3	3	3	70	60 < 70	Stop → Block 3

Sequential Search in Block 3

Element Checked	70	72	78	85
Comparison with 60	60 < 70	60 < 72	60 < 78	60 < 85
Result	Not Found			

Key 60 is NOT FOUND.

7. PSEUDO CODE

```
IndexedSequentialSearch(A, Index, n, b, key)
 1.  low ← 0, high ← ⌈n/b⌉ - 1
     // number of blocks
 2.  // Binary search on index
 3.  while (low ≤ high)
 4.      mid ← (low + high) / 2
 5.      if (Index[mid].key == key) return Index[mid].addr
 6.      else if (Index[mid].key < key) low ← mid + 1
 7.      else  high ← mid - 1
 8.  // Get block address
 9.  block ← (low - 1)   // last index where key < Index[mid].key
10.  start ← block * b
11.  end ← min(start + b - 1, n - 1)
12.  // Sequential search in the block
13.  for i ← start to end
14.      if (A[i] == key)  return i
15.  return NOT FOUND
```

8. C CODE (Example)

```c
#define MAX 100
#define BLOCK 4
typedef struct { int key; int addr; } Index;
int indexedSequentialSearch(int A[], Index idx[], int n, int b, int key)
{
   int low = 0, high = (n + b - 1) / b - 1, mid, block, start, end, i;
   // Binary search on index
   while (low <= high)
   {
      mid = (low + high) / 2;
      if (idx[mid].key == key) return idx[mid].addr;
      else if (idx[mid].key < key) low = mid + 1;
      else  high = mid - 1;
   }
   block = low - 1;
   start = block * b;
   end = (start + b - 1 < n - 1) ? start + b - 1 : n - 1;
   // Sequential search in block
   for (i = start; i <= end; i++)
      if (A[i] == key) return i;
   return -1; // Not Found
}
```

9. COMPLEXITY ANALYSIS

Let n = total number of elements
b = block size
Number of blocks = ⌈n/b⌉

- Binary search on index: O(log (n/b))
- Sequential search in block: O(b)
- Total Time Complexity: O(log (n/b) + b)

Best Case: O(1) (key found in index)
Worst Case: O(log (n/b) + b)
Space Complexity: O(n/b) for index table

10. ADVANTAGES / DISADVANTAGES

Advantages

- Faster than linear search for large datasets.
- Efficient for data that is stored on secondary storage (e.g., disk).
- Requires less comparison than linear search.

Disadvantages

- Extra space needed for index table.
- Performance depends on block size.
- More complex to implement than linear search.

11. APPLICATIONS

- Searching in large ordered files stored on disks.
- Databases and library management systems.
- Used in indexing of dictionaries, phone books, and catalogs.
- Any system where data is stored in sorted blocks.

KEY TAKEAWAY

Indexed Sequential Search reduces the number of comparisons by first searching the index (using binary search) to locate the block and then performing sequential search within that block.

SORTING

DEFINITION AND CONCEPT

Sorting is the process of arranging data or elements in a particular order (ascending or descending) based on a key or value.

1. DEFINITION

Sorting is the systematic arrangement of data elements in a specific order. The order may be either ascending (smallest to largest) or descending (largest to smallest) according to the values of the elements or a specified key.

In Simple Words

Sorting means putting the items in order so that we can easily search, process, and manage the data.

2. CONCEPT

- Sorting arranges data in a meaningful sequence.
- It is used to organize data for efficient searching, processing, and presentation.
- Elements are compared and rearranged based on a sorting order.
- After sorting, data can be in:
  - Ascending Order : Smallest to Largest
  - Descending Order : Largest to Smallest

3. EXAMPLE

Ascending Order (Smallest to Largest)

Unsorted:	45	12	78	33	9	56	23	→	Sorted:	9	12	23	33	45	56	78

Descending Order (Largest to Smallest)

Unsorted:	45	12	78	33	9	56	23	→	Sorted:	78	56	45	33	23	12	9

4. IMPORTANCE OF SORTING

Improves Search Efficiency
Sorted data allows faster searching (using binary search).

Better Data Organization
Makes data easier to read, understand, and manage.

Efficient Processing
Many algorithms and operations work better on sorted data.

Useful in Real Life
Used in databases, file systems, reporting, leaderboards, scheduling, etc.

5. GENERAL SORTING PROCESS

Input Data
Take the list/array of elements.

↓

Compare Elements
Compare elements based on a key.

↓

Rearrange Elements
Swap or shift elements to place them in correct order.

↓

Repeat
Repeat the process until the entire list is sorted.

↓

Sorted Output
The list is now in ascending or descending order.

6. TYPES OF SORTING ORDER

Ascending Order

Example:	5	2	9	1	7	→	1	2	5	7	9

Descending Order

Example:	5	2	9	1	7	→	9	7	5	2	1

7. APPLICATIONS OF SORTING

- Database Management
- File and Record Organization
- E-commerce (Product Listing)
- Rankings and Leaderboards
- Scheduling and Timetabling
- Data Analysis and Reporting

8. CHARACTERISTICS OF A GOOD SORTING ALGORITHM

Characteristic	Description
Correctness	Always produces correctly sorted output.
Efficiency	Uses minimum time and memory.
Stability	Maintains the relative order of equal elements.
In-place	Requires less extra memory.
Scalability	Performs well for large datasets.

KEY TAKEAWAY

Sorting organizes data in a specific order to make it easier to search, process, and analyze.
It is a fundamental operation in computer science with numerous real-world applications.

SORTING TECHNIQUES
BUBBLE SORT

Bubble Sort is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements and swaps them if they are in the wrong order. This process is repeated until the list is sorted.

1. DEFINITION

Bubble Sort is a comparison-based sorting algorithm. It gets its name because larger elements "bubble" toward the end of the list in each pass, just like bubbles rise to the surface in water.

In Simple Words

Keep comparing neighboring elements and swap them if they are in the wrong order. Repeat until no more swaps are needed.

2. CONCEPT

- Compare adjacent elements.
- If the left element is greater than the right element, swap them.
- After each pass, the largest unsorted element moves to its correct position at the end.
- Reduce the range of comparison by one after each pass.
- Repeat until no swaps are needed in a pass.

How it "Bubbles"

Pass 1 ↓
Pass 2 ↓
Pass 3 ↓
...
Pass n-1 ↓

Largest element at its correct place

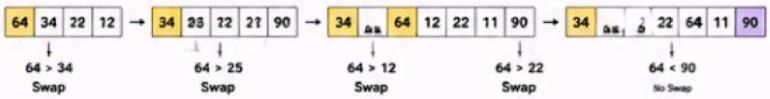
3. WORKING EXAMPLE (Ascending Order)

Array: 64, 34, 25, 12, 22, 11, 90 (n = 7)

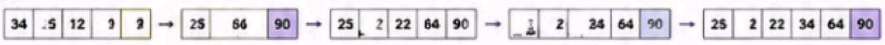
Key

- 64 — Element being compared
- 34 — Element being compared with
- 64 — Element after swapping
- 90 — Element at correct position after each pass

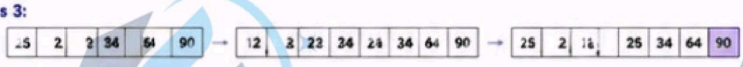
Pass 1: (Largest element goes to last position)



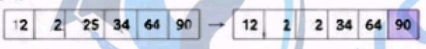
Pass 2:



Pass 3:



Pass 4:



(No Swaps in next pass → Sorted)

4. ALGORITHM

1. Start
2. For i = 0 to n-2
3. For j = 0 to n-i-2
4. If A[j] > A[j+1]
   Swap A[j] and A[j+1]
5. Next j
6. Next i
7. Stop

Note

After i'th pass, the largest element in the unsorted part is placed at position n-i-1.

5. PSEUDO CODE

```
BubbleSort(A, n)
1.  for i = 0 to n-2
2.      for j = 0 to n-i-2
3.          if A[j] > A[j+1]
4.              swap A[j] and A[j+1]
5.      next j
6.  next i
End BubbleSort
```

6. C CODE

```
void bubbleSort(int A[], int n)
{
   int i, j, temp;
   for (i = 0; i < n-1; i++)
   {
      for (j = 0; j < n-i-1; j++)
      {
         if (A[j] > A[j+1])
         {
            temp = A[j];
            A[j] = A[j+1];
            A[j+1] = temp;
         }
      }
   }
}
```

7. TRACE TABLE EXAMPLE (Array: [64, 34, 25, 12, 22, 11, 90])

Pass	Comparisons & Swaps	Array After Pass	Largest at Position
Initial	–	64, 34, 25, 12, 22, 11, 90	–
Pass 1	Multiple swaps	34, 25, 12, 22, 11, 64, 90	Index 6 (90)
Pass 2	Multiple swaps	25, 12, 22, 11, 34, 64, 90	Index 5 (64)
Pass 3	Multiple swaps	12, 22, 11, 25, 34, 64, 90	Index 4 (34)
Pass 4	Multiple swaps	12, 11, 22, 25, 34, 64, 90	Index 3 (25)
Pass 5	1 swap	11, 12, 22, 25, 34, 64, 90	Index 2 (22)
Pass 6	0 swap → Stop	11, 12, 22, 25, 34, 64, 90 (Sorted)	–

8. CHARACTERISTICS

- Simple and easy to understand
- In-place sorting (no extra space)
- Stable sort (maintains relative order of equal elements)
- Adaptive (can stop early if already sorted using a flag)
- Not efficient for large datasets

9. COMPLEXITY ANALYSIS

Case	Comparisons	Swaps	Time Complexity
Best Case (Already Sorted)	n - 1	0	O(n)
Average Case	n(n - 1) / 2	~ $n^2/4$	$O(n^2)$
Worst Case (Reversed Order)	n(n - 1) / 2	n(n - 1) / 2	$O(n^2)$

Space Complexity: O(1) (In-place)

10. ADVANTAGES

- Easy to implement
- Requires only constant extra space
- Works well for small or nearly sorted lists
- Stable sorting algorithm

11. DISADVANTAGES

- Very inefficient for large datasets
- Higher number of comparisons and swaps
- Slower performance compared to advanced algorithms

KEY TAKEAWAY

Bubble Sort is a simple comparison-based algorithm where larger elements "bubble" to the end in each pass until the entire list is sorted.

SELECTION SORT

Simple Idea, Powerful Concept!

WHAT IS SELECTION SORT?

Selection Sort is a simple comparison-based sorting algorithm. It repeatedly selects the smallest (or largest) element from the unsorted part and moves it to the sorted part.

KEY IDEA

Find the minimum element from the unsorted part and place it at the beginning.

ALGORITHM

1. Start from the first element.
2. Find the minimum element in the unsorted part.
3. Swap the minimum element with the first element of the unsorted part.
4. Move the boundary of the sorted part one step forward.
5. Repeat steps 2–4 until the array is sorted.

PSEUDOCODE

```
for i = 0 to n-2
   min_idx = i
   for j = i+1 to n-1
     if arr[j] < arr[min_idx]
        min_idx = j
   swap(arr[i], arr[min_idx])
```

COMPLEXITY ANALYSIS

Best Case	$O(n^2)$
Average Case	$O(n^2)$
Worst Case	$O(n^2)$
Space Complexity	$O(1)$

EXAMPLE

Sort the following array in ascending order.

Initial Array : | 64 | 25 | 12 | 22 | 11 |

STEP-BY-STEP DRY RUN

Pass 1 — Find the minimum (11) in the whole array and swap it with the first element.

64	25	12	22	11	→ Swap 64 and 11

11	25	12	22	64	11 is in correct position

Pass 2 — Find the minimum (12) in the remaining unsorted part and swap it with the first element of unsorted part.

11	25	12	22	64	→ Swap 25 and 12

11	12	25	22	64	12 is in correct position

Pass 3 — Find the minimum (22) in the remaining unsorted part and swap it.

11	12	25	22	64	→ Swap 25 and 22

11	12	22	25	64	22 is in correct position

Pass 4 — Find the minimum (25) in the remaining unsorted part and swap it.

11	12	22	25	64	→ Swap 25 and 25 (No change)

11	12	22	25	64	25 is in correct position

ARRAY IS SORTED! | 11 | 12 | 22 | 25 | 64 |

CHARACTERISTICS

- In-place sorting (no extra memory required)
- Not stable (relative order of equal elements may change)
- Simple to understand and implement
- Performs well on small datasets

PROS & CONS

PROS

- ✓ Easy to implement
- ✓ In-place (O(1) space)
- ✓ Works well for small arrays

CONS

- ✗ Inefficient for large datasets ($O(n^2)$)
- ✗ Not stable
- ✗ Many comparisons even if array is sorted

INSERTION SORT

Builds the sorted list one item at a time!

WHAT IS INSERTION SORT?

Insertion Sort is a simple sorting algorithm that builds the final sorted array one item at a time. It takes each element from the unsorted part and inserts it into its correct position in the sorted part.

KEY IDEA

Take one element (key) from the unsorted part and insert it into the correct position in the sorted part.

ALGORITHM

1. Start from the second element (index 1).
2. Take the current element (key).
3. Compare it with elements in the sorted part (left side).
4. Shift all greater elements one position to the right.
5. Insert the key in its correct position.
6. Repeat steps 2–5 until the array is sorted.

PSEUDOCODE

```
for i = 1 to n-1
    key = arr[i]
    j = i - 1
    while j >= 0 and arr[j] > key
        arr[j+1] = arr[j]  // shift right
        j = j - 1
    arr[j+1] = key         // insert key
```

EXAMPLE

Sort the following array in ascending order.

Initial Array : 12 | 11 | 13 | 5 | 6

Sorted Part

Key (Current)

STEP-BY-STEP DRY RUN

Pass 1 (i = 1)

- Key = 11
- Compare with 12 (greater), shift 12 right.
- Insert 11 in its correct position.

12 | 11 | 13 | 5 | 6 — Key = 11

↓

12 | 12 | 13 | 5 | 6

↓

11 | 12 | 13 | 5 | 6 — 11 is inserted at the beginning.

Pass 2 (i = 2)

- Key = 13
- Compare with 12 (smaller).
- No shifting needed.
- Insert 13 after 12.

11 | 12 | 13 | 5 | 6

↓

11 | 12 | 13 | 5 | 6 — 13 is already in correct position.

Pass 3 (i = 3)

- Key = 5
- Compare with 13, 12, 11 (all greater).
- Shift all to the right.
- Insert 5 at the beginning.

11 | 12 | 13 | 5 | 6

↓

11 | 12 | 13 | 13 | 6

↓

11 | 12 | 12 | 13 | 6

↓

11 | 11 | 12 | 13 | 6

↓

5 | 11 | 12 | 13 | 6 — 5 is inserted at the beginning.

Pass 4 (i = 4)

- Key = 6
- Compare with 13, 12, 11, 5.
- Shift 13, 12, 11 to the right.
- Insert 6 after 5.

5 | 11 | 12 | 13 | 6

↓

5 | 11 | 12 | 13 | 13

↓

5 | 11 | 12 | 12 | 13

↓

5 | 11 | 11 | 12 | 13

↓

5 | 6 | 11 | 12 | 13 — 6 is inserted in its correct position.

ARRAY IS SORTED! 5 | 6 | 11 | 12 | 13

COMPLEXITY ANALYSIS

Best Case (Already Sorted)	O(n)
Average Case	$O(n^2)$
Worst Case (Reverse Sorted)	$O(n^2)$
Space Complexity	O(1)

CHARACTERISTICS

- In-place sorting algorithm
- Stable (maintains relative order of equal elements)
- Efficient for small datasets
- Online algorithm (works as data comes in)
- Adaptive (good for nearly sorted data)

PROS & CONS

PROS

- ✓ Simple to understand and implement
- ✓ In-place (O(1) space)
- ✓ Stable
- ✓ Works well for small or nearly sorted arrays

CONS

- ✗ Inefficient for large datasets (O(n^2))
- ✗ More comparisons and shifts in worst case

TIP: Insertion Sort is efficient for small arrays or when the input array is nearly sorted.

RADIX SORT

Sorts numbers digit by digit!

WHAT IS RADIX SORT?

Radix Sort is a non-comparison based sorting algorithm. It sorts numbers digit by digit using a stable sorting technique (like Counting Sort) as a subroutine.

KEY IDEA

Sort by the least significant digit (LSD) first, then next digit, and so on until the most significant digit (MSD).

ALGORITHM

1. Find the maximum number to know the number of digits.
2. For each digit position starting from LSD to MSD:
   - Perform Counting Sort based on that digit.
   - Maintain stability.
3. After the last digit is processed, the array is sorted.

EXAMPLE

Sort the following numbers:

170	45	75	90	802	24	2	66

Pass 1: Sort by Ones place (LSD)

Ones :	0	5	5	0	2	4	2	
After sorting:	90	170	802	2	24	45	75	66

Sorted by ones digit

Pass 2: Sort by Tens place

Tens :	9	7	0	0	2	4	7	6
After sorting:	802	2	24	45	66	170	75	90

Sorted by tens digit

Pass 3: Sort by Hundreds place (MSD)

Hundreds :	8	0	0	0	0	1	0	0
After sorting:	2	24	45	66	75	90	170	802

Sorted by hundreds digit

ARRAY IS SORTED! 2 24 45 66 75 90 170 802

COMPLEXITY

Time Complexity:
$O(d \times (n + k))$
Where n = numbers,
d = number of digits,
k = base (usually 10)

Space Complexity: $O(n + k)$

PROS

- ✓ Very efficient for large numbers.
- ✓ Linear time complexity for fixed number of digits.
- ✓ Stable sorting algorithm.

CONS

- ✗ Not suitable for comparison-based data (e.g., strings without conversion).
- ✗ Requires extra space.
- ✗ Performance depends on the range of digits.

Best for: Sorting large integers or strings with fixed length.

MERGE SORT

Divide, Conquer and Combine!

WHAT IS MERGE SORT?

Merge Sort is a comparison-based sorting algorithm that uses the Divide and Conquer approach.

ALGORITHM

1. Divide the array into two halves.
2. Recursively sort both halves.
3. Merge the two sorted halves into a single sorted array.

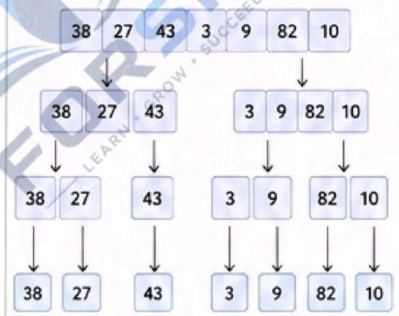
COMPLEXITY

Time Complexity:
Best / Average / Worst: O(n log n)
Space Complexity: O(n)
Stable: Yes

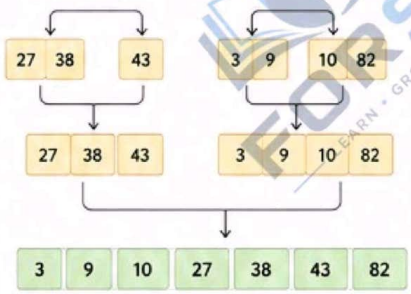
EXAMPLE

Sort the following array: 38 27 43 3 9 82 10

DIVIDE



CONQUER & MERGE



PROS

- ✓ Guaranteed O(n log n) time.
- ✓ Stable sorting.
- ✓ Efficient for large datasets.

CONS

- ✗ Requires extra space O(n).
- ✗ Not an in-place sorting algorithm.

Best for: Large datasets, linked lists, external sorting.

QUICK SORT

Pick a Pivot and Partition!

WHAT IS QUICK SORT?

Quick Sort is a comparison-based sorting algorithm. It picks an element as pivot and partitions the array into two parts: elements less than pivot and greater than pivot.

ALGORITHM

1. Choose a pivot element.
2. Partition the array:
   - Elements < pivot to the left
   - Elements > pivot to the right
3. Recursively apply steps 1–2 to the left and right subarrays.
4. Base case: If subarray has 0 or 1 element, it is already sorted.

EXAMPLE

Sort the following array: 10 7 8 9 1 5

Step 1: Choose pivot = 10 (last element)

10 7 8 9 1 5 → Partition → 7 8 9 1 5 10 (Pivot in correct place)

Step 2: Sort left part [7, 8, 9, 1, 5], pivot = 5

7 8 9 1 5 → Partition → 1 5 9 7 8

Step 3: Sort left part [1], pivot = 1

1 1 → Partition

Step 4: Sort right part [9, 7, 8], pivot = 8

9 7 8 → Partition → 7 8 9 (Pivot in correct place)

Final Sorted Array: 1 5 7 8 9 10

COMPLEXITY

Time Complexity:
Best Case: O(n log n)
Average Case: O(n log n)
Worst Case: O(n^2)
Space Complexity:
O(log n) average (recursion stack)

PROS

- ✓ In-place sorting (less extra space).
- ✓ Usually faster in practice.
- ✓ Efficient for large datasets.

CONS

- ✗ Worst case O(n^2) when pivot is worst (e.g., already sorted).
- ✗ Not stable.
- ✗ Performance depends on pivot choice.

Best for: General purpose sorting, in-memory sorting.

Tip: Choosing a good pivot (like random or median-of-three) improves performance.