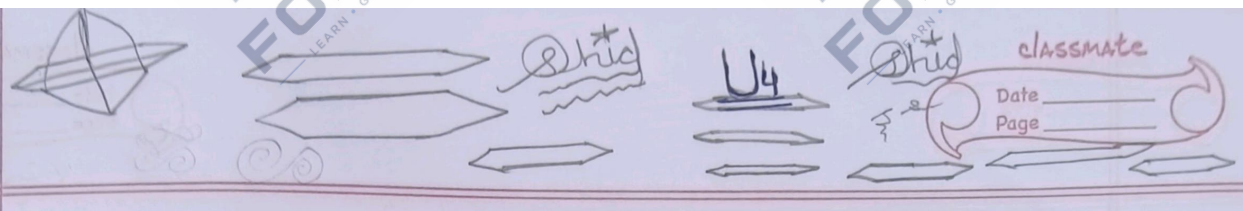
Unit - IV

DATA STRUCTURES
ALGORITHMS
COMPUTER NETWORKS
DATABASE SYSTEMS
#include <iostream>
using namespace std;
int main()
{
cout << "Hello, World!";
return 0;
}



* Searching -

A searching is the process of finding element or value from the collection of data items such as an array, list or files.

Eg -
If you have a list of numbers [10, 20, 30, 40, 50, 60, 70] and you have to find whether 40 exists or not This is the searching operation.

Types of Searching Techniques.

Searching techniques are mainly classified into two types -

1) Sequential (Linear) Search
2) Binary Search

i) Sequential Search - (Linear) Search.

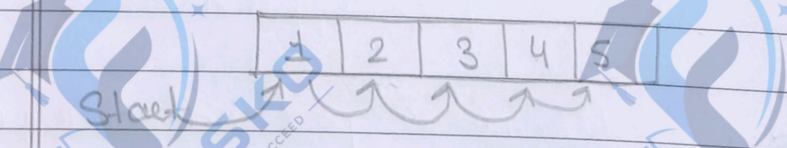
Linear Search is a searching technique in which each element of data structure is checked one by one until the required element (key) is found or the list ends.

It works on unsorted or sorted data no special order required.
It is the simplest searching technique.

Working Principle.

i) Searching starts from the first element
ii) Each element compared with key element.
iii) Searching continue until -

Element is found, or All elements are chec



Time Complexity -

- Best Case: O(1)
- Wort Case: O(n)
- Average Case: O(n)

Algorithm:- Linear Search

Input : Array A of size n, element KEY

Step1 : Set i = 0

Step2 : While i < n

- If A[i] == KEY
  print "Element found at position i".
  Stop

Else
  i = i + 1

Step3 : If i == n
  print "Element not found".

Step4 : Stop

Example-

Array : A = {10, 20, 30, 40, 45, 60};

Key : 30

Step - by - Step

Compare 10 → Not match
Compare 20 → Not match
Compare 30 → Match found at index 2

Result :
Element found at position 2

ii) Binary Search -

Binary Search is the efficient and searching technique used to find an element (key) in a sorted array.

Instead of checking element one-by-one like linear search, it repeadedly halves the searching range. It divids the searc interval into two halves.

Condition -

Binary search works only on sorted data (ascending or descending)

Working Principle.

i) Compare the key with the middle element of the array.

ii) If the key is equal to middle element searching is success-full.

iii) If the key is smaller than middle element then search in the left half.

iv) If the key is greater than middle eleme then search in right half.

v) Repeat until the element is found or the search space becomes empty.

Binary Search Algorithm -

Input : Sorted array A, size n, element KEY

Step 1: Set low = 0 , high = n-1

Step 2: While low ≤ high

- mid = (low + high) / 2
- If A[mid] == KEY,
  print "Element found at position mid".
  Stop
- Else if KEY < A[mid]
  high = mid - 1
- Else
  low = mid + 1

Step 3: If low > high

print " element not found "

Step 4: Stop

Example-

Array (sorted):

A = {10, 20, 30, 40, 50};

Key to search: 20

Steps:

middle element = 30

If Key < mid, search in left half
If Key > mid, search in right half.

Therefore,
Key = 20

Key < mid
20 < 30

then search left half

Result:

Element found at index position 1.

Time Complexity-

- Best Case: O(1)
- Worst Case: O(log n)
- Average Case: O(log n)

Sorting

Definition: Sorting is the process of arranging the elements in a particular order (Ascending or Descending).

Types of Sorting

1. Bubble Sort

Idea: Compare adjacent elements and swap if they are in wrong order.

Example: [5, 1, 4, 2, 8]

Time Complexity:

- Best Case : $O(n)$
- Average Case : $O(n^2)$
- Worst Case : $O(n^2)$

---

2. Selection Sort

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

Example: [64, 25, 12, 22, 11]

Time Complexity:

- Best Case : $O(n^2)$
- Average Case : $O(n^2)$
- Worst Case : $O(n^2)$

---

3. Insertion Sort

Idea: Take one element and place it in its correct position in the sorted part.

Example: [12, 11, 13, 5, 6]

Time Complexity:

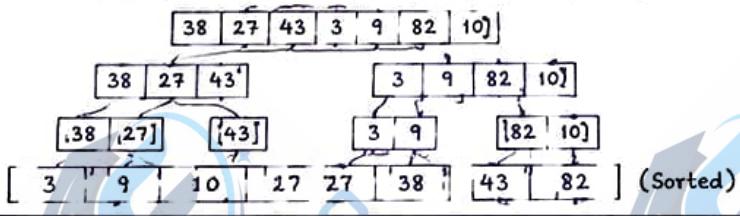
- Best Case : $O(n)$
- Average Case : $O(n^2)$
- Worst Case : $O(n^2)$

---

4. Merge Sort

Idea: Divide the array into halves, sort them and merge.

Example: [38, 27, 43, 3, 9, 82, 10]



Time Complexity:

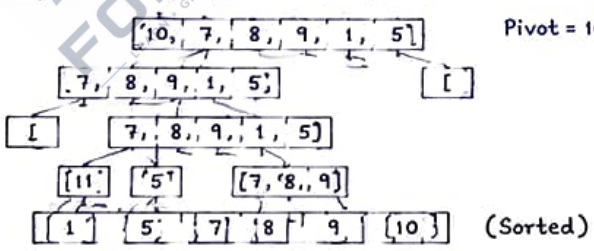
- Best Case : $O(n \log n)$
- Average Case : $O(n \log n)$
- Worst Case : $O(n \log n)$

---

5. Quick Sort

Idea: Choose a pivot, place it in correct position and sort left and right parts recursively.

Example: [10, 7, 8, 9, 1, 5]



Time Complexity:

- Best Case : $O(n \log n)$
- Average Case : $O(n \log n)$
- Worst Case : $O(n^2)$

*** Note:**

1. Stable Sorts: Bubble, Insertion, Merge
2. Unstable Sorts: Selection, Quick

Stable → Order of equal elements remains same.

* Bubble Sort *

1. Definition : 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.

> It "bubbles" the largest element to the end in each pass.

Key Points :

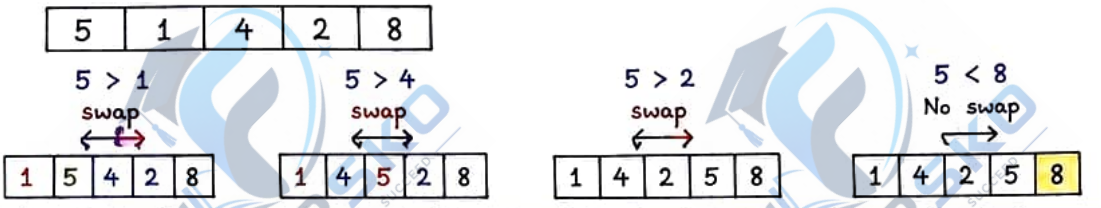
- In each pass, the largest element comes to its correct position.
- After i^{th} pass, last i elements are sorted.
- It is a stable sort.
- In-place sorting algorithm.

2. Algorithm :

1. Compare the first and second elements.
2. If first > second, swap them.
3. Move to the next pair and repeat Step 1 and 2.
4. After each pass, the largest element will be at the end.
5. Repeat for remaining unsorted elements.

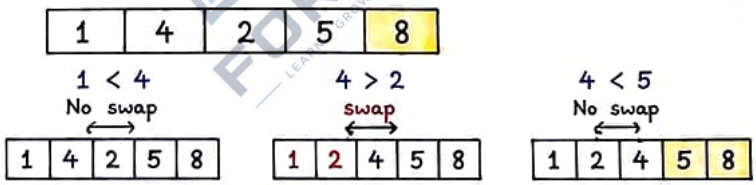
3. Example : Sort the array [5 , 1 , 4 , 2 , 8] in ascending order.

Pass 1 :



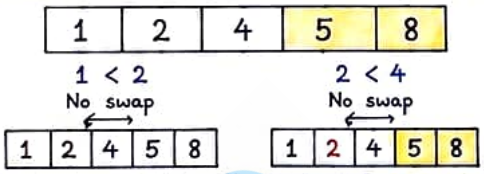
8 is in correct position.

Pass 2 :



5 is in correct position.

Pass 3 :



4 is in correct position.

Pass 4 :

2 is in correct position.

Pass 5 :

Array is sorted. ☺

Time Complexity :

- Best Case : O(n) (Already sorted with optimization)
- Average Case : $O(n^2)$
- Worst Case : $O(n^2)$
- Space Complexity : O(1)

Note :

- Bubble Sort is easy to understand but not efficient for large data.
- It is useful for small datasets or nearly sorted arrays.

* Selection Sort *

1. Definition :

Selection Sort is a comparison sorting algorithm. It works by dividing the list into two parts :

- Sorted part (at the beginning)
- Unsorted part (at the end)

> It repeatedly selects the minimum element from the unsorted part and puts it at the beginning of the unsorted part.

Key Points :

- In each pass, the minimum element from the unsorted part is selected and placed at the beginning.
- The sorted part grows from left to right.
- It does the minimum number of swaps.
- It is not suitable for large datasets.

2. Algorithm :

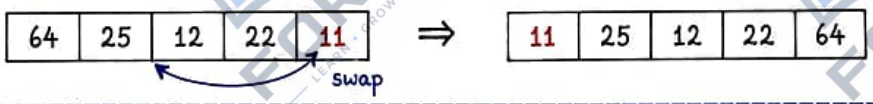
1. Set the first element as the minimum.
2. Compare it with the rest of the elements in the unsorted part.
3. If any element is smaller, update the minimum.
4. After checking all elements, swap the minimum element with the first element of the unsorted part.
5. Move the boundary of the sorted part one step ahead.
6. Repeat steps 1 to 5 until the whole array is sorted.

Time Complexity :

- Best Case : $O(n^2)$
- Average Case : $O(n^2)$
- Worst Case : $O(n^2)$

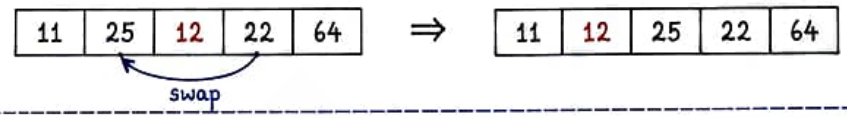
3. Example : Sort the array [64 , 25 , 12 , 22 , 11] in ascending order.

Pass 1 : Find minimum in [64 , 25 , 12 , 22 , 11] → minimum = 11



11 is in correct position.

Pass 2 : Find minimum in [25 , 12 , 22 , 64] → minimum = 12

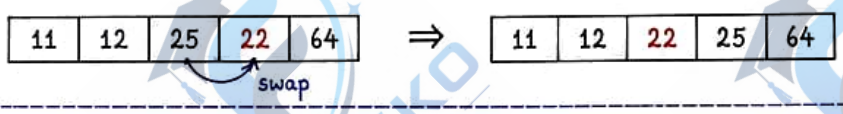


12 is in correct position.

Space Complexity :

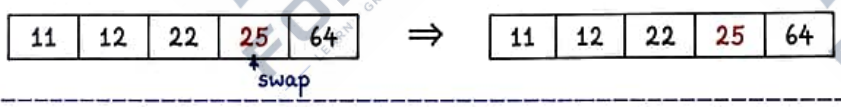
- $O(1)$

Pass 3 : Find minimum in [25 , 22 , 64] → minimum = 22



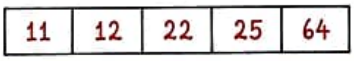
22 is in correct position.

Pass 4 : Find minimum in [25 , 64] → minimum = 25



25 is in correct position.

Pass 5 : Only one element [64] is left.
Array is already sorted.



64 is in correct position.

> **Note :**
> - Selection Sort is In-place (no extra space).
> - It is not stable (relative order of equal elements may change).
> - It is not adaptive (best case is also same as worst case).

* Insertion Sort *

1. Definition :

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

Idea : Always keep the left part of the array sorted. Insert the next element in its correct position.

2. Algorithm :

1. Start from the second element (index 1).
2. Take the current element (key).
3. Compare it with the 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. Move to the next element and repeat Steps 2 to 5 until the whole array is sorted.

Key Points :

- Builds the sorted array one element at a time.
- Efficient for small or nearly sorted arrays.
- In-place sorting algorithm.
- Stable sort.

Time Complexity :

- Best Case : $O(n)$
- Average Case : $O(n^2)$
- Worst Case : $O(n^2)$

Space Complexity :

- $O(1)$

3. Example : Sort the array [12, 11, 13, 5, 6] in ascending order.

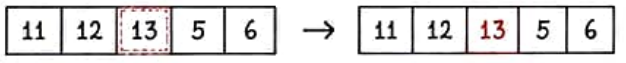
Initial array :

Pass 1 : Key = 11

11 < 12, shift 12 to right and insert 11.

After Pass 1 : 11, 12 | 13, 5, 6 (sorted | unsorted)

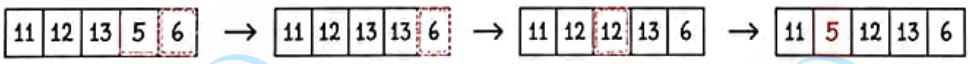
Pass 2 : Key = 13



13 > 12, already in correct position.

After Pass 2 : 11, 12, 13 | 5, 6 (sorted | unsorted)

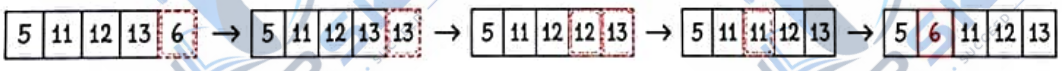
Pass 3 : Key = 5



5 < 13, shift 13 right. 5 < 12, shift 12 right. 5 < 11, shift 11 right and insert 5.

After Pass 3 : 5, 11, 12, 13 | 6 (sorted | unsorted)

Pass 4 : Key = 6



6 < 13, shift 13 right. 6 < 12, shift 12 right. 6 < 11, shift 11 right. 6 > 5, stop and insert 6.

After Pass 4 : 5, 6, 11, 12, 13 (sorted)

Final Sorted Array : [5, 6, 11, 12, 13]

Note :
- Insertion Sort is simple and easy to implement.
- It works well for small data or nearly sorted data.
- It is adaptive (performs better when the array is already partially sorted).