DATA STRUCTURES

LECTURE 1: SEARCHING AND SORTING ALGORITHMS

MADE BY: KEVIN HARVEY

Prerequisite (Must learn before entering)

Data Structures PREREQUISITE

C++ From Scratch

Made By: Kevin Harvey

Definition Of Searching

A *record* in data structures is usually represented as:

Key + Data

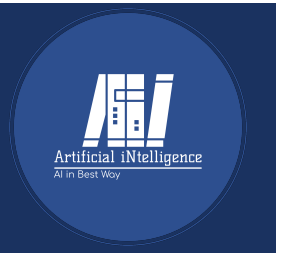
📘 **Example:**

A person's info → SSN (Key) + name, address, phone number, etc.

When we search for something (like a person), we use their **Key** — not the entire record.

However, when converting an algorithm to an actual program, we must also handle the **data** linked with that key.

Types Of Searching Algorithms



- Linear (Sequential) Search
- Binary Search

⚡ 1. Linear Search (Sequential Search)

This is the **simplest** and most basic searching technique.

The “scroll through the list until you find it” of algorithms.

💡 Concept:

Whether your list is **ordered or unordered**, Linear Search just checks **each element one by one** from the beginning.

👉 If it finds the target → **Search successful** ✅

👉 If it reaches the end without finding → **Search failed** ❌

⚡ 1. Linear Search Algorithm

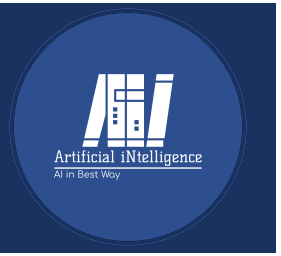
🧾 Algorithm (Python)

```
key = 30
list = [10,20,30,40,50,60,70,80,90,100]
for i in list:
    if i == key:
        print("Element found")
        break
else:
    print("Element not found")
```

Or step-by-step 🔢

1. Start from the **first element** in the array.
2. Compare it with the **target key**.
3. If it matches → returns **0 internally (Element Found)**.
4. If not → move to the next element.
5. If no match till the end → returns **1 internally (Element not found)**.

⚡ 1. Linear Search Algorithm



💻 C++ Implementation

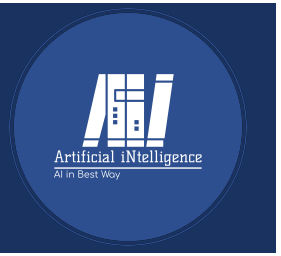
```cpp
#include <iostream>
using namespace std;

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

int main() {
    int arr[] = {5, 2, 8, 4, 9};
    int n = 5, key = 8;
    int result = linearSearch(arr, n, key);

    if (result != -1)
        cout << "Element found at index " << result;
    else
        cout << "Element not found.";
}
```

📊 Linear Search Advantages



📊 Advantages:

✅ **Simplicity:** easy to understand and implement.

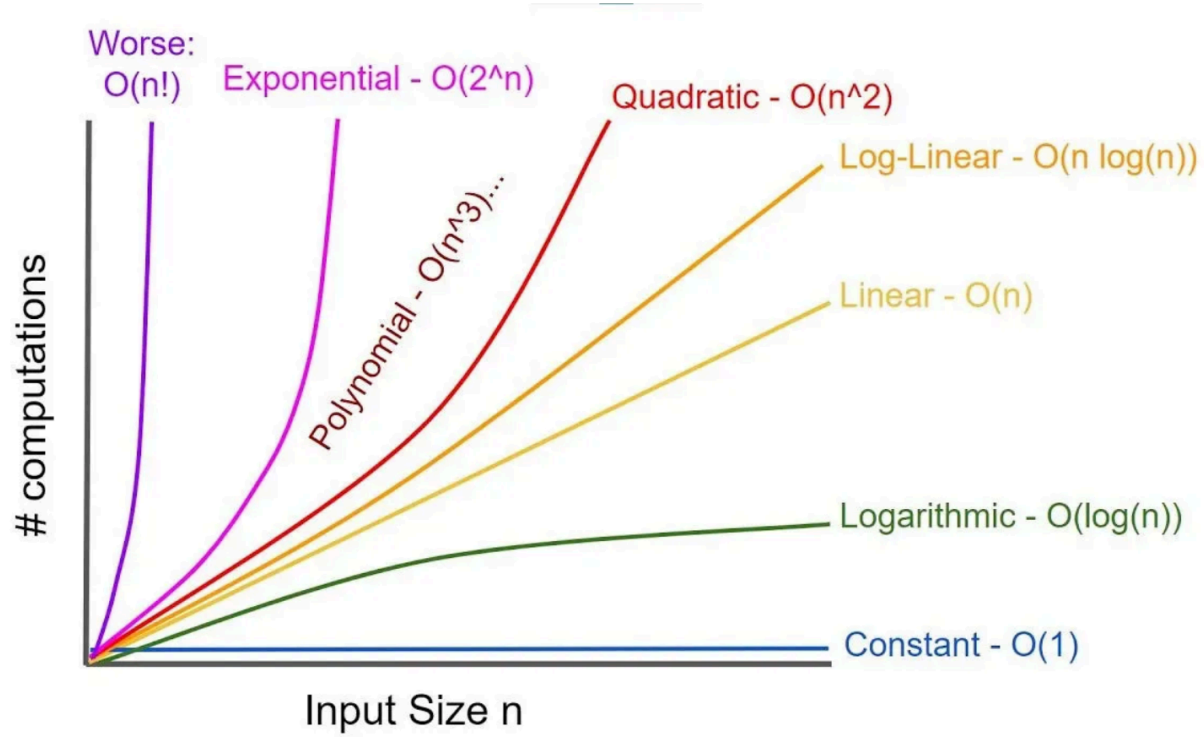
✅ **Flexibility:** doesn't require the array to be sorted.

⚠️ Linear Search Disadvantages

⚠️ Disadvantages:

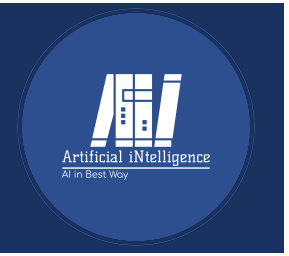
🚫 **Poor efficiency** — takes too many comparisons for large lists.

🚫 **Linear scaling:** performance grows *linearly* with input size (O(n)).



🚫 **Slower** than other searching algorithms like Binary Search.

Linear Search Example



🧮 Example:

Goal: Search for `-86`

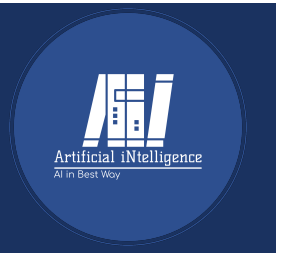
Array: `[-23, 97, 18, 21, 5, -86, 64, 0, -37]`

💬 Process:

- Compare `23` → no match
- Compare `97` → no match
- Compare `18` → no
- Compare `21` → no
- Compare `5` → no
- Compare `86` → 🎯 FOUND!

So, the **element is found successfully.**

🧭 2. Binary Search



⚙️ Concept:

Binary Search is the **faster algorithm** — but it only works if your data is **sorted**.

Let's say we have `n` records ordered like this:

`x1 < x2 < x3 < ... < xn`

`[10,20,30,40,50,..., 100]`

Now, when we want to find an element `x`:

- If it exists → find index `i` where `a[i] = x`
- If not → return 0 (unsuccessful search)

💡 How Binary Search Works:

Instead of checking one by one, it **jumps straight to the middle**.

1. Find the **middle** index → `mid = (low + high) / 2`
2. Compare:
   - If `x == a[mid]` → 🎯 found!
   - If `x < a[mid]` → look in **left half**
   - If `x > a[mid]` → look in **right half**
3. Repeat until found or until `low > high` (means not found)

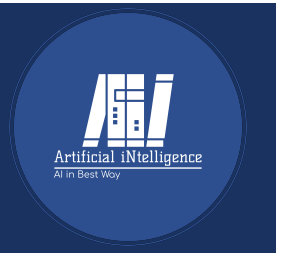
2. Binary Search Example

📘 Example:

Find the number **42** in the sorted list below 👇

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Value	-4	2	7	10	15	20	22	25	30	36	**42**	50	56	68	85	92	103

2. Binary Search C++ Implementation



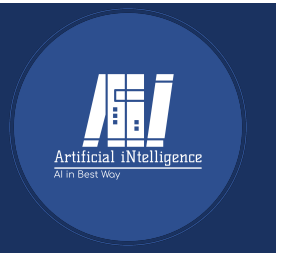

```cpp
#include <iostream>
using namespace std;

int binarySearch(int arr[], int n, int key) { // declaring function that searches for an element using binary search
    int low = 0, high = n - 1; // initializing low and high
    while (low <= high) { // while loop
        int mid = (low + high) / 2; // finding middle
        if (arr[mid] == key) // if element is found
            return mid; // return index
        else if (arr[mid] > key) // elif element is less than middle
            high = mid - 1; // decrease high
        else // if element is greater than middle
            low = mid + 1; // increase low
    }
    return -1;
}

int main() {
    int arr[] = {2, 4, 6, 8, 10, 12}; // initializing array
    int n = 6, key = 10; // initializing n and key
    int result = binarySearch(arr, n, key); // calling function

    if (result != -1) // if element is found
        cout << "Element found at index " << result; // print index
    else // elif element is not found
        cout << "Element not found."; // print not found
}
//output: Element found at index 4
```

⚠️ 2. Binary Search Disadvantages



1. 🧾 Requires sorting first (no unsorted arrays allowed).
2. 🧩 Linear search may be better for small datasets.
3. 💥 Not ideal for structures like linked lists (since it needs “jump access” to middle elements).

🌀 1. Sorting Algorithms

Sorting is the art of taking a **messy array** and turning it into something *actually usable*

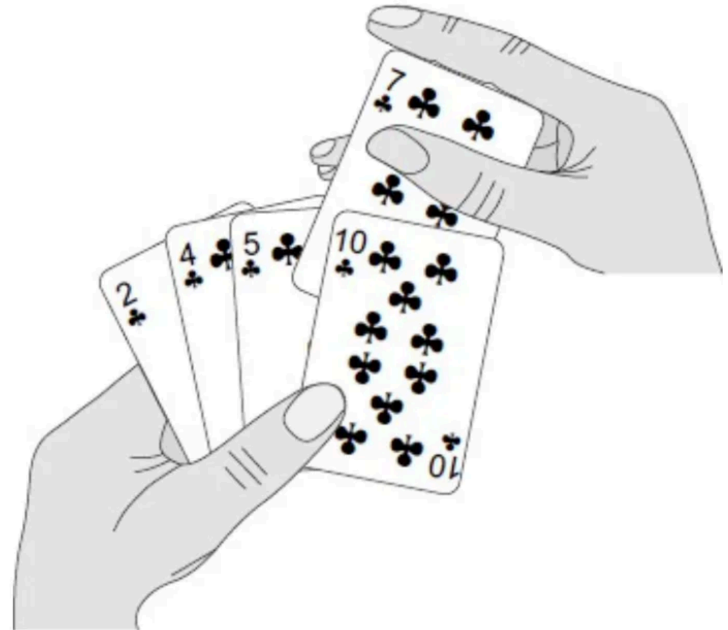
It’s like organizing a Spotify playlist or a deck of cards.

1. Insertion Sort

💡 Idea Behind Insertion Sort

Imagine you're holding **playing cards** 🎴 and picking them up **one by one**.

Every time you pick a new card, you **insert** it into your hand so that your cards are **always sorted**.

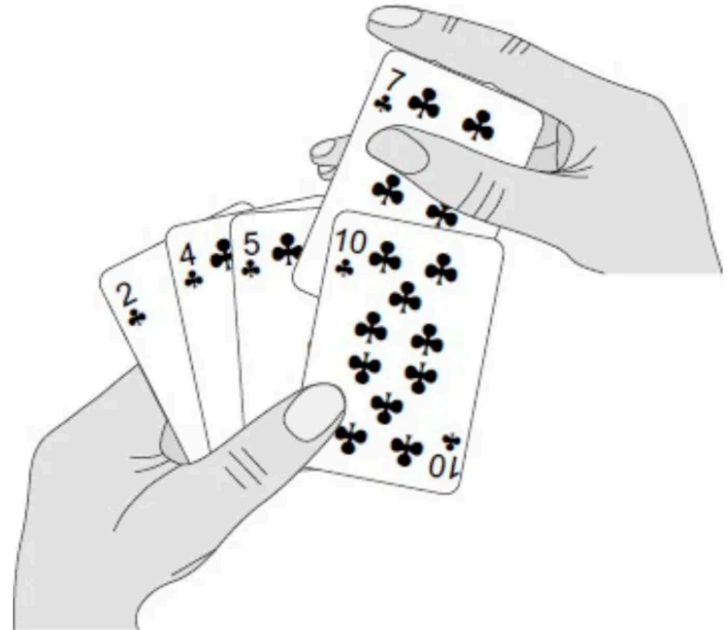


1. Insertion Sort

💡 Idea Behind Insertion Sort

Imagine you're holding **playing cards** 🎴 and picking them up **one by one**.

Every time you pick a new card, you **insert** it into your hand so that your cards are **always sorted**.



🧩 1. Insertion Sorting Process Explained

Let's say you'll pick up cards with these numbers:

`[57, 48, 79, 65, 52]`

👉 Start by picking the **first card** (57). It's automatically "sorted" (since it's alone).

👉 Pick the next one (48) and put it in the **correct spot** before 57.

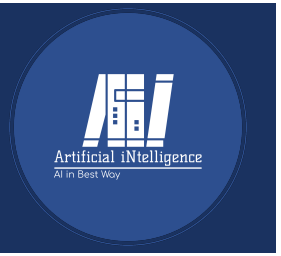
👉 Then take the next one (79) — it's already bigger, so just append.

👉 Keep going till your "hand" (the left side of the array) is always sorted.

At the end:

🎯 Sorted array = `[48, 52, 57, 65, 79]`

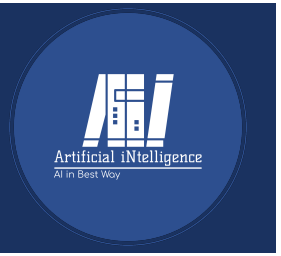
🧠 1. Insertion Sorting Algorithm



1. Start from the **second element** (index 1).
2. Compare it with all **previous elements**.
3. **Shift** every element greater than it *one position forward*.
4. Insert the current element in the correct position.
5. Repeat till the entire array is sorted.

🌀 The process runs until there's nothing left unsorted.

1. Insertion Sorting Algorithm C++ Code




```cpp
#include <iostream>
using namespace std;

void insertionSort(int arr[], int n) {
    for (int i = 1; i < n; i++) {
        int key = arr[i];
        int j = i - 1;

        // Move elements greater than key one step ahead
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];  // shifting
            j--;
        }
        arr[j + 1] = key;
    }
}

int main() {
    int arr[] = {9, 5, 1, 4, 3};
    int n = 5;

    insertionSort(arr, n);

    cout << "Sorted array: ";
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
}
```

1. Insertion Sort Algorithm Notes

- It's efficient for **small** or **almost sorted** datasets.
- It uses **shifting**, not swapping.
- Time complexity:
  - 🟢 Best case: O(n) → already sorted.
  - 🔴 Worst case: O(n²) → completely reversed order.

🎮 Visualization Example:

Let's sort `[32, 51, 27, 85, 66, 23, 13, 57]`

First pass: compare 51 with 32 → stays same.

Second pass: compare 27 with [32,51] → goes to the first place.

Third pass: 85 → stays last (it's biggest).

Fourth pass: 66 → slides before 85.

...and so on until you get a fully sorted list 🧩

🫧 2. Bubble Sorting

🫧 Concept:

Bubble Sort is like bubbles in soda 🥤 — the **biggest values "bubble up" to the top** after each round.

🧠 Algorithm (Step-by-Step)

1. Start at the **beginning** of the array.
2. Compare each **pair of adjacent elements**.
3. If the first is greater → **swap** them 🔁.
4. At the end of each pass, the **largest element moves to the end**.
5. Repeat until the entire array is sorted.

2. Bubble Sorting

A1 < A2 Not Altered

32	51	27	85	66	23	13	57

A2 > A3 Altered

32	51	27	85	66	23	13	57

A3 < A4 Not Altered

32	27	51	85	66	23	13	57

A4 > A5 Altered

32	27	51	85	66	23	13	57

2. Bubble Sorting

A5 > A6 Altered

32	27	51	66	85	23	13	57

A6 > A7 Altered

32	27	51	66	23	85	13	57

A7 > A8 Altered

32	27	51	66	23	13	85	57

32	27	51	66	23	13	57	85

🫧 2. Bubble Sorting

⚖️ Bubble Sort Properties

Case	Behavior	Time Complexity
🟢 **Best Case**	Array already sorted	O(n)
🟠 **Average Case**	Random array	O(n²)
🔴 **Worst Case**	Reversed array	O(n²)

✅ Works fine for small datasets

❌ Not efficient for large ones

🔁 Swapping vs Shifting:

Let's clear this up once and for all 👇

Operation	Meaning	Used In
Swapping	Exchange two elements directly	🫧 Bubble Sort
Shifting	Move elements to make room for insertion	🧍 Insertion Sort

---

🧩 📊 Comparison Table

🔢 Algorithm	🔍 Type	💡 Works On	⚙️ Method	🧠 Best Case	💀 Worst Case	⏱️ Time Complexity	🧱 Space Complexity	✅ Advantages	⚠️ Disadvantages
Linear Search	Searching	🔶 Sorted or Unsorted	Compare each element sequentially	O(1) (found first)	O(n) (found last/not found)	O(n)	O(1)	✅ Easy to implement✅ Works on any list	❌ Super slow for big data❌ Linear performance
Binary Search	Searching	🔷 Sorted lists only	Divide list into halves repeatedly	O(1)	O(log n)	O(log n)	O(1)	✅ Very fast✅ Fewer comparisons	❌ Requires sorted data❌ Not for linked lists
Insertion Sort	Sorting	🔷 Any (works best on small or nearly sorted data)	Inserts each element into its correct place	O(n)	O(n²)	O(n²)	O(1)	✅ Simple logic✅ Efficient on small datasets	❌ Slow for large lists❌ Quadratic time complexity
Bubble Sort	Sorting	🔷 Any	Repeatedly swaps adjacent elements	O(n)	O(n²)	O(n²)	O(1)	✅ Easy to code✅ Detects already sorted data	❌ Painfully slow❌ Too many swaps

Thank you