CLASS NOTES

UNIT–I: Algorithmic Thinking & Complexity

- What is an algorithm? Characteristics and examples
- Types of algorithms (brute force, greedy, recursive, divide and conquer)
- Introduction to Big-O notation
- Time and space complexity with examples
- Practice: Designing pseudocode and estimating complexity

1. What is an Algorithm?

An **algorithm** is a **clear, finite, step-by-step method** to solve a problem or perform a task **correctly and efficiently**.

> In industry, an algorithm is not just "something that works" —
> it is **something that works fast, scales well, and is maintainable**.

Example

Making Tea Algorithm

1. Take water
2. Boil water
3. Add tea leaves
4. Add sugar
5. Add milk
6. Boil
7. Serve

This is an algorithm because:

- Steps are **ordered**
- Each step is **clear**
- It **terminates**
- It gives a **desired output**

Programming Example (Python mindset)

Problem: Find the maximum number in a list

```
arr = [3, 7, 2, 9, 4]

max_val = arr[0]
for num in arr:
    if num > max_val:
        max_val = num

print(max_val)
```

This logic is an **algorithm**, not just code.

Why Algorithms Matter in Real Industry

In real projects:

- Millions of users
- Huge datasets
- Limited memory
- Strict response time (milliseconds)

A bad algorithm =
Slow APIs
App crashes
Higher cloud cost
Poor user experience

2. Characteristics of a Good Algorithm

Every **good algorithm** must satisfy these:

1. Input

Takes zero or more inputs
Example: list of numbers

2. Output

Produces at least one output
Example: maximum number

3. Definiteness

Steps must be clear and unambiguous

“Process data properly”
“Sort list using merge sort”

4. Finiteness

Must terminate after finite steps
Infinite loop = bad algorithm

5. Effectiveness

Each step should be simple and executable

6. Efficiency (Industry-critical)

Uses **minimum time & memory**

3. Types of Algorithms (With Intuition + Industry Use)

1. Brute Force Algorithms

Try **all possible solutions** and pick the correct one.

Example (Beginner level)

Check if a number exists in a list

```
def search(arr, target):
    for x in arr:
        if x == target:
            return True
    return False
```

Industry Reality

- Very easy to write
- Very bad for large data

Real-Life Use Case

Small input size
Prototyping
Proof of concept

- Never used for large-scale systems

2. Greedy Algorithms

Idea

At every step, **choose the best local option**, hoping it leads to global optimum.

Example: Coin Change (Intuition)

Suppose coins: ₹10, ₹5, ₹2, ₹1
Amount = ₹28

Greedy approach:

- Take ₹10 → remaining 18
- Take ₹10 → remaining 8
- Take ₹5 → remaining 3
- Take ₹2 → remaining 1
- Take ₹1 → remaining 0

Industry Use Cases

- Network routing
- CPU scheduling
- Resource allocation
- Load balancing

Important Teaching Point

- Greedy **does NOT always work**, but when it works, it is **very fast**.

3. Recursive Algorithms

A problem solved by **breaking it into smaller subproblems of the same type**.

Simple Example: Factorial

```
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n - 1)
```

Key Components

- **Base case** (stopping condition)
- **Recursive case**

Industry Use Cases

- Tree traversal
- File system traversal
- Parsing expressions
- Backtracking problems

Industry Insight

Recursion is powerful but:

- Uses stack memory
- Can cause stack overflow
- Often converted to iteration in production

4. Divide and Conquer Algorithms

Idea

1. Divide problem into smaller parts
2. Solve them independently
3. Combine results

Example: Merge Sort (High Level)

- Divide array into halves
- Sort each half
- Merge sorted halves

Industry Use Cases

- Sorting (merge sort, quick sort)
- Search engines
- Big data processing
- Parallel computing

Why Industry Loves It

- Highly scalable
- Efficient for large data

- Easy to parallelize

4. Introduction to Big-O Notation

Big-O tells us **how algorithm performance grows** as input size increases.

> Industry question is NOT:
> “How fast is your code on my laptop?”
>
> Industry question IS:
> “What happens when data becomes 10× or 1000×?”

Common Big-O Complexities

Big-O	Meaning	Example
$O(1)$	Constant time	Access array element
$O(n)$	Linear time	Loop through array
$O(n^2)$	Quadratic	Nested loops
$O(\log n)$	Logarithmic	Binary search
$O(n \log n)$	Efficient sorting	Merge sort

Example: O(1)

```
def get_first(arr):
    return arr[0]
```

No matter list size → same time.

Example: O(n)

```
def sum_array(arr):
    total = 0
    for x in arr:
        total += x
    return total
```

Example: O(n²)

```
for i in range(n):
    for j in range(n):
        print(i, j)
```

Dangerous for large input.

5. Time Complexity (With Real Intuition)

Time complexity measures **how execution time grows** with input size.

Rule of Thumb (Industry)

- Ignore constants
- Ignore lower-order terms
- Focus on **dominant term**

Example:

$5n^2 + 3n + 100 \rightarrow O(n^2)$

6. Space Complexity

Memory used by algorithm **apart from input**.

Example: O(1) Space

```
def find_max(arr):
```

```
    max_val = arr[0]
    for x in arr:
        if x > max_val:
            max_val = x
    return max_val
```

Uses constant extra space.

Example: O(n) Space

```
def copy_array(arr):
    new_arr = []
    for x in arr:
        new_arr.append(x)
    return new_arr
```

Industry Insight

Often:

- Trade time for space
- Or space for time
  Examples:
- Caching
- Memoization
- Precomputed results

7. Practice Section (Must-Do for Students)

1. Pseudocode Practice

Problem: Find sum of first N natural numbers

Pseudocode

```
START
READ N
SET sum = 0
FOR i from 1 to N
   sum = sum + i
PRINT sum
END
```

2. Complexity Estimation Practice

Code	Time Complexity
Single loop	$O(n)$
Nested loop	$O(n^2)$
Loop + recursion	Depends
Binary search	$O(\log n)$

Are you able to answer below questions?

- Can this scale?
- What if data grows 100×?
- Can we optimize?
- Can we trade space for time?