🧠 Algorithm Analysis & Design — Quick Revision Sheet

Lecture 1 — Introduction to Algorithms

Algorithm:
A *well-defined sequence of steps* that transforms **input** → **output**.

Properties of an Algorithm:

- Input (from a defined set)
- Output (clearly specified result)
- Definiteness (each step clear)
- Correctness (gives right results)
- Finiteness (must end)
- Effectiveness (each step doable)
- Generality (works for similar problems)

Basic Commands:

Command	Syntax	Meaning
Assignment	`X = 5Y + 16`	Stores value in memory
Input	`Get X`	Take value from user
Output	`Give X`	Display value

Note:
Assignment (`A = A + 3`) ≠ Mathematical equality — in CS, it updates the variable's value.

Algorithm Structure

1. Understand problem
2. Identify *Givens* (inputs)
3. Identify *Results* (outputs)
4. Name the algorithm
5. Write method using `Get`, `int`, `Give`

Example – Sum of Three Numbers

Get N1
Get N2
Get N3
int Total = N1 + N2 + N3
Give Total

Tracing an Algorithm

Used to verify correctness.

1. Number each line.
2. Make a table for variables.
3. Execute step-by-step.
4. Update values and check outputs.

Flowchart Symbols

Shape	Meaning
Oval	Start / End

Rectangle	Process
Parallelogram	Input / Output
Diamond	Decision
Arrows	Flow of control

Lecture 2 — Complexity of Algorithms

Why analyze algorithms?
When multiple algorithms solve the same problem, choose based on:

1. Ease of implementation
2. Running time
3. Memory usage

We focus on **time** and **space complexity**.

Linear Search Example

Find if `3` exists in list `A[1...n]`.

for i = 1 to n
  if A[i] == 3 → found

Observation:

- Time depends on **list size (n)**
- Position of element affects runtime

Cases:

Case	Meaning	Example

Best	Fewest steps	Found first
Worst	Most steps	Not found
Average	Middle number	Random position

Complexity Terms

- **Time Complexity:** number of steps to complete
- **Space Complexity:** memory required

Asymptotic Notation

Used to describe **growth rate** of running time.

Notation	Meaning	Example
O (Big Oh)	Upper bound → Worst case	$O(n^2)$
Ω (Big Omega)	Lower bound → Best case	$\Omega(n)$
Θ (Big Theta)	Tight bound → Average case	$\Theta(n \log n)$

Think:
O = “At most this slow”
Ω = “At least this fast”
Θ = “Typical growth rate”

Lecture 3 — Counting Operations

To analyze runtime, **count primitive operations**:

- Assignments

- Arithmetic (+, −, *, /)
- Comparisons (<, >)
- Function calls
- Loop iterations

Example 1 — Sum of n elements

```
s = 0;
for (i = 0; i < n; i++) {
    s = s + A[i];
}
return s;
```

Total Operations: ~2n + 2
→ **O(n)**

Example 2 — Add Two n×n Matrices

```
for i = 0 to n-1
  for j = 0 to n-1
    C[i,j] = A[i,j] + B[i,j];
```

Total: n^2
→ **$O(n^2)$**

Example 3 — Multiply Two n×n Matrices

```
for i = 0 to n-1
  for j = 0 to n-1
    for k = 0 to n-1
      C[i,j] += A[i,k] * B[k,j];
```

Total: n^3
→ **$O(n^3)$**

Loop Patterns

Code	Iterations	Complexity
`for(i=0; i<n; i++)`	n	O(n)
`for(i=1; i<n; i+=2)`	n/2	O(n)
Nested Loops	n × n	$O(n^2)$
Triple Nested	n × n × n	$O(n^3)$

Quick Review

Algorithm	Example	Complexity
Linear Search	Check every element	O(n)
Binary Search	Divide by 2 each time	O(log n)
Bubble Sort	Nested loops	$O(n^2)$
Matrix Addition	Double loop	$O(n^2)$
Matrix Multiplication	Triple loop	$O(n^3)$

🔥 1-Day Crash Study Plan

1. Read this summary slowly (≈40 min).
2. Trace one algorithm manually.
3. Memorize O, Ω, Θ notation meanings.
4. Understand why nested loops → $O(n^2)$.
5. Skim this again before bed.

⚡ Bonus: Big-O Growth (small → large)

$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n)$

Tip: Always look for patterns in loops — they reveal the time complexity faster than counting steps.