

Notes n Hindi

SUBJECT

ALGORITHMS

PREPARED BY

TEAM NOTESINHINDI

<https://notesinhindi.com/>

2026 Edition

Unit 1: Fundamentals of Algorithms

Introduction to Algorithms, Programming Models (Pseudocode, Flowcharts, Iterative vs Recursive), aur Algorithm Analysis (Time/Space Complexity, Asymptotic Notations, Best/Worst/Average Case)

Introduction

Computer Science में किसी भी problem को solve करने से पहले यह ज़रूरी होता है कि उसे step-by-step तरीके से सोचा जाए — यही काम **Algorithm** करता है। इस unit में हम algorithms को define करना, उन्हें represent करने के तरीके, aur उनकी performance को measure करना सीखेंगे।

Is Unit ki Structure:

- 1) Introduction to Algorithms:** Definition, Characteristics, Importance, Steps in Algorithm Design
- 2) Programming Models:** Pseudocode, Flowcharts, Iterative vs Recursive
- 3) Algorithm Analysis:** Time/Space Complexity, Asymptotic Notations (Big-O, Omega, Theta), Best/Worst/Average Case, Simple Algorithms Analysis

1. Introduction to Algorithms

Algorithm एक finite, well-defined steps का sequence होता है, जो किसी particular problem को solve करने के लिए एक निश्चित input को लेकर उसे एक desired output में convert करता है। आसान भाषा में कहें तो, यह एक "recipe" या "तरीका" है जिसमें step-by-step बताया जाता है कि कोई काम कैसे पूरा करना है।

- Algorithm किसी भी programming language से independent होता है — इसे पहले plain steps में सोचा जाता है, फिर बाद में किसी भी language (C, Python, Java) में code किया जा सकता है।
- एक अच्छा algorithm यह सुनिश्चित करता है कि problem को कम से कम समय aur resources (memory) में efficiently solve किया जाए।
- Algorithms सिर्फ computer science तक सीमित नहीं हैं — रोज़मर्रा की ज़िंदगी में भी हम बिना जाने कई algorithms follow करते हैं, जैसे खाना बनाने की recipe या किसी जगह पहुँचने का रास्ता।

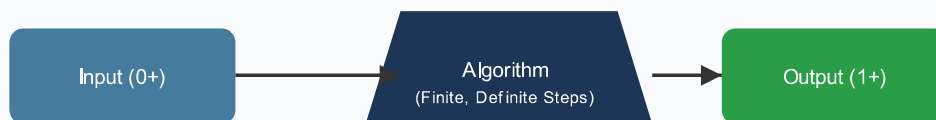
1.1 Characteristics of a Good Algorithm

हर algorithm में कुछ खास **characteristics** होनी चाहिए, तभी उसे एक valid aur "अच्छा" algorithm माना जाता है। यह characteristics यह सुनिश्चित करती हैं कि algorithm सही तरीके से काम करे aur किसी को भी आसानी से समझ में आ जाए।

Characteristic	Description
Input	Algorithm के पास zero या ज़्यादा well-defined inputs होने चाहिए, जिन पर वह काम करे।

Output	Algorithm को कम से कम एक well-defined output ज़रूर देना चाहिए, जो problem का solution हो।
Finiteness	Algorithm को finite (सीमित) संख्या में steps के बाद ज़रूर खत्म होना चाहिए — यह हमेशा के लिए loop में नहीं चल सकता।
Definiteness (Unambiguous)	हर step का meaning बिल्कुल clear aur specific होना चाहिए, जिसमें किसी तरह का confusion ना हो।
Effectiveness	हर step इतना basic aur simple होना चाहिए कि उसे हाथ से (manually), बिना किसी computer के भी, थोड़े समय में perform किया जा सके।

Algorithm — Input, Processing, Output



Zero या ज़्यादा well-defined inputs → finite, definite steps से processing → कम से कम एक output

चित्र 1.1 — कोई भी algorithm well-defined input(s) लेकर, finite aur definite steps से process करके, एक well-defined output देता है।

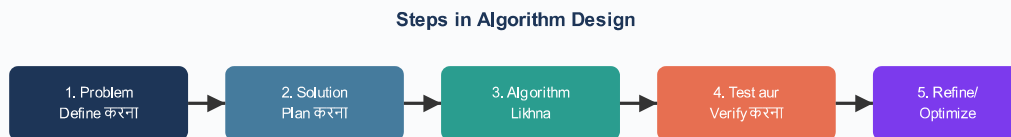
1.2 Importance of Algorithms

- Algorithms किसी भी software या application को बनाने की सबसे पहली aur सबसे ज़रूरी नींव (foundation) होते हैं — बिना सही algorithm के, efficient program लिखना possible नहीं है।
- यह problem को programming language में convert करने से पहले, logic को clearly plan करने में मदद करते हैं, जिससे coding करते समय गलतियाँ कम होती हैं।

- अलग-अलग algorithms का comparison करके यह पता लगाया जा सकता है कि कौन-सा algorithm कम समय या कम memory में problem solve करता है — यह बड़े systems में performance के लिए बहुत ज़रूरी है।
- एक बार सही algorithm design हो जाए, तो उसे किसी भी programming language में implement किया जा सकता है, जिससे algorithm reusable बन जाता है।

1.3 Steps in Algorithm Design (Process)

किसी भी problem के लिए एक अच्छा algorithm design करने के लिए एक systematic process follow किया जाता है, ताकि solution efficient aur सही हो।



चित्र 1.2 — Algorithm design एक systematic process है, जो problem को समझने से शुरू होकर उसे refine करने तक चलता है।

1. **Problem को Define करना:** सबसे पहले यह clearly समझना ज़रूरी है कि problem क्या है, input क्या दिया जाएगा, aur output क्या चाहिए।
2. **Solution को Plan करना:** Problem को छोटे-छोटे sub-problems में divide किया जाता है, aur यह decide किया जाता है कि उसे किस approach से solve किया जाएगा।
3. **Algorithm लिखना:** Solution को step-by-step instructions के रूप में लिखा जाता है, जिसे pseudocode या flowchart के रूप में represent किया जा सकता है।
4. **Test aur Verify करना:** अलग-अलग test cases (जैसे normal case, edge case) से algorithm को check किया जाता है, ताकि यह सुनिश्चित हो कि वह सही output दे रहा है।
5. **Refine/Optimize करना:** अगर algorithm की performance (time/space) बेहतर की जा सकती है, तो उसे optimize करके final रूप दिया जाता है।

Algorithms इस्तेमाल करने के Advantages

- Problem-solving को systematic aur step-by-step बनाता है, जिससे गलतियों की गुंजाइश कम हो जाती है।
- अलग-अलग algorithms का comparison करके सबसे efficient solution चुना जा सकता है।
- Algorithm एक बार design हो जाए, तो उसे किसी भी programming language में आसानी से implement किया जा सकता है।

Limitations

- Complex problems के लिए algorithm design करना समय लेने वाला aur मुश्किल हो सकता है।
- एक ही problem के लिए कई algorithms possible होते हैं, इसलिए सही चुनाव करने के लिए experience चाहिए होता है।
- Algorithm सिर्फ logic बताता है — उसे actual program में convert करना अलग skill है, जिसमें errors आ सकती हैं।

Real-life Example: किसी नई जगह जाने के लिए Google Maps पर route search करना एक algorithm जैसा ही process है — पहले start aur destination (input) दिया जाता है, फिर सबसे अच्छा रास्ता (algorithm द्वारा) निकाला जाता है, aur अंत में best route (output) दिखाया जाता है।

2. Programming Models

Algorithm को actual code लिखने से पहले किसी ऐसे तरीके से represent करना ज़रूरी होता है, जो आसानी से समझा जा सके — इन्हीं तरीकों को **Programming Models** कहा जाता है। यह models algorithm की logic को visually या text के रूप में स्पष्ट करते हैं, ताकि कोई भी उसे पढ़कर समझ सके, चाहे वह किसी particular programming language को जानता हो या नहीं।

- Programming models algorithm aur actual code के बीच का एक "पुल" (bridge) होते हैं — यह logic को language-independent तरीके से represent करते हैं।
- यह design करते समय गलतियाँ पकड़ने में मदद करते हैं, क्योंकि coding शुरू करने से पहले ही logic clearly दिख जाता है।

2.1 Pseudocode Representation

Pseudocode एक informal, English जैसी भाषा में लिखा गया तरीका है, जिससे किसी algorithm के logic को represent किया जाता है — बिना किसी particular programming language की strict syntax का पालन किए।

- यह किसी भी programming language की तरह strict rules follow नहीं करता, इसीलिए इसे लिखना aur पढ़ना दोनों आसान होते हैं।
- Pseudocode को बाद में आसानी से किसी भी programming language (C, Python, Java) में convert किया जा सकता है।
- यह मुख्य रूप से algorithm की logic पर focus करता है, ना कि language-specific syntax की details पर।

Example Pseudocode: दो numbers का sum निकालना

START

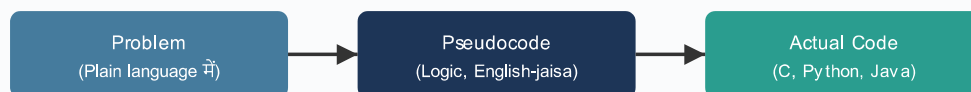
INPUT a, b

sum = a + b

OUTPUT sum

END

Problem → Pseudocode → Actual Code



Pseudocode बीच का पड़ाव है — यह language-independent होता है, इसलिए किसी भी language में convert हो सकता है

चित्र 2.1 — Pseudocode, problem की plain-language समझ aur actual programming code के बीच का पुल (bridge) है।

2.2 Flowcharts and Structured Approach

Flowchart किसी algorithm का **graphical/visual representation** होता है, जिसमें अलग-अलग standard symbols (जैसे boxes, diamonds, arrows) का इस्तेमाल करके steps aur उनके बीच के flow को दिखाया जाता है।

- Flowchart में हर तरह के step के लिए एक specific symbol होता है — जैसे process के लिए rectangle, decision के लिए diamond, aur start/end के लिए oval।
- यह उन लोगों के लिए भी आसानी से समझ में आ जाता है, जिन्हें programming की ज़्यादा जानकारी नहीं है, क्योंकि यह पूरी तरह visual होता है।
- **Structured Approach** का मतलब है कि program को तीन basic control structures — **Sequence**, **Selection (decision)**, aur **Iteration (loop)** — के through व्यवस्थित तरीके से design किया जाए, ताकि logic clean aur आसानी से समझने योग्य रहे।

तीन Basic Control Structures

Sequence

Steps एक के बाद एक चलते हैं

Selection

Condition के according रास्ता चुना जाता है

Iteration

जब तक condition true रहे,
तब तक repeat होता है

चित्र 2.2 — Structured Programming तीन basic control structures पर based होती है: Sequence (एक के बाद एक), Selection (condition के according रास्ता चुनना), aur Iteration (repeat करना)।

Common Flowchart Symbols

इन symbols को arrows से जोड़कर algorithm के steps aur उनकी sequence दिखाई जाती है

चित्र 2.3 — Flowchart में हर step के लिए एक standard symbol होता है, जिन्हें arrows से जोड़कर पूरे algorithm का flow दिखाया जाता है।

Symbol	नाम	Use
Oval	Start/End (Terminal)	Algorithm/flowchart की शुरुआत aur अंत दिखाने के लिए
Rectangle	Process	कोई calculation या processing step दिखाने के लिए

about:blank

10/22

Diamond	Decision	कोई condition check करने के लिए (Yes/No branch)
Parallelogram	Input/Output	Data input लेने या output दिखाने के लिए
Arrow	Flow Line	Steps के बीच execution की direction दिखाने के लिए

Pseudocode के Advantages

- लिखना तेज़ है aur किसी strict syntax rules का पालन नहीं करना पड़ता।
- किसी भी programming language में आसानी से convert किया जा सकता है।

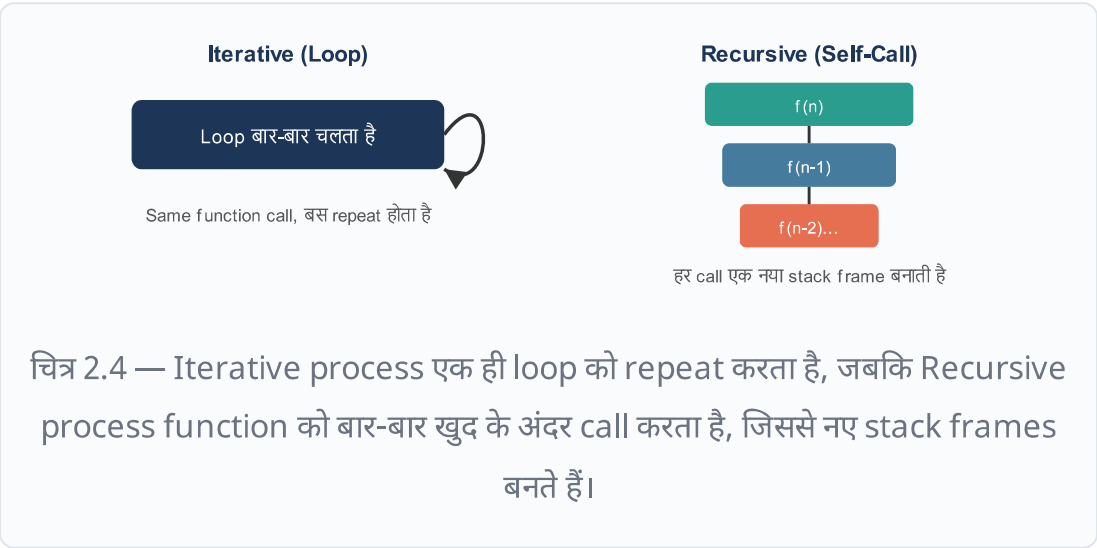
Flowchart के Advantages

- Visual होने की वजह से logic को समझना ज़्यादा आसान होता है, especially beginners के लिए।
- Complex decision-making वाले logic को दिखाने में ज़्यादा प्रभावी होता है।

2.3 Iterative vs Recursive Processes

किसी भी repetitive task को perform करने के दो मुख्य तरीके होते हैं — **Iterative** (loops का इस्तेमाल करके) aur **Recursive** (function द्वारा खुद को call करके)।

- **Iterative Process** में `for` या `while` जैसे loops का इस्तेमाल करके किसी task को बार-बार repeat किया जाता है, जब तक एक condition पूरी ना हो जाए।
- **Recursive Process** में एक function खुद को (directly या indirectly) call करता है, जब तक वह एक base case तक ना पहुँच जाए — जैसे हमने पिछले units में recursion पढ़ा था।



Example: Factorial — Iterative vs Recursive

```
// Iterative
function factorialIterative(n):
    result = 1
    for i = 1 to n:
        result = result * i
    return result

// Recursive
function factorialRecursive(n):
    if n == 0:
        return 1 // Base case
    return n * factorialRecursive(n - 1) // Recursive case
```

2.4 Iterative vs Recursive — Comparison Table

Basis	Iterative	Recursive
तरीका	Loop (for/while) का इस्तेमाल	Function खुद को call करता है

Memory Usage	कम (extra stack frames नहीं बनते)	ज़्यादा (हर call के लिए नया stack frame)
Speed	तेज़ (function-call overhead नहीं)	Comparatively धीमा
Code Readability	कभी-कभी लंबा हो सकता है	छोटा aur clean, especially tree/recursive problems के लिए
Termination	Loop condition की ज़रूरत	Base case की ज़रूरत

Iterative Process के Advantages

- Memory efficient होता है, क्योंकि extra stack frames नहीं बनते।
- ज़्यादातर मामलों में recursion से तेज़ होता है, क्योंकि function-call overhead नहीं होता।
- Simple repetitive tasks के लिए समझना aur debug करना आसान होता है।

Recursive Process के Advantages

- Tree, graph, या divide-and-conquer जैसी problems के लिए code बहुत छोटा aur clean हो जाता है।
- Complex problems को छोटे sub-problems में तोड़कर solve करना आसान हो जाता है।
- Mathematical problems (जैसे factorial, Fibonacci) को naturally represent कर पाता है।

Real-life Example: किसी box के अंदर कई छोटे-छोटे boxes रखे हों, aur हर box के अंदर फिर से box हो — इन्हें एक-एक करके खोलना Recursive process जैसा है,

जबकि किसी list के हर item को एक-एक करके गिनना (जैसे 1 से 10 तक counting)
Iterative process जैसा है।

3. Algorithm Analysis

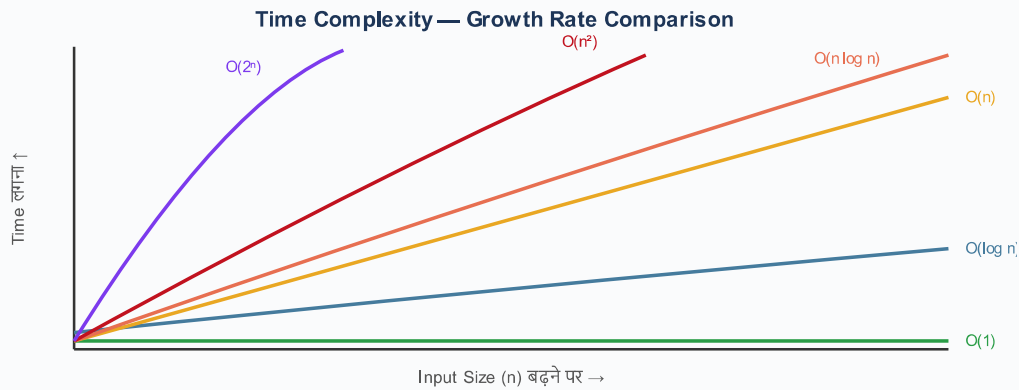
Algorithm Analysis वह process है जिसमें यह measure किया जाता है कि कोई algorithm कितना **efficient** है — यानी वह कितना समय (time) लेता है aur कितनी memory (space) इस्तेमाल करता है। यह analysis किसी algorithm को actual computer पर run किए बिना, सिर्फ theoretical तरीके से किया जाता है।

- एक ही problem को solve करने के कई तरीके (algorithms) हो सकते हैं — analysis से यह पता चलता है कि इनमें से कौन-सा सबसे efficient है।
- यह analysis "machine-independent" होता है, यानी यह किसी particular computer की speed पर depend नहीं करता, बल्कि यह देखता है कि input की size बढ़ने पर algorithm कैसे behave करता है।

3.1 Time aur Space Complexity — Concept

Time Complexity यह बताती है कि किसी algorithm को complete होने में input की size (n) के according कितने "operations" या "steps" लगेंगे। **Space Complexity** यह बताती है कि algorithm को चलाने के लिए कितनी extra memory चाहिए होगी।

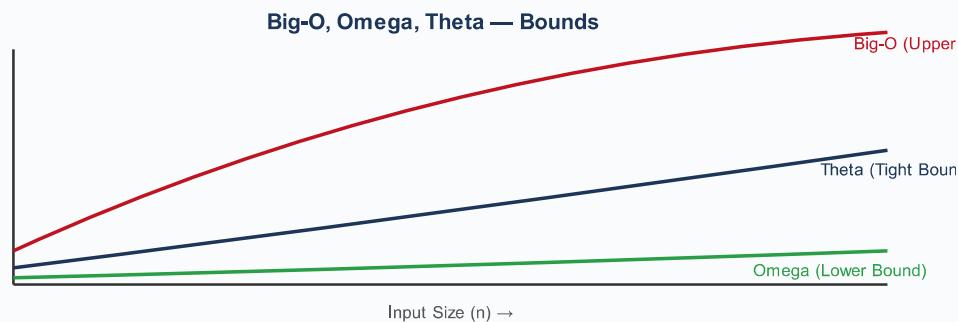
- Time Complexity को actual seconds/minutes में नहीं, बल्कि input size (n) के function के रूप में express किया जाता है — जैसे $O(n)$, $O(n^2)$ ।
- Space Complexity में algorithm द्वारा इस्तेमाल की गई extra variables, arrays, या recursive call stack की memory शामिल होती है (input को store करने वाली memory नहीं गिनी जाती)।
- ज़्यादातर algorithms में time aur space के बीच एक trade-off होता है — कभी-कभी कम time के लिए ज़्यादा memory खर्च करनी पड़ती है, या इसके उल्टा।



चित्र 3.1 — जैसे-जैसे input size (n) बढ़ती है, अलग-अलग time complexities कितनी तेज़ी से बढ़ती हैं यह दिखाया गया है — $O(1)$ सबसे बेहतर (flat) है, जबकि $O(2^n)$ सबसे ज़्यादा तेज़ी से बढ़ता है (सबसे खराब)।

3.2 Asymptotic Notations: Big-O, Omega, Theta

Asymptotic Notations वह mathematical तरीके हैं, जिनसे किसी algorithm की time/space complexity को input की size बहुत बड़ी होने पर (यानी "asymptotically") express किया जाता है। यह तीन मुख्य notations से मिलकर बनते हैं — Big-O, Omega, aur Theta — जो algorithm के performance की अलग-अलग bounds (सीमाएं) बताते हैं।



चित्र 3.2 — Big-O algorithm की worst (upper) limit बताता है, Omega best (lower) limit बताता है, aur Theta दोनों के बीच की tight (exact) bound बताता है।

Notation	क्या बताता है	Meaning
----------	---------------	---------

Big-O, $O(f(n))$	Upper Bound (Worst Case)	Algorithm ज़्यादा से ज़्यादा कितना समय ले सकता है — यह सबसे ज़्यादा इस्तेमाल होने वाला notation है।
Omega, $\Omega(f(n))$	Lower Bound (Best Case)	Algorithm कम से कम कितना समय लेगा — यानी सबसे अच्छी स्थिति में performance।
Theta, $\Theta(f(n))$	Tight Bound (Average/Exact)	जब upper aur lower bound दोनों एक जैसे हों, तो यह algorithm की exact growth rate बताता है।

3.2.1 Big-O Notation, $O(f(n))$

Big-O यह बताता है कि किसी algorithm का running time, input size बढ़ने पर **ज़्यादा से ज़्यादा** कितनी तेज़ी से बढ़ सकता है — यानी यह algorithm की **worst-case** performance की guarantee देता है।

- यह सबसे commonly इस्तेमाल होने वाला notation है, क्योंकि worst-case जानना practical रूप से सबसे ज़्यादा ज़रूरी होता है।
- Example: अगर किसी algorithm का time complexity $O(n^2)$ है, तो इसका मतलब है कि input बढ़ा होने पर उसका time ज़्यादा से ज़्यादा n^2 के हिसाब से बढ़ेगा।

3.2.2 Omega Notation, $\Omega(f(n))$

Omega यह बताता है कि किसी algorithm को **कम से कम** कितना समय लगेगा — यानी यह algorithm की **best-case** performance को represent करता है।

- यह हमें बताता है कि चाहे कितनी भी अच्छी स्थिति क्यों ना हो, algorithm इससे कम समय में कभी complete नहीं होगा।
- Practical use में Big-O जितना common नहीं है, लेकिन theoretical analysis में इसका महत्व है।

3.2.3 Theta Notation, $\Theta(f(n))$

Theta तब इस्तेमाल होता है जब किसी algorithm की upper bound (Big-O) aur lower bound (Omega) दोनों **एक जैसी** हों — यह algorithm की **exact (tight) growth rate** बताता है।

- Theta notation सबसे precise होता है, क्योंकि यह best aur worst case के बीच कोई gap नहीं छोड़ता — दोनों बराबर होते हैं।
- अगर किसी algorithm का $\Theta(n \log n)$ है, तो इसका मतलब है कि वह हमेशा (हर case में) लगभग $n \log n$ समय ही लेगा।

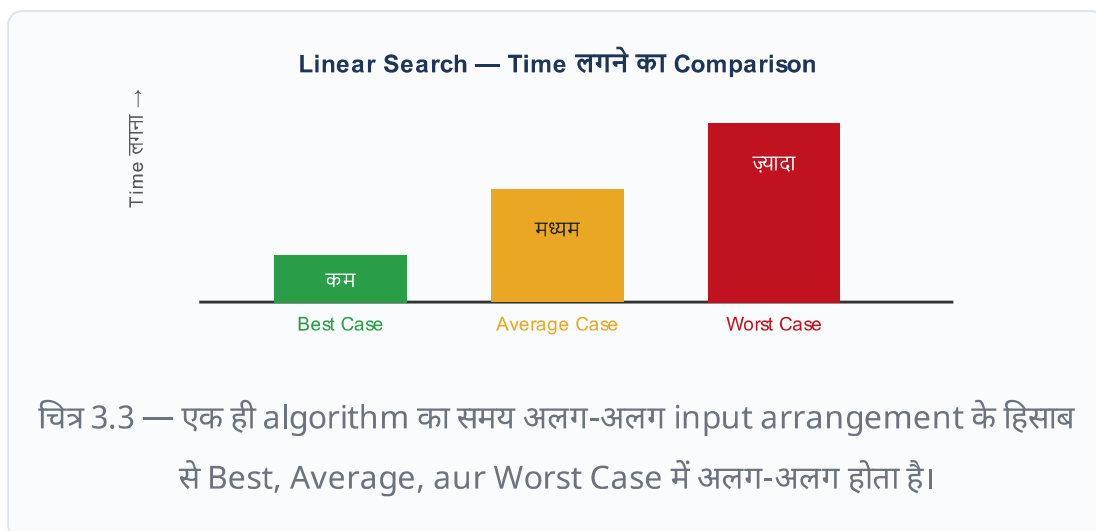
याद रखें: आसान तरीके से याद रखने के लिए — **Big-O** = " $\leq$ " (ज़्यादा से ज़्यादा), **Omega** = " $\geq$ " (कम से कम), aur **Theta** = "=" (लगभग बराबर, tight bound)।

3.3 Best-case, Worst-case, aur Average-case Analysis

किसी भी algorithm का performance input की **arrangement/nature** पर भी depend कर सकता है, ना सिर्फ उसकी size पर। इसीलिए algorithm को तीन अलग-अलग scenarios में analyze किया जाता है — Best Case, Worst Case, aur Average Case।

Case	Description	Example (Linear Search)
------	-------------	-------------------------

Best Case	जब input ऐसी हो कि algorithm सबसे कम समय में complete हो जाए	Target element पहले ही position पर मिल जाए
Worst Case	जब input ऐसी हो कि algorithm को सबसे ज़्यादा समय लगे	Target element आखिरी position पर हो या मौजूद ही ना हो
Average Case	Random/typical input पर algorithm का average performance	Target element बीच में कहीं भी मिल सकता है



- Best case हमेशा सबसे optimistic scenario होता है, जो practical रूप से बहुत उपयोगी नहीं माना जाता क्योंकि यह हमेशा नहीं होता।
- Worst case सबसे ज़्यादा important माना जाता है, क्योंकि यह algorithm की "guarantee" देता है कि वह इससे ज़्यादा समय कभी नहीं लेगा।
- Average case को calculate करना सबसे मुश्किल होता है, क्योंकि इसके लिए सभी possible inputs की probability जाननी पड़ती है।

Worst-Case Analysis के Advantages

- Algorithm की एक निश्चित guarantee देता है, जो critical/real-time systems के लिए बहुत ज़रूरी होता है।
- Calculate करना Average Case के comparison में आसान होता है।

Limitation

- Worst case हमेशा नहीं होता, इसलिए यह कभी-कभी algorithm की actual real-world performance को ज़्यादा negative दिखा सकता है।
- Average case ज़्यादा realistic होता है, लेकिन उसे calculate करना statistically complex होता है।

3.4 Analysis of Simple Algorithms (Addition aur Multiplication of Array)

Time aur Space complexity को practically समझने के लिए, आइए array के दो सबसे basic operations — **addition (सभी elements का sum)** aur **multiplication (दो arrays/matrices का multiplication)** — का analysis करते हैं।

3.4.1 Array के Elements का Addition (Sum)

```
function sumArray(array, n):  
    sum = 0  
    for i = 0 to n - 1:  
        sum = sum + array[i]    // यह statement n बार चलेगा  
    return sum
```

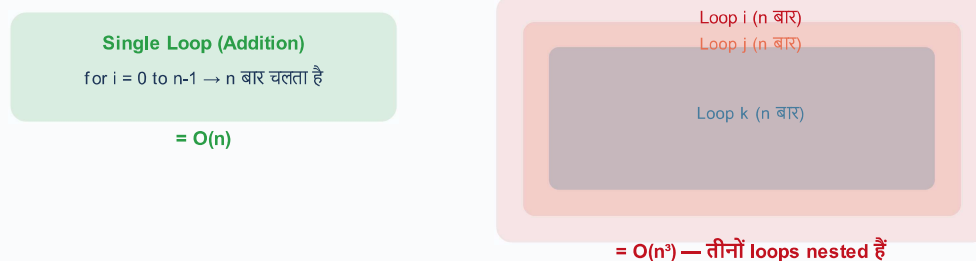
- इस algorithm में loop array के n elements पर exactly **n बार** चलता है, इसलिए इसकी Time Complexity **$O(n)$** होती है — यानी यह "Linear" algorithm है।
- Space Complexity **$O(1)$** होती है, क्योंकि सिर्फ एक extra variable (sum) इस्तेमाल हो रहा है, चाहे array कितनी भी बड़ी क्यों ना हो।

3.4.2 दो Matrices का Multiplication

```
function multiplyMatrix(A, B, n):
    for i = 0 to n - 1:
        for j = 0 to n - 1:
            C[i][j] = 0
            for k = 0 to n - 1:
                C[i][j] = C[i][j] + A[i][k] * B[k][j]    // यह
innermost statement n × n × n बार चलेगा
    return C
```

- इसमें तीन nested loops होती हैं (हर एक n बार चलती है), इसलिए innermost statement कुल $n \times n \times n = n^3$ बार execute होता है।
- इसीलिए इस algorithm की Time Complexity $O(n^3)$ होती है — यह Addition के comparison में बहुत ज़्यादा है, क्योंकि इसमें नेस्टेड (nested) loops शामिल हैं।
- Space Complexity $O(n^2)$ होती है, क्योंकि result matrix C को store करने के लिए $n \times n$ साइज़ की extra memory चाहिए होती है।

Single Loop vs Nested Loops — Visual



चित्र 3.4 — एक single loop n elements पर $O(n)$ time लेता है; तीन nested loops (Matrix Multiplication) एक-दूसरे के अंदर होने की वजह से $n \times n \times n = O(n^3)$ time लेते हैं।

Operation	Time Complexity	Space Complexity	वजह
-----------	-----------------	------------------	-----

Array Elements का Addition	$O(n)$	$O(1)$	एक single loop, n elements पर एक बार चलता है
Matrix Multiplication	$O(n^3)$	$O(n^2)$	तीन nested loops ($n \times n \times n$), result store करने के लिए $n \times n$ memory

Exam Tip: Loops की complexity पहचानने का आसान तरीका — एक single loop (n elements पर) आमतौर पर $O(n)$ देता है, दो nested loops $O(n^2)$ देते हैं, aur तीन nested loops $O(n^3)$ देते हैं। जितनी nested loops होंगी, complexity उतनी ही तेज़ी से बढ़ती जाएगी।

Real-life Example: एक class में सभी students के marks जोड़कर total निकालना (Addition, $O(n)$) तेज़ है aur सीधा-सादा है, लेकिन अगर दो अलग-अलग subjects के हर student का combination बनाकर compare करना हो (जैसे Matrix Multiplication जैसा nested comparison), तो वह कहीं ज़्यादा समय लेने वाला काम होगा।