Time Complexity

By Mani Abedii

View Full DSA Repository

A clear, compact guide covering essential data structures and algorithms, designed for easy understanding and quick learning.

Time complexity is a way to represent how the runtime of an algorithm grows relative to the **size of input**.

- Helps us compare algorithms.
- Crucial for writing scalable codes.
- Predicts performance before implementation.

Case Scenarios:

- **Best Case:** The scenario where the algorithm performs the least amount of work – the most favorable input.
    - Shows the minimum time an algorithm can take.
    - Represented with Ω (Omega) notation.
- **Worst Case:** The scenario where the algorithm performs the most work – the least favorable input.
    - Commonly used to guarantee performance boundaries.
    - Represented with O (big O) notation.
- **Average Case:** The expected performance of the algorithm across all possible inputs.
    - Gives a realistic performance estimation.
    - Represented with θ (Theta) notation.

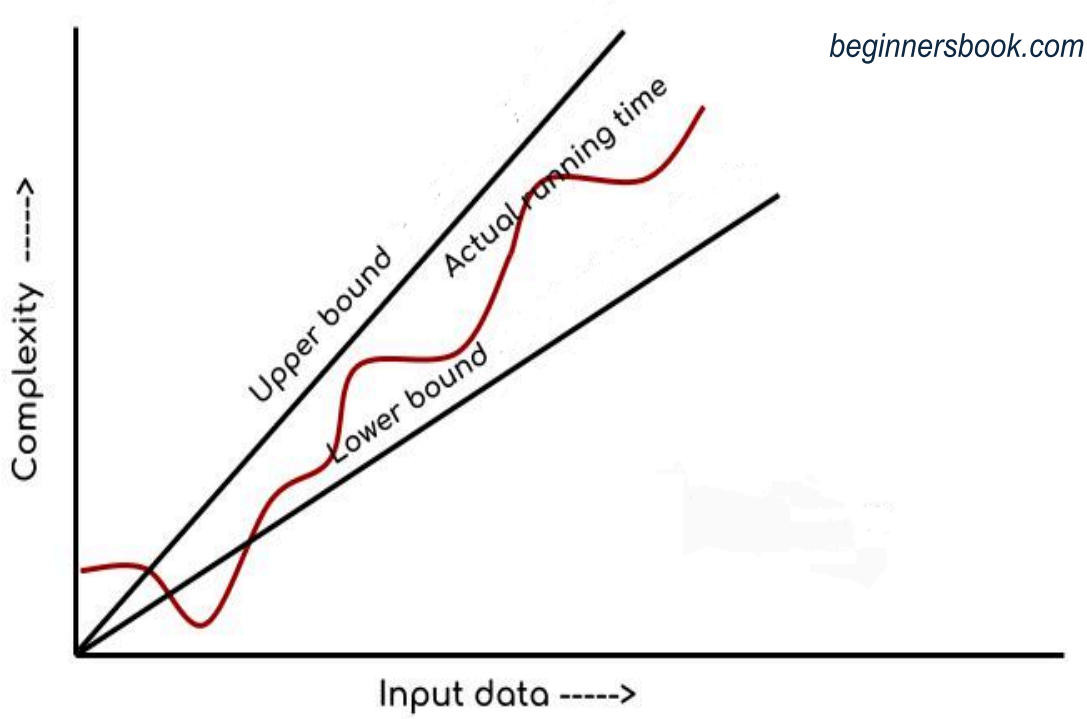
Big O, Omega, and Theta:

To describe how an algorithm behaves as the input size grows, we use three symbols:

- **Big O (O) – Worst-Case:**
    - The maximum time an algorithm can take.
    - No matter what, it will never take longer than this.
- **Omega (Ω) – Best-Case:**
    - The minimum time an algorithm might take.
    - No matter what, it will never take shorter than this.
- **Theta (θ) – Balanced-Case:**
    - The average or typical case – not too bad, not too perfect.
    - How long the algorithm usually takes.

We'll mostly use **Big O** notation to describe the time complexity of algorithms – because it's the most common and safest way to analyze performance.

© 2025 Mani Abedii

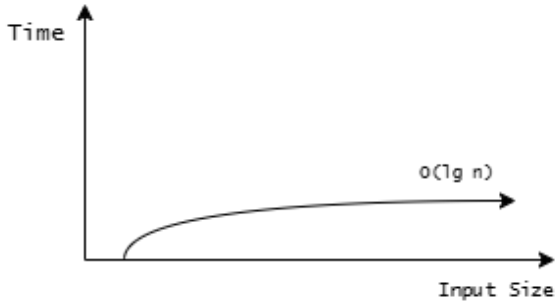

beginnersbook.com
Complexity ----->
Upper bound
Actual running time
Lower bound
Input data ----->
T(n)
f(n) = O(g(n))
c*g(n)
f(n)
g(n)
Time Complexity
n
n₀
Input Size
|f(n)| is asymptotically bounded
above by g(n) up to constant factor c

Common Time Complexities in Big O Notation

The following are commonly encountered time complexities from worst to best, along with details. The Java implementations illustrating the following time complexity cases are provided in the attached source code folder.

1. **O(1) – Constant Time**
   An algorithm is said to have constant time complexity if the execution time does not grow with the size of the input. It performs a fixed number of operations regardless of input length.
   - Extremely efficient
   - Desired wherever possible

2. **O(log n) – Logarithmic Time**
   An algorithm runs in logarithmic time if it reduces the size of the input data in each step – typically by dividing it in half. The number of steps grows proportionally to the logarithm of the input size.
   - Extremely efficient for large datasets
   - Typical of divide-and-conquer strategies

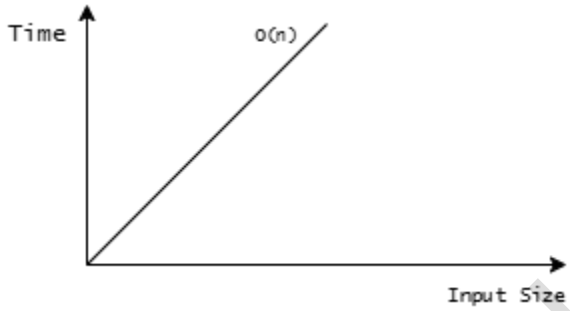


© 2025 Mani Abedi

3. **O(n) – Linear Time**

   Linear time complexity means that the execution time grows linearly with the input size. Each input element is processed exactly once.

   - Simple & predictable
   - Often used as a baseline for performance comparison



4. **O(n log n) – Log-linear Time**

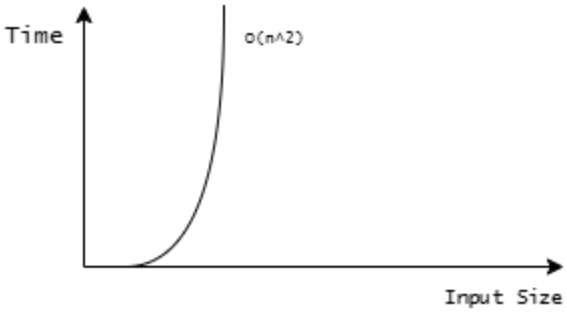
   An algorithm with log-linear complexity typically performs a logarithmic operation in time. It's faster than quadratic time and is the optimal time complexity for many comparison-based sorting algorithms.

   - Efficient for sorting large datasets
   - Represents a balance between complexity and performance

5. **O(n^2) – Quadratic Time**

   Quadratic time means the time taken increases quadratically with the input size. Usually occurs when there are two nested loops, each depending on input size n.

   - Not efficient for large datasets
   - Usually easy to implement but should be avoided if possible

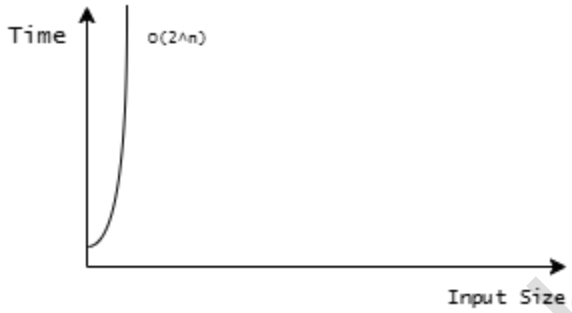


© 2025 Mani Abedii

6. **$O(2^n)$ – Exponential Time**

   Exponential algorithms double their runtime with each additional input element. These are generally infeasible for even moderately large inputs.

   - Extremely inefficient
   - Often arises in brute-force solution



7. **O(n!) – Factorial Time**

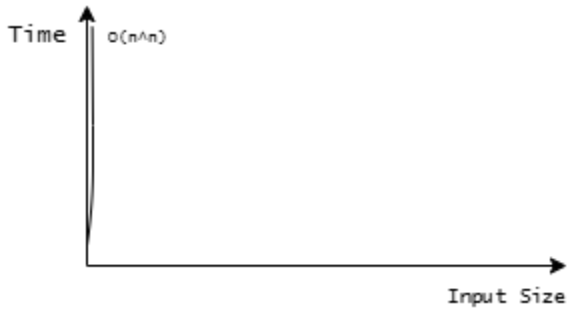
   Factorial time complexity increases faster than any other common complexity. These algorithms evaluate all possible permutations of input data.

   - Unusable for large n
   - Acceptable only for very small input sizes

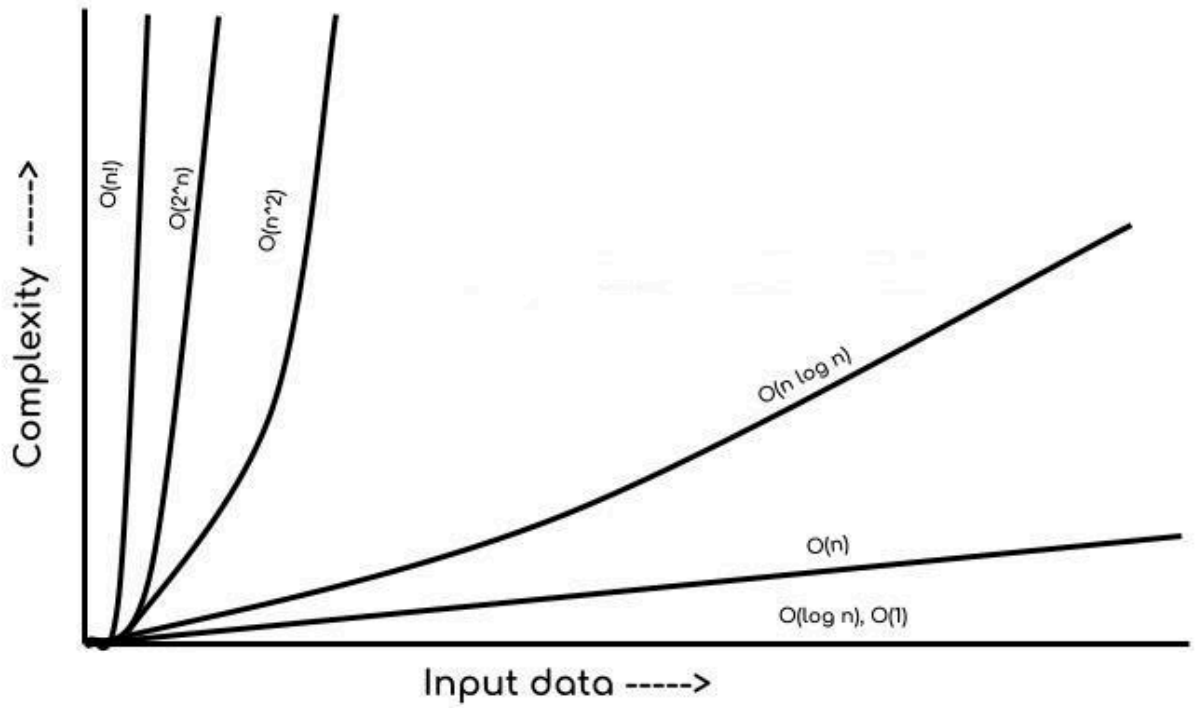
8. **$O(n^n)$ – Super-Exponential Time**

   An algorithm is said to have super-exponential time complexity where each increase in input size results in an exponential increase of exponential scale. This is one of the slowest growing complexities encountered in algorithm analysis.

   - Extremely inefficient for any input size greater than a few units
   - Rarely used in practical algorithm design



© 2025 Mani Abedii



Comparison of input size growth relative to time complexity

Key Takeaways on Big O Notation:

1. **Dominant Terms & Constant Factors:** Big O notation focuses on the dominant terms and ignores constant multipliers, and less-dominant factors, so 2n and n or both O(n) because constant factors do not significantly affect the growth rate of an algorithm. Also, the time complexity of $O(n^2) + O(n) + O(n \lg(n))$ is $O(n^2)$ because it's the dominant term.

2. **Higher-Degree & Logarithmic Complexities:** While polynomial complexities like $O(n^3)$ or $O(n^4)$ do exist, they are less common in practice. Among them, quadratic time is the most frequently encountered and widely studied. Also, in CS, logarithmic complexity is typically assumed to be base 2, since many algorithms halve the input size at each step. However, in Big O notation, the base of the logarithm is omitted because all logarithmic bases differ only by a constant factor.

© 2024

3. **Time Complexity vs Actual Runtime:** Time complexity measures how the number of operations grows as the input size increases – not how long an algorithm takes to run in real time. Two algorithms may both have a time complexity of O(n), but one might take significantly longer due to larger constant factors, inefficient implementation, or hardware limitations. Therefore, Time complexity allows us to compare algorithms independently of hardware, software, or implementation details, focusing solely on their scalability.

Time Complexity Analysis Techniques

1. Polynomial Function:

If $f(n) = a_m n^m + a_{m-1} n^{m-1} + a_{m-2} n_{m-2} + \dots + a_2 n^2 + a_1 n + a_0$

$\Rightarrow f(n) = O(n^m)$

2. If Blocks:

```
if (condition):
      block1;

else:

      block 2;

=> Time Complexity = Max(time(block1), time(block2))
```

3. Loops:

```
(a <= b)
for (i = a; i <= b; i += k):
      statement();
or

for (i = b; i >= a: i -= k):
      statement();

=> Number of Iterations = (b - a + 1) / k

e.g.:
for (int i = 0; i < n; i++):
      statements();

=> Number of Iterations = (n - 1 + 1) / 1 = O(n)
```

```
for (i = a; i <= b; i *= k):
      statement();
or

for (i = b; i >= a; i /= k):
      statement();

=> Number of Iterations = log_k b - log_k a + 1

e.g.:
for (int i = 1; i <= n; i *= 2)
      statements();

=> Number of Iterations = lg(n)-lg(1) + 1 = O(lg(n))
```

4. **Nested Loops**: For each iteration of the outer loop, the inner loop is executed completely; Therefore, time complexities are multiplied.

```
e.g.:

for (int i = 0; i < n; i++):
      for (int j = 0; j < n; j++):
            statement();

=> Time Complexity = O(n) * O(n) = O(n^2)

for (int i = 1; i <= n; i *= 2)
      for (int j = 0; j < n; j++)
            statement();

=> Time Complexity = O(lg(n)) * O(n) = O(n lg(n))
```

5. **Sequential Loops**: Time complexities add up instead of multiplying.

```
e.g.:

for (int i = 0; i < n; i++):

      statement();

for (int j = 1; j <= n; j *= 2):

      statement();

=> Time Complexity = O(lg(n)) + O(n) = O(n)
```

6. **Dependent Nested Loops:** The inner loop's range depends on the outer loop's counter variable. Analyzing their time complexity is more involved because boundaries vary with each outer loop iteration. Unlike regular nested loops where complexities multiply, here we often need to use summation notation (∑) or build a step-by-step table to count iterations.

```
for (int i = 0; i < n; i++):
        for (int j = 0; j <= i; j++):
                statement();
```

Step-by-step table:

i	0	1	2	…	n
Num of Iterations	1	2	3	…	n

=> Time Complexity = 1 + 2 + 3 + … + n = n(n+1)/2

= $O(n^2)$

Summation notation:

$$\sum_{i=0}^{n-1} \sum_{j=0}^{i} 1 = \sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

```
for (int i = 1; i <= n; i *= 2):
        for (int j = 0; j < i; j++):
                statement();
```

i	1	2	4	…	n
Num of Iterations	1	2	4	…	n

=> Time Complexity = 1 + 2 + 4 + … + n

= $2^0 + 2^1 + 2^2 + \dots + 2^{lg(n)}$

= $(2^{lg(n+1)} - 1) / (2 - 1) = 2n - 1 = O(n)$

7. **Recursive Functions:** Analyzing recursive functions has its own toolkit. We first must translate the code into a recurrence relation that reflects how the function calls itself and how much word is done per call. Once the recurrence is formed, we can solve it using the following methods to determine the overall time complexity.

Techniques to Analyze the Time Complexity of Recursive Functions:

1. **Iterative Expansion/Substitution:** Expand the recurrence step by step to find a pattern, then solve.

e.g.:

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

The function does one recursive call to factorial(n-1)

It does 1 constant work (n) before returning:*

So, the recurrence is: ***T(n) = T(n – 1) + 1***

Solve Using Iterative Expansion:

$T(n) = T(n - 1) + 1$

$= T(n - 2) + 1 + 1$

$= T(n - 3) + 1 + 1 + 1$

$= \ldots$

$= T(0) + n * 1 = 0 + n = n$

$\Rightarrow T(n) = O(n)$

e.g.:

```
int func(int n):
      if (n <= 1) return 1;
      return func(n-1) + func(n-1);
```

The function does two recursive calls & one constant work on each call, So the recurrence is:

T(n) = T(n-1) + T(n-1) + 1 = 2T(n-1) + 1

Solve Using Iterative Expansion:

$T(n) = 2T(n-1) + 1$

$= 2.2T(n-2) + 3 = 2^2T(n-2) + 3$

$= 2^3T(n-3) + 7$

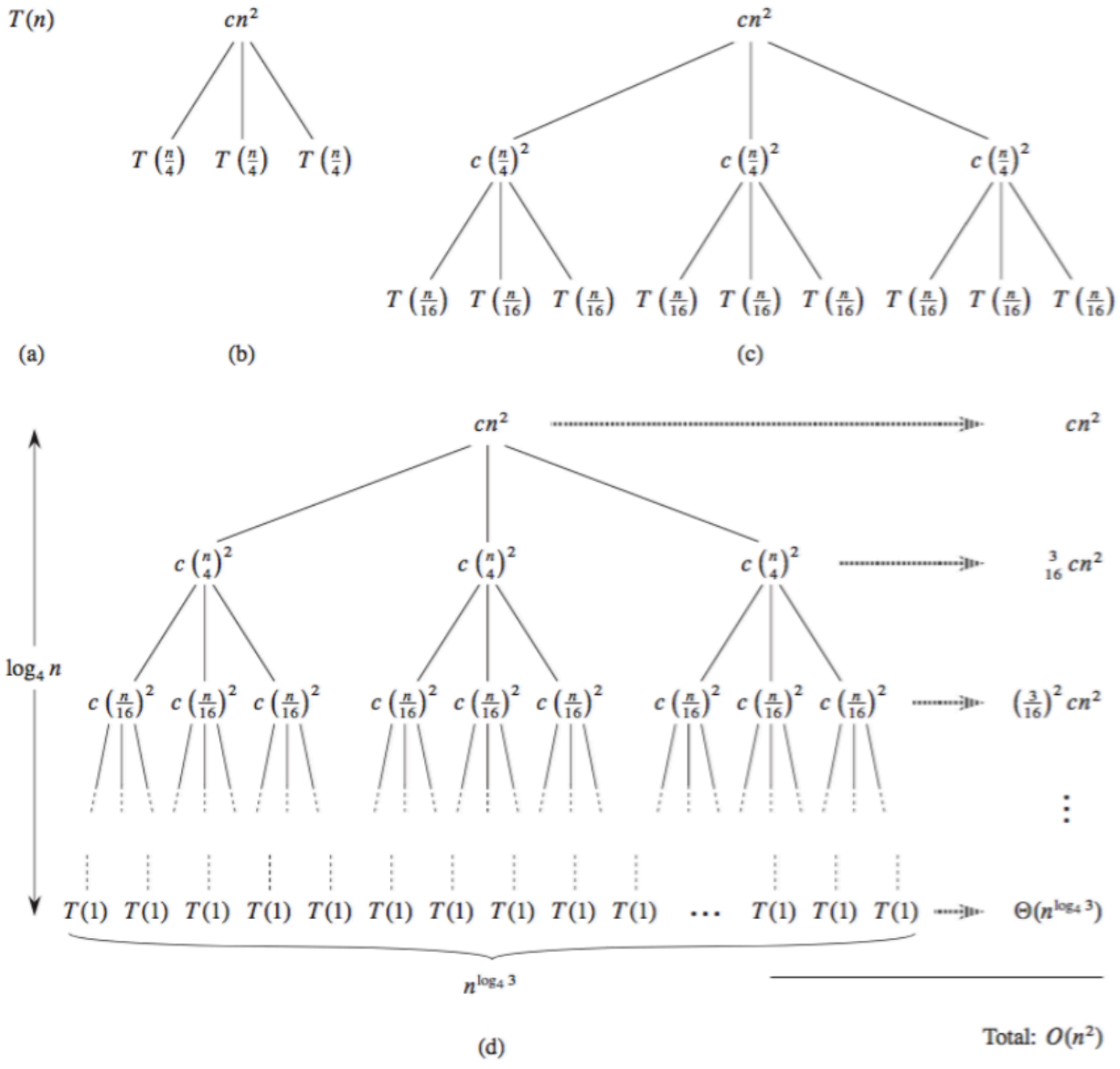
$= \ldots = 2^kT(n-k) + (2^k-1)$

$= \text{base case: } T(n) = 2^{n-1}T(1) = 2^{n-1}$

$\Rightarrow T(n) = O(2^n)$

2. **Recursion Tree**: A recursion tree is a tree where each node represents the cost of a certain recursive subproblem. Then you can sum up the numbers in each node to get the cost of the entire algorithm. Note: We would usually use a recursion tree to generate possible guesses for the runtime, and then use the substitution method to prove them. However, if you are very careful when drawing out a recursion tree and summing the costs, you can actually use a recursion tree as a direct proof of a solution to a recurrence.

e.g.: Recurrence: $T(n) = 3T(n/4) + \theta(n^2)$



3. **Master Theorem:** The Master Theorem helps you find the time complexity of recursive functions of the form:

   T(n) = a.T(n/b) + f(n)

   Where:
   a >= 1: number of subproblems
   n/b: size of each subproblem
   f(n): cost of non-recursive work

   Steps: Let $C = \log_b a$

   First, compare f(n) to n^C. One of the following cases will apply:

   Case I:

   If f(n) grows polynomially slower than n^C, then the majority of the cost comes from the recursive calls. In other words, the work done at each level increases faster than the non-recursive part f(n), and f(n) becomes asymptotically insignificant.

   As a result, the overall time complexity is dominated by the work from the final levels of recursion, and we get: **$T(n) = O(n^C)$**

   Case II:

   If f(n) & n^C grow at the same rate, then both the non-recursive work and the recursive calls also grow at the same rate. This means the cost is evenly distributed across all levels of the recursion tree. Each level contributes the same amount of work, and there are log n levels.

   As a result, the total time complexity is the cost per level times the number of levels: **$T(n) = O(n^C . \log(n))$**

   Case III:

   If f(n) grows faster than n^C, then the non-recursive part grows faster than the recursive part. This means the majority of the cost comes from the work done outside the recursive calls.

   As a result, the overall time is dominated by the non-recursive part, and we get:

   T(n) = O(f(n))

e.g.:

Merge Sort: $T(n) = 2T(n/2) + O(n)$

$a = b = 2$

$f(n) = O(n)$

$\Rightarrow C = \log_b^a = \log_2^2 = 1$

$\Rightarrow n^C$ *and* $f(n)$ *grow at the same rate*

$\Rightarrow T(n) = O(n \log(n))$

4. **Generating Functions:** Advanced mathematical technique for solving recurrences.
5. **Akra-Bazzi Theorem:** Generalizes Master Theorem (for more complex cases).
6. **Characteristic Equation:** Used mostly in linear recurrences.

© 2025 Mani Abdii