**1. What does "log" mean in algorithms?**

When you see **log(n)** or **log₂(n)** in **time complexity**, it usually means:

👉 "How many times can I divide n by 2 until I reach 1?"

That’s why it’s often written as **log₂(n)**, but in computer science we usually just write **log(n)** (and assume base 2).

### 2. ****Example with numbers****

* If n = 8
  + Divide by 2: 8 → 4 → 2 → 1
  + That’s **3 steps** → log₂(8) = 3
* If n = 16
  + Divide by 2: 16 → 8 → 4 → 2 → 1
  + That’s **4 steps** → log₂(16) = 4
* If n = 1000
  + log₂(1000) ≈ 10 (since 2¹⁰ = 1024)

So **log(n) grows very slowly compared to n**.

### 3. ****Why do we care in algorithms?****

* **Binary Search** → O(log n)  
  (Because you keep dividing the array in half until one element is left.)
* **Balanced Trees (like AVL, Red-Black Tree)** → search/insert/delete are O(log n)
* **Heap operations** → O(log n)

### 4. ****log vs log₂****

* log₂(n) = logarithm base 2 (divide by 2 each step).
* In algorithms, unless otherwise written, **log(n) means log₂(n)**.
* Mathematicians sometimes use natural log (ln, base e), but in DSA → always assume **base 2**.

💡 Quick way to remember:  
**log₂(n) ≈ the number of times you can split n things in half before reaching 1.**

### 🔎 1. General Rule

A code is **O(log n)** when:

* The input size gets **reduced by a constant factor each step** (usually divided by 2).
* Example: Binary Search halves the array each time → log n steps.

int binarySearch(int[] arr, int target) {

int left = 0, right = arr.length - 1;

while (left <= right) {

int mid = left + (right - left) / 2;

if (arr[mid] == target) return mid;

else if (arr[mid] < target) left = mid + 1;

else right = mid - 1;

}

return -1;

}

* First iteration: look at n elements
* Next iteration: look at n/2
* Then n/4 → n/8 → … until 1

👉 Number of iterations = log₂(n) → **O(log n)**

Example: Loop that divides

int count = 0;

while (n > 1) {

n = n / 2;

count++;

}

 Each loop divides n by 2

 How many times until n = 1? → log₂(n) times

 Complexity = **O(log n)**

**Shortcut to identify O(log n) code**:

* If inside the loop, the variable gets divided (/2, /3, etc.), or the search space halves → usually **O(log n)**.
* If it subtracts (-1, -2), that’s **O(n)**.

int i = 1;

while (i < n) {

i = i \* 2;

}

👉 Here, i grows:

* Step 1: 1
* Step 2: 2
* Step 3: 4
* Step 4: 8
* … until i ≥ n

ow many steps?

* If i = 2^k, then when 2^k ≥ n, loop stops.
* That means k = log₂(n) steps.

Do you want me to draw a **small table (visual)** showing:

* i++ → O(n)
* i \*= 2 → O(log n)
* i-- → O(n)
* i /= 2 → O(log n)`

## 1. What is Space Complexity?

👉 **Space Complexity** = the **amount of memory** an algorithm needs to run, based on the input size n.

It measures **extra memory** beyond the input data.  
This includes:

1. **Fixed part** → memory that doesn’t depend on input size (e.g., constants, program code).
2. **Variable part** → memory that depends on input size:
   * Variables
   * Data structures (arrays, hash tables, trees, etc.)
   * Function call stack (recursion uses memory!)

## 🔹 2. Components of Space Complexity

1. **Instruction Space** → memory for the code itself (ignore in analysis).
2. **Fixed Part** → simple variables, constants.
   * Example: int x, y; → O(1)
3. **Dynamic Part** → memory that changes with input size.
   * Example: int arr[n]; → O(n)
4. **Recursion Stack Space**
   * Each recursive call adds to the stack.

void printArray(int[] arr) {

for (int i = 0; i < arr.length; i++) {

System.out.println(arr[i]);

}

}

 Variables: i → O(1)

 Input array is given (not counted as extra).

 No recursion.  
👉 Space Complexity = **O(1)**

int[] copyArray(int[] arr) {

int[] newArr = new int[arr.length];

for (int i = 0; i < arr.length; i++) {

newArr[i] = arr[i];

}

return newArr;

}

* New array size = n → O(n)
* Variables: i → O(1)  
  👉 Space Complexity = **O(n)**

int factorial(int n) {

if (n == 0) return 1;

return n \* factorial(n - 1);

}

 Each recursive call adds a stack frame (stores n and return address).

 Depth of recursion = n  
👉 Space Complexity = **O(n)**

int binarySearch(int[] arr, int x) {

int left = 0, right = arr.length - 1;

while (left <= right) {

int mid = left + (right - left) / 2;

if (arr[mid] == x) return mid;

else if (arr[mid] < x) left = mid + 1;

else right = mid - 1;

}

return -1;

}

 Variables: left, right, mid → constant

 No recursion  
👉 Space Complexity = **O(1)**

## 🔹 4. Shortcut to Calculate

✅ Ask yourself:

1. Does the algorithm use **extra arrays/lists**?
   * Yes → O(n) or more
2. Does it use **recursion**?
   * Yes → consider stack depth
3. Otherwise → usually O(1)

💡 **Summary:**

* **O(1)** → constant extra memory (binary search, simple loops)
* **O(n)** → extra array/structure same size as input
* **O(log n)** → recursive divide & conquer (like binary search recursive version)
* **O(n²)** → 2D arrays or matrices

