⏳ What Is Time Complexity?

Time Complexity tells us how long an algorithm takes to run as the input size increases.
It describes the growth rate of an algorithm's running time, not the actual time in seconds.

🚀 Why Are We Learning Time and Space Complexity?

When solving coding problems on platforms like **LeetCode**, you might encounter a **TLE (Time Limit Exceeded) and Memory Limit** error after submitting your solution.

By understanding how much **time** and **space** your code consumes, you can:

- Optimize your logic according to the **problem's constraints**
- Evaluate your solution's **efficiency**
- Prevent performance issues like **TLE**
- Make smarter decisions when choosing **algorithms and data structures**

Learning time and space complexity helps you write **faster**, **more efficient**, and **scalable** code.

Which algorithm is better?: the one with the least time and space complexity

⏱️ Ways to Determine Time Complexity

Time complexity can be determined using two methods:

1. **Experimental Method**
2. **Asymptotic Notation**

🧪 Experimental Analysis

Method: Measure actual runtime using system clock.
We will use Java's built-in functionality to record the start and end time, rather than manually checking the clock, for greater accuracy and convenience.

```java
long start = System.currentTimeMillis();
for (int i = 0; i < 1000000; i++) {
    // System.out.println("Hello Akarsh");
}
long end = System.currentTimeMillis();
System.out.println("Time taken to print Akarsh: " + (end - start));
```

Limitation: Depends on machine(hardware), compiler, and input.

The reason this logic fails is that I cannot be biased based on hardware. Suppose someone is using a supercomputer — that doesn't mean their solution is always the most optimized or the smartest. I want to judge an algorithm based on consistent, fair parameters. Everyone should be evaluated on the same criteria, regardless of hardware.

For example, if one person is using a 32GB RAM machine and another is using a 4GB RAM machine, the person with the 4GB machine might have written a more efficient algorithm, but it may not execute as fast due to hardware limitations. That doesn't make the logic any less effective.

🧮 Asymptotic Notation

Dependent on user input, not on the system.

“Number of times a repetitive statement is running”

It is divided into three main types:

1. Worst Case (Big-O)
2. Best Case (Omega)
3. Average Case (Theta)

📈 Graph Setup

- **X-axis**: Input size `n`
- **Y-axis**: Time taken `f(n)`
- **g(n)**: A reference function (e.g., `n`, `n log n`, etc.)

We’ll draw curves to compare `f(n)` (your algorithm’s actual time) with `g(n)` (the benchmark).

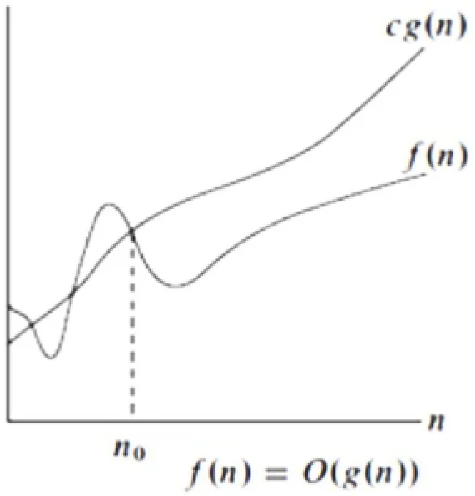
🚨 1. Worst Case – Big-O Notation (O) – Upper Bound

When we talk about the **worst case**, we refer to the scenario **where an algorithm takes the maximum possible time or resources to complete.**

💼 **Example:**

Think of Anil Ambani. Even during his worst time — when he was declared bankrupt — he still had a net worth of over a billion dollars.

📉 So even in the most unfavorable situation, the worst case tells us how much you still retain or how bad things can get.



$f(n) = O(g(n))$

📐 **Definition**:

There exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$:

f(n) ≤ c · g(n)

✅ **Meaning:**

Your algorithm will **never take more time than** c · g(n) for large enough n.

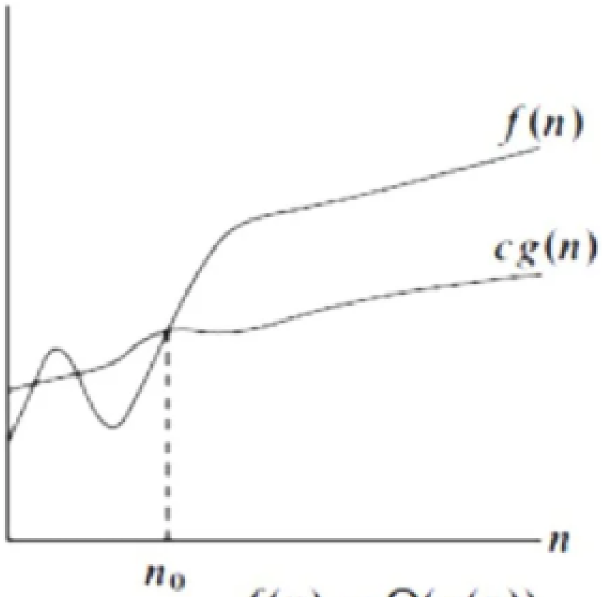
🎯 **Interpretation:**

- `f(n)` stays **below or touches** `c·g(n)`
- Think of `c·g(n)` as a **ceiling** your algorithm never crosses

🌟 2. Best Case – Omega Notation (Ω) – Lower Bound

The **best case** describes the minimum time or space an algorithm will take — **the most optimistic scenario**, usually when the input is already in the most favorable form.

🚀 **Example:**
Imagine you go to a restaurant and there's **no queue at all**. You instantly get seated, place your order, and receive your food.
🍽️ This is the **best case** — everything is perfect, minimal waiting, and everything goes your way.



$f(n) = \Omega(g(n))$

📐 **Definition**:

There exist constants $c > 0$ and $n_0 \ge 0$ such that for all $n \ge n_0$:

f(n) ≥ c·g(n)

✅ **Meaning:**

Your algorithm will **always take at least** $c \cdot g(n)$ time for large enough n.

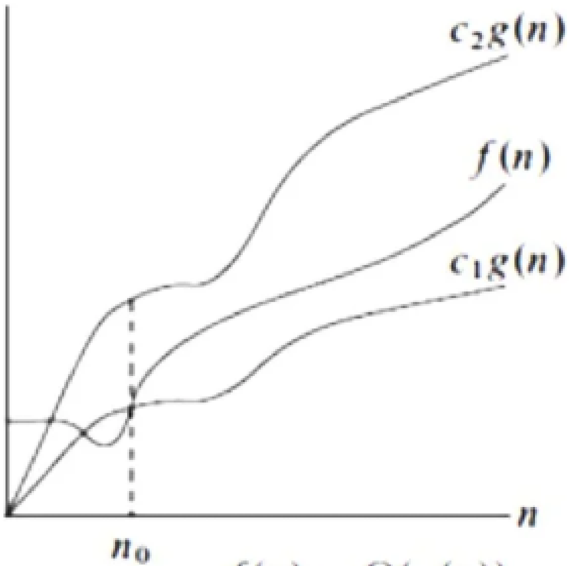
🎯 **Interpretation:**

- $f(n)$ stays **above or touches** $c \cdot g(n)$
- Think of $c \cdot g(n)$ as a **floor** your algorithm never dips below

⚖️ 3. Average Case – Theta Notation (Θ) – Tight Bound

The **average case** provides an estimate of the time or space an algorithm will take on **typical or random inputs**. It gives a more realistic performance measure.

📊 **Example:**
Think of **daily traffic** during your commute.
🚗 Some days it's smooth, other days it's jammed — but over time, you get an **average** idea of how long your trip usually takes.
⏱️ That's the **average case** — not too good, not too bad, just the expected behavior in most scenarios.



$f(n) = \Theta(g(n))$

📐 **Definition**:

There exist constants $c_1 > 0$, $c_2 > 0$, and $n_0 \geq 0$ such that for all $n \geq n_0$:

c1 · g(n) ≤ f(n) ≤ c2 · g(n)

✅ **Meaning:**

Your algorithm's time is **sandwiched** between two multiples of $g(n)$—not too fast, not too slow.

🎯 **Interpretation:**

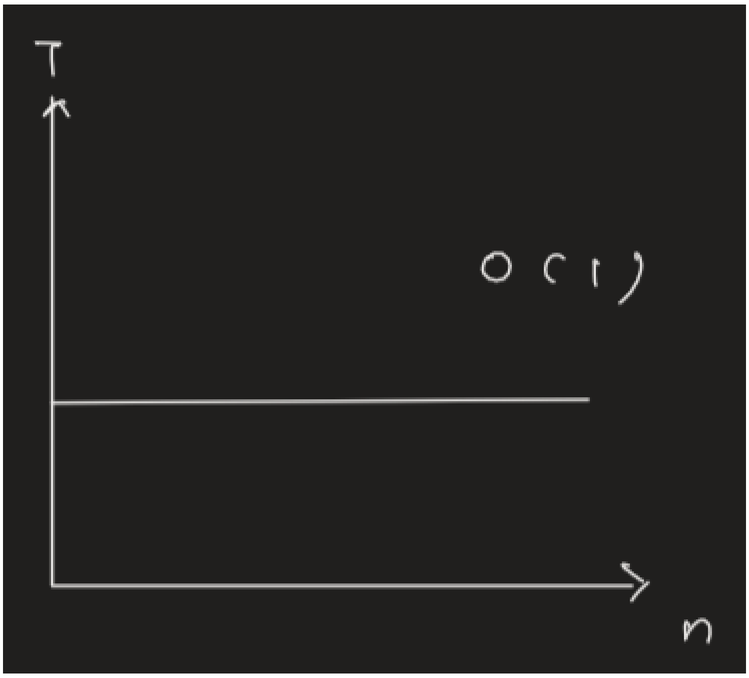
- $f(n)$ lies **between** $c_1 \cdot g(n)$ and $c_2 \cdot g(n)$
- Think of it as a **speed range** your algorithm always stays within

☕ Understanding Constant Time Complexity – O(1)

In Java, operations like **mathematical expressions, logical operators, relational operators, bitwise operators, variable declarations, and assignments** generally execute in **constant time**, denoted as **O(1)**.

❓ What Does Constant Time, O(1), Mean?

O(1) means that **as the size of the input (n) grows, the time taken by the operation remains the same**.
In simple terms: **The operation does not scale with input size — it always takes the same amount of time to run.**



✅ O(1) Operations Examples

1. Mathematical Expressions

```
int sum = a + b;
int product = x * y;
```

2. Logical Operators

```
boolean result = (a > 0) && (b < 10);
```

3. Relational Operators

```
boolean isEqual = a == b;
boolean isGreater = x > y;
```

4. Bitwise Operators

```
int result = a & b;
int shifted = x << 1;
```

5. Array Access by Index

```
int value = arr[5];
arr[2] = 100;
```

✅ Variable Declaration

Declaring a variable tells the Java compiler what type it is and reserves space in memory, This is a **constant-time operation** — O(1).

```
int a;
String name;
double salary;
```

✅ Variable Assignment

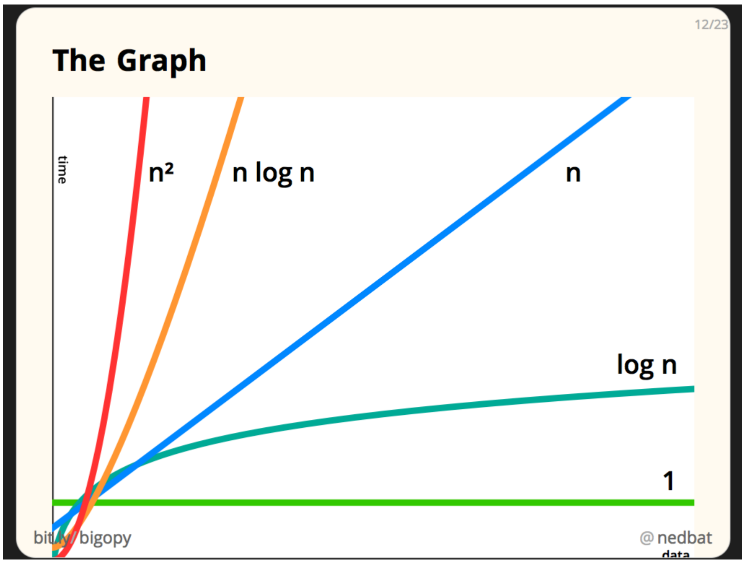
Assigning a value stores data in memory, Also **O(1)**, since you're just storing a value.

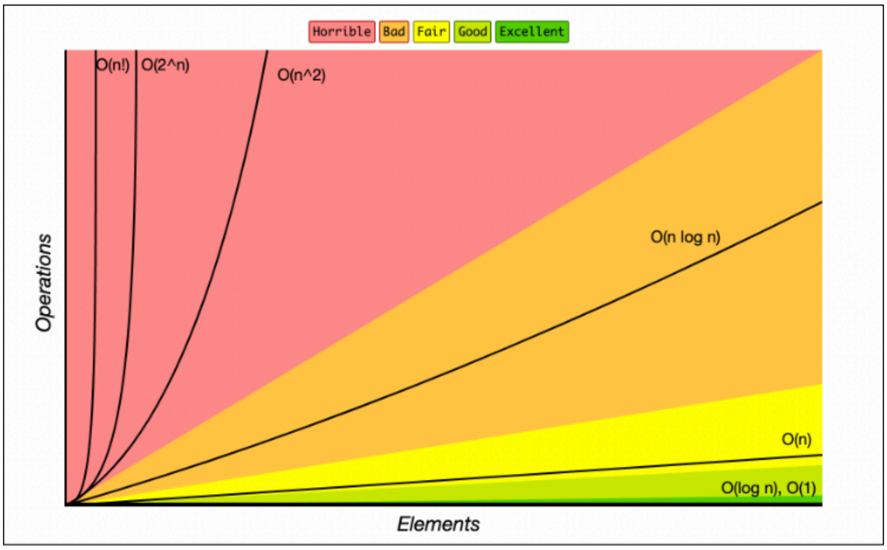
```
a = 10;
name = "Alice";
salary = 50000.00;
```

🚀 Always pick the largest power/value & Ignore Constants

🔍 **Example1:** $f(n) = 5n^2+2n^5+3n+2$, Time complexity = $O(n^5)$

🔍 **Example2:** $f(n) = 2^n+2n^5+3n+2$, Time complexity = $O(2^n)$





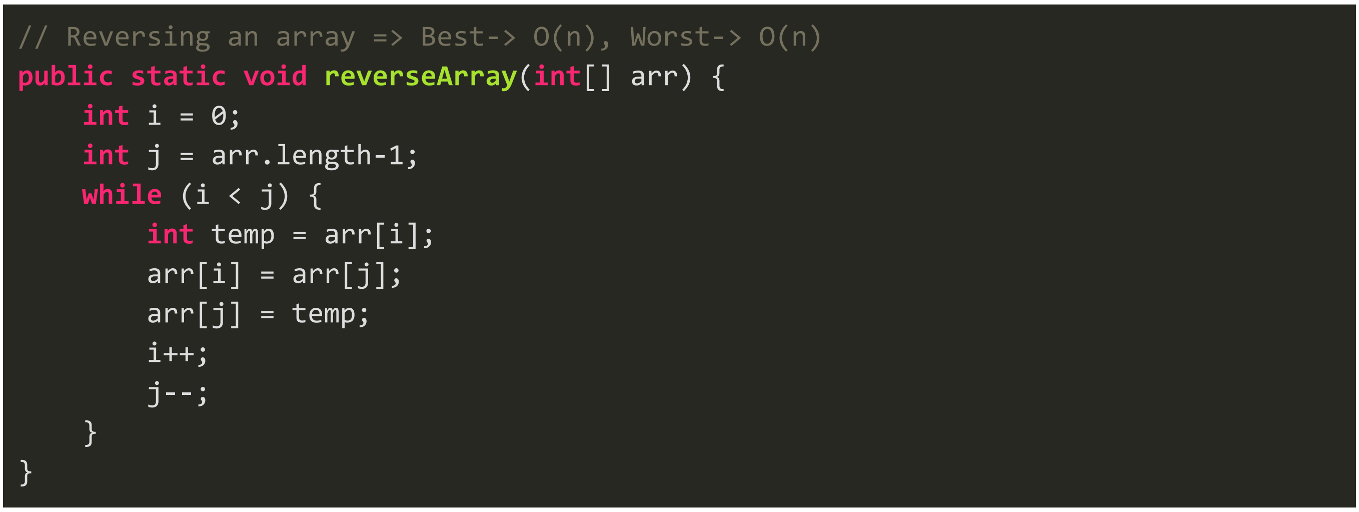
🎯 Code Snippets

```java
// Best/Worst-> O(5) = O(1)
System.out.println("Hello Akarsh");
System.out.println("Hello Akarsh");
System.out.println("Hello Akarsh");
System.out.println("Hello Akarsh");
System.out.println("Hello Akarsh");
```

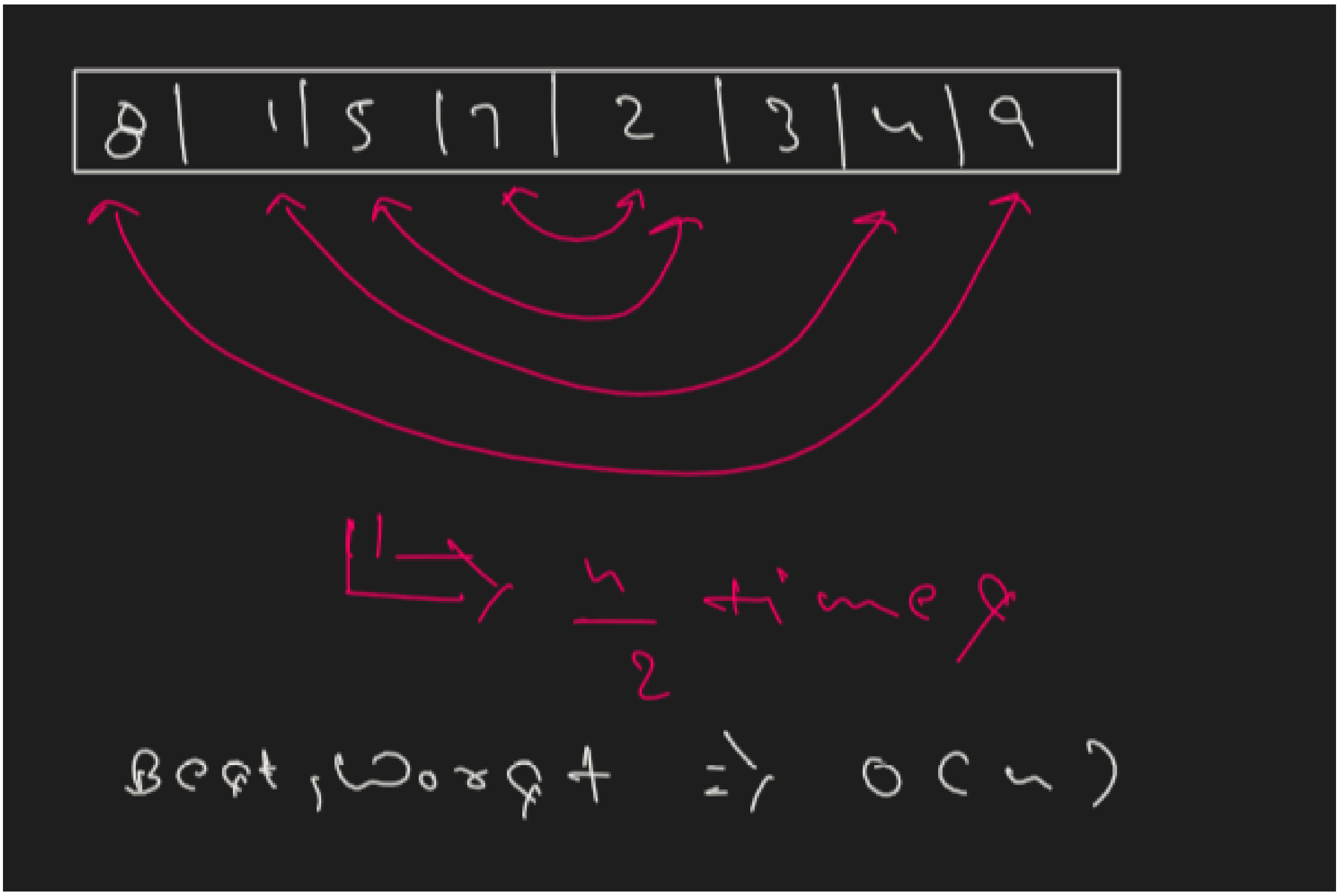
```java
// Linear Search => Best-> O(1), Worst-> O(n)
public static int linearSearch(int[] arr, int item) {
    for (int i = 0; i < arr.length; i++) {
        if (arr[i] == item) {
            return i;
        }
    }
    return -1;
}
```

```java
// Maximum value in an array => Best/Worst-> O(n)
public static int maxValue2(int[] arr) {
    int max = Integer.MIN_VALUE;
    for (int i = 0; i < arr.length; i++) {
        if (arr[i] > max) {
            max = arr[i];
        }
    }
    return max;
}
```

```java
// Reverse printing an array => Best-> O(n), Worst-> O(n)
public static void reversePrint(int[] arr) {
    for (int i = arr.length - 1; i >= 0; i--) {
        System.out.print(arr[i] + " ");
    }
    System.out.println();
}
```



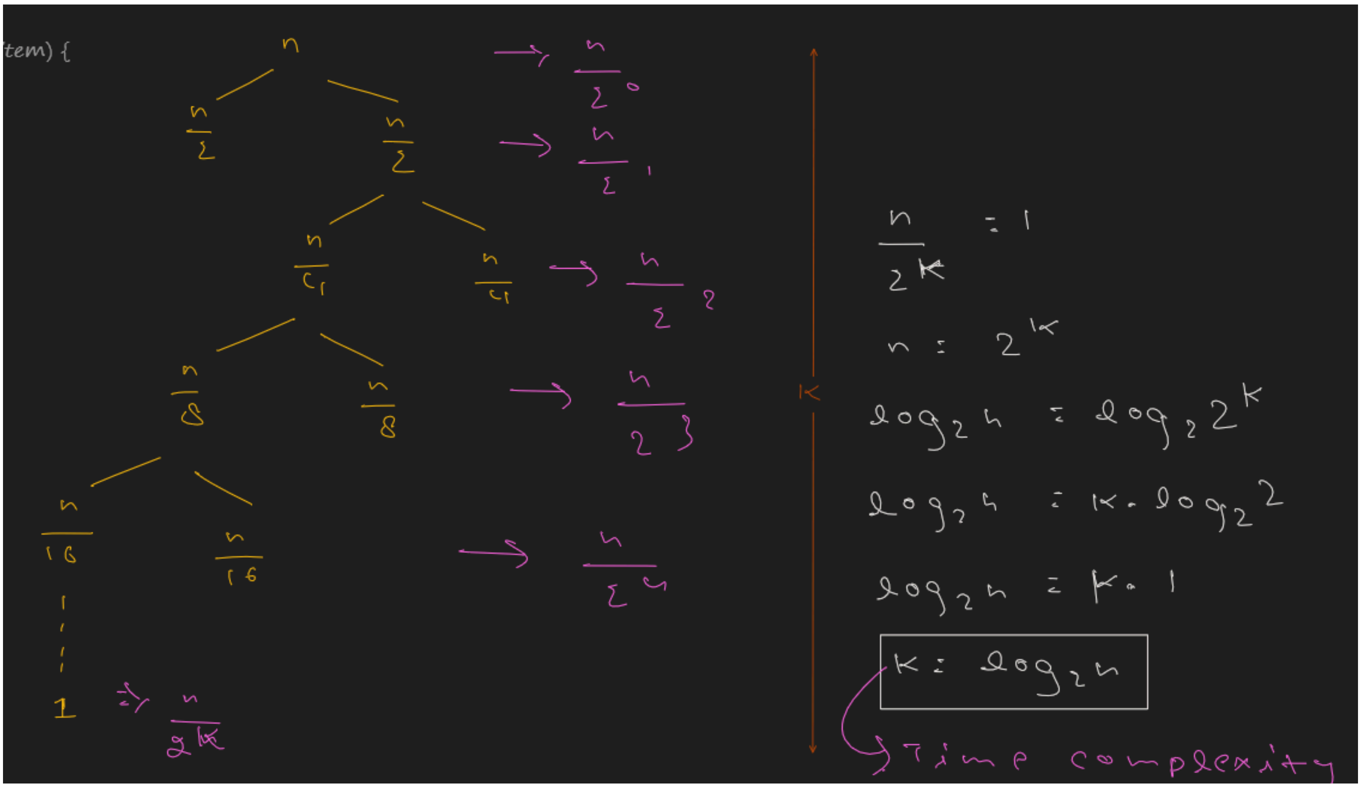
```java
// Reversing an array => Best-> O(n), Worst-> O(n)
public static void reverseArray(int[] arr) {
    int i = 0;
    int j = arr.length-1;
    while (i < j) {
        int temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
        i++;
        j--;
    }
}
```



$\Rightarrow \frac{n}{2}$ times

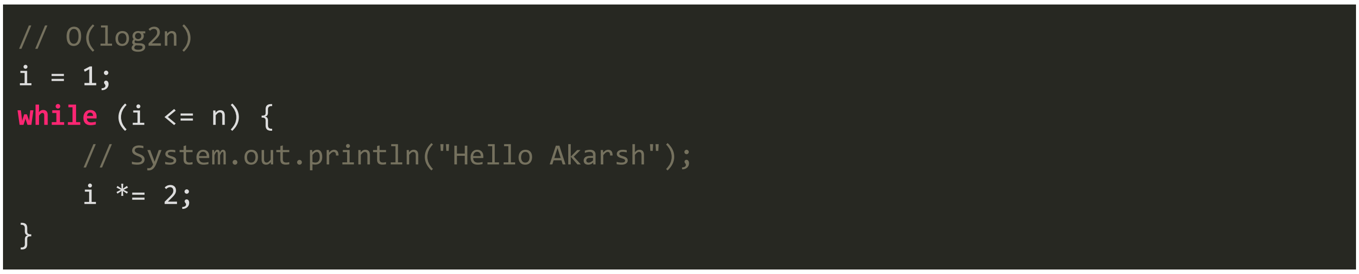
Best, Worst $\Rightarrow$ O(n)

```java
// Binary Search => Best-> O(1), Worst-> O(log₂n)
public static int binarySearch(int[] arr, int item) {
    int n = arr.length;
    int low = 0;
    int high = n - 1;
    while (low <= high) {
        int mid = (low + high) / 2;
        if (arr[mid] == item) {
            return mid;
        } else if (arr[mid] > item) {
            high = mid - 1;
        } else {
            low = mid + 1;
        }
    }
    return -1;
}
```



```java
int n = 566789;
int i = 0;
```

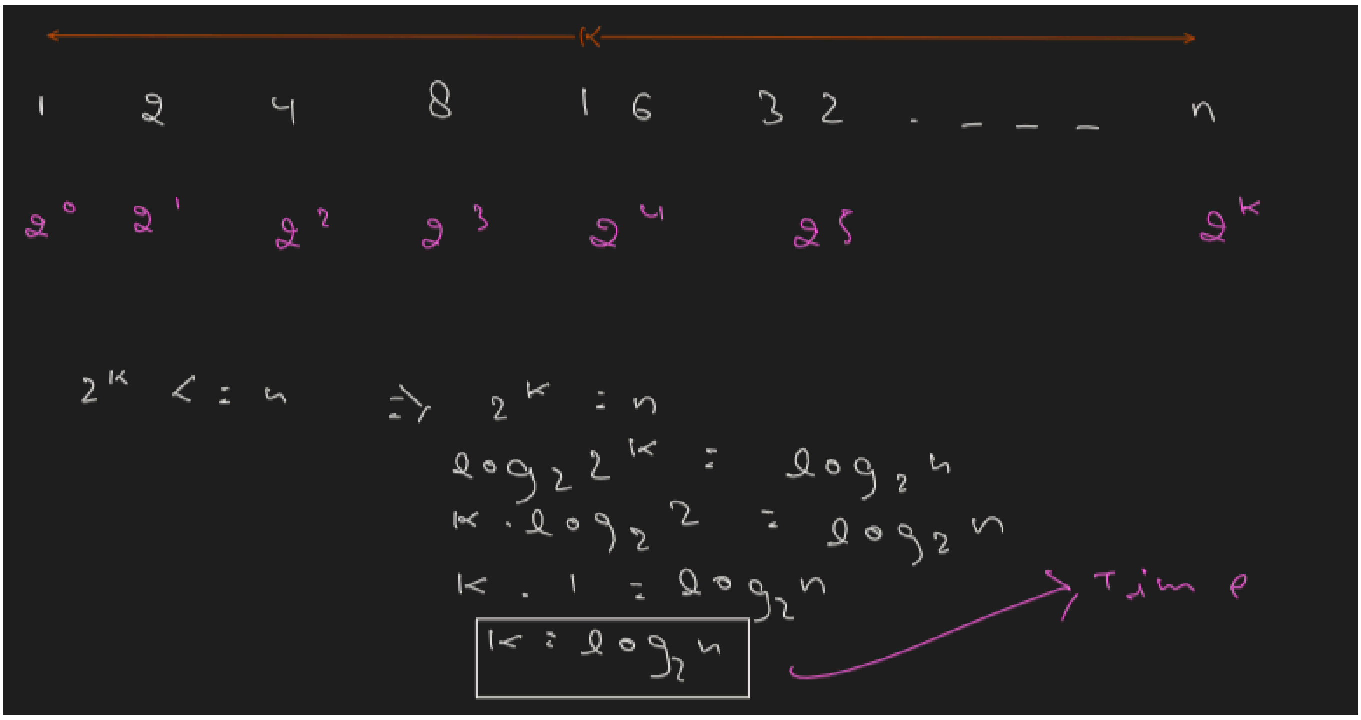
```java
// O(n)
while (i < n) {
    // System.out.println("Hello Akarsh");
    i++;
}
```



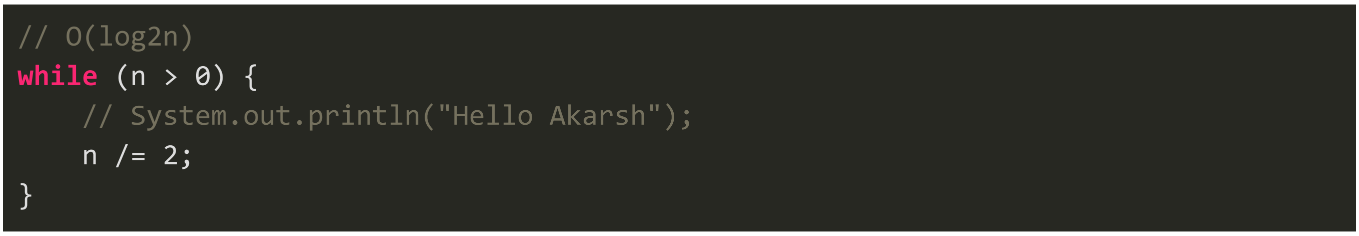
```
// O(log2n)
i = 1;
while (i <= n) {
    // System.out.println("Hello Akarsh");
    i *= 2;
}
```

What's happening?

- `i` starts at 1 and doubles each time: 1, 2, 4, 8, 16, ...
- It stops when `i > n`
- So it runs until: **$2^k > n$**
- Taking log base 2: **$k > \log_2(n)$**



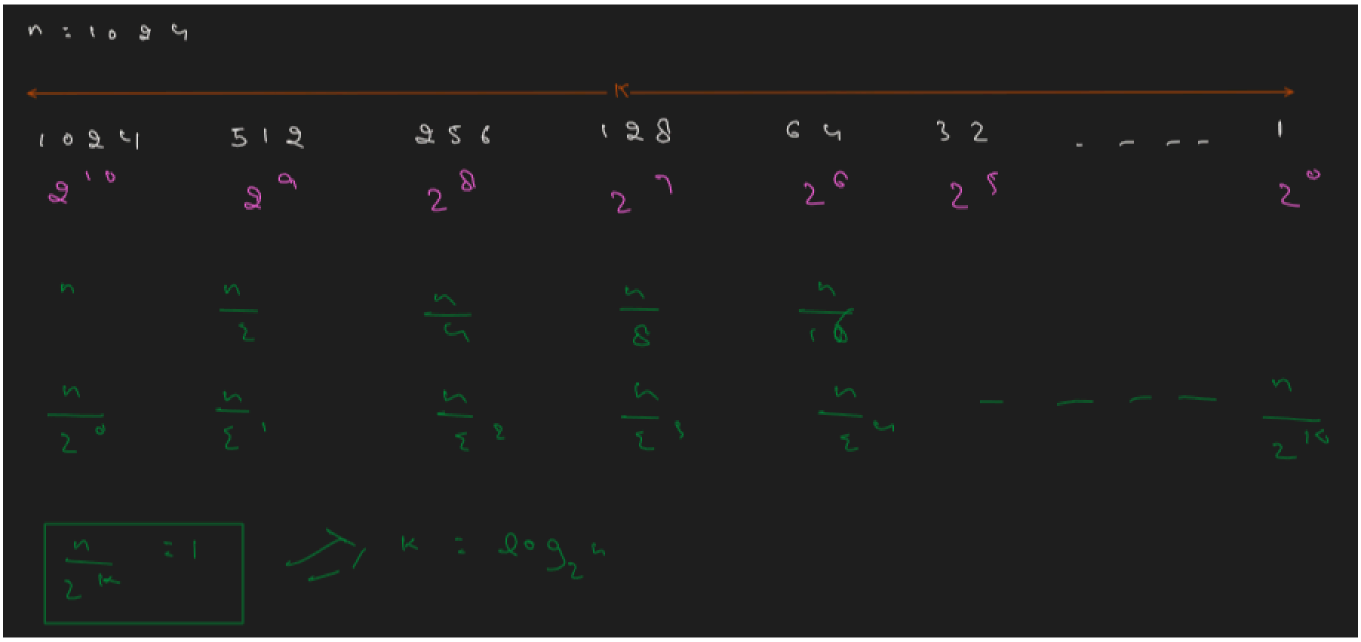
$$2^k <= n \Rightarrow 2^k = n$$
$$\log_2 2^k = \log_2 n$$
$$k \cdot \log_2 2 = \log_2 n$$
$$k \cdot 1 = \log_2 n$$
$$\boxed{k = \log_2 n} \rightarrow \text{Time}$$



```
// O(log2n)
while (n > 0) {
    // System.out.println("Hello Akarsh");
    n /= 2;
}
```

What's happening?

- Each iteration: n=n/2 [64, 32, 16, 8, 4, 2, 1, 0]
- After k steps: $n \leq n_0/2^k$
- Loop stops when: $n_0/2^k < 1 \Rightarrow 2^k > n_0 \Rightarrow k = \log_2 n_0$



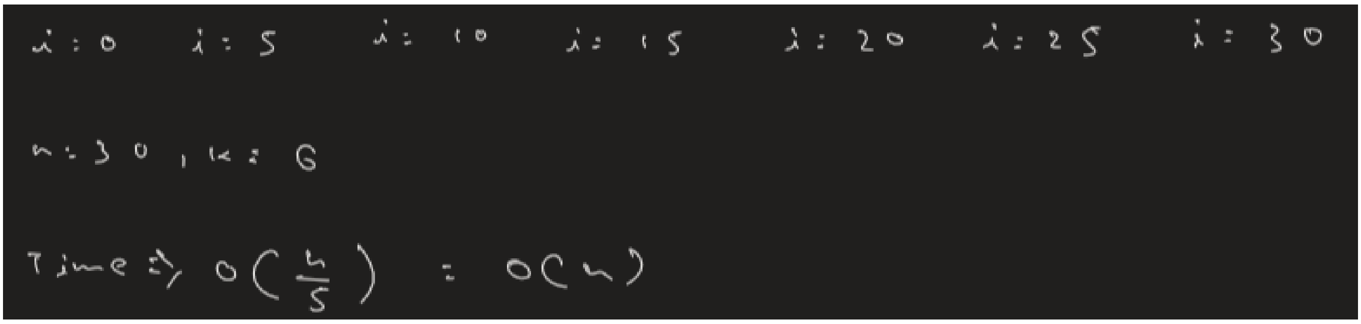
n : 1024

k

1024	512	256	128	64	32	----	1
2^{10}	2^9	2^8	2^7	2^6	2^5		2^0
n	$\frac{n}{2}$	$\frac{n}{4}$	$\frac{n}{8}$	$\frac{n}{16}$			
$\frac{n}{2^0}$	$\frac{n}{2^1}$	$\frac{n}{2^2}$	$\frac{n}{2^3}$	$\frac{n}{2^4}$	----		$\frac{n}{2^{10}}$

$\boxed{\frac{n}{2^k} = 1} \Rightarrow k = \log_2 n$

```
// O(n)
while (i <= n) {
    // System.out.println("Hello Akarsh");
    i += 2;
    i += 3;
}
```

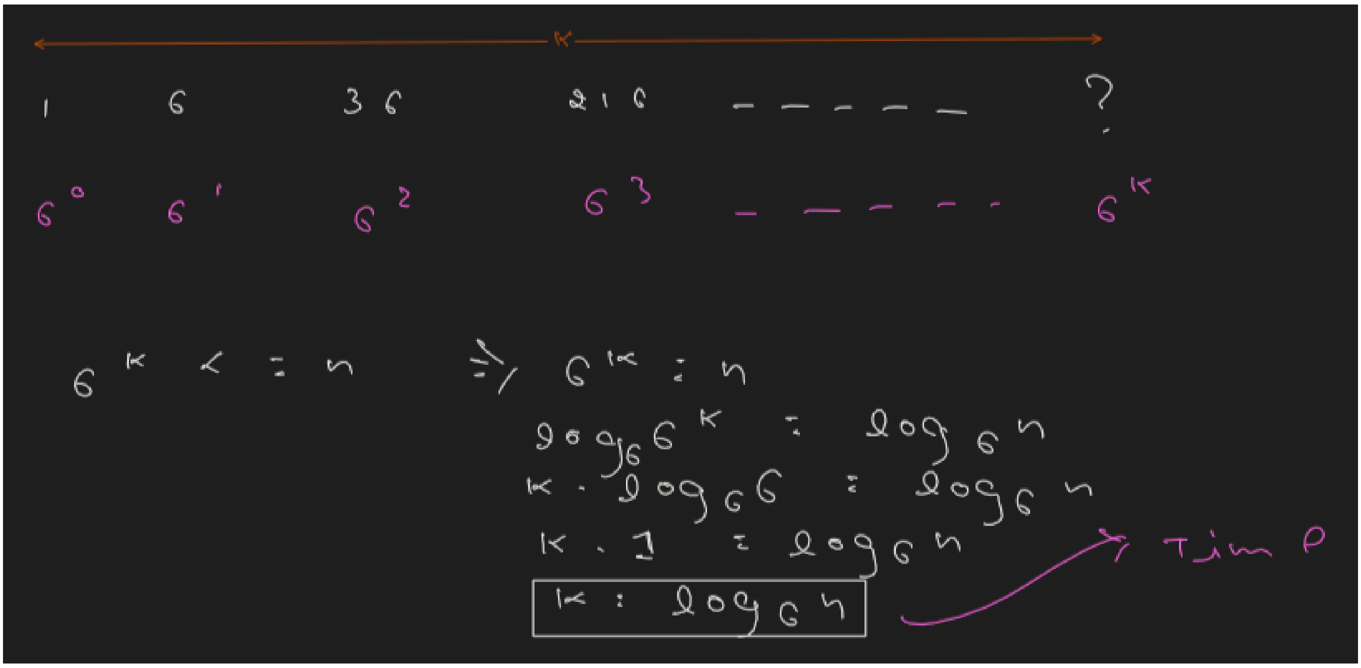


i : 0 i : 5 i : 10 i : 15 i : 20 i : 25 i : 30

n : 30 , k : 6

Time $\Rightarrow O(\frac{n}{5}) = O(n)$

```
// O(log6n)
while (i <= n) {
    // System.out.println("Hello Akarsh");
    i *= 2;
    i *= 3;
}
```



$$\overleftrightarrow{\quad k \quad}$$

$1 \quad 6 \quad 36 \quad 216 \quad ----- \quad ?$

$6^0 \quad 6^1 \quad 6^2 \quad 6^3 \quad ----- \quad 6^k$

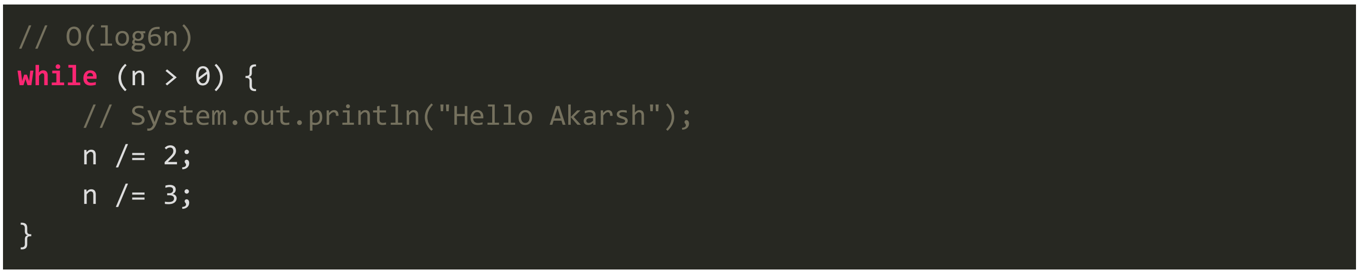
$6^k <= n \Rightarrow 6^k = n$

$\log_6 6^k = \log_6 n$

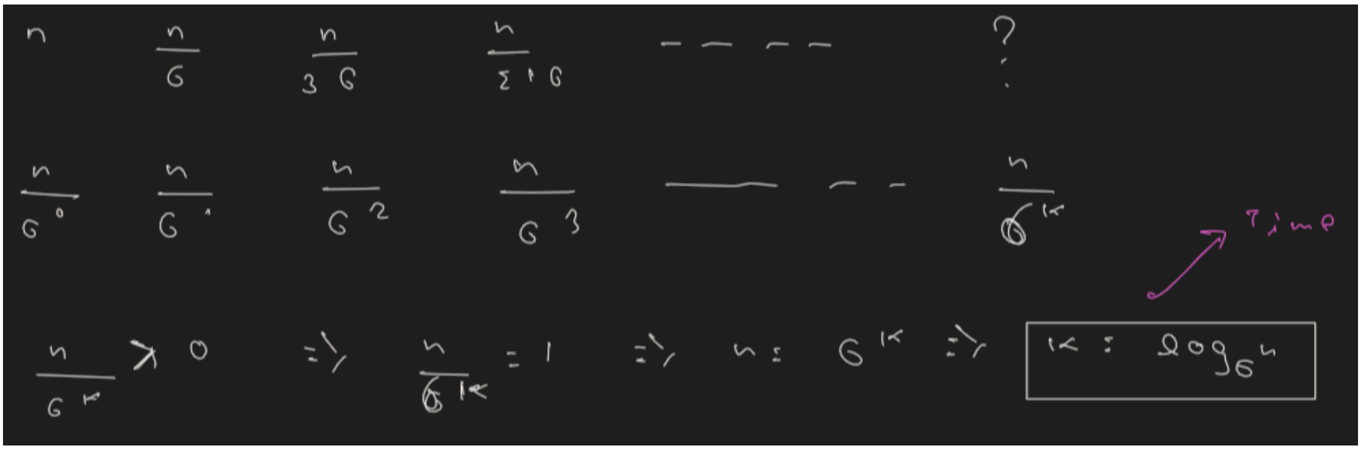
$k \cdot \log_6 6 = \log_6 n$

$k \cdot 1 = \log_6 n$

$\boxed{k = \log_6 n}$ → Time



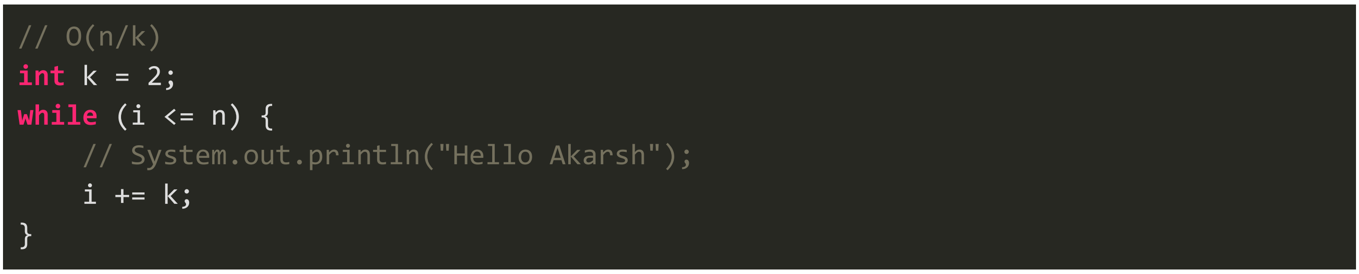
```
// O(log6n)
while (n > 0) {
    // System.out.println("Hello Akarsh");
    n /= 2;
    n /= 3;
}
```



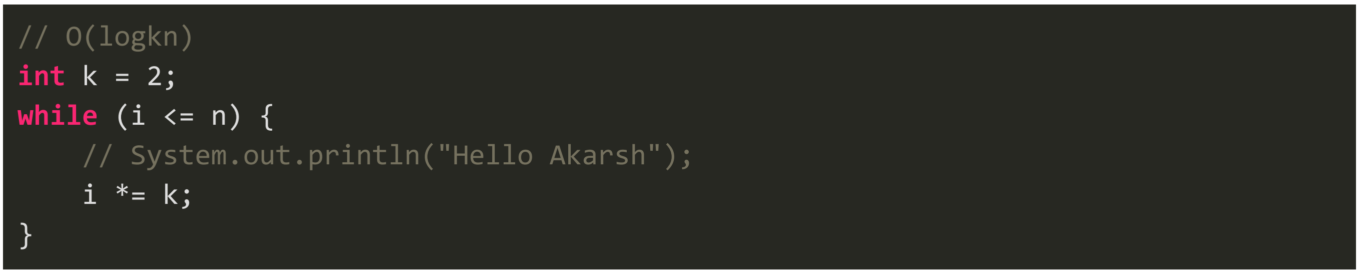
$n \quad \frac{n}{6} \quad \frac{n}{36} \quad \frac{n}{216} \quad ---- \quad ?$

$\frac{n}{6^0} \quad \frac{n}{6^1} \quad \frac{n}{6^2} \quad \frac{n}{6^3} \quad ---- \quad \frac{n}{6^k}$

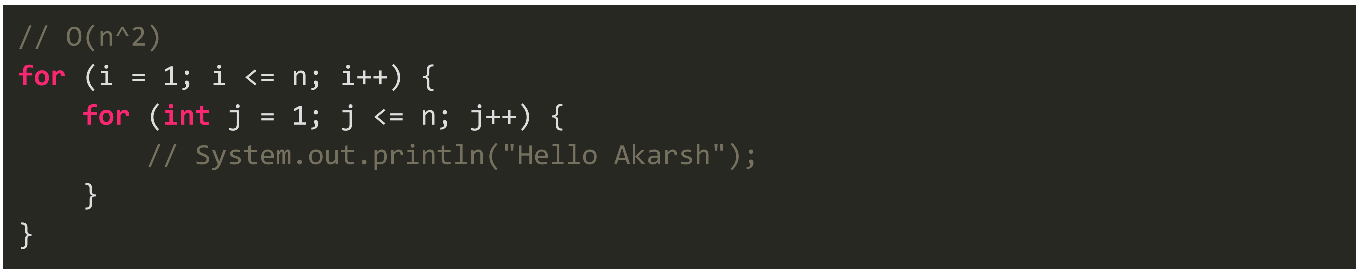
$\frac{n}{6^k} > 0 \Rightarrow \frac{n}{6^k} = 1 \Rightarrow n = 6^k \Rightarrow \boxed{k = \log_6 n}$ → Time



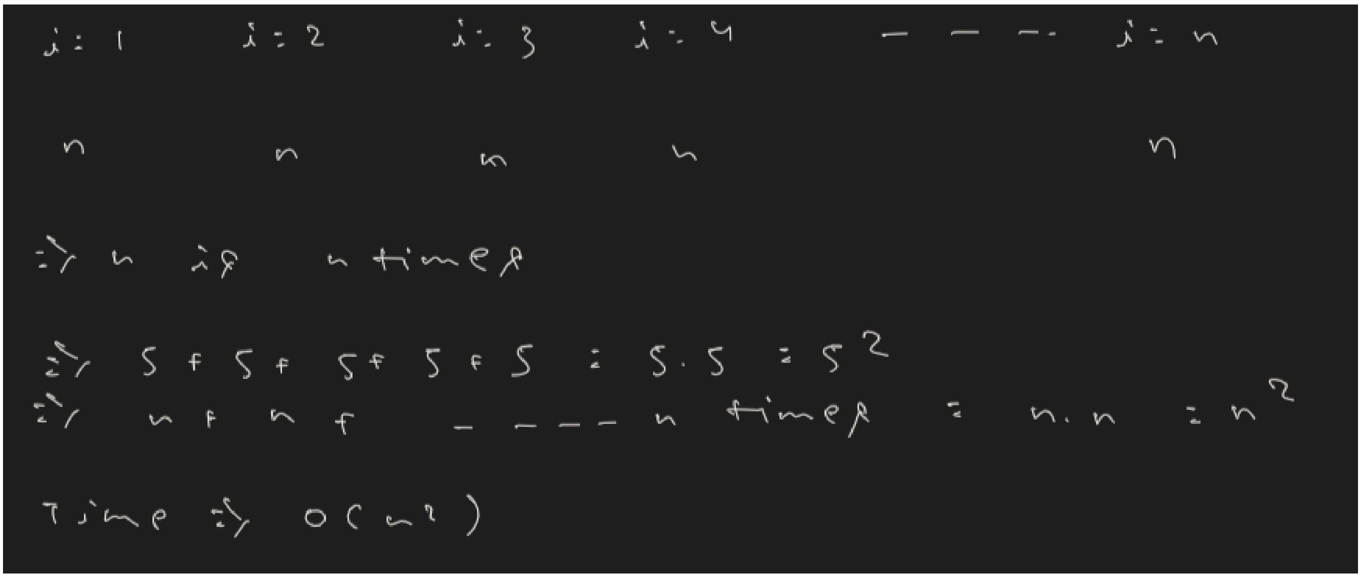
```
// O(n/k)
int k = 2;
while (i <= n) {
    // System.out.println("Hello Akarsh");
    i += k;
}
```



```
// O(logkn)
int k = 2;
while (i <= n) {
    // System.out.println("Hello Akarsh");
    i *= k;
}
```



```
// O(n^2)
for (i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
        // System.out.println("Hello Akarsh");
    }
}
```



$i:1 \quad i:2 \quad i:3 \quad i:4 \quad - - - \quad i:n$

$n \quad n \quad n \quad n \quad n$

$\Rightarrow$ n is n times

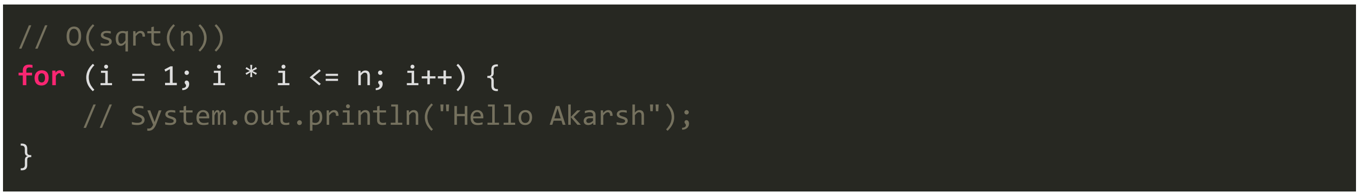
$\Rightarrow 5 + 5 + 5 + 5 + 5 = 5 \cdot 5 = 5^2$

$\Rightarrow n + n + - - - -$ n times $= n \cdot n = n^2$

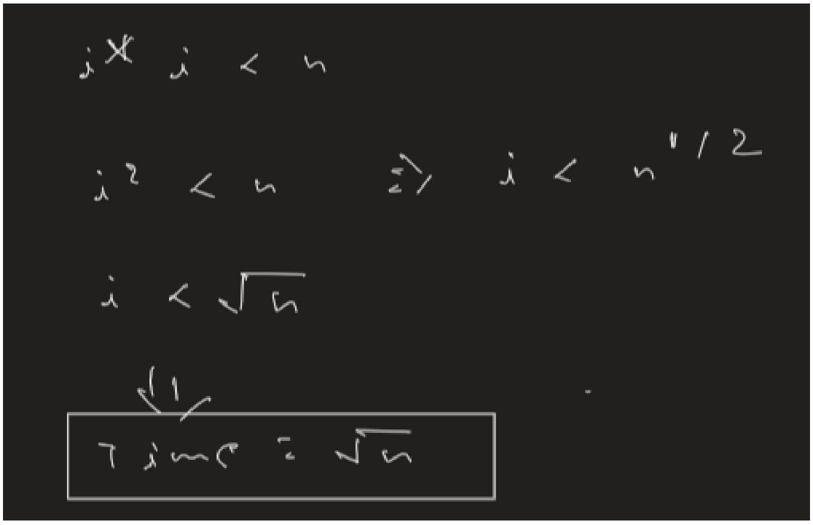
Time $\Rightarrow O(n^2)$

Nested loops can be of two types: dependent and independent.

- **Dependent loops** mean that the inner loop depends on the outer loop. This means that the range or condition of the inner loop uses a variable that is defined or controlled by the outer loop.
- **Independent loops** are loops where both the outer and inner loops operate independently. The inner loop does not use any variables from the outer loop.



```
// O(sqrt(n))
for (i = 1; i * i <= n; i++) {
    // System.out.println("Hello Akarsh");
}
```

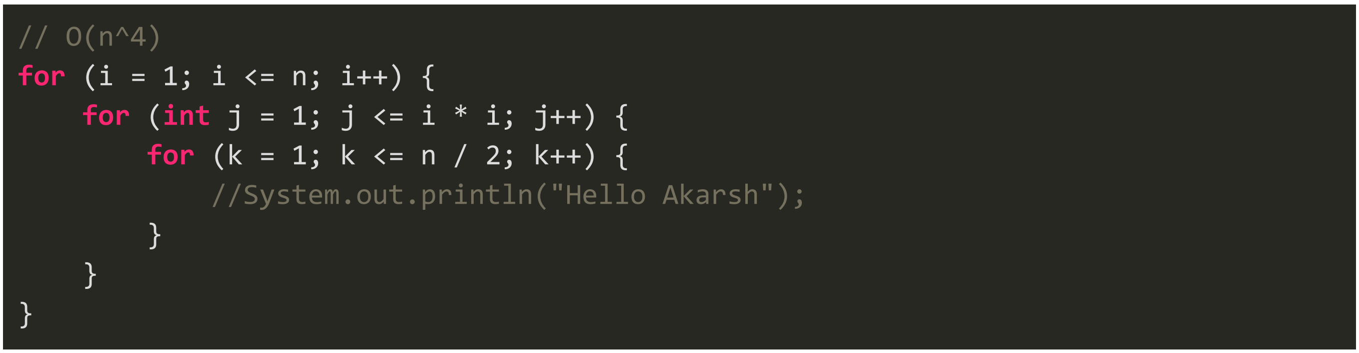


$i \times i < n$

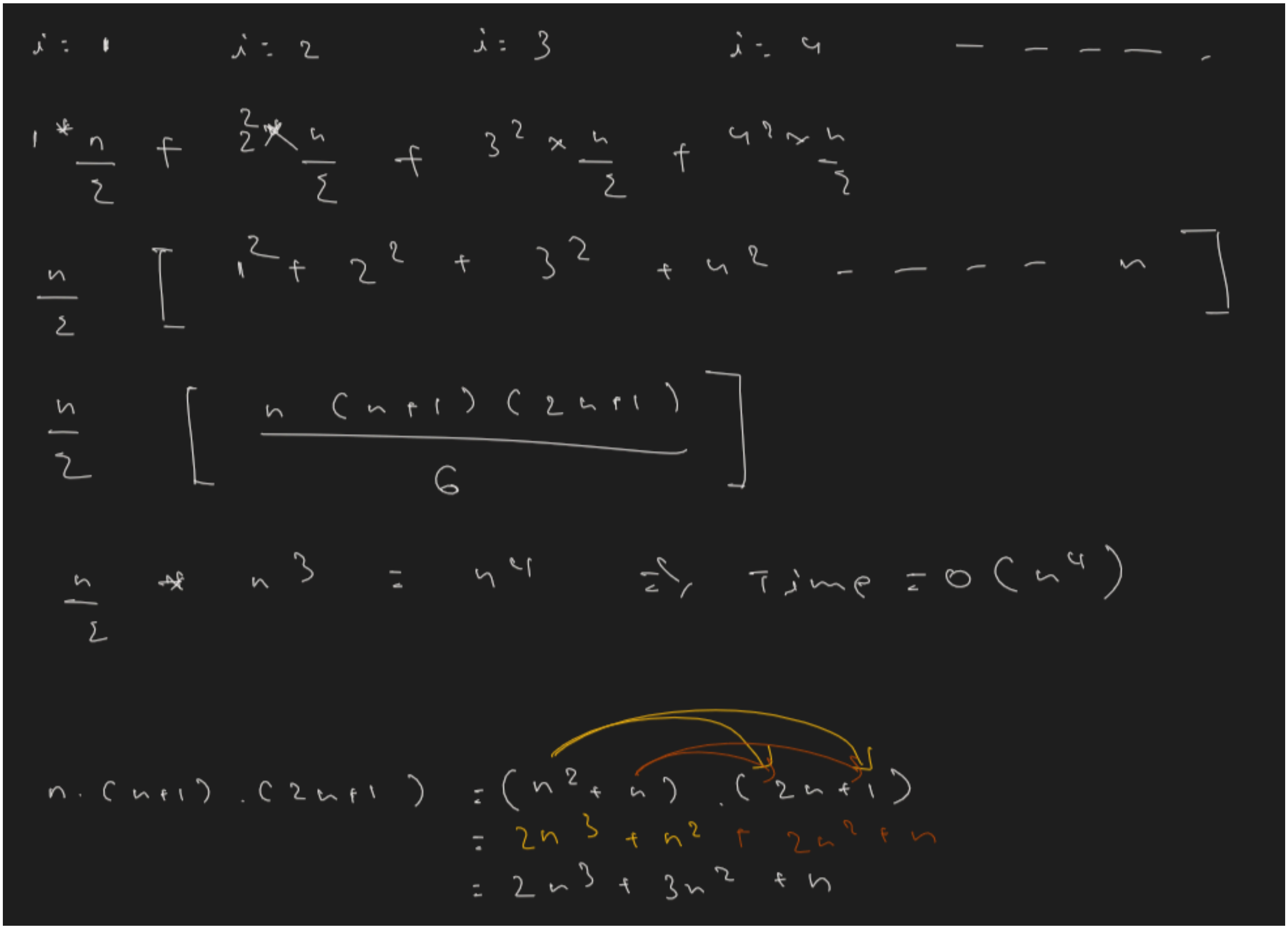
$i^2 < n \Rightarrow i < n^{1/2}$

$i < \sqrt{n}$

$\boxed{\text{Time} = \sqrt{n}}$



```
// O(n^4)
for (i = 1; i <= n; i++) {
    for (int j = 1; j <= i * i; j++) {
        for (k = 1; k <= n / 2; k++) {
            //System.out.println("Hello Akarsh");
        }
    }
}
```



$i : 1$ $i : 2$ $i : 3$ $i : 4$ ---- .

$1 * \frac{n}{2} + 2^2 * \frac{n}{2} + 3^2 \times \frac{n}{2} + 4^2 \times \frac{n}{2}$

$\frac{n}{2} \left[1^2 + 2^2 + 3^2 + 4^2 - - - - n \right]$

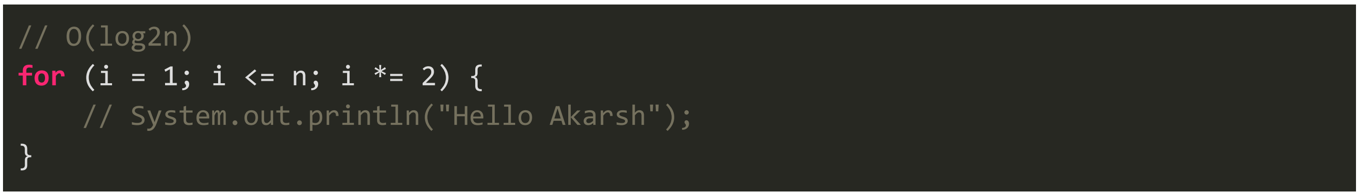
$\frac{n}{2} \left[\frac{n(n+1)(2n+1)}{6} \right]$

$\frac{n}{2} * n^3 = n^4 \Rightarrow$ Time $= O(n^4)$

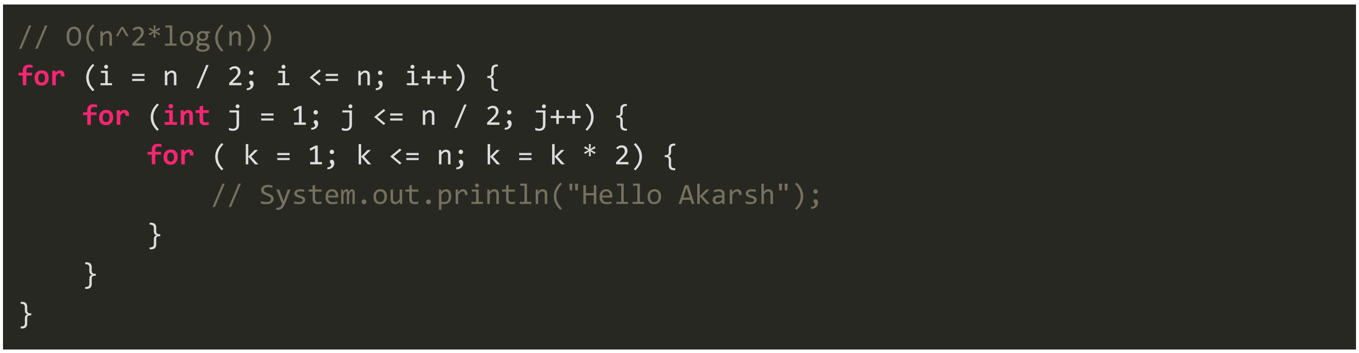
$n.(n+1).(2n+1) = (n^2 + n).(2n+1)$

$= 2n^3 + n^2 + 2n^2 + n$

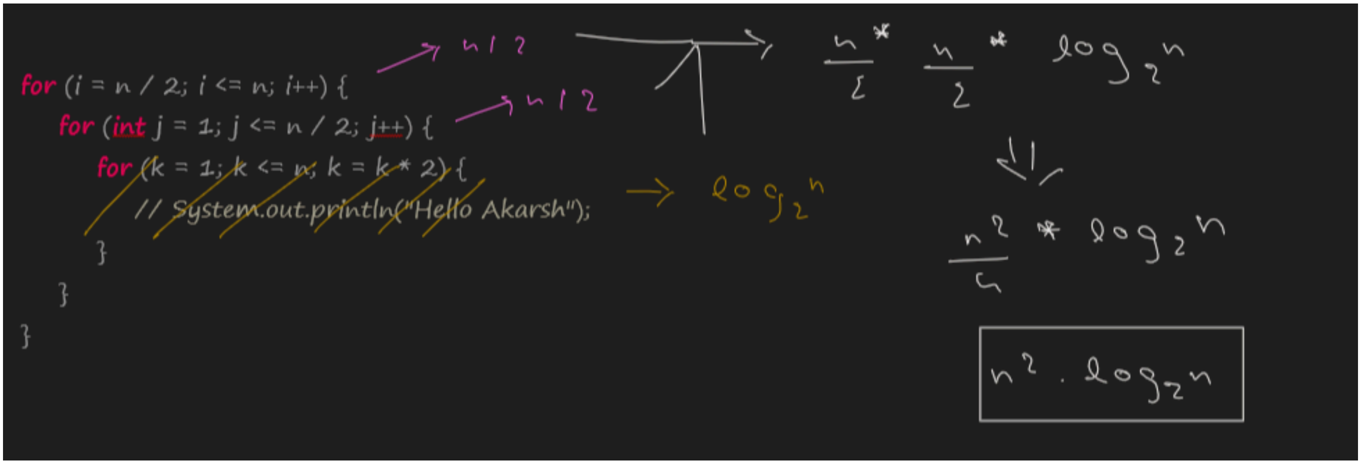
$= 2n^3 + 3n^2 + n$



```
// O(log2n)
for (i = 1; i <= n; i *= 2) {
    // System.out.println("Hello Akarsh");
}
```



```
// O(n^2*log(n))
for (i = n / 2; i <= n; i++) {
    for (int j = 1; j <= n / 2; j++) {
        for ( k = 1; k <= n; k = k * 2) {
            // System.out.println("Hello Akarsh");
        }
    }
}
```



```
for (i = n / 2; i <= n; i++) {
    for (int j = 1; j <= n / 2; j++) {
        for (k = 1; k <= n; k = k * 2) {
            // System.out.println("Hello Akarsh");
        }
    }
}
```

→ n / 2

→ n / 2

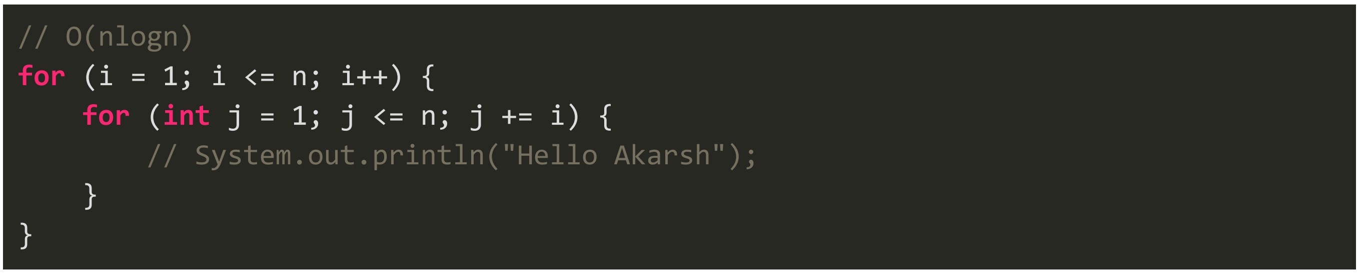
→ $\log_2 n$

$\frac{n}{2} * \frac{n}{2} * \log_2 n$

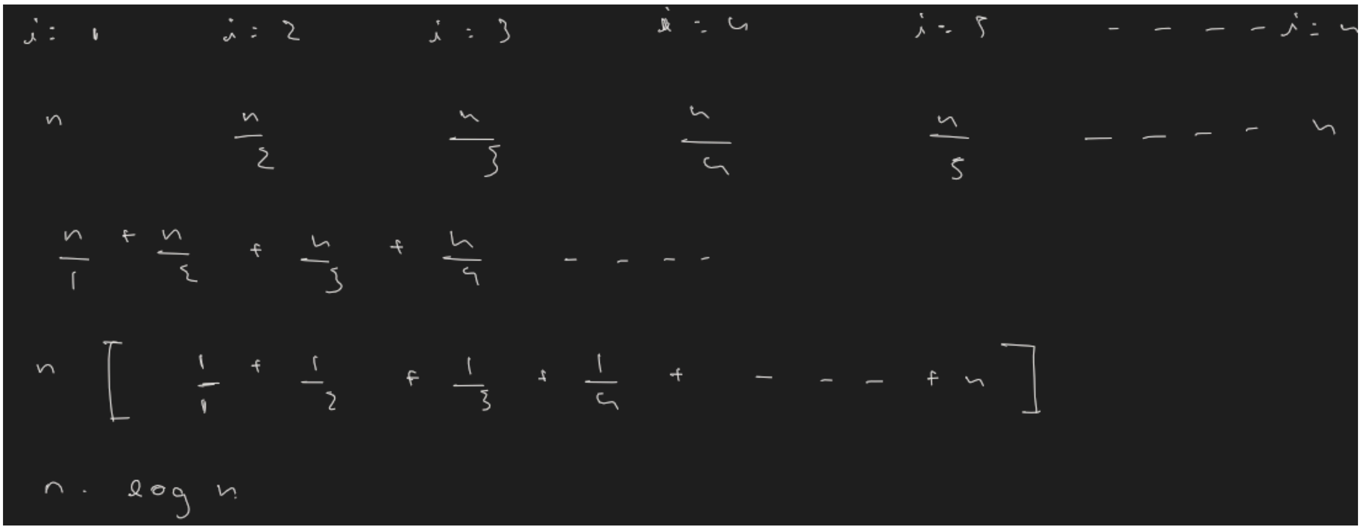
$\Downarrow$

$\frac{n^2}{4} * \log_2 n$

$\boxed{n^2 . \log_2 n}$



```
// O(nlogn)
for (i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j += i) {
        // System.out.println("Hello Akarsh");
    }
}
```



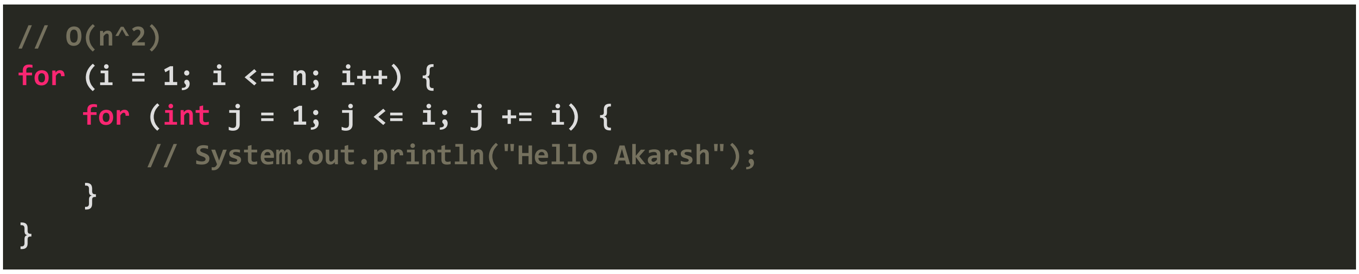
i: 1 i: 2 i: 3 i: 4 i: 5 - - - - i: n

$n \quad \frac{n}{2} \quad \frac{n}{3} \quad \frac{n}{4} \quad \frac{n}{5} \quad - - - - \quad n$

$\frac{n}{1} + \frac{n}{2} + \frac{n}{3} + \frac{n}{4} - - - -$

$n \left[\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + - - - + n \right]$

$n . \log n$



```
// O(n^2)
for (i = 1; i <= n; i++) {
    for (int j = 1; j <= i; j += i) {
        // System.out.println("Hello Akarsh");
    }
}
```

```
// O(n^2)
for (i = 1; i <= n; i++) {
    for (int j = i; j <= 1; j += i) {
        // System.out.println("Hello Akarsh");
    }
}
```

Note: Try to find Time Complexity of all the previous questions solved.

💾 What Is Space Complexity?

“Extra space we use in program”

Space Complexity in Java refers to how much **memory** (RAM) an algorithm uses with respect to the **input size**. It includes: **Auxiliary space** (like additional arrays, recursive stack, objects, etc.)

Space complexity of primitive variables is constant.

Also we **ignore the space complexity of the input** that is required.

🔍 Example1

```
int[] arr = {1, 2, 3, 4, 5};
for (int i : arr) {
    System.out.println(i);
}
```

No extra space used → **Space Complexity: O(1)**

🔍 Example2

```
int[] original = {1, 2, 3};
int[] copy = new int[original.length];
for (int i = 0; i < original.length; i++) {
    copy[i] = original[i];
}
```

You created a new array → **Space Complexity: O(n)**

🚀 How to Decide If Your Solution Will Work

In most online judges, you typically get ⏱ **1 second** to run your code.

🧠 **Rule of Thumb:** A modern computer can execute about `10^8` **operations per second**.

To check if your solution will work, consider:

- The **input size** n
- The **time complexity** of your algorithm

Example 1:

If the input size `n ≤ 10^3`:

- **An O(n^3)** solution would require around 10^9 operations (since 10^3^3 = 10^9).
  This is **too much** for 1 second and **will likely TLE** (Time Limit Exceeded).

- **An O(n^2)** solution would require 10^6 operations, which is acceptable.
- **An O(nlogn)** solution would also work easily.

So in this case, **O(n^2) or better** is acceptable.

Example 2:

If `n ≤ 10^5`:

- **An O(n^2)** solution would perform around 10^10 operations, which is far too slow.

- **An O(nlogn)** solution would perform roughly $10^5 \cdot \log_2(10^5) \approx 10^5 * 17 \approx 1.7 * 10^6$ operations — this is acceptable.

So for n up to 10^5, you should aim for **O(nlogn) or faster**.

📊 What Time Complexities Are Acceptable for Given Input Sizes:

Input Size n	Acceptable Time Complexity
n ≤ 10	Up to O(n!) or O(2^n)
n ≤ 100	Up to O(n^3)
n ≤ 1000	Up to O(n^2)
n ≤ 10^5	Up to O(nlogn)
n ≤ 10^6	Up to O(n)