Problem Name: Check whether a given number is even or odd DSA???

Step 01:

Approach: By Finding the Basic Remainder....?

Code and its time complexity.

Time Complexity:

- `input()` → Constant time, **O(1)**
- `int()` → Converting a string to an integer takes time **O(1)** (assuming reasonably small input).
- `num % 2` → Modulo operation is a basic arithmetic operation → **O(1)**
- `if...else` + `print()` → Both are constant-time operations → **O(1)**

Final Time Complexity:
✅ **O(1)** — constant time.
This means the program runs in the same amount of time regardless of the input size.

? Space Complexity:
Also O(1) — only one variable (num) is used.
Let me know if you'd like this explained in Bengali or need it for a larger set of numbers (like checking even/odd in a list).

✅ Common Time Complexities:

Complexity	Name	Example	Explanation
`O(1)`	Constant Time	Checking a single number `if num % 2 == 0`	The operation runs once no matter the input size.
`O(n)`	Linear Time	Checking all elements in a list	Time increases directly with the input size.

Complexity	Name	Example	Explanation
`O(n^2)`	Quadratic Time	Nested loops (a loop inside a loop)	If input doubles, time increases four times.
`O(log n)`	Logarithmic Time	Binary search	Time increases slowly even if input size increases a lot.
`O(n log n)`	Linearithmic Time	Merge Sort	Some efficient sorting algorithms run in this time complexity.

Step 02:

Approach: By Finding the Remainder….?

Logic Used:

Approach: By Finding the Remainder

- `n % 2` → This operation finds the remainder when `n` is divided by `2`
- If the remainder is `0`, then `n` is **even**
- If the remainder is `1`, then `n` is **odd**

This method is **mathematically accurate** and **computationally efficient**.

Time Complexity Analysis:

Let's analyze each part of the code:

Line	Operation	Time Complexity	Reason
`input()`	Takes user input	`O(1)`	Constant time to read input
`int()`	Converts string to int	`O(1)`	Simple type conversion

Line	Operation	Time Complexity	Reason
`n % 2`	Modulo operation	`O(1)`	Basic arithmetic operation
`print()`	Displays result	`O(1)`	Constant time

✅ Final Time Complexity: `O(1)`

It runs in **constant time**, no matter what number you input.

📦 Space Complexity:

- Only one variable `n` is used → so space usage is constant.

✅ **Final Space Complexity: `O(1)`**

📌 Summary:

Metric	Value
Time Complexity	`O(1)`
Space Complexity	`O(1)`
Approach Used	By checking remainder using modulo `%`
Efficient?	✅ Yes, it's the most efficient method for this task

Step 03:

Approach: Using Bitwise AND Operator...?

Logic Behind Bitwise AND

- Binary of **even** numbers always ends with `0`
  Example: `4` → `100`, `10` → `1010`
- Binary of **odd** numbers always ends with `1`
  Example: `5` → `101`, `11` → `1011`

So, when you do `n & 1`:

- For even: `n & 1` → `0`
- For odd: `n & 1` → `1`

📌 Example:

- `n = 6` → binary: `110`
  `6 & 1 = 0` → even
- `n = 7` → binary: `111`
  `7 & 1 = 1` → odd

Time Complexity Analysis:

Line	Operation	Time Complexity	Why?
`input()`	Take input from user	**O(1)**	One-time user input
`int()`	Convert input to integer	**O(1)**	Constant time conversion
`n & 1`	Bitwise AND operation	**O(1)**	Direct binary operation
`print()`	Output result	**O(1)**	Constant time printing

✅ Final Time Complexity: `O(1)` (Constant Time)

- The program's speed does **not** depend on the size of the number.
- Bitwise operations are extremely fast and always take constant time.

💼 Space Complexity:

- Only one variable `n` is used.

- So, **Space Complexity:** `O(1)`

📋 Summary:

Metric	Value
Approach	Bitwise AND Operator
Time Complexity	`O(1)`
Space Complexity	`O(1)`
Efficient?	✅ Yes – Bitwise ops are very fast
Why it works	Because LSB (least significant bit) of odd = 1, even = 0

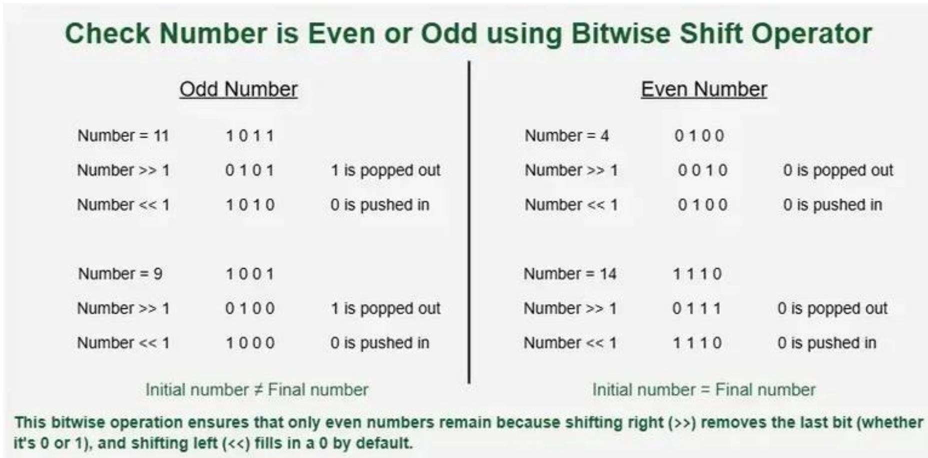
Step 04:
Approach: Using Bitwise Shift Operators

🔍 Logic Used: Using Bitwise Shift Operators

- `n >> 1`: This shifts all bits of `n` **one place to the right** (i.e., divides the number by 2 and drops the remainder).
- `<< 1`: This shifts all bits **one place to the left** (i.e., multiplies the number by 2).

✅ Explanation with Example:

Let's say:



So the logic is:

- For **even** numbers: shifting right then left gives back the original number
- For **odd** numbers: you lose the 1-bit on the right, so the number changes

Time Complexity Analysis

Operation	Time Complexity	Why?
`input()`	`O(1)`	Constant-time user input
`int()`	`O(1)`	Simple type conversion
`n >> 1` and `<< 1`	`O(1)`	Bitwise shift = constant time
`==` comparison	`O(1)`	Just one comparison
`print()`	`O(1)`	Prints output once

✔ Final Time Complexity: `O(1)`

(Independent of input size)

Space Complexity

- You're only using one integer variable (`n`), and a few temporary values during the comparison.

✔ Space Complexity: `O(1)` (Constant)

📋 Summary Table:

Aspect	Details
Approach	Bitwise shift (right → left)
Time Complexity	`O(1)`
Space Complexity	`O(1)`
Efficiency	✅ Very fast – bitwise ops are CPU-level fast
Why It Works	Only even numbers retain value after `>> 1` then `<< 1`

🛠 Bonus Tip:

Your function name has a small typo: `chack_odd_even`
Correct spelling: `check_odd_even`

```
python
CopyEdit
def check_odd_even(n):
    return n == (n >> 1) << 1
```