Set<I>

✅ What is Set<E> in Java?

- Set is an interface in java.util package.
- It extends Collection<E>.
- It does not allow duplicate elements.
- Elements in a Set are unordered, unless you use a specific implementation like LinkedHashSet or TreeSet.

🛠 Methods of Set<E> Interface

These methods are inherited from Collection and optionally overridden in Set.

🔹 Basic Methods

Method	Description
int size()	Returns number of elements.
boolean isEmpty()	Returns true if set is empty.
boolean contains(Object o)	Checks if element exists.
boolean add(E e)	Adds element if not already present.
boolean remove(Object o)	Removes the element if present.
void clear()	Removes all elements.

SH_23

- **Bulk Operations**

Method	Description
boolean containsAll(Collection<?>)	Checks if this set contains all of another collection.
boolean addAll(Collection<? extends E>)	Adds all elements from another collection.
boolean removeAll(Collection<?>)	Removes all elements from another collection.
boolean retainAll(Collection<?>)	Retains only elements also in another collection.

- **Iterator & Array Conversion**

Method	Description
Iterator<E> iterator()	Returns an iterator to loop through the set.
Object[] toArray()	Returns elements in Object array.
<T> T[] toArray(T[] a)	Returns elements in given array type.

- **Equality & Hashing**

Method	Description
boolean equals(Object o)	Compares two sets for equality.

Method	Description
int hashCode()	Returns hash code based on elements.

- **Java 8+ Default Method**

Method	Description
default Spliterator<E> spliterator()	Used for parallel stream processing.

- **Java 9+ Factory Methods**
- **Immutable Set Creation**

Method	Description
Set.of()	Empty immutable set.
Set.of(e1, e2, ...)	Set with 1-10 elements (no duplicates, no nulls).
Set.of(E...)	Varargs version – any number of elements.
Set.copyOf(Collection<? extends E>)	Immutable copy of a collection.

⚠️ All Set.of() and copyOf() methods:

- Do **not allow nulls.**
- Do **not allow duplicates.**
- Throw NullPointerException or IllegalArgumentException.

Note :

✅ **1. Unmodifiable**

- You **cannot add or remove elements.**
- If you try to do this, you get a UnsupportedOperationException.

```
Set<String> set = Set.of("A", "B");
set.add("C"); // ❌ Error
```

🚫 **2. No null values**

- If you try to add null, it will throw NullPointerException.

```
Set<String> set = Set.of("A", null); // ❌ Error
```

⚠️ **3. No duplicates allowed**

- You can't create a set with duplicate values.
- If you do, it throws IllegalArgumentException.

```
Set<String> set = Set.of("A", "A"); // ❌ Error
```

🔁 **4. Iteration order is not fixed**

- The order in which you get elements when looping is **not guaranteed.**

```
Set<String> set = Set.of("A", "B", "C");
// Output order may vary
```

ASHISH_23

📦 5. Serializable

- If all the elements are serializable, the Set can be saved to a file or transferred (serialization).

⚙️ 6. Value-based

- Two sets with same elements are considered **equal**.
- Don't use these sets for synchronization or rely on their exact memory address.

```
Set<String> s1 = Set.of("A", "B");
Set<String> s2 = Set.of("A", "B");

System.out.println(s1 == s2); // Could be false (not same object)
System.out.println(s1.equals(s2)); // true
```

📄 7. Serialization format

- The way they are stored/serialized follows Java’s standard rules (you don’t need to worry unless doing advanced I/O).

ASHISH_23

📝 Summary Table

Feature	Behavior
Modifiable?	❌ No – cannot add or remove elements
Null allowed?	❌ No
Duplicates allowed?	❌ No
Serializable?	✅ Yes, if all elements are
Iteration order fixed?	❌ No – it can change
Use for synchronization?	❌ Not recommended
Value-based?	✅ Equal sets are interchangeable

✅ Set.of(...) vs Set.copyOf(...)

Feature	Set.of(...)	Set.copyOf(...)
📦 Purpose	Creates a new unmodifiable set with given elements	Creates a new unmodifiable set **from a collection**
🔁 Input	Individual elements (varargs)	A collection (like List, Set, etc.)
🚫 Null allowed?	❌ No (throws NullPointerException)	❌ No (throws NullPointerException)

Feature	Set.of(...)	Set.copyOf(...)
❌ Duplicates allowed?	❌ No (IllegalArgumentException if duplicates)	Allowed
🔓 Modifiable?	❌ No	❌ No
🧠 Value-based?	✅ Yes	✅ Yes
📌 Java Version	Java 9+	Java 10+ (part of Java 9+ API as well)

ASHISH

Note :

copyOf() :

“Returns an **unmodifiable Set** containing the elements of the given collection. The collection must:

- not be null,
- must not contain null elements,
- if it has duplicates, only **one** of them is kept,
- and any changes to the original collection **will not** affect the Set.”

"If the given Collection contains duplicate elements, an arbitrary element of the duplicates is preserved."

🔄 **What Does This Actually Mean?**

Let’s decode:

- Set.copyOf() takes a collection (like List, ArrayList, etc.).
- Internally, it creates a Set (which **automatically removes duplicates**).
- You do **not get an IllegalArgumentException**, unlike Set.of(...).
- So if your collection has [1, 2, 1], the result will be [1, 2].

🧠 **Why Is Set.of(...) Different?**

Set.of(...) is varargs-based, so it can check for duplicates at creation.

```
Set<Integer> s = Set.of(1, 2, 1); // ❌ IllegalArgumentException
```

But Set.copyOf() accepts a Collection, not individual values — so it just uses a regular set internally and removes duplicates automatically.

ASHISH 23

HashSet<C>

✅ What is HashSet<E>?

- HashSet<E> is a concrete class in java.util.
- Implements the **Set interface**, which means:
  - **No duplicates allowed**
  - **No guaranteed order**
- Internally backed by a **HashMap**
- HashSet is a collection that stores unique values only.
- It is very fast for checking whether an element is present or not.

✅ Why use HashSet?

Need this 👇	**HashSet is perfect** ✅
No duplicates	✅ Keeps only unique values
Fast search (contains())	✅ O(1) time
Don’t care about order	✅ Ignores order
Want to remove duplicates from a list	✅ Easy!

SH_23

🧠 Internal Implementation

```
transient HashMap<E, Object> map;
```

- Internally, HashSet uses a **HashMap** to store elements.
- Each element of the set becomes a **key** in the map.
- All values are the same dummy object PRESENT.

```
private static final Object PRESENT = new Object();
```

So, adding to HashSet is like:

```
map.put(element, PRESENT);
```

🔧 Constructors

Constructor	Description
HashSet()	Creates an empty HashSet with default capacity (16) and load factor (0.75)
HashSet(Collection<? extends E>)	Creates a HashSet containing elements from another collection(removes duplicates)
HashSet(int initialCapacity)	Sets initial capacity of the internal HashMap
HashSet(int initialCapacity, float loadFactor)	Controls resizing of underlying HashMap
HashSet(int, float, boolean)	Internal use only (package-private)
static <T> HashSet<T> newHashSet(int)	Static factory method (not part of standard Java SE; possibly added in a framework or custom build)

✅ Basic Methods

Method	Description
int size()	Returns number of elements in the set
boolean isEmpty()	true if set has no elements
boolean contains(Object o)	Checks if element o is in the set
boolean add(E e)	Adds element e, returns true if it wasn't already present
boolean remove(Object o)	Removes element o if found
void clear()	Removes all elements from the set

✅ What is Load Factor?

Load Factor is a number that controls **when the HashSet grows (resizes).**

📖 Definition:

The **load factor** is the **maximum ratio** of elements to buckets (capacity) **before** resizing happens.

🧠 In Simple Words:

- Load Factor = how full the HashSet (or HashMap) **can get** before it automatically **grows in size.**
- Default load factor = **0.75** (75%)

✅ Default Constructor

ASHISH

```
Set<String> set = new HashSet<>();
```

This uses:

- **Default capacity** = 16
- **Default load factor** = 0.75

🔁 **Resize threshold:**

16 × 0.75 = 12

So after inserting the **13th element**, the set **automatically resizes** to reduce collisions.

🧪 **Example:**

```
Set<Integer> set = new HashSet<>(16, 0.75f);
```

This means:

- Start with **16 buckets**
- Once you add more than **12 elements**, HashSet **doubles** the number of buckets to 32(16*2).

📊 **Why Resize?**

- HashSet stores elements using **hashing**.
- If too many items go into the same bucket → **collisions** → performance drops.
- Resizing **reduces collisions**, keeps operations like .add(), .contains() fast.

Note :

```
HashSet<Integer> set4 = (HashSet<Integer>)Set.of(1,2,3,4,5);

System.out.println("set4 :"+set4);//java.lang.ClassCastException
```

ASHISH_23

LinkedHashSet<C>

LinkedHashSet is a **Set implementation** that:

- Stores **unique elements only**
- **Maintains insertion order**
- Is **backed internally** by a LinkedHashMap

🔍 Constructor Overview

Constructor	Purpose
LinkedHashSet()	Creates an empty set with default capacity(16) and load factor(0.75)
LinkedHashSet(int)	Sets initial capacity
LinkedHashSet(int, float)	Sets initial capacity and load factor
LinkedHashSet(Collection<? extends E>)	Copies elements from another collection (preserves order)

✅ All these remove duplicates and preserve the original insertion order.

Note : *it allows one null*

Difference Between HashSet and LinkedHashSet :

AS 03

✅ Core Difference

Feature	HashSet	LinkedHashSet
Ordering	❌ No guarantee of order	✅ Maintains **insertion order**
Internal Storage	Backed by HashMap	Backed by LinkedHashMap
Performance	Slightly faster	Slightly slower (due to order tracking)
Duplicates	❌ Not allowed	❌ Not allowed
Null values	✅ One null allowed	✅ One null allowed
Load Factor	Default 0.75	Default 0.75 (same as HashSet)
Initial Capacity	Default 16	Default 16
Thread-safe?	❌ No	❌ No
Ordering Use Cases	❌ Not suitable	✅ Suitable when order matters

📊 Performance Comparison

Operation	HashSet	LinkedHashSet
add()	O(1)	O(1) (but slightly slower)
contains()	O(1)	O(1)
remove()	O(1)	O(1)
Memory usage	Lower	Higher (due to linked list for order)

✅ When to Use What?

Use Case	Recommended Set Type
Fast lookups, don't care about order	HashSet ✅
Need unique items **in same order** added	LinkedHashSet ✅
Configurations, user inputs, dropdowns	LinkedHashSet ✅
Just deduplication, performance critical	HashSet ✅

SortedSet<I>

✅ What is SortedSet<E>?

A **SortedSet** is a special kind of Set that:

- Contains **unique** elements
- Maintains elements in **sorted (natural or custom)** order
- Allows **range-based operations** like subSet(), headSet(), tailSet()
- A **SortedSet** is a subtype of Set that stores **unique elements in sorted order.**

```
Set<E>
 └── SortedSet<E>  (Interface)
      └── NavigableSet<E>  (adds navigation)
           └── TreeSet<E>  (concrete implementation)
```

✅ SortedSet Key Features

Feature	Description
Sorted order	Maintains elements in natural (Comparable) or custom (Comparator) order
No duplicates	Like all sets

Feature	Description
Range views	Get part of the set via subSet(), headSet(), tailSet()
First/last element	Access with first() / last()
New methods (Java 21)	Double-ended operations and reversed views from SequencedSet

✅ Interface: java.util.SortedSet<E>

A **SortedSet** is a subtype of Set that stores **unique elements in sorted order.**

In Java 8, it's primarily used through its most common implementation: **TreeSet**.

✅ Java 8 Methods in SortedSet (Excluding Java 21)

- **Comparator<? super E> comparator()**
  - Returns the **Comparator** used to sort elements.
  - If it returns null, then the set uses **natural ordering** (i.e., elements must implement Comparable).

```
SortedSet<String> set = new TreeSet<>();
System.out.println(set.comparator()); // null => natural order
```

- **SortedSet<E> subSet(E fromElement, E toElement)**
  - Returns a **view** of the portion of this set:

ASHISH 2

- from **fromElement (inclusive)** to **toElement (exclusive)**

Note :

Returns a view of the portion of this set whose elements range from fromElement, inclusive, to toElement, exclusive. (If fromElement and toElement are equal, the returned set is empty.) The ***returned set is backed by this set***, so changes in the returned set are reflected in this set, and vice-versa. The returned set supports all optional set operations that this set supports.

The returned set will throw an **IllegalArgumentException** on an attempt to ***insert an element outside its range***.

Parameters:

fromElement low endpoint (inclusive) of the returned set

toElement high endpoint (exclusive) of the returned set

Returns:

a view of the portion of this set whose elements range from fromElement, inclusive, to toElement, exclusive

```
SortedSet<Integer> set = new TreeSet<>(Arrays.asList(10, 20, 30, 40, 50));
SortedSet<Integer> sub = set.subSet(20, 50);
System.out.println(sub); // [20, 30, 40]
Sub.add(12); //java.lang.IllegalArgumentException: key out of range
Sub.add(50);//java.lang.IllegalArgumentException: key out of range
```

- **SortedSet<E> headSet(E toElement)**
  - Returns a **view** of elements **less than** the specified toElement.

ASHISH_23

- Returns a view of the portion of this set whose elements are strictly less than toElement. ***The returned set is backed by this set, so changes in the returned set are reflected in this set, and vice-versa.*** The returned set supports all optional set operations that this set supports.
- The returned set will throw an ***IllegalArgumentException*** on an attempt to insert an element outside its range

```
SortedSet<Integer> set = new TreeSet<>(Arrays.asList(10, 20, 30, 40));
System.out.println(set.headSet(30)); // [10, 20]
```

- **SortedSet<E> tailSet(E fromElement)**
  - Returns a **view** of elements **greater than or equal to** fromElement.

  Returns a view of the portion of this set whose elements are greater than or equal to fromElement. **The returned set is backed by this set, so changes in the returned set are reflected in this set, and vice-versa.** The returned set supports all optional set operations that this set supports.

  The returned set will throw an ***IllegalArgumentException*** *on an attempt to insert an element outside its range.*

  Parameters:

  fromElement low endpoint (inclusive) of the returned set

  Returns:

  a view of the portion of this set whose elements are greater than or equal to fromElement

```
SortedSet<Integer> set = new TreeSet<>(Arrays.asList(10, 20, 30, 40));
```

ASHISH_23

```
System.out.println(set.tailSet(30)); // [30, 40]
```

- **E first()**
  - Returns the **smallest (first) element** in the set.

  Returns the first (lowest) element currently in this set.

  Returns:

  the first (lowest) element currently in this set

  Throws:

  NoSuchElementException - if this set is empty

```
SortedSet<Integer> set = new TreeSet<>(Arrays.asList(20, 10, 40));
System.out.println(set.first()); // 10
```

- **E last()**
  - Returns the **largest (last) element** in the set.

  Returns the last (highest) element currently in this set.

Returns:

the last (highest) element currently in this set

Throws:

NoSuchElementException - if this set is empty

```
SortedSet<Integer> set = new TreeSet<>(Arrays.asList(20, 10, 40));
System.out.println(set.last()); // 40
```

- **Spliterator<E> spliterator()**
  - Returns a **spliterator** for the set.

ASHISH_23

- Mainly used with Java 8 **Stream API** and **parallel processing.**

```
SortedSet<String> set = new TreeSet<>(Arrays.asList("Java", "Spring", "Boot"));
set.spliterator().forEachRemaining(System.out::println);
```

⚠️ Java 8 SortedSet Summary Table

Method	Description	Java Version
comparator()	Get sorting comparator	✅ Java 8
subSet(a, b)	Elements in [a, b)	✅ Java 8
headSet(x)	Elements less than x	✅ Java 8
tailSet(x)	Elements ≥ x	✅ Java 8
first() / last()	Min / Max elements	✅ Java 8
spliterator()	For stream support	✅ Java 8
addFirst() / reversed() etc.	❌ Not available in Java 8	🚫 Java 21+ only

✅ Is it required to learn NavigableSet<I>?

🔸 **Short Answer:**

Not strictly required, but **definitely recommended** if:

- You work with **sorted data**
- You use or customize **TreeSet**

- You want fine-grained control over ranges, **nearest values**, or **reverse ordering**

✅ Why Learn NavigableSet?

NavigableSet extends SortedSet and adds **powerful methods**:

Feature	Benefit
lower(E e)	Element **less than** e
floor(E e)	Element ≤ e
ceiling(E e)	Element ≥ e
higher(E e)	Element **greater than** e
pollFirst() / pollLast()	Remove and return first/last
descendingSet()	Reverse view
descendingIterator()	Iterate in reverse order

ASH

TreeSet<C>

✅ What is TreeSet?

- A ***Red-Black tree-based*** implementation of the NavigableSet interface.
- Elements are stored in **sorted order**.
- Duplicates are **not allowed**.
- No null allowed (if using natural/comparable ordering).
- Backed internally by a **TreeMap**.

✅ Key Features

Feature	Supported?
Maintains order (sorted)	✅ Yes
Allows null values	❌ No (throws NPE for null in comparisons)
Thread-safe	❌ No
Backed by	TreeMap<E, Object>
Uses Comparable or Comparator	✅ Yes

✅ Constructor Variants

Constructor	Description
TreeSet()	Natural ordering (elements must be Comparable)
TreeSet(Comparator<? super E>)	Custom sorting
TreeSet(Collection<? extends E>)	Creates from another collection
TreeSet(SortedSet<E>)	Clones another sorted set
TreeSet(NavigableMap<E, Object>)	Internal use (advanced)Internal use (used internally in Map-backed construction

✅ Core Methods Breakdown

🔹 Basic Set Operations

add(E e)

remove(Object o)

contains(Object o)

clear()

isEmpty()

size()

addAll(Collection<? extends E>)

These behave like regular Set methods.

🔹 NavigableSet Methods (Range Queries)

ASHISH_

Method	Description	Example
lower(E e)	Greatest element < e	lower(50) → 40
floor(E e)	Greatest element ≤ e	floor(50) → 50
ceiling(E e)	Smallest element ≥ e	ceiling(50) → 50
higher(E e)	Smallest element > e	higher(50) → 60
pollFirst()	Removes & returns first (lowest) element	
pollLast()	Removes & returns last (highest) element	

- **Range Views**

Method	Description
subSet(from, to)	[from, to) view of set
subSet(from, boolean fromInclusive, to, boolean toInclusive)	Precise control on bounds
headSet(to)	Elements < to
headSet(to, boolean inclusive)	Include to if needed
tailSet(from)	Elements ≥ from
tailSet(from, boolean inclusive)	Include from or not

✅ These are **views**, not copies — changes in view affect the original set.

🔹 **Ordering & Traversal**

Method	Description
iterator()	Ascending iterator
descendingIterator()	Reverse iterator (Returns an iterator over the elements in this set in descending order.)
descendingSet()	Returns a reverse-order view
first()	Smallest element
last()	Largest element
comparator()	Returns custom comparator or null (if natural order)

descendingSet()

Returns a reverse order view of the elements contained in this set. ***The descending set is backed by this set, so changes to the set are reflected in the descending set, and vice-versa.*** If either set is modified while an iteration over either set is in progress (except through the iterator's own remove operation), the results of the iteration are undefined.

⚠️ **Iteration Warning**

If you're **iterating** and modifying at the same time **(not using iterator's remove()),** it leads to **ConcurrentModificationException** or **undefined behavior.**

```
for (String s : set.descendingSet()) {
    set.remove(s); // ❌ BAD: modifies backing set during iteration
}
```

✅ Use an Iterator with iterator.remove() if needed.

Note :

comparator() :

```
     TreeSet<Integer> num = new TreeSet<Integer>(List.of(10, 20, 50, 90, 70, 10, 80, 40, 30, 45, 50));
          System.out.println(num);


          System.out.println("Comparator used  : "+num.comparator()); //if null default sorting order

          TreeSet<Integer> num1 = new TreeSet<Integer>(Comparator.reverseOrder());

          num1.addAll(List.of(10, 20, 50, 90, 70, 10, 80, 40, 30, 45, 50));

          System.out.println("Comparator used ? : "+num1.comparator());
```

🔹 Cloning and Splitting

- clone() → Deep copy (but elements are **shallow copied**).
- spliterator() → Used for Java Stream API.

🔹 Java 21 Specific (SequencedSet)

Java 21 Method	Description
addFirst(E) / addLast(E)	Add at start or end
reversed()	Reversed view

🛑 These methods are **Java 21+ only** (available via SequencedSet interface)

❌ Limitations

- ❌ Not thread-safe
  → Use Collections.synchronizedSortedSet() if needed
- ❌ Does **not** allow null values
  → Throws NullPointerException

✅ Real-Time Use Cases

Scenario	TreeSet Benefits
Leaderboard System	Always keeps scores sorted
Unique timestamps	Retrieve events by range (e.g., tailSet())

ASHISH_23

Scenario	TreeSet Benefits
Auto-suggestions	Use ceiling() and subSet() to filter
Calendar / Scheduler	Use higher() for "next" scheduled task
Sorted student roll numbers	No duplicates, always ordered

✅ **Example Code**

```
TreeSet<Integer> scores = new TreeSet<>();
scores.add(80);
scores.add(60);
scores.add(90);

System.out.println(scores); // [60, 80, 90]

System.out.println(scores.lower(80));   // 60
System.out.println(scores.higher(80));  // 90
System.out.println(scores.subSet(60, 90)); // [60, 80]

System.out.println(scores.descendingSet()); // [90, 80, 60]
```

ASHISH_2

Note:

✅ **TreeSet — Does it have Default Capacity & Load Factor?**

🔹 **No, TreeSet does not have:**

- ✅ **Default capacity**
- ✅ **Load factor**

❓ **Why?**

Because TreeSet is backed by a **Red-Black Tree** (via TreeMap) — **not a hash table.**

Key Set Implementations

Implementation	Ordering	Sorted?	Null Allowed	Performance	Backed By
HashSet	❌ No ordering	❌ No	✅ Yes (1 max)	✅ Fast (O(1))	Hash table
LinkedHashSet	✅ Insertion order	❌ No	✅ Yes (1 max)	✅ Medium	Hash table + linked list
TreeSet	✅ Sorted (natural/custom)	✅ Yes	❌ No	❌ Slower (O(log n))	Red-Black tree

✅ HashSet vs TreeSet in Java

Feature	HashSet	TreeSet
Ordering	❌ No order maintained	✅ Maintains **sorted (ascending)** order
Underlying Data Structure	HashMap	TreeMap (Red-Black Tree)
Performance (Big-O)	✅ Faster: O(1) for add/remove/contains	❌ Slower: O(log n) for add/remove/contains
Allows null?	✅ Yes (only one null allowed)	❌ No (throws NullPointerException)
Duplicates Allowed?	❌ No	❌ No
Custom Sorting	❌ Not supported	✅ Yes, via Comparator
Use Case	Fast lookups, set operations	Sorted data, range views, leaderboard
Thread-safe?	❌ No	❌ No
Implements	Set	NavigableSet, SortedSet, Set
Memory Usage	Lower (uses hash table)	Slightly higher (due to tree structure)

Feature	HashSet	TreeSet
Null Element Handling	Allows null	Throws NullPointerException

Iterators in Set – fail-fast behavior

Java Set implementations use **fail-fast iterators**, especially HashSet and LinkedHashSet.

🔸 **What is fail-fast?**

If you modify the set **structurally** (add/remove) **while iterating**, you'll get a ***ConcurrentModificationException.***

```
Set<String> data = new HashSet<>();
data.add("A");
data.add("B");

for (String s : data) {
    data.remove(s); // ❌ throws ConcurrentModificationException
}
```

✅ **How to fix it?**

Use an explicit iterator:

```
Iterator<String> it = data.iterator();
while (it.hasNext()) {
```

```
    String s = it.next();
    it.remove(); // ☑ Allowed
```

Internal Working (Simplified)

- **HashSet**
  - Uses HashMap internally
  - Adds keys to the map with a dummy value (PRESENT)
  - Fast lookup but **no ordering**
- **LinkedHashSet**
  - Same as HashSet but also keeps a **linked list** of elements
  - Preserves insertion order
- **TreeSet**
  - Uses TreeMap internally
  - Maintains **sorted order**
  - Elements must be Comparable or use Comparator

Real-World Use Cases

Use Case	Best Set
Fast lookup & deduplication	HashSet
Maintain order (recent search)	LinkedHashSet
Sorted data (leaderboard, dates)	TreeSet

ASHISH_23

Comparison Chart

Feature	HashSet	LinkedHashSet	TreeSet
Duplicate Allowed	❌ No	❌ No	❌ No
Null Allowed	✅ Yes	✅ Yes	❌ No
Maintains Order	❌ No	✅ Insertion	✅ Sorted
Performance	✅ Fastest	✅ Fast	❌ Slower
Backed By	HashMap	HashMap + LL	TreeMap
Fail-Fast	✅ Yes	✅ Yes	✅ Yes

ASH

========== The End ==========

Author : Ashish Baipalli

Mail : baipalliashish@gmail.com

==========