// nums.erase(nums.begin() + i) remove the element of current index and all further elements shift     left by 1 and size of the vector is decreses by one ...

// so some rules regarding erase

// vector<int> vec = {1, 2, 3, 4, 5, 6, 7};

// Erase elements from index 2 to 4 (i.e., 3rd to 5th elements)

// vec.erase(vec.begin() + 2, vec.begin() + 5); // it is actually vec.erase(index = 2, index = 4)

// vec.erase(inclusive length, exclusive length);

//so a.erase(a.begin()+m, a.begin()+n-1) does not give expected output

a.erase(a.begin()+m , a.begin()+n); // But it gives expected outut

// How to use set

unordered\_set<int> unionSet(a.begin(), a.end()); // create set and Insert elements of `a`

unionSet.insert(b.begin(), b.end()); // Insert elements of `b`

return unionSet.size();

//

sum = 10

a = 5

sum += a # Equivalent to sum = sum + a

print(sum) # Output: 15

//

sum = 10

a = 5

sum = +a # Equivalent to sum = a (because +a has no effect on positive numbers)

print(sum) # Output: 5

//

**Limitations of int in Different Languages:**

| **Language** | **int Size** | **Approximate Max Value** |
| --- | --- | --- |
| **C / C++** | 32-bit | ~2.1 × 10⁹ (2^31 - 1) |
| **C / C++** | 64-bit (long long) | ~9.2 × 10¹⁸ (2^63 - 1) |
| **Java** | int (32-bit) | ~2.1 × 10⁹ |
| **Java** | long (64-bit) | ~9.2 × 10¹⁸ |
| **Python** | No fixed limit | Can store 100+ digit numbers |

//

No, \_\_uint128\_t **cannot** store a 100-digit number in C++.

**Why Not?**

* \_\_uint128\_t is a **128-bit unsigned integer**.
* The maximum value it can store is **2¹²⁸ - 1**, which is approximately **3.4 × 10³⁸**.
* A **100-digit number** is around **10⁹⁹**, which is much larger than \_\_uint128\_t can hold.
* **Alternative Solutions in C++**
* If you need to work with **100-digit numbers**, you must use a **big integer library**, because built-in integer types won’t be sufficient.
* **1. Use GMP Library (mpz\_t)**
* The **GMP (GNU Multiple Precision) library** supports arbitrarily large numbers.

#include <gmp.h> #include <iostream>

**2. Use boost::multiprecision::cpp\_int (Boost Library)**

If you don’t want to use GMP, Boost provides a **header-only** big integer solution.

#include <boost/multiprecision/cpp\_int.hpp> #include <iostream>

//

**Integer Storage in Programming (int, long, long long Differences)**

In programming languages like **C, C++, and Java**, integers are stored in memory using **binary representation** with a fixed number of bits. The size of integer types depends on the **compiler, architecture (32-bit vs 64-bit), and language**.

**1. Integer Data Types and Their Sizes**

| **Data Type** | **Size (bytes)** | **Typical Range (Signed)** | **Range (Unsigned)** |
| --- | --- | --- | --- |
| **int** | 4 | -2,147,483,648 to 2,147,483,647 | 0 to 4,294,967,295 |
| **long** | 4 or 8 | -2,147,483,648 to 2,147,483,647 (4 bytes) or larger if 8 bytes | 0 to 4,294,967,295 (or more) |
| **long long** | 8 | -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 | 0 to 18,446,744,073,709,551,615 |

**Key Differences**

1. **int** is usually 4 bytes, sufficient for most applications.
2. **long** may be the same size as int (4 bytes) on some systems but 8 bytes on others.
3. **long long** is always **at least 8 bytes**, allowing much larger values.

**2. Signed vs. Unsigned Integers**

* **Signed integers**: Can store negative and positive values (default).
* **Unsigned integers**: Only store non-negative values (0 and above), effectively doubling the max positive range.

**3. Memory Representation (Binary)**

Integers are stored using **two’s complement** representation:

* **Positive numbers**: Stored as regular binary.
* **Negative numbers**: Stored by inverting bits and adding 1 (Two’s complement).

Example:

* 5 in 8-bit binary: 00000101
* -5 in 8-bit binary (Two’s complement): 11111011

**4. Integer Size in Different Architectures**

| **Architecture** | **int** | **long** | **long long** |
| --- | --- | --- | --- |
| **32-bit** | 4B | 4B | 8B |
| **64-bit** | 4B | 8B | 8B |

* In **Java**, int is always **4 bytes**, and long is always **8 bytes**, regardless of architecture.

**5. When to Use Each Type?**

* Use **int** when you know the values will fit within ±2 billion.
* Use **long** for larger values if needed.
* Use **long long** when working with very large numbers, like in **financial calculations, cryptography, or large file sizes**.