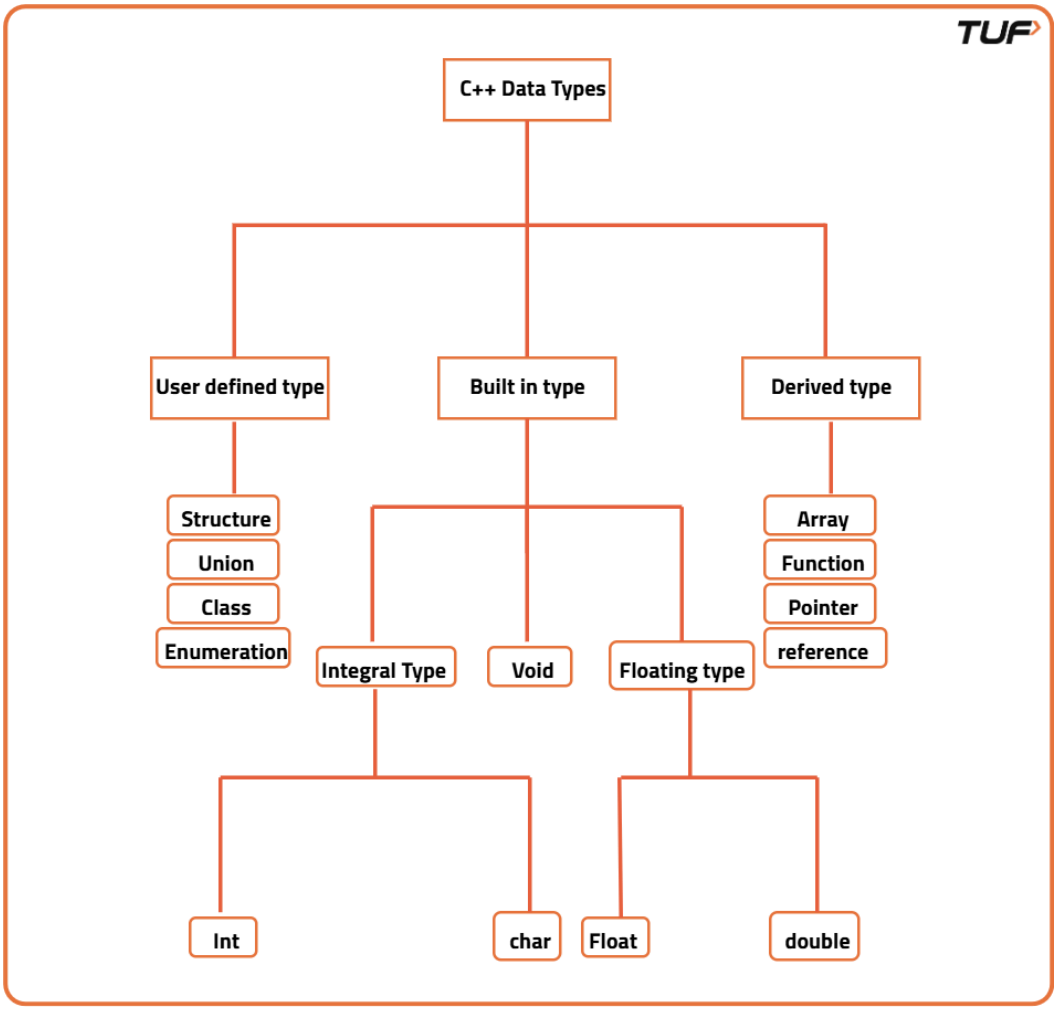
Step 1: Learn the basics

Data Types

Data Types in C++

C++ is a statically typed language, meaning you must specify the type of every variable at compile time. This ensures type safety and better performance, but it also means the compiler will throw errors if you try to assign incompatible types.



- **Primitive Types:**
  - int for integers (whole numbers)
  - float and double for decimal values, with double providing more precision

    - char for single characters
    - bool for Boolean values (true or false)
- **Derived Types:**
    - Arrays, pointers, references, and function types fall under this
- **User- Defined Types:**
    - Structures (struct), classes (class), and enumerations (enum)

C++ also supports modifiers like unsigned, short, and long to tweak the size and range of numeric types. For example, unsigned int only stores non-negative integers but allows larger maximum values than a signed int.

Data Type	Size (in Bytes)	Range / Approximate Values	Description	32-bit vs 64-bit Compiler
bool	1	true (1) or false (0)	Logical data type used for Boolean values.	Same on both (1 byte)
char	1	-128 to 127 (signed) or 0 to 255 (unsigned)	Stores a single character or ASCII code.	Same on both (1 byte)
wchar_t	2 or 4	System dependent	Used for wide (Unicode) characters.	32-bit: 2 bytes | 64-bit: 4 bytes
int	4	-2,147,483,648 to 2,147,483,647 (4 bytes)	Integer data type for whole numbers.	Same on both (4 bytes)
short int	2	-32,768 to 32,767	Short integer with smaller range.	Same on both (2 bytes)
long int	4 or 8	-2,147,483,648 to 2,147,483,647 (4 bytes) or larger (8 bytes)	Larger integer range for bigger values.	32-bit: 4 bytes | 64-bit: 8 bytes
unsigned int	4	0 to 4,294,967,295	Only non-negative integers.	Same on both (4 bytes)

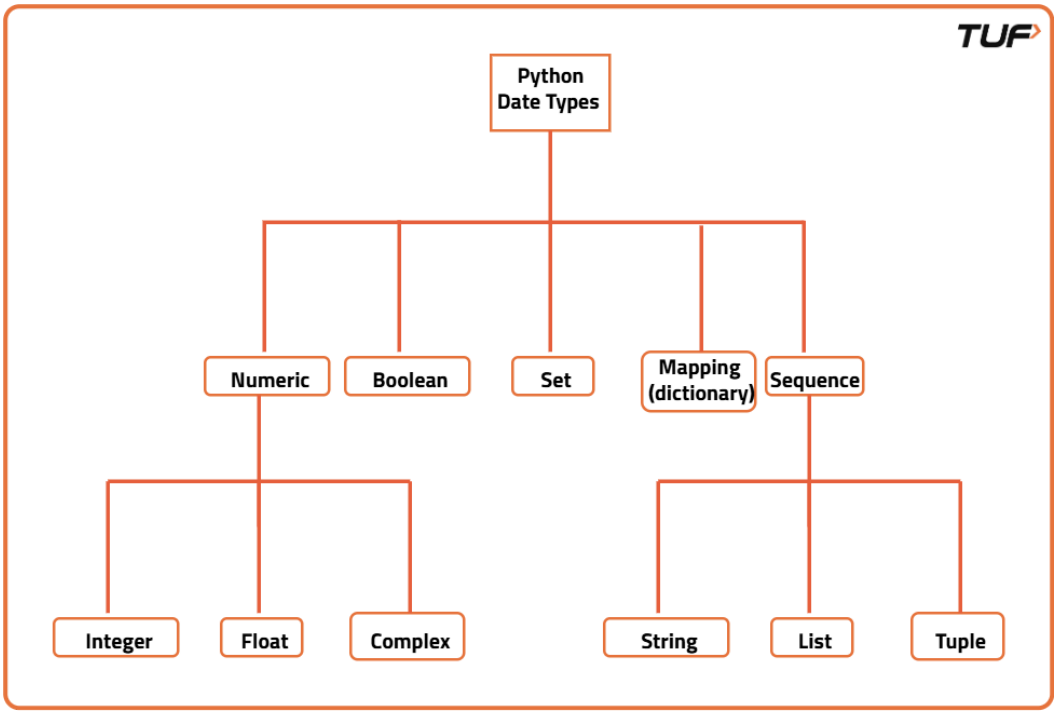
Data Type	Size (in Bytes)	Range / Approximate Values	Description	32-bit vs 64-bit Compiler
float	4	$\pm 3.4 \times 10^{38}$	Floating-point for single precision decimals.	Same on both (4 bytes)
double	8	$\pm 1.7 \times 10^{308}$	Double precision floating-point.	Same on both (8 bytes)
long double	10 or more	$\pm 1.2 \times 10^{4932}$	Extended precision floating-point.	32-bit: 12 bytes | 64-bit: 16 bytes
void	0	—	Represents the absence of type/value.	Not applicable (no size)

Notes

- The **size and range** may vary depending on the **compiler** and **system architecture** (32-bit vs 64-bit).
- The **modifiers** `signed`, `unsigned`, `short`, and `long` modify integer and character ranges.
- Floating-point precision differs between **float** (≈6-7 digits) and **double** (≈15 digits)

Data Types in Python

Python is dynamically typed, meaning you don’t need to declare the data type the interpreter figures it out at runtime. This makes coding faster but can also lead to subtle bugs if you’re not careful



- **Numeric Types:**
  - int for integers (unlimited size)
  - float for decimal numbers (double precision)
  - complex for complex numbers (3 + 5j)
- **Sequence Types:**
  - list, tuple, and range for ordered collections
- **Text Type:**
  - str for strings
- **Mapping Type:**
  - dict for key-value pairs
- **Set Types:**
  - set and frozenset for unordered collection of unique elements
- **Boolean Type:**
  - bool for logical operations

Summary:

Feature	C++ (Typical)	Python (Standard)
Integer Range	Fixed (e.g., 32-bit: -2^{31} to $2^{31}-1$)	Unlimited (dynamic, grows as needed)
int Size	Usually 4 bytes (32-bit/64-bit)	Dynamic (varies with value)
Overflow Risk	Possible (exceeds type limit)	No (auto-converts to big integer)
float Size/Range	4 bytes (float), 8 bytes (double)	8 bytes (double precision, IEEE 754)
float Overflow	Yes (exceeds $\pm 1.7\times10^{308}$ → inf/err)	Yes (exceeds $\pm 1.7\times10^{308}$ → inf/err)
char Type	1 byte (stores a character)	No char type ; str for text/char
bool Type	Present (1 byte, true/false)	Present (True/False, no fixed size)
Static/Dynamic	Static (declare type before use)	Dynamic (type assigned automatically)
DSA Practicality	Must watch for limits	No integer limit—safer for big numbers

Conditional Statements

Conditional statements are indispensable tools for **controlling the flow of your program**. Whether you're making simple decisions or handling complex logic, if-else statements and their variants empower you to write code that **responds dynamically to changing conditions**. Mastering these fundamentals is essential for any aspiring programmer.

```
#include<bits/stdc++.h>
using namespace std;

int main() {
int age=10;


if (age >= 18) {
   cout << "You are an adult." << endl;
} else {
   cout << "You are not an adult." << endl;
}

}
```

Output: You are not an adult.

```
#include <iostream>
using namespace std;

int main() {
   int marks = 54;


   if (marks < 25) {
      cout << "Grade: F" << endl;
   } else if (marks >= 25 && marks <= 44) {
      cout << "Grade: E" << endl;
   } else if (marks >= 45 && marks <= 49) {
      cout << "Grade: D" << endl;
   } else if (marks >= 50 && marks <= 59) {
      cout << "Grade: C" << endl;
   } else if (marks >= 60 && marks <= 69) {
      cout << "Grade: B" << endl;
   } else if (marks >= 70) {
      cout << "Grade: A" << endl;
   } else {
      cout << "Invalid marks entered." << endl;
   }

   return 0;
}
```

Output: Grade: C

```
num = int(input("Enter a number: "))
if num > 0:
   print("Positive number")
elif num < 0:
   print("Negative number")
else:
   print("Zero")
```

Output:

Enter a number: 10

Positive number

Switch Case Statements

- Used to select one out of multiple possible execution paths, based on the value of a variable or expression.
- **How it Works**:
  - The `expression` (usually an integer or character) is evaluated.
  - Execution jumps to the matching `case`.
  - If no match, `default` block executes (if present).
  - `break` statements prevent “fall-through” to the next case. Omitting `break` makes execution continue to subsequent cases.

```
#include <iostream>
using namespace std;

int main() {
    int day = 4;

    switch (day) {
        case 1:
            cout << "Monday";
            break;
        case 2:
            cout << "Tuesday";
            break;
        case 3:
            cout << "Wednesday";
            break;
        case 4:
            cout << "Thursday";
            break;
        case 5:
            cout << "Friday";
            break;
        case 6:
```

```
            cout << "Saturday";
            break;
        case 7:
            cout << "Sunday";
            break;
        default:
            cout << "Invalid";
    }

    return 0;
}
```

Output: Thursday

Common Uses:

- Menu-driven programs
- State machines
- Command selection

Python **does not have a traditional** `switch-case` **statement** like C++ or Java, but you can achieve similar logic using `match-case` (Python 3.10 and later) or with **dictionaries**.

1. Using `match-case` (Python 3.10+):

```
value = int(input("Enter a number (1-3): "))

match value:
    case 1:
        print("One")
    case 2:
        print("Two")
    case 3:
        print("Three")
    case _:
        print("Other number")
```

Output:

Enter a number (1-3): 10
Other number

2. Using Dictionary Mapping (works in all Python versions):

```
def switch_case(value):
    cases = {
        1: "One",
        2: "Two",
        3: "Three"
    }
    print(cases.get(value, "Other number"))

num = int(input("Enter a number (1-3): "))
switch_case(num)
```

Output:

Enter a number (1-3): 10
Other number

Both approaches let you implement switch-like logic in Python.

- Prefer `match-case` if you use Python 3.10 or newer.
- Use dictionary mapping for older versions.

Arrays and Strings

Arrays

What is an Array?

- **Definition:**
    - An array is a collection of elements (all must be the same type) stored in contiguous memory locations.

    - You can access any element using its **index** (starting from 0).
- **Key Properties:**
    - Random access using index is fast (O(1)).
    - Fixed size when created in C++, but flexible in Python.
    - All elements are of **homogeneous** type.

Arrays in C++

```
int arr[5]; // declares an array of 5 integers
arr[0] = 10; // assigns value to the first element
```

- **Fixed size:** You must declare the size at creation.
- **Accessing:** Use square brackets with index (starts at 0).
- **Finding length:** Use `sizeof(arr)/sizeof(arr[0])` for plain arrays.

Arrays in Python

```
arr = [10, 20, 30, 40, 50] # creates a list (dynamic array)
print(arr[0]) # outputs: 10
```

- **Lists:** Python's `list` type behaves like an array but can grow/shrink.
- **No fixed size:** You can append or remove items.
- **Accessing:** Same as C++, with square brackets.
- **Finding length:** Use `len(arr)`.

Why Zero Indexing?

- Starts from 0 for simpler memory offset calculation. The first item is at the starting address.

Strings

What is a String?

- A string is a sequence of characters arranged in order, indexed from 0.

- Examples: "hello", "abc123"

Strings in C++

```
#include <string>
std::string s = "striver";
char ch = s[0]; // 's'
s.length(); // 7
```

- **Standard Library:** Use `std::string` for easier string handling.
- **Access:** Same as arrays, with `[index]`.
- **Length:** `s.length()` or `s.size()`.
- **Comparison:** `==` and `!=` work for strings.

Strings in Python

```
s = "striver"
ch = s[0] # 's'
len(s)    # 7
```

- **Access:** Square brackets, 0-based index.
- **Length:** `len(s)`.
- **Comparison:** `==` and `!=` for string equality.

💡 Deep Copy Note

- **C++:** Assigning or passing a string variable creates a new copy of the string (unless passed by reference).
- **Python:** Strings are immutable, so assignments still reference the same object, but changes always result in new strings.

Quick Summary Table

Feature	C++ Array	Python List	C++ String	Python String
Memory Layout	Contiguous	Contiguous	Contiguous	Contiguous
Homogeneous?	Yes	Yes	Yes	Yes
Zero Indexing	Yes	Yes	Yes	Yes
Length	Fixed	Dynamic	Dynamic	Dynamic
Element Access	arr[i]	arr[i]	s[i]	s[i]
Find Length	`sizeof/size()`	`len()`	`length()/size()`	`len()`

💡 Mini Review

- **Arrays:** Indexed collections, fixed (C++) or dynamic (Python), fast access.
- **Strings:** Ordered sequence of characters, indexed, immutable in Python, easily handled in C++ using `std::string`.

Understanding For Loop

What Is a For Loop?

A **for loop** is a control structure used to repeat a block of code multiple times, making it essential for Data Structures & Algorithms (DSA) tasks such as array traversal, searching, pattern printing, and working with multi-dimensional data.

C++ For Loop Syntax

C++ for loops have three main components in their header:

```
for (initialization; condition; increment/decrement) {
    // loop body
}
```

- **Initialization:** Run once before the loop starts, typically used to initialize the loop counter.
- **Condition:** Checked before every iteration; if true, the loop continues.
- **Increment/Decrement:** Changes (usually increments) the counter variable after each iteration.

Example: Print numbers from 1 to 5

```
cppfor (int i = 1; i <= 5; i++) {
    cout << i << " ";
}
// Output: 1 2 3 4 5
```

Python For Loop Syntax

Python for loops use iterators (commonly `range(n)`):

```
for variable in iterable:
    # loop body
```

Example: Print numbers from 1 to 5

```
for i in range(1, 6):
    print(i, end=" ")
# Output: 1 2 3 4 5
```

Note: `range(start, stop)` includes `start` but not `stop`.

Core DSA Loop Patterns

- **Array Traversal:**
  - C++: `for (int i = 0; i < n; i++) arr[i] ...`
  - Python: `for i in range(n): arr[i] ...` or `for val in arr: ...`
- **Matrix/Nested Loop:**
  - C++:

    ```
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            // arr[i][j] ...
        }
    }
    ```

  - Python:

```
for i in range(rows):
    for j in range(cols):
        # arr[i][j] ...
```

Customizing For Loops

- **Increment/Step Size:**
  - C++: `for (int i = 1; i <= 25; i += 5)`
  - Python: `for i in range(1, 26, 5)`
- **Reverse Order:**
  - C++: `for (int i = n-1; i >= 0; i--)`
  - Python: `for i in range(n-1, -1, -1)`

Conditional Logic Within Loops

Both C++ and Python allow `if`, `else` blocks inside loops for logic such as checking even/odd numbers, searching values, etc.

Example (Find Even Numbers):

- C++:

```
for (int i = 1; i <= 10; i++) {
    if (i % 2 == 0) { cout << i << " "; }
}
```

- Python:

```
for i in range(1, 11):
    if i % 2 == 0:
        print(i, end=" ")
```

Key DSA Applications of For Loops

- Array and matrix traversal
- Searching and pattern matching

- Sorting implementations
- Dynamic programming table updates

While Loops in Programming

Core Concept

A **while loop** repeatedly executes a block of code **as long as a condition is true**. The loop:

1. **Checks the condition.**
2. If **true**, runs the body of the loop.
3. After each iteration, **re-checks** the condition.
4. **Stops** when the condition becomes false.

If the initial condition is false, the loop body **does not execute even once**.

Syntax Comparison

C++

```
while (condition) {
    // loop body
}
```

Python

```
while condition:
    # loop body
```

DSA Example: Factorial Calculation

- **DSA Use:** Loops like `while` are foundational in problems involving iteration, searching, validity checks, etc.
- **Factorial Algorithm:** Multiply all positive integers from 1 to nnn.

C++ Implementation

```
int n = 5;
int factorial = 1;
while (n > 0) {
    factorial *= n;
    n--;
}
cout << "Factorial of 5 is: " << factorial << endl;
```

```
Factorial of 5 is: 120
```

Python Implementation

```
n = 5
factorial = 1
while n > 0:
    factorial *= n
    n -= 1
print("Factorial of 5 is:", factorial)
```

Output:

```
Factorial of 5 is: 120
```

Key DSA Points

- **Termination Condition:**
  - Always define a clear exit condition to avoid infinite loops.
  - In DSA, infinite loops are a major source of bugs, especially in searching, sorting, or data processing algorithms.
- **Loop Control (Optimization):**
  - **break:** Terminates the loop early. Useful for search or early exit in algorithms.

- **continue:** Skips the current iteration. Helps when you want to ignore certain cases (like skipping even numbers).

C++ Example: Search with break and continue

```
int numbers[] = {1,2,3,4,5,6,7,8,9,10};
int target = 6;
// Using break
for (int num : numbers) {
    if (num == target) {
        cout << "Target found: " << target << endl;
        break; // Exit loop
    }
    cout << "Checking: " << num << endl;
}
// Using continue
for (int num : numbers) {
    if (num % 2 == 0) {
        continue; // Skip even numbers
    }
    cout << "Odd number: " << num << endl;
}
```

Python Example: Same logic

```
numbers = [1,2,3,4,5,6,7,8,9,10]
target = 6
# Using break
for num in numbers:
    if num == target:
        print("Target found:", target)
        break  # Exit loop
    print("Checking:", num)
# Using continue
for num in numbers:
    if num % 2 == 0:
```

```
        continue  # Skip even numbers
    print("Odd number:", num)
```

Output:

```
Checking: 1
Checking: 2
Checking: 3
Checking: 4
Checking: 5
Target found: 6
Odd number: 1
Odd number: 3
Odd number: 5
Odd number: 7
Odd number: 9
```

💡 Checklist: While Loops in Algorithms

- Always define **clear exit conditions**.
- Use **break** for early exit, and **continue** for skipping cases.
- Test for **edge cases**: What if the condition is false from the start?
- In competitive programming and interviews, **while loops** are used for:
  - Input validation
  - Repeated calculation (like factorial, sum until zero)
  - Traversing data structures (linked lists, arrays)

Functions (Pass by Reference and Value)

Core Concepts

- **Pass by Value:** Function receives a copy of the variable. Changes inside the function do **not** affect the original variable.

- **Pass by Reference:** Function receives a reference (address) to the variable. Changes **do** affect the original.

C++ in DSA

- **Default**: Primitives (like `int`, `char`, `double`) are passed **by value**.
- **Explicit Reference**: Use the `&` symbol in the function parameter to pass **by reference** and allow the function to modify the arguments.

Example: Pass by Value (C++)

```
#include <iostream>
using namespace std;
void modify(int a) {
    a = a + 10;
}
int main() {
    int x = 5;
    modify(x);
    cout << x << endl; // Output: 5
    return 0;
}
```

- **Result**: The value of `x` in `main` is **unchanged**.

Example: Pass by Reference (C++)

```
#include <iostream>
using namespace std;
void modify(int &a) {
    a = a + 10;
}
int main() {
    int x = 5;
    modify(x);
    cout << x << endl; // Output: 15
```

```
    return 0;
}
```

- **Result**: The value of `x` **changes** because `&a` refers to the original `x`.

Application in DSA:

- Use pass by reference for efficiency (avoid copying big structures, like vectors or trees).
- Essential when a function must modify the input (e.g., swapping values, updating an array, marking nodes as visited).

Python in DSA

- **Everything** in Python is passed by a mechanism often called *pass-by-object-reference* (or *pass-by-assignment*).
- **Immutable Types** (like `int`, `str`, `tuple`): Behave **like pass by value** (cannot modify the object in place).
- **Mutable Types** (like `list`, `dict`, `set`): Behave **like pass by reference** (can modify the object in place).

Example: Immutable type (int)

```
def modify(a):
    a = a + 10
x = 5
modify(x)
print(x)  # Output: 5
```

- **Result**: The value of `x` is **unchanged**.

Example: Mutable type (list)

```
def modify(lst):
    lst.append(10)
nums = [5]
modify(nums)
print(nums)  # Output: [5, 10]
```

- **Result**: The contents of `nums` **change** because lists are mutable and the reference to the same object is passed.

Application in DSA:

- For efficiency, large lists (arrays), sets, dictionaries are often modified in place to save memory.
- Side effects: Be careful when passing mutable objects—unintended modifications can cause subtle bugs.
- If you do **not** want a function to modify the passed list/dict, pass a copy (e.g., `my_list.copy()`).

Quick Recap Table (C++ vs Python DSA)

Operation	C++ Default	C++ with `&`	Python (int)	Python (list/dict)
Function gets	Copy	Reference (original)	Ref to immutable	Ref to mutable
Original changes?	No	Yes	No	Yes

Mnemonic:

- "If you want to change it, pass the REAL thing." (use `&` in C++, pass a mutable in Python)
- "If you want to keep it safe, pass a COPY." (no `&` in C++, or pass an immutable or a copy in Python)

Time and Space Complexity

1. What Is Time Complexity?

Time complexity measures how the execution time of an algorithm changes as the input size () grows. It's not the real elapsed time on a machine, but an *abstract* measure based on the number of steps or operations in the code, irrespective of hardware differences. The same program will run faster on a better machine, but its **time complexity remains unchanged**.

2. Big O Notation

- Time complexity is represented using Big O notation, which describes the *upper bound* of the growth rate.
- **Example:**
  - If a loop runs 5 times, and each iteration has 3 fixed steps: assigning, comparing, and printing, the total steps are: 5×3=15, so O(15).
  - If the loop runs N times, O(3N), but we ignore constants focusing on the dominant term as N gets very large.

3. Rules For Calculating Time Complexity

- **Worst Case**: Always analyze for the scenario requiring the most steps, ensuring robust performance.
- **Ignore Constants**: As N gets large, constants become insignificant. O(3N) is simply O(N).
- **Ignore Lower Order Terms**: Only the term with the fastest growth rate matters. Example:

  $O(4N^3 + 3N^2 + 8)$ simplifies to $O(N^3)$ when N is very large.

Best/Average/Worst Case

- **Best Case**: Least steps required.
- **Average Case**: Expected number of steps.
- **Worst Case**: Most steps required.

In interviews, always prioritize worst case unless specified!

4. Examples

Nested Loops

Code 1:

```
for (int i=0; i<N; i++)
    for (int j=0; j<N; j++)
        // Constant time operation
```

- Outer loop: N times
- Inner loop: N times for each outer
- **Total Steps:** $N \times N = N^2$
- **Time Complexity:** $O(N^2)$

Code 2:

```
for (int i=0; i<N; i++)
    for (int j=0; j<=i; j++)
        // Constant time operation
```

- Inner runs i+1 times per outer.
- **Total Steps:** 1 + 2 + 3 + …N = $\frac{N(N+1)}{2}$
- **Time Complexity:** $O(N^2)$

5. Other Complexity Notations

- **Big O (O)**: Upper bound (worst case)
- **Theta (Θ)**: Tight bound (exact growth rate)
- **Omega (Ω)**: Lower bound (best case)

  For interviews, Big O is most relevant.

6. Space Complexity

Space complexity refers to how much *extra* memory an algorithm needs as the input size grows. Calculated as the sum of:

- **Input Space**: Memory for input data.
- **Auxiliary Space**: Extra variables and data structures created while solving the problem.

Examples:

- Variables a, b, c: O(3) (constants usually ignored).
- Array of size N: O(N).

> Interview Tip: Never modify given inputs to save space unless explicitly allowed! Data may be used elsewhere in the system.

7. Competitive Programming Guidance

- Online judges typically allow ~10^8 operations in 1 second.
- If limit is 2s, target 2×10^8; for 5s, 5×10^8.
- So your code must have time complexity not worse than $O(10^8)$ per second for typical problems.

Quick Review Table

Concept	Definition & Key Points
Time Complexity	Predicts algorithm running time by input size N; uses Big O; ignore constants/low terms.
Space Complexity	Predicts extra memory an algorithm needs; sum of input + auxiliary; use Big O.
Worst/Best/Average Case	Analyze for max/least/expected steps taken; focus on **worst case** in interviews.
Big O, Θ, Ω	Notations for upper/tight/lower bounds; Big O is standard for DSA interviews.
CP Time Limits	10^8 operations per second; always check limits for efficient coding.

💡 Mnemonic For Big O Calculation

WIC: Always focus on **Worst case**, **Ignore constants**, **Cut lower terms**.

Patterns

1. **Square Pattern**

```
*****
*****
*****
*****
*****
```

```
int n;
cin >> n;
for(int i = 0; i < n; i++) {
    for(int j = 0; j < n; j++) {
        cout << "* ";
    }
    cout << endl;
}
```

T.C : O(n^2)

Input: 6

Output:

```
* * * * * *
* * * * * *
* * * * * *
* * * * * *
* * * * * *
* * * * * *
```

```
n = int(input())
for i in range(n):
    print("* " * n )
```

Input: 4

Output:

```
* * * *
* * * *
* * * *
* * * *
```

- **Outer loop**: Runs `n` times (`i` from `0` to `n-1`)
- **Each iteration**:
  - String repetition `"* " * n` creates a string of length `2n` → **O(n)** time
  - Printing the string also takes **O(n)** time
- **Total operations**: `n × O(n) = O(n²)`
- **Time complexity**: **O(n²)**
- **Benchmark typically shows**: String multiplication is 2-5x faster than nested loop

2. **Left Triangle**

```
*
**
***
****
*****
```

```
int n;
cin >> n;
for(int i = 0; i<n; i++){
   for(int j = 0; j <= i; j++){
      cout << "* " ;
   }
   cout << endl;
}
```

Input: 5

Output:

```
*
* *
* * *
* * * *
* * * * *
```

Time complexity:

- The **outer loop** runs from i=0 to n-1, so **n** iterations.
- The **inner loop** runs from j=0 to j = i
  inner loop executes
  $1 + 2 + 3 + \ldots + n = \frac{n(n+1)}{2}$
  T.C : O (n^2)

```
n = int(input())
for i in range(n):
print("* " * (i + 1))
```

Input: 7

Output:

```
*
* *
* * *
* * * *
* * * * *
* * * * * *
* * * * * * *
```

- **Outer loop:** Runs n times
- **Inner work per loop:** Prints i+1 stars in each iteration
- **Total work:** $1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2}$
- **Time complexity:** $O(n^2)$

3. **Left Triangle Numbers**

```
1
12
123
1234
12345
```

```
int n;
cin >> n;

for(int i = 1; i <= n; i++){
   for(int j = 1; j <=i; j++){
      cout << j << " ";
   }
   cout << endl;
}
```

Input: 5

Output:

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```

T.C : $O(n^2)$

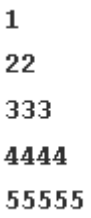
```
n = int(input())
for i in range(1,n+1):
    for j in range(1, i+1):
        print(j, end = " ")
    print()
```

Input: 3

Output:

```
1
1 2
1 2 3
```

4. **Left Triangle repeating pattern**




```
int n;
cin >> n;

for(int i = 1; i <= n; i++){
    for(int j = 1; j <=i; j++){
        cout << i << " ";
    }
    cout << endl;
}
```

Input: 5

Output:

```
1
2 2
3 3 3
4 4 4 4
5 5 5 5 5
```

```
n = int(input())
for i in range(1,n+1):
    for j in range(1, i+1):
        print(i, end = " ")
    print()
```

Input: 3

Output:

```
1
2 2
3 3 3
```

5. **Left Inverse Triangel**

```
*****
****
***
**
*
```

```
    int n;
    cin >> n;

    for(int i = n; i >=1; i--){
        for (int j = 1; j <= i; j++){
            cout << "* " ;
        }
```

```
        cout << endl;

    }
```

Input: 5

Output:

```
* * * * *
* * * *
* * *
* *
*
```

```
n = int(input())
for i in range(n, 0, -1):
  print("* "*i)
```

Input: 7

Output:

```
* * * * * * *
* * * * * *
* * * * *
* * * *
* * *
* *
*
```

6. **Left Inverse numbered triangle**

```
12345
1234
123
12
1
```

```
int n;
cin >> n;

for(int i = n; i >=1; i--){
  for (int j = 1; j <= i; j++){
      cout << j << " ";
  }
  cout << endl;

}
```

Input: 5

Output:

```
1 2 3 4 5
1 2 3 4
1 2 3
1 2
1
```

```
n = int(input())
for i in range(n, 0, -1):
  for j in range (1, i + 1):
      print(j, end=' ')
  print()
```

Input: 7

Output:

```
1 2 3 4 5 6 7
1 2 3 4 5 6
1 2 3 4 5
1 2 3 4
1 2 3
1 2
1
```

7. **Pyramid**

```
int n;
cin >> n;

for(int i = 1; i <= n; i++){
for(int j = n; j > i; j--){
   cout << " ";
}
for(int k = 1; k <= (2 * i - 1); k++){
   cout << "*";
}
cout << endl;

}
```

Input: 5

Output:

```
  *
 ***
```

```
  *****
 *******
```

```
n = int(input())
for i in range(1, n+1):
    print(' ' * (n-i), end = ' ')
    print('*' * (2 * i - 1))
```

Input: 7

Output:

```
     *
    ***
   *****
  *******
 *********
***********
*************
```

8. **Inverted Pyramid**

```
*********
 *******
  *****
   ***
    *
```

```
int n;
cin >> n;

for(int i = n; i >= 1; i--) {
for(int j = 1; j <= n - i; j++) {
```

```
    cout << " ";
}
for(int j = 1; j <= 2*i - 1; j++) {
    cout << "*";
}
cout << endl;
}
```

Input: 5

Output:

```
*********
 *******
  *****
   ***
    *
```

```
n = int(input())
for i in range(n, 0, -1):
    print(' ' * (n - i), end = ' ')
    print('*' * (2 * i - 1))
```

Input: 7

Output:

```
*************
 ***********
  *********
   *******
    *****
     ***
      *
```

9. Combined Pyramid

```
int n;
cin >> n;

for(int i = 1; i <= n; i++){
    for(int j = n; j > i; j--){
        cout << " ";
    }
    for(int k = 1; k <= (2 * i - 1); k++){
        cout << "*";
    }
    cout << endl;

}


for(int i = n; i >= 1; i--) {
    for(int j = 1; j <= n - i; j++) {
        cout << " ";
    }
    for(int j = 1; j <= 2*i - 1; j++) {
        cout << "*";
    }
    cout << endl;
}
```

Input: 5

Output:

```
   *
  ***
 *****
*******
*********
*********
 *******
  *****
```

```
  ***
   *
```

```
n = int(input())
for i in range(1, n+1):
    print(' ' * (n-i), end = ' ')
    print('*' * (2 * i - 1))

for i in range(n, 0, -1):
    print(' ' * (n - i), end = ' ')
    print('*' * (2 * i - 1))
```

Input: 7

Output:

```
      *
     ***
    *****
   *******
  *********
 ***********
*************
*************
 ***********
  *********
   *******
    *****
     ***
      *
```

10. Left Pyramid

```
*
**
***
****
*****
****
***
**
*
```

```
int n;
cin >> n;
for(int i = 1; i <= n; i++){
    for(int j = 1; j <= i; j++){
        cout << "*" ;
    }
    cout << endl;

}

for(int i = (n-1); i >=1; i--){
    for(int j = 1; j <= i; j++){
        cout << "*" ;
    }
    cout << endl;
}
```

Input: 5

Output:

```
*
**
***
****
*****
****
***
```

```
**
*
```

```
n = int(input())
for i in range(1, n+1):
    print("*" * i)
for i in range(n-1, 0, -1):
    print("*" * i)
```

Input: 7

Output:

```
*
**
***
****
*****
******
*******
******
*****
****
***
**
*
```

11. evenOdd left triangle

```
1
0 1
1 0 1
0 1 0 1
1 0 1 0 1
```

```
int n;
cin >> n;
for(int i = 1; i <=n; i++){
   for(int j = 1; j<= i ; j++){
         int val = ((i + j) % 2 == 0) ? 1 : 0;
         cout << val << (j == i ? "" : " ");
   }
   cout << "\n";
}
```

Input: 5

Output:

```
1
0 1
1 0 1
0 1 0 1
1 0 1 0 1
```

```
n = int(input())
for row in range(1, n+1):
   for col in range(1, row+1):
      val = 1 if (row + col) % 2 == 0 else 0
      if(row == col):
         print(val)
      else:
         print(val, end=" ")
```

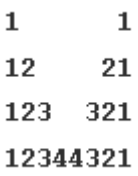
Input: 7

Output:

```
1
0 1
1 0 1
0 1 0 1
1 0 1 0 1
```

```
0 1 0 1 0 1
1 0 1 0 1 0 1
```

12. number cone




```
int n;
cin >> n;
for(int row = 1; row <= n; row++) {
   for(int col1 = 1; col1 <= row; col1++){
      cout << col1 ;
   }

   for(int space = 2 * (n - row); space >= 1; space--){
      cout << " " ;
   }
   for(int col2  = row; col2 >= 1; col2--){
      cout << col2 ;
   }
   cout << endl;
```

Input: 5

Output:

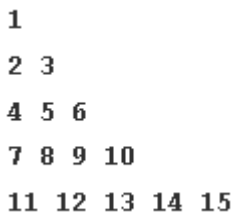
```
1       1
12      21
123    321
1234  4321
1234554321
```

```
n = int(input())
for row in range(1, n+1):
    for col1 in range(1, row + 1):
        print(col1, end = "")
    print(" " * (2 * ( n - row)), end = "")

    for col2 in range(row, 0, -1):
        print(col2, end="")
    print()
```

```
1          1
12         21
123        321
1234      4321
12345    54321
123456  654321
12345677654321
```

13. number triangle




```
int n;
cin >> n;
int count = 0;
for(int row = 1; row <= n; row++){
    for(int col = 1; col <= row ; col++){
        count++;
        cout << count << " ";
    }
    cout <<  "\n";
```

```
}
```

Input: 5

Output:

```
1
2 3
4 5 6
7 8 9 10
11 12 13 14 15
```

```
n = int(input())
count = 0
for row in range (1, n+1):
    for col in range(1, row+1):
        count += 1
        print(count, end = " ")
    print()
```

Input: 7

Output:

```
1
2 3
4 5 6
7 8 9 10
11 12 13 14 15
16 17 18 19 20 21
22 23 24 25 26 27 28
```

14. Alphabet left Triangle

```
A
AB
ABC
ABCD
ABCDE
```

```
int n;
cin >> n;
for(int row = 1; row <= n; row++){
   for(int col = 0; col < row ; col++){
      char chr = (65 + col);
      cout << chr << " ";
   }
   cout <<  "\n";

}
```

Input: 5

Output:

```
A
A B
A B C
A B C D
A B C D E
```

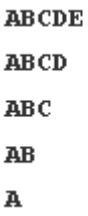
```
n = int(input())
for row in range (1, n+1):
   for col in range(row):
      print(chr(65 + col), end = " ")
   print()
```

Input: 7

Output:

```
A
A B
A B C
A B C D
A B C D E
A B C D E F
A B C D E F G
```

15. Alphabets Inverse Triangle




```
int n;
cin >> n;
for(int row = 1; row <= n; row++){
   for(int col = 0; col <= (n-row) ; col++){
      char chr = (65 + col);
      cout << chr << " ";
   }
   cout <<  "\n";

}
```

Input: 5

Output:

```
A B C D E
A B C D
A B C
```

```
A B
A
```

```
n = int(input())
for row in range (1, n+1):
    for col in range(n-row+1):
        print(chr(65 + col), end = " ")
    print()
```

Input: 7

Output:

```
A B C D E F G
A B C D E F
A B C D E
A B C D
A B C
A B
A
```