DSA

- https://github.com/kishanrajput23/Love-Babbar-CPP-DSA-Course/tree/main
- https://www.youtube.com/playlist?list=PLDzeHZWIZsTryvtXdMr6rPh4IDexB5NIA

Table of Contents

1 Flowcharts, Pseudocode & Programming Languages

Table of Contents

Course Intent & Problem Solving

- Problem Solving Approach

```
1. UNDERSTAND THE PROBLEM
   ↓
2. IDENTIFY GIVEN VALUES
   ↓
3. THINK ABOUT APPROACH
   ↓
4. WRITE THE CODE
```

- Development Flow

```
Problem → Rough Solution → High Level Code → Machine Code → Executable
         (Flowchart/      (C++/Java/      (Binary)    (Run Program)
          Pseudocode)      Python)
```

Flowcharts

- Flowcharts Class Notes

Pseudocode

- What is Pseudocode?
- **Pseudo** = Rough, **Code** = Instructions
- Generic way of representing logic
- Language-independent approach
- Easy to understand English-like statements

- Example Pseudocodes

- - Sum of Two Numbers

```
1. Read a, b
2. sum = a + b
3. Print sum
```

- - Even/Odd Check

```
1. Read N
2. If N%2 == 0
```

```
   Then Print "EVEN"
   Else Print "ODD"
```

- - Sum 1 to N

```
1. Read N
2. sum = 0
3. num = 1
4. While num <= N
   - sum = sum + num
   - num = num + 1
5. Print sum
```

- - Prime Number Check

```
1. Read N
2. num = 2
3. While num < N
   - If N%num == 0
     Then Print "NOT PRIME" and Exit
   - num = num + 1
4. Print "PRIME"
```

Programming Languages

- What are Programming Languages? Languages used to give instructions to computers, just like we use human languages to communicate.

- Why Do We Need Them?

```
Human Instructions → Programming Language → Compiler → Machine Code
"Give me food"       "cout << message"     Binary    Computer Execute
```

- Language Translation Process

```
┌─────────────┐   ┌─────────────┐   ┌─────────────┐
│ Source Code │ → │   Compiler  │ → │ Executable  │
│   (C++)     │   │  (gcc)      │   │   (.exe)    │
│ Human Read │    │ Translator │    │ Machine Run │
└────────────┘    └────────────┘    └─────────────┘
```

- Key Points
- **Source Code**: Human-readable instructions
- **Compiler**: Translator (Hindi to French analogy)
- **Binary Code**: Machine language (0s and 1s)
- **Executable File**: Ready-to-run program

- Language Rules Every programming language has:
- **Syntax**: Grammar rules
- **Semantics**: Meaning of constructs
- **Keywords**: Reserved words
- **Operators**: Mathematical and logical symbols

Practice Problems

- Homework Questions

- - 1. Triangle Validity Check **Problem**: Check if triangle with sides A, B, C is valid **Logic**: A+B > C AND B+C > A AND C+A > B

```
Input: A, B, C
Conditions: All three inequalities must be true
Output: "VALID" or "INVALID"
```

- - 2. Print Odd Numbers 1 to N (Inclusive) **Problem**: Print all odd numbers from 1 to N where N is included

```
Input: N = 5
Output: 1, 3, 5
Logic: Start with 1, increment by 2, condition num <= N
```

- - 3. Factorial of N **Problem**: Calculate N! = N × (N-1) × (N-2) × ... × 1

```
Input: N = 5
Output: 120
Logic: fact = 1, multiply by each number from 1 to N
```

- Additional Practice Write pseudocode for all the flowchart examples covered in the lecture.

Mod Operator

- gives remainder
- a%b = remainder of a/b
- `5%2=1`
- `8%2=0`
- `4%9=4`, when a>b, a%b=a
- if n%2 =0 , n is even, else n is odd

Key Takeaways

- Problem Solving Steps

1. **Understand** the problem statement
2. **Identify** given inputs and required outputs
3. **Design** approach using flowchart/pseudocode
4. **Implement** in programming language

- Flowchart Components
- **Oval**: Start/End points
- **Parallelogram**: Input/Output operations
- **Rectangle**: Processing/Calculations
- **Diamond**: Decision making
- **Arrows**: Program flow direction

- Programming Fundamentals
- Code needs compilation to run on computer
- Flowcharts help visualize logic before coding
- Pseudocode provides language-independent solutions
- Loops enable repetitive operations
- Conditions enable decision making

---

2 First Program & Data Types

Love Babbar - CodeHelp Series

- 2_first_prof_in_cpp.pdf [IMP] [HowDatatypeStoredInMemory]

Complete Example Program

```
#include <iostream>
```

```
using namespace std;

int main() {
   // Variable declarations
    int age = 20;
    char grade = 'A';
    bool isPassed = true;
    float percentage = 85.5f;

    // Output
    cout << "Age: " << age << endl;
    cout << "Grade: " << grade << endl;
    cout << "Passed: " << isPassed << endl;
    cout << "Percentage: " << percentage << endl;

    // Size information
    cout << "Size of int: " << sizeof(int) << " bytes" << endl;
    cout << "Size of char: " << sizeof(char) << " bytes" << endl;
    // Age: 20
    // Grade: A
    // Passed: 1
    // Percentage: 85.5
    // Size of int: 4 bytes
    // Size of char: 1 bytes

    cout<<"Hello World"
        <<endl
        <<'h'
        <<"\n";
    int a=5;
    char ch ='a';
    bool b=1;
    bool b1=true;
    float f=1.32;
    double d=1.234234234;
    cout<<sizeof(a)
        <<endl;
    //hi
    cout<<(int)'a'<<endl;
    cout<<char(65)<<endl;
    int aa = 'a';
    cout<<aa<<endl;
    char chh = 99;
    unsigned z=1122;
     unsigned z11=-1122;//wrong way

    //comment
    /*
    comment

    comment*/
    int a1= 2/5;
    cout<<a1;
    int a22=2.0/5;
    double a2=2.0/5;

    int zz =2;
    int yy=3;
    bool z1 = (a==b),
    z2= (a>b),
    z3= (a<b),
    z4= (a<=b),
    z5= (a>=b),
    z6= (a!=b)
    ;
cout<<endl;
    cout<< (true && false)<< (true || false)<< (!true);

    cout<<endl;
    cout<<(7&4)<<(7|4)<<(7^4)<<(~4)<<(5<<2)<<(16>>2);

    int x=10;
    float y=x;

    float x11=0.14;
    int y11= (int)x11;




    return 0;
}
```

Table of Contents

Compilation Process

- How Programs Work

```
Source Code → Compiler → Executable File → Output
    (.cpp)       (gcc)       (.exe/.out)    (result)
```

- C/ C++ is inbuilt in Linux, as Linux is made in C/ C++ !!!

- Compiler Functions

1. **Translation**: Converts source code to machine code
2. **Error Detection**: Finds compile-time and runtime errors

---

First Program - Hello World

- Basic Structure

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

int main() {
    cout << "NAMASTEY DUNIYA" << endl;
    return 0;
}
```

- IDE
  - Integrated Development Environment
  - Let's You code, debug, execute
  - gives you a lab to experiment
  - eg: VSC(Local IDE), Replit(Remote IDE)
- `;` end of statement, EOL
- `<<` means enterline, syntax of cout function
- `return 0;` : end/exit
- `"abcd"` : string & syntax
- library : collection of pre-built code essentials to directy plug and play
- `endl` : newline + flush operation, slower than `\n`
- `iostream` : library contains all input output codes
- `\n` : newline character, faster

- Code Breakdown

```
#include <iostream>  ← Include file for input/output functions
using namespace std; ← Use standard namespace
int main() {         ← START block (entry point)
    // code here     ← Program logic
}                    ← Boundary markers
```

- Output Methods

```
// Method 1: Using endl
cout << "Hello World" << endl;

// Method 2: Using \n
cout << "Hello World\n";

// Method 3: Combined
cout << "Hello World" << endl << "Next Line" << endl;
```

Output:

```
Hello World
Next Line
```

Data Types & Variables

- `dataType VarName = Value;`
- `int` = 4bytes, 2bytes etc. DEPENDS ON COMPUTER
- once you defined a Variable with one datatype, YOU CAN'T REDEFINE IN CODE, USE A DIFFERENT VAR !!!
- `char a = '123';` [NO], Can't store more than 1 char
- `char a = "b";` [NO], use single-quotes `'`, not `"` double-quotes
- even `bool b = true;` then when cout, it gives `1`

- Primary Data Types

Data Type	Size	Range	Example
int	4 bytes	-2^{31} to 2^{31}	int a = 5
char	1 byte	0 to 255	char c='a'
bool	1 byte	true/false	bool b=true
float	4 bytes	6-7 digits	float f=1.2
double	8 bytes	15 digits	double d=1.2

- Variable Declaration Examples

```
// Integer
int a = 123;
unsigned int ua = 456;  // Only positive values

// Character
```

```cpp
char ch = 'a';           // Single quotes for char

// Boolean
bool flag = true;        // or false, 1, 0
bool isValid = 1;

// Floating point
float price = 99.99f;
double precision = 3.14159265;

// Finding size
cout << sizeof(a) << endl;  // Output: 4
```

- Variable Naming Rules

```
✅ Valid Names:          ❌ Invalid Names:
   abc1                     1abc (starts with number)
   _variable                var-name (hyphen not allowed)
   myVariable               my variable (spaces not allowed)
   MAX_SIZE                 #count (special chars not allowed)
```

Memory Storage

- 2_first_prof_in_cpp.pdf [IMP] [HowDatatypeStoredInMemory]

- Integer Storage (4 bytes = 32 bits)

```
int a = 8;

Binary of 8: 1000
Memory Layout (32 bits):
┌─┬─┬─┬─┬─┬─┬─┬─┬─┬─┬─┬─┬─┬─┬─┬─┬─┬─┬─┬─┬─┬─┬─┬─┬─┬─┬─┬─┬─┬─┬─┬─┐
│0│0│0│0│0│0│0│0│0│0│0│0│0│0│0│0│0│0│0│0│0│0│0│0│0│0│0│0│1│0│0│0│
└─┴─┴─┴─┴─┴─┴─┴─┴─┴─┴─┴─┴─┴─┴─┴─┴─┴─┴─┴─┴─┴─┴─┴─┴─┴─┴─┴─┴─┴─┴─┴─┘
 ←────────────────── 28 zeros ──────────────────→ ←─ 1000 ─→
```

- Character Storage (ASCII)

```cpp
char ch = 'a';
```

```
'a' → ASCII value 97 → Binary 01100001
Memory Layout (8 bits):
┌─┬─┬─┬─┬─┬─┬─┬─┐
│0│1│1│0│0│0│0│1│
└─┴─┴─┴─┴─┴─┴─┴─┘
```

- Negative Number Storage (2's Complement)

```
For -5:
Step 1: Binary of 5 = 101
Step 2: Pad to 32 bits = 00000000000000000000000000000101
Step 3: 1's Complement = 11111111111111111111111111111010
Step 4: 2's Complement = 11111111111111111111111111111011
        (Add 1 to 1's complement)

First bit = 1 (indicates negative)
```

- Signed vs Unsigned

```
Signed int:     -2³¹ to (2³¹ - 1)
Unsigned int:   0 to (2³² - 1)

┌─────────────┬─────────────┬─────────────┐
│    Type     │  Min Value  │  Max Value  │
├─────────────┼─────────────┼─────────────┤
│ signed int  │ -2,147,483,648 │ 2,147,483,647 │
│ unsigned int│      0      │ 4,294,967,295 │
└─────────────┴─────────────┴─────────────┘
```

Type Casting

- Implicit Type Casting

```
// Character to Integer
int a = 'a';          // a = 97 (ASCII of 'a')
cout << a;            // Output: 97

// Integer to Character
char ch = 98;         // ch = 'b' (ASCII 98)
cout << ch;           // Output: b

// Overflow Example
```

```
char ch1 = 123456;   // Only last 8 bits stored
cout << ch1;         // Output: @ (ASCII 64)
```

- Data Type Conversion Chart

```
char → int → float → double
 ↑                      ↓
 └─── Explicit Cast ───┘
```

Operators

- Arithmetic Operators

```
int a = 10, b = 3;

cout << a + b;     // Addition: 13
cout << a - b;     // Subtraction: 7
cout << a * b;     // Multiplication: 30
cout << a / b;     // Division: 3 (integer division)
cout << a % b;     // Modulus: 1
```

- Division Rules

```
// Integer / Integer = Integer
int result1 = 2 / 5;        // result1 = 0

// Float / Integer = Float (if stored in float)
float result2 = 2.0 / 5;    // result2 = 0.4
float result3 = 2 / 5;      // result3 = 0 (still integer division)
```

- Relational Operators

```
int a = 5, b = 3;

cout << (a == b);    // Equal to: 0 (false)
cout << (a > b);     // Greater than: 1 (true)
cout << (a < b);     // Less than: 0 (false)
cout << (a >= b);    // Greater equal: 1 (true)
cout << (a <= b);    // Less equal: 0 (false)
cout << (a != b);    // Not equal: 1 (true)
```

- Logical Operators

```
bool condition1 = true, condition2 = false;

// AND (&&) - All conditions must be true
cout << (condition1 && condition2);  // 0 (false)

// OR (||) - At least one condition must be true
cout << (condition1 || condition2);  // 1 (true)

// NOT (!) - Reverses the condition
cout << (!condition1);               // 0 (false)
cout << (!condition2);               // 1 (true)
```

- Logical Operators Truth Table

A	B	A && B	A || B	!A
T	T	T	T	F
T	F	F	T	F
F	T	F	T	T
F	F	F	F	T

- Assignment Operator

```
int a = 5;       // Assigns value 5 to variable a
a = 10;          // Updates value of a to 10
```

Key Takeaways

- Program Structure

1. **#include** - For input/output operations
2. **using namespace std** - Use standard namespace
3. **int main()** - Entry point of program
4. **cout** - For output, **endl** - For newline
5. **Semicolon (;)** - Statement terminator

- Memory Management
- **int**: 4 bytes, stores integers
- **char**: 1 byte, stores single character (ASCII)
- **bool**: 1 byte, stores true/false
- **float**: 4 bytes, decimal numbers
- **double**: 8 bytes, high precision decimals

- Important Rules
- Variable names can't start with numbers
- Integer division gives integer result
- Negative numbers stored using 2's complement
- Type casting happens automatically in mixed operations

---

3 Conditionals, Loops & Patterns

Conditionals

- 1.1 Introduction to Conditionals

Conditionals are fundamental programming constructs that allow different blocks of code to be executed based on whether certain conditions are met. The choice of which code to execute depends on the evaluation of an expression to either `true` or `false`.

Flowchart Representation: A conditional statement can be visualized using a flowchart, where a diamond shape represents the decision point.

```
          Code?
            |
            V
      +----------+
      |  a > b?  |<-- YES ----> Executes code if true
      +----------+
            |
            NO
            |
            V
          ...
```

- 1.2 `if` Statement

The `if` statement allows a block of code to be executed only if a specified condition is true.

Problem Statement (PS): Given two inputs, `a` and `b`. **Output (O/P):**

- If `a > b`, the answer is "Answer is A".
- If `b > a`, the answer is "Answer is B".

Pseudocode/C++ Example:

```
if (a > b) {
    cout << "Answer is A";
}
```

```
if (b > a) {
    cout << "Answer is B";
}
```

Example Trace: Consider `a = 5` and `b = 14`.

- The condition `if (a > b)` (i.e., `5 > 14`) evaluates to `false`. The code block inside this `if` statement would not execute.
- Similarly, `if (a >= b)` (i.e., `5 >= 14`) also evaluates to `false`, and its code block would not execute.

- 1.3 `if-else` Statement

The `if-else` statement provides two paths of execution: one if the condition is `true` and another if it is `false`.

Example: Checking if a number is positive or not Input: `a = 5` **Logic:** Determine if `a` is positive (`+ve`) or not positive (`not +ve`).

Pseudocode/C++ Example:

```
if (a > 0) {
    print("positive"); // or cout << "+ve";
} else {
    print("not positive"); // or cout << "not +ve";
}
```

Example Trace:

- If `a = 5`, `(a > 0)` is `true`, so "positive" is printed.
- If `a = -2`, `(a > 0)` is `false`, so the `else` block executes, and "not positive" is printed.
- If `a = 0`, `(a > 0)` is `false`, so the `else` block executes, and "not positive" is printed.

- 1.4 `if-else if-else` Ladder

This structure allows for multiple conditions to be checked sequentially. Once a condition is `true`, its corresponding block is executed, and the rest of the ladder is skipped. If none of the `if` or `else if` conditions are met, the `else` block (if present) is executed.

Structure:

```
if (condition1) {
    // Code for condition1
} else if (condition2) {
    // Code for condition2
} else if (condition3) {
    // Code for condition3
} else {
    // Code if none of the above conditions are true
}
```

- 1.5 Nested Conditionals

Conditionals can be nested within other conditional statements. This allows for more complex decision-making processes.

Structure Example:

```
if (conditionA) {
    if (conditionB) {
        // Code for A and B
    } else {
        // Code for A but not B
    }
} else {
    // Code for not A
}
```

- 1.6 Character Classification Example

A common application of `if-else if-else` is to classify characters.

Problem Statement: Given a character `ch`, determine if it is a lowercase letter, an uppercase letter, or a numeric digit.

Pseudocode/Logic:

```
char ch; // Declare a character variable
// Input ch

if (ch >= 'a' && ch <= 'z') {
    // This is lowercase small case
} else if (ch >= 'A' && ch <= 'Z') {
    // This is Upper case
} else if (ch >= '0' && ch <= '9') {
    // This is numeric
} else {
    // Other character
}
```

- 1.7 Key Takeaways for Conditionals

- Conditionals (`if`, `if-else`, `if-else if-else`) control program flow based on boolean expressions.
- They enable programs to make decisions and execute specific code blocks accordingly.
- Nesting conditionals allows for handling more intricate decision logic.
- `else` blocks provide a fallback execution path when preceding conditions are false.

Loops

- 2.1 Introduction to Loops

Loops are programming constructs that execute a block of code repeatedly until a certain condition is met. This is useful for tasks that require repetitive actions, such as iterating through a range of numbers or processing elements in a collection.

- 2.2 `while` Loop Flowchart

The `while` loop continues to execute "jab tak" (as long as) its condition remains true.

Flowchart for Printing Numbers 1 to N:

```
        Start
          |
          V
      +--------+
      | Read N |
      +--------+
          |
          V
      +--------+
      | Let i = 1 |
      +--------+
          |
          V
      +-----------+
      |  i <= N? |<--- YES -----+
      +-----------+             |
           | NO                 |
           V                    |
          End                   |
                                |
          +---------------------+
          |
          V
```

```
     +-------+
     | Print i |
     +-------+
         |
         V
     +---------+
     | i = i + 1 |
     +---------+
```

- 2.3 `while` Loop: Printing Numbers 1 to N

Problem Statement: Print numbers from `1` to `N`.

Pseudocode/C++ Example:

```
int n;
cin >> n; // Input for N (e.g., n = 5)

int i = 1; // Initialize counter
while (i <= n) { // Condition: loop as long as i is less than or equal to N
    cout << i << " "; // Print current value of i
    i = i + 1; // Increment i
}
// Output for n=5: 1 2 3 4 5
```

Trace for `n = 5`:

- `i = 1`: `1 <= 5` (True), prints `1`, `i` becomes `2`
- `i = 2`: `2 <= 5` (True), prints `2`, `i` becomes `3`
- `i = 3`: `3 <= 5` (True), prints `3`, `i` becomes `4`
- `i = 4`: `4 <= 5` (True), prints `4`, `i` becomes `5`
- `i = 5`: `5 <= 5` (True), prints `5`, `i` becomes `6`
- `i = 6`: `6 <= 5` (False), loop terminates. The output is `1 2 3 4 5`.

- 2.4 `while` Loop: Sum of Numbers 1 to N

Problem Statement: Find the sum of all numbers from `1` to `N`.

Pseudocode/C++ Example:

```
int n;
cin >> n; // Input N

int sum = 0; // Initialize sum to 0
int i = 1; // Initialize counter
while (i <= n) { // Loop as long as i is less than or equal to N
    sum = sum + i; // Add current i to sum
    i = i + 1; // Increment i
}
// After loop, sum holds the total (e.g., for n=5, sum is 15)
```

Trace for `n = 5`:

`i`	`sum` (before `sum = sum + i`)	`sum` (after `sum = sum + i`)	`i` (after `i = i + 1`)	Condition `i <= n`
1	0	1	2	True
2	1	3	3	True
3	3	6	4	True
4	6	10	5	True
5	10	15	6	True
6	15	15	6	False

Arithmetic Series Formula: The sum of an arithmetic series from `1` to `N` can also be calculated using the formula: `sum = n/2 * (first_term + last_term)`. For `1` to `N`, this becomes `sum = n/2 * (1 + N)`. **Example:** For `N = 100`, `sum = 100/2 * (1 + 100) = 50 * 101 = 5050`.

- 2.5 Practice Problems (Homework)

- Find the sum of all even numbers from `1` to `N`.
- Generate a multiplication table for a given number.
- Determine if a number is `Prime` or `Not`.

- 2.6 Prime or Not Algorithm

Problem Statement: Given an integer `n`, determine if it is a prime number. A prime number is a natural number greater than 1 that has no positive divisors other than 1 and itself.

Logic: To check if a number `n` is prime, you can iterate from `2` up to `n-1`. If `n` is divisible by any of these numbers (i.e., the modulo operation `%` results in a remainder of `0`), then `n` is not prime. If no such divisor is found, `n` is prime.

- The `%` (modulo operator) gives the remainder of a division.
- If `n % i == 0` for any `i` in the range `[2, n-1]`, then `n` is `Not prime`.
- If `n % i != 0` for all `i` in the range `[2, n-1]`, then `n` is `Prime`.
- It is important to loop `i < n` (or `i <= n-1`), not `i <= n`, because `n % n` will always be `0`, which would incorrectly mark any number `n > 1` as not prime.

Pseudocode/C++ Example:

```
int n;
cin >> n; // Input n (e.g., n = 5)
```

```
bool isPrime = true; // Assume n is prime initially

// Check for divisibility from 2 up to n-1
int i = 2;
while (i < n) { // Loop condition: i less than n
    if (n % i == 0) { // If n is divisible by i
        isPrime = false; // Mark as not prime
        break; // Exit loop early
    }
    i = i + 1;
}

if (isPrime) {
    cout << n << " is Prime";
} else {
    cout << n << " is Not Prime";
}
```

Example Trace for `n = 7`:

- `i = 2`: `7 % 2 != 0` (True)
- `i = 3`: `7 % 3 != 0` (True)
- `i = 4`: `7 % 4 != 0` (True)
- `i = 5`: `7 % 5 != 0` (True)
- `i = 6`: `7 % 6 != 0` (True)
- `i = 7`: Condition `i < n` (i.e., `7 < 7`) is `false`, loop terminates. Since `isPrime` remains `true`, `7` is determined to be `Prime`.

- 2.7 Key Takeaways for Loops

- Loops (`while`) are used for repetitive execution of code blocks.
- They rely on a condition that controls their termination.
- Loops are essential for tasks like counting, summing, and iterating through data.
- The `break` statement can be used to exit a loop prematurely, for example, once a condition is met (like finding a divisor for a prime check).

Patterns

- 3.1 Introduction to Patterns

Pattern printing involves using nested loops to create various shapes and designs, often using characters or numbers, in a grid-like output. These exercises help in understanding the control flow of nested loops and the relationship between row and column indices. The outer loop typically controls the rows, and the inner loop controls the columns.

General structure for patterns:

```
int n; // Input for number of rows/columns (e.g., n=4 for a 4x4 pattern)
cin >> n;

int row = 1;
while (row <= n) { // Outer loop for rows
    int col = 1;
    while (col <= n) { // Inner loop for columns (or based on row)
        // Print character/number
        // cout << value;
        col = col + 1;
    }
    cout << endl; // Move to next line after each row
    row = row + 1;
}
```

- 3.2 Basic Rectangular Patterns

These patterns typically involve a fixed number of rows and columns, usually `n` by `n`.

- - 3.2.1 Printing Stars

Pattern: `n` rows, each with `n` stars. **Example for `n=4`:**

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

Logic: Outer loop for `row` from `1` to `n`. Inner loop for `col` from `1` to `n`. Print `*` in the inner loop.

Pseudocode (using the general structure):

```
// Outer loop for rows
int row = 1;
while (row <= n) {
    // Inner loop for columns
    int col = 1;
    while (col <= n) {
        cout << "*";
        col = col + 1;
    }
    cout << endl; // Newline after each row
    row = row + 1;
}
```

- - 3.2.2 Printing Row Numbers

Pattern: Each row prints its row number `n` times. **Example for `n=3`:**

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

Logic: Print `row` in the inner loop.

Pseudocode:

```
// Outer loop for rows
int row = 1;
while (row <= n) {
    // Inner loop for columns
    int col = 1;
    while (col <= n) {
        cout << row << " "; // Print the current row number
        col = col + 1;
    }
    cout << endl;
    row = row + 1;
}
```

- - 3.2.3 Printing Column Numbers

Pattern: Each row prints numbers from `1` to `n`. **Example for `n=3`:**

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

Logic: Print `col` in the inner loop.

Pseudocode:

```
// Outer loop for rows
int row = 1;
while (row <= n) {
    // Inner loop for columns
    int col = 1;
    while (col <= n) {
        cout << col << " "; // Print the current column number
        col = col + 1;
    }
    cout << endl;
    row = row + 1;
}
```

- - 3.2.4 Printing Decreasing Column Numbers

Pattern: Each row prints numbers from `n` down to `1`. **Example for `n=3`:**

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

Logic: Print `(n - col + 1)` in the inner loop.

Pseudocode:

```
// Outer loop for rows
int row = 1;
while (row <= n) {
    // Inner loop for columns
    int col = 1;
    while (col <= n) {
        cout << (n - col + 1) << " "; // Print calculated value
        col = col + 1;
    }
    cout << endl;
    row = row + 1;
}
```

- 3.3 Basic Triangular Patterns

These patterns typically have the number of columns in each row dependent on the current row number.

- - 3.3.1 Printing Row Numbers (Left Triangle)

Pattern: The `row`th row prints `row` number of times the `row` number. **Example for `n=4`:**

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

Logic: Inner loop for `col` from `1` to `row`. Print `row` in the inner loop.

Pseudocode:

```
// Outer loop for rows
int row = 1;
while (row <= n) {
    // Inner loop for columns
    int col = 1;
    while (col <= row) { // Column count depends on row
        cout << row << " ";
        col = col + 1;
    }
    cout << endl;
    row = row + 1;
}
```

- - 3.3.2 Printing Column Numbers (Left Triangle)

Pattern: The `row`th row prints numbers from `1` to `row`. **Example for `n=4`:**

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

Logic: Inner loop for `col` from `1` to `row`. Print `col` in the inner loop.

Pseudocode:

```
// Outer loop for rows
int row = 1;
while (row <= n) {
    // Inner loop for columns
    int col = 1;
    while (col <= row) { // Column count depends on row
        cout << col << " ";
        col = col + 1;
    }
    cout << endl;
    row = row + 1;
}
```

- - 3.3.3 Printing Increasing Count (Left Triangle)

Pattern: Prints numbers in an increasing sequence, filling a left triangle shape. **Example for `n=4`:**

```
1
2 3
4 5 6
7 8 9 10
```

Logic: Introduce a `count` variable, initialized to `1` before the loops. Print `count` and then increment `count` in the inner loop.

Pseudocode:

```
int count = 1; // Initialize count outside loops
// Outer loop for rows
int row = 1;
while (row <= n) {
    // Inner loop for columns
    int col = 1;
    while (col <= row) {
        cout << count << " ";
        count = count + 1; // Increment count
        col = col + 1;
    }
    cout << endl;
    row = row + 1;
}
```

- - 3.3.4 Printing Row-Based Increasing Sequence

Pattern: For each row, the numbers start from `row` and increase sequentially. **Example for `n=4`:**

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

Logic: The value to print can be derived as `row + col - 1`. Or, initialize `value = row` at the start of each row's inner loop, then print `value` and increment `value`.

Pseudocode:

```
// Outer loop for rows
int row = 1;
while (row <= n) {
    int value = row; // Start value for current row
    // Inner loop for columns
    int col = 1;
    while (col <= row) {
        cout << value << " "; // Print current value
        value = value + 1; // Increment value
        col = col + 1;
    }
    cout << endl;
    row = row + 1;
}
// Alternative using formula: cout << (row + col - 1) << " ";
```

- - 3.3.5 Printing `(n-row+col-1)` Decreasing Sequence [Clarified]

Pattern: Numbers decrease based on `n` and `row` for each column in a triangle. **Example for `n=4`:**

```
4
3 2
2 1 0
1 0 -1 -2
```

Logic: The starting value for each row is `n - row + 1`. Subsequent values in the row decrement. The value to print is `(n - row + 1) - (col - 1)` which simplifies to `n - row - col + 2`.

Pseudocode:

```
// Outer loop for rows
int row = 1;
while (row <= n) {
    int col = 1;
    // Initial value for the current row
    int val = n - row + 1;
    while (col <= row) {
        cout << val << " "; // Print starting value and decrement
        val = val - 1;
        col = col + 1;
    }
    cout << endl;
    row = row + 1;
}
// Alternative using formula: cout << (n - row + 1 - (col - 1)) << " ";
```

- 3.4 Character Patterns

Similar to number patterns, but using characters, often derived from `'A'` by adding an offset.

- - 3.4.1 Rectangular Character Pattern (Row-based)

Pattern: Each row prints the same character, which changes based on the row number. **Example for `n=3`:**

```
A A A
B B B
C C C
```

Logic: The character is `'A' + row - 1`.

Pseudocode:

```
// Outer loop for rows
int row = 1;
while (row <= n) {
    // Inner loop for columns
    int col = 1;
    while (col <= n) {
        char ch = 'A' + row - 1; // Calculate character based on row
        cout << ch << " ";
        col = col + 1;
    }
    cout << endl;
    row = row + 1;
}
```

- - 3.4.2 Triangular Character Pattern (Row-based)

Pattern: Each row prints its corresponding character `row` times. **Example for `n=3`:**

```
A
B B
C C C
```

Logic: The character is `'A' + row - 1`. Inner loop runs `col <= row`.

Pseudocode:

```
// Outer loop for rows
int row = 1;
while (row <= n) {
    // Inner loop for columns
    int col = 1;
    while (col <= row) { // Columns depend on row
        char ch = 'A' + row - 1;
        cout << ch << " ";
        col = col + 1;
    }
    cout << endl;
    row = row + 1;
}
```

- - 3.4.3 Triangular Character Pattern (Column-based)

Pattern: Each row prints characters `A`, `B`, `C`... up to the `row`th character. **Example for `n=3`:**

```
A
A B
A B C
```

Logic: The character is `'A' + col - 1`. Inner loop runs `col <= row`.

Pseudocode:

```
// Outer loop for rows
int row = 1;
while (row <= n) {
    // Inner loop for columns
    int col = 1;
    while (col <= row) { // Columns depend on row
        char ch = 'A' + col - 1; // Calculate character based on column
        cout << ch << " ";
        col = col + 1;
    }
    cout << endl;
    row = row + 1;
}
```

- - 3.4.4 Rectangular Character Pattern (Counting)

Pattern: Prints characters in an increasing sequence, filling an `n` by `n` grid. **Example for `n=3`:**

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

Logic: Initialize a `char value = 'A'` before loops. Print `value` and then increment `value` in the inner loop.

Pseudocode:

```
char value = 'A'; // Initialize starting character
// Outer loop for rows
int row = 1;
while (row <= n) {
    // Inner loop for columns
    int col = 1;
    while (col <= n) {
        cout << value << " ";
        value = value + 1; // Increment character value
        col = col + 1;
    }
    cout << endl;
    row = row + 1;
}
```

- - 3.4.5 Rectangular Character Pattern (Formula `A + row + col - 2`)

Pattern: Prints characters where the value is a sum of row and column indices relative to `'A'`. **Example for `n=3`:**

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

Logic: The character to print is `char('A' + row + col - 2)`.

- For `row=1, col=1`: `'A' + 1 + 1 - 2 = 'A'`
- For `row=1, col=2`: `'A' + 1 + 2 - 2 = 'B'`
- For `row=2, col=1`: `'A' + 2 + 1 - 2 = 'B'`

Pseudocode:

```
// Outer loop for rows
int row = 1;
while (row <= n) {
    // Inner loop for columns
    int col = 1;
    while (col <= n) {
        char ch = 'A' + row + col - 2; // Calculate character using formula
        cout << ch << " ";
        col = col + 1;
    }
    cout << endl;
    row = row + 1;
}
```

- - 3.4.6 Triangular Character Pattern (Formula `A + n - row + col - 1`) [Clarified]

Pattern: Prints a character triangle that starts from a higher character and increases left to right. **Example for `n=4`:**

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

Logic: The starting character for each row decreases, and then characters increment within the row. The character to print is `char('A' + n - row + col - 1)`.

- For `n=4, row=1, col=1`: `'A' + 4 - 1 + 1 - 1 = 'A' + 3 = 'D'`
- For `n=4, row=2, col=1`: `'A' + 4 - 2 + 1 - 1 = 'A' + 2 = 'C'`
- For `n=4, row=3, col=2`: `'A' + 4 - 3 + 2 - 1 = 'A' + 2 = 'C'`

Pseudocode:

```
// Outer loop for rows
int row = 1;
while (row <= n) {
    // Inner loop for columns
    int col = 1;
    while (col <= row) { // Columns depend on row
        char ch = 'A' + n - row + col - 1; // Calculate character using
formula
        cout << ch << " ";
        col = col + 1;
    }
    cout << endl;
    row = row + 1;
}
```

- 3.5 Patterns with Spaces

These patterns combine printing characters/numbers with printing spaces to achieve specific alignments.

- - 3.5.1 Right-Aligned Star Triangle

Pattern: A triangle of stars aligned to the right. **Example for `n=4`:**

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

Logic: For each row `row`:

1. Print `n - row` spaces.
2. Print `row` stars.

Pseudocode:

```
// Outer loop for rows
int row = 1;
while (row <= n) {
    // Inner loop for spaces
    int space = n - row;
    while (space > 0) {
        cout << " ";
        space = space - 1;
    }
    // Inner loop for stars
    int col = 1;
    while (col <= row) {
        cout << "*";
        col = col + 1;
    }
    cout << endl;
    row = row + 1;
}
```

- - 3.5.2 Inverted Right-Aligned Star Triangle

Pattern: An inverted triangle of stars aligned to the right. **Example for `n=4`:**

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

Logic: For each row `row`:

1. Print `row - 1` spaces.
2. Print `n - row + 1` stars.

Pseudocode:

```
// Outer loop for rows
int row = 1;
while (row <= n) {
    // Inner loop for spaces
```

```
        int space = row - 1;
        while (space > 0) {
            cout << " ";
            space = space - 1;
        }
        // Inner loop for stars
        int col = 1;
        while (col <= (n - row + 1)) { // Number of stars decreases
            cout << "*";
            col = col + 1;
        }
        cout << endl;
        row = row + 1;
    }
```

- 3.6 Homework Patterns

Several patterns are provided as homework to practice.

Pattern 1: Inverted Decreasing Row Numbers (Right Aligned) Example for `n=4`:

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

Pattern 2: Inverted Increasing Row Numbers Example for `n=4`:

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

Pattern 3: Inverted Increasing Column Numbers Example for `n=4`:

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

Pattern 4: Right-Aligned Increasing Row Numbers Example for `n=4`:

```
   1
  2 2
```

```
   3  3  3
  4  4  4  4
```

Pattern 5: Right-Aligned Increasing Count Triangle Example for `n=4`:

```
   1
  2 3
 4 5 6
7 8 9 10
```

- 3.7 The "Dabang" Pattern

This is a complex pattern combining numbers and stars, typically symmetrical around a central axis.

Example for `n=5`:

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

Logic (Based on visual observation for `n=5` from source [Clarified]): For each `row` from `1` to `n`:

1. **Print Left Numbers:** Print numbers `j` from `1` up to `n - row + 1`.
2. **Print Middle Stars:**
   - For `row = 1`, print `0` stars.
   - For `row = 2`, print `1` star.
   - For `row = 3`, print `3` stars.
   - For `row = 4`, print `6` stars.
   - For `row = 5`, print `8` stars. *(Note: A simple general formula for the number of stars was not explicitly clear from the handwritten notes and varied from common "Dabang" patterns. The counts above are direct observations from the provided example.)*
3. **Print Right Numbers:** Print numbers `j` from `(n - row + 1) - 1` down to `1`.
4. After each row, move to a new line.

- 3.8 Key Takeaways for Patterns

- Nested loops (outer for rows, inner for columns) are the primary tool for generating patterns.
- The condition for the inner loop often depends on the current `row` number for triangular patterns.
- Formulas involving `row`, `col`, and `n` can be used to determine the value or character to print at each position.
- Introducing auxiliary variables like `count` or `value` can simplify patterns with sequential numbers or characters.

- Spaces (   ) can be printed strategically using additional loops to align patterns.
- Complex patterns often combine multiple simple pattern-generating techniques.
