Name: Prathamesh Arvind Jadhav

DSA Course

DAY-1:

CPP Basics:

1. Input and Output in C++

Header File Required

#include <iostream>

using namespace std;

Input

- Uses cin (Console Input) with the extraction operator >>

int age;

cin >> age; // User inputs a value for age

Output

- Uses cout (Console Output) with the insertion operator <<

cout << "Age is: " << age;

2. Variables in C++

- A **variable** is a container for storing data.
- Must be **declared with a data type** before using.

int age = 20;

float weight = 55.5;

Rules for Naming Variables

- Can contain letters, digits, and underscores.
- Cannot start with a digit.
- Cannot be a keyword (like int, float, etc.).

Name: Prathamesh Arvind Jadhav

3. Data Types in C++

Type	Description	Example
int	Integer (4 bytes)	10
float	Floating point (4 bytes)	3.14
double	Double precision float(8 Byte)	3.14159
char	Character (1 byte)	'A'
bool	Boolean (true/false)	true, false
string	Text (needs <string>)	"Hello"

Example

```
#include <string>

string name = "John";
```

4. Operators in C++

Arithmetic Operators

```
+   // Addition

-   // Subtraction

*   // Multiplication

/   // Division

%   // Modulus (remainder)
```

Relational (Comparison) Operators

```
==  // Equal to
```

!= // Not equal to

> // Greater than

< // Less than

>= // Greater than or equal to

<= // Less than or equal to

Logical Operators

&& // Logical AND

|| // Logical OR

! // Logical NOT

Assignment Operators

= // Assign

+= // Add and assign

-= // Subtract and assign

*= // Multiply and assign

/= // Divide and assign

Increment/Decrement

++i; // Pre-increment

i++; // Post-increment

--i; // Pre-decrement

i--; // Post-decrement

5. Type Casting in C++

- **Type casting** converts a variable from one type to another.

Syntax

Name: Prathamesh Arvind Jadhav

data_type(variable)

Example

int a = 10, b = 3;

float result = (float)a / b; // Output: 3.33333

- **Implicit casting**: Automatic by compiler when safe.
- **Explicit casting**: Done by the programmer.

Conditional Statements (Decision Making)

Conditional statements help in making decisions based on certain conditions.

 Types of Conditional Statements:

---

 if Statement

- **Syntax:**

  ```
  if (condition) {
     // code to execute if condition is true
  }
  ```

- **Example:**

  ```
  int age = 20;
  if (age >= 18) {
     cout << "You are eligible to vote.";
  }
  ```

---

 if-else Statement

- **Syntax:**

  ```
  if (condition) {
     // true block
  } else {
     // false block
  ```

```
}
```

- **Example:**

```
int marks = 45;
if (marks >= 50) {
   cout << "Passed";
} else {
   cout << "Failed";
}
```

---

 else if Ladder

- Useful when checking multiple conditions.
- **Syntax:**

```
if (condition1) {
   // block1
} else if (condition2) {
   // block2
} else {
   // default block
}
```

- **Example:**

```
int score = 75;
if (score >= 90) {
   cout << "Grade A";
} else if (score >= 80) {
   cout << "Grade B";
} else if (score >= 70) {
   cout << "Grade C";
} else {
   cout << "Fail";
}
```

---

 switch Statement

- Best for checking equality against multiple values.
- **Syntax:**

```
switch (expression) {
   case value1:
      // code block
      break;
   case value2:
      // code block
      break;
   default:
      // default block
}
```

- **Example:**

```
int day = 3;
switch (day) {
   case 1: cout << "Monday"; break;
   case 2: cout << "Tuesday"; break;
   case 3: cout << "Wednesday"; break;
   default: cout << "Invalid Day";
}
```

□ 2. Loops (Repetitive Tasks)

Loops allow executing a block of code multiple times.

□ for Loop

- Best when you know how many times to loop.
- **Syntax:**

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

- **Example:**

```
for (int i = 1; i <= 5; i++) {
    cout << i << " ";
}
```

□ while Loop

- Condition is checked **before** execution.
- Best when the number of iterations is unknown.
- **Syntax:**

```
while (condition) {
    // code
}
```

- **Example:**

```
int i = 1;
while (i <= 5) {
    cout << i << " ";
    i++;
}
```

□ do-while Loop

- Condition is checked **after** the loop body.
- Executes at least once.
- **Syntax:**

```
do {
    // code
} while (condition);
```

- **Example:**

```
int i = 1;
do {
    cout << i << " ";
    i++;
} while (i <= 5);
```

Name: Prathamesh Arvind Jadhav

□ Nested Loops

- A loop inside another loop.
- **Example:**

```
for (int i = 1; i <= 3; i++) {
   for (int j = 1; j <= 3; j++) {
      cout << "(" << i << "," << j << ") ";
   }
   cout << endl;
}
```

□ Loop Control Statements

These control the flow inside loops.

□ break:

- Exits the loop immediately.

```
for (int i = 1; i <= 10; i++) {
   if (i == 5) break;
   cout << i << " ";
}
```

□ continue:

- Skips the current iteration and moves to the next.

```
for (int i = 1; i <= 5; i++) {
   if (i == 3) continue;
   cout << i << " ";
}
```

□ Summary Table

Name: Prathamesh Arvind Jadhav

Statement	Used For	Checks Condition When?	Executes At Least Once?
if, if-else, else-if	Conditional logic	N/A	Depends on condition
switch	Multi-value condition	N/A	Based on case match
for	Fixed iteration	Before loop	No
while	Unknown iteration	Before loop	No
do-while	Unknown iteration	After loop	Yes

Day-2:

Pattern Problems:

□ 1. Right-Angled Triangle of Stars

□ Pattern:

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

□ Code:

```
int n = 5;
for (int i = 1; i <= n; i++) {
   for (int j = 1; j <= i; j++) {
      cout << "* ";
   }
   cout << endl;
}
```

□ 2. Inverted Triangle of Stars

□ Pattern:

Name: Prathamesh Arvind Jadhav

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

□ Code:

```
int n = 5;
for (int i = n; i >= 1; i--) {
    for (int j = 1; j <= i; j++) {
        cout << "* ";
    }
    cout << endl;
}
```

□ 3. Pyramid Pattern

□ Pattern:

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

□ Code:

```
int n = 5;
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n - i; j++) {
        cout << " ";
    }
    for (int k = 1; k <= i; k++) {
        cout << "* ";
    }
    cout << endl;
}
```

Name: Prathamesh Arvind Jadhav

□ 4. Number Triangle

□ Pattern:

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

□ Code:

```
int n = 4;
for (int i = 1; i <= n; i++) {
   for (int j = 1; j <= i; j++) {
      cout << j << " ";
   }
   cout << endl;
}
```

□ 5. Floyd's Triangle

□ Pattern:

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

□ Code:

```
int n = 4, num = 1;
for (int i = 1; i <= n; i++) {
   for (int j = 1; j <= i; j++) {
      cout << num++ << " ";
   }
   cout << endl;
}
```

□ 6. Checkerboard (Using if condition)

Name: Prathamesh Arvind Jadhav

 Pattern:

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

 Code:

```
int n = 4;
for (int i = 1; i <= n; i++) {
   for (int j = 1; j <= n; j++) {
      if ((i + j) % 2 == 0)
         cout << "1 ";
      else
         cout << "0 ";
   }
   cout << endl;
}
```

---

 7. Hollow Rectangle

 Pattern:

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

 Code:

```
int rows = 4, cols = 5;
for (int i = 1; i <= rows; i++) {
   for (int j = 1; j <= cols; j++) {
      if (i == 1 || i == rows || j == 1 || j == cols)
         cout << "* ";
      else
         cout << "  ";
   }
   cout << endl;
```

}

□ 8. Diamond Pattern

□ Pattern:

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

□ Code:

```
int n = 3;
// Upper part
for (int i = 1; i <= n; i++) {
    for (int j = i; j < n; j++) cout << " ";
    for (int k = 1; k <= i; k++) cout << "* ";
    cout << endl;
}
// Lower part
for (int i = n - 1; i >= 1; i--) {
    for (int j = n; j > i; j--) cout << " ";
    for (int k = 1; k <= i; k++) cout << "* ";
    cout << endl;
}
```

DAY-3:

Functions:

□ What is a Function?

A **function** is a block of code that performs a specific task. It helps you **reuse code**, **reduce redundancy**, and **organize** your program efficiently.

□ Why Use Functions?

Name: Prathamesh Arvind Jadhav

- **Modular Code**: Breaks large problems into smaller, manageable parts.
- **Reusability**: Write once, use many times.
- **Readability**: Easier to read and debug.
- **Avoid Redundancy**: No need to repeat the same code again.

□ Types of Functions

Type	Description
Built-in Functions	Provided by C++ like sqrt(), pow()
User-defined Functions	Functions created by the programmer

□ Syntax of a Function

```
return_type function_name(parameter_list) {
   // body of function
   return value; // if return_type is not void
}
```

□ Example:

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

int add(int a, int b) {
   return a + b;
}

int main() {
   int result = add(5, 3);
   cout << "Sum = " << result;
   return 0;
}
```

□ Parts of a Function

Name: Prathamesh Arvind Jadhav

Part	Description
return_type	Type of value the function returns (int, void, etc.)
function_name	Unique name for identification
parameter_list	Values passed into the function
function body	Code block that defines what the function does
return	Sends value back to the caller (if not void)

□ Function Declaration (Prototype)

- Used before main() to declare a function's existence.

```
int add(int, int); // Declaration

int main() {
   cout << add(2, 3);
}

int add(int a, int b) {
   return a + b;
}
```

□ Calling a Function

- To **use** the function, you "call" it with arguments:

```
int result = add(4, 7); // Function call
```

□ Function with No Return (void)

```
void greet() {
   cout << "Hello, User!";
}
```

□ Types of User-defined Functions

Name: Prathamesh Arvind Jadhav

Type	Example
No arguments, no return	void greet()
With arguments, no return	void greet(string name)
No arguments, returns value	int getInput()
With arguments, returns value	int sum(int a, int b)

 Example for All Types

1. No Argument, No Return

```
void showMessage() {
   cout << "Welcome!";
}
```

2. With Argument, No Return

```
void greet(string name) {
   cout << "Hello " << name;
}
```

3. No Argument, With Return

```
int giveNumber() {
   return 10;
}
```

4. With Argument, With Return

```
int square(int x) {
   return x * x;
}
```

 Pass by Value vs Pass by Reference

 Pass by Value (Copy is passed)

```
void change(int a) {
   a = 10;
```

```
}
```

 Pass by Reference (Original is modified)

```
void change(int &a) {
   a = 10;
}
```

 Recursive Functions

A **function calling itself**.

 Example: Factorial

```
int factorial(int n) {
   if (n == 0) return 1;
   return n * factorial(n - 1);
}
```

 Inline Functions

- Used for **short** functions.
- Increases performance by replacing function call with the function code.
- Syntax:

```
inline int square(int x) {
   return x * x;
}
```

 Default Arguments

- Allows **default values** for parameters.

```
int power(int base, int exp = 2) {
   return pow(base, exp);
}
```

Name: Prathamesh Arvind Jadhav

Summary Table

Concept	Example	Description
Declaration	int add(int, int);	Introduce function before use
Definition	int add(int a, int b) {}	Actual code of the function
Call	add(5, 3);	Using the function
Return type	int, void, float	Type of value returned
Recursion	factorial(n)	Function calling itself
Inline function	inline int square(int x)	Suggests compiler to expand inline
Default arguments	power(3)	Optional parameters
Reference vs Value	void func(int &x)	Modify original value

Day-4:

Pointers:

 What is a Pointer?

A **pointer** is a variable that **stores the memory address** of another variable.

 Think of it like:

A pointer is a signboard that shows *where* a house (variable) is, not the house itself.

 Why Use Pointers?

- To **access and modify** variables indirectly.
- To work with **arrays, functions, and dynamic memory**.
- To improve **performance** in large data operations.
- To **pass large data efficiently** to functions.

 Pointer Syntax

data_type *pointer_name;

Name: Prathamesh Arvind Jadhav

* means this variable is a pointer to the given data_type.

 Example 1: Basic Pointer Usage

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

int main() {
   int x = 10;
   int *ptr = &x;  // ptr stores the address of x

   cout << "Value of x: " << x << endl;
   cout << "Address of x: " << &x << endl;
   cout << "Pointer ptr holds: " << ptr << endl;
   cout << "Value pointed by ptr: " << *ptr << endl;

   return 0;
}
```

 Output Explanation:

- &x gives address of x.
- *ptr gives value **stored at** that address.

 Pointer Terms

Term	Meaning
&	Address-of operator
*	Dereference operator (get value)
ptr	Pointer variable (holds address)
*ptr	Value at the address stored in ptr

 Changing Values Using Pointers

```
int x = 5;
```

```
int *ptr = &x;

*ptr = 20;  // changes value of x directly

cout << x;  // Output: 20
```

□ Pointer to Different Data Types

```
int a = 5;
float b = 3.14;
char c = 'A';

int* p1 = &a;
float* p2 = &b;
char* p3 = &c;
```

□ Pointer and Arrays

```
int arr[] = {10, 20, 30};
int *ptr = arr;  // array name is the address of the first element

cout << *ptr;      // 10
cout << *(ptr+1);  // 20
cout << *(ptr+2);  // 30
```

*(ptr + i) gives the i-th element of the array.

□ Pointer to Pointer (Double Pointer)

```
int a = 10;
int *ptr = &a;
int **pptr = &ptr;

cout << **pptr;  // Output: 10
```

First * gets ptr, second * gets a.

Name: Prathamesh Arvind Jadhav

□ Functions and Pointers (Pass by Reference)

□ Example: Swapping two values

```
void swap(int *a, int *b) {
   int temp = *a;
   *a = *b;
   *b = temp;
}

int main() {
   int x = 5, y = 10;
   swap(&x, &y);
   cout << x << " " << y;  // Output: 10 5
}
```

Changes reflect in original variables because we're modifying their addresses.

□ Pointers vs Normal Variables

Normal Variable	Pointer
Stores value	Stores address
int x = 5;	int *p = &x;
Access: x	Access: *p
Address: &x	Address: p

□ Null Pointer

Used to initialize a pointer when it is not assigned yet.

```
int *ptr = nullptr;  // Or NULL in older C++
```

□ Dangling Pointer

Name: Prathamesh Arvind Jadhav

When a pointer refers to memory that has been freed or deleted.

```
int *ptr = new int(10);
delete ptr;       // Memory deleted
// Now ptr is dangling unless reset
ptr = nullptr;
```

 Pointers and Dynamic Memory (new/delete)

 Example: Allocating memory at runtime

```
int *ptr = new int;     // allocate memory
*ptr = 100;
cout << *ptr;

delete ptr;             // free memory
```

Always use delete to avoid memory leaks.

 Summary of Pointer Operators

Operator	Name	Usage
*	Dereference	Get value at address
&	Address-of	Get address of variable
->	Member access	Access members from pointer to object/struct

---

Real-life Analogy

- Variable = **house**
- Address = **house number**
- Pointer = **someone holding the house number**
- *pointer = **going to that house and seeing what's inside**

Name: Prathamesh Arvind Jadhav

DAY-5:

Binary Number System:

□ What is the Binary Number System?

The **Binary Number System** is a **base-2** numeral system that uses only **two digits**:

0 and 1

□ Why Binary?

- Computers and digital devices use **binary logic**.
- Every value in memory (data, text, image, etc.) is ultimately stored as a series of **0s and 1s**.
- It is **efficient** and **easy** to represent with electronic signals (ON = 1, OFF = 0).

□ Basic Terminology

Term	Description
Bit	A single binary digit (0 or 1)
Nibble	4 bits
Byte	8 bits
LSB	Least Significant Bit (rightmost bit)
MSB	Most Significant Bit (leftmost bit)

□ Binary Positional Value System

Binary is a **positional number system**, just like decimal. Each digit has a **place value** which is a power of 2.

For a binary number 1011:

```
 1   0   1   1
(8) (4) (2) (1)   ← 2^3, 2^2, 2^1, 2^0
```

= 1×8 + 0×4 + 1×2 + 1×1 = 11 (in decimal)

 Conversions Between Number Systems

 Binary → Decimal

Method: Multiply each bit with 2 raised to its position (right to left) and sum them.

Example:

Binary: 1101
Decimal = $1\times2^3 + 1\times2^2 + 0\times2^1 + 1\times2^0$
= 8 + 4 + 0 + 1 = 13

 Decimal → Binary

Method: Divide the number by 2 repeatedly and write remainders in reverse order.

Example: Convert 13 to binary

Division	Quotient	Remainder
13 ÷ 2	6	1
6 ÷ 2	3	0
3 ÷ 2	1	1
1 ÷ 2	0	1

Binary = 1101

 Binary → Octal

Group binary digits in sets of 3 (right to left) and convert each to octal.

Example:

Binary: 110101

Grouped: 110 101
Octal: 6 5 → 65 (base 8)

 Binary → Hexadecimal

Group digits in sets of 4 (right to left) and convert each to hexadecimal.

Example:

Binary: 11010110

Grouped: 1101 0110
Hex: D 6 → D6 (base 16)

Binary Arithmetic

1. Binary Addition Rules

A	B	A + B	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Example:

```
  1101
+ 1011
------
11000
```

---

2. Binary Subtraction Rules

A	B	A − B	Borrow
0	0	0	0

Name: Prathamesh Arvind Jadhav

A	B	A − B	Borrow
1	0	1	0
1	1	0	0
0	1	1	1

Example:

```
  1010
− 0011
-------
  0111
```

 Binary Number Representation

Type	Example	Range (for 8 bits)
Unsigned Binary	00001101	0 to 255
Signed Binary (2’s)	11110011	-128 to +127 (2’s complement)
Binary Coded Decimal	0001 0011 (for 13)	0–9 per 4 bits

Real-Life Applications

- Memory addressing
- Networking (IP addressing)
- Machine-level programming
- Digital circuit design
- Encryption & data storage

Bitwise Operators in C++

Bitwise operators are used to perform operations on **bits** (binary representations).

Operator	Description	Example (a = 5, b = 3)	Result (in binary)
&	AND	a & b (5 & 3)	0101 & 0011 = 0001 (1)
`	`	OR	`a
^	XOR	a ^ b (5 ^ 3)	0101 ^ 0011 = 0110 (6)

Name: Prathamesh Arvind Jadhav

Operator	Description	Example (a = 5, b = 3)	Result (in binary)
~	NOT (1's complement)	~a (~5)	~0101 = 1010 (in 2's complement = -6)
<<	Left Shift	a << 1	0101 << 1 = 1010 (10)
>>	Right Shift	a >> 1	0101 >> 1 = 0010 (2)

 Use cases: Encryption, Graphics, Low-level programming.

---

2. Data Type Modifiers in C++

Used to **change the size** or **sign** of the data types.

Modifier	Purpose	Example
signed	Can hold both +ve and -ve	signed int a = -10;
unsigned	Only +ve (doubles max value)	unsigned int a = 10;
short	Smaller range than int	short int a = 100;
long	Larger range than int	long int a = 100000;
long long	Very large integer	long long int a = 1e18;

 unsigned int x = -1; → Causes wraparound.

---

3. Type Conversion (Type Casting)

- **Implicit**: Automatically done by compiler.

  ```
  int a = 5;
  float b = a; // int to float automatically
  ```

- **Explicit**: Manual casting.

  ```
  float a = 5.5;
  int b = (int)a; // truncates decimal
  ```

Name: Prathamesh Arvind Jadhav

□ 4. Storage Classes in C++

Define scope, lifetime, and linkage of variables.

Storage Class	Scope	Lifetime	Default Value	Keyword
auto	Local	Function block	Garbage	auto
register	Local (in CPU)	Fast access	Garbage	register
static	Local	Entire program	Zero	static
extern	Global	Entire program	Zero	extern

□ 5. Constants and Macros

- **const**: Constant variable.

  const int x = 10;

- **#define**: Preprocessor macro.

  #define PI 3.14159

DAY-6:

Arrays:

What is an Array?

An **array** is a **collection of elements** of the **same data type** stored in **contiguous memory locations** and accessed using **indexing**.

Syntax:

data_type array_name[size];

□ Types of Arrays

1. **One-Dimensional Array**
2. **Two-Dimensional Array (Matrix)**
3. **Multi-Dimensional Array**

Name: Prathamesh Arvind Jadhav

□ 1. One-Dimensional Array

Declaration:

```
int arr[5];  // Uninitialized array of 5 integers
```

Initialization:

```
int arr[5] = {1, 2, 3, 4, 5};
```

Accessing Elements:

```
cout << arr[2];  // Outputs 3
```

Input and Output:

```
for (int i = 0; i < 5; i++)
   cin >> arr[i];

for (int i = 0; i < 5; i++)
   cout << arr[i] << " ";
```

□ 2. Two-Dimensional Array

A 2D array is like a **matrix**: rows × columns.

Declaration:

```
int mat[3][4];  // 3 rows, 4 columns
```

Initialization:

```
int mat[2][3] = {{1, 2, 3}, {4, 5, 6}};
```

Access:

```
cout << mat[1][2];  // Outputs 6
```

Input/Output:

```
for (int i = 0; i < 2; i++)
   for (int j = 0; j < 3; j++)
      cin >> mat[i][j];
```

□ 3. Multi-Dimensional Arrays

Name: Prathamesh Arvind Jadhav

- 3D arrays example:

int cube[2][3][4];

- Think of it as: 2 layers of 3×4 matrices.

□ Array Memory Allocation

- Arrays are stored in **contiguous memory**.
- Indexing starts at **0**.
- arr[i] refers to *(base_address + i × size_of_datatype)*

□ Common Operations

Operation	Description
Traversal	Loop through elements
Insertion	Insert at index (manual shifting required)
Deletion	Remove element by shifting
Searching	Find element using linear/binary search
Sorting	Bubble, Selection, Insertion, Merge, Quick etc.

□ Limitations of Arrays

- **Fixed size**: Cannot resize after declaration.
- **No bounds checking**: Accessing out-of-bounds is undefined behavior.
- **Inflexible insertion/deletion**

Name: Prathamesh Arvind Jadhav

□ Advantages of Arrays

- Fast access via index
- Efficient in memory for homogeneous data
- Easy to implement static data structures

□ Array Example Program

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

int main() {
   int n;
   cout << "Enter number of elements: ";
   cin >> n;

   int arr[n];
   cout << "Enter elements:\n";
   for (int i = 0; i < n; i++)
      cin >> arr[i];

   cout << "You entered: ";
   for (int i = 0; i < n; i++)
      cout << arr[i] << " ";

   return 0;
}
```

□ Array with Functions

```
void displayArray(int arr[], int size) {
   for (int i = 0; i < size; i++)
      cout << arr[i] << " ";
}
```

Name: Prathamesh Arvind Jadhav

Vectors:

What is a Vector?

A **vector** in C++ is a part of the **STL (Standard Template Library)** that acts as a **dynamic array** — it can **resize itself automatically** when elements are added or removed.

Vectors are more powerful and flexible than traditional arrays.

□ Header File

#include <vector>

You can also use:

using namespace std;

□ Declaration and Initialization

```
vector<int> v;                  // Empty vector of int
vector<int> v(5);                // Vector with 5 default-initialized elements (0)
vector<int> v(5, 10);            // Vector with 5 elements, each initialized to 10
vector<int> v2 = {1, 2, 3, 4};   // Initialization using list
```

□ Common Member Functions

Function	Description
v.size()	Returns number of elements
v.push_back(x)	Adds element x at the end
v.pop_back()	Removes last element

Name: Prathamesh Arvind Jadhav

Function	Description
v.front()	Returns first element
v.back()	Returns last element
v[i] or v.at(i)	Access element at index i
v.clear()	Removes all elements
v.empty()	Returns true if vector is empty
v.insert(pos, val)	Inserts val at specified position
v.erase(pos)	Erases element at position
v.begin() / v.end()	Returns iterator to start/end of vector
v.resize(n)	Resizes vector to n elements
v.swap(v2)	Swaps contents with another vector
v.assign(n, val)	Assigns n copies of val to vector

 Example: Basic Vector Usage

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

int main() {
    vector<int> v;

    v.push_back(10);
    v.push_back(20);
    v.push_back(30);
```

```
    for (int i = 0; i < v.size(); i++)
        cout << v[i] << " ";  // Output: 10 20 30

    v.pop_back();  // Removes 30

    cout << "\nFront: " << v.front();  // 10
    cout << "\nBack: " << v.back();    // 20

    return 0;
}
```

Accessing Elements

- Using [] operator:

```
cout << v[1];  // Fast but unsafe (no bounds checking)
```

- Using at():

```
cout << v.at(1);  // Safe (throws out_of_range if invalid)
```

Traversing Vectors

➢ ***Using index-based loop:***

```
for (int i = 0; i < v.size(); i++)
    cout << v[i];
```

➢ ***Using auto and range-based loop:***

```
for (auto x : v)
    cout << x << " ";
```

➢ ***Using iterator:***

```
for (vector<int>::iterator it = v.begin(); it != v.end(); ++it)
    cout << *it << " ";
```

Sorting a Vector

Name: Prathamesh Arvind Jadhav

```
#include <algorithm>
sort(v.begin(), v.end());          // Ascending
sort(v.rbegin(), v.rend());        // Descending
```

□ Useful Vector Tricks

Copying a vector:

```
vector<int> v2 = v;  // Shallow copy
```

Removing duplicate elements:

```
sort(v.begin(), v.end());
v.erase(unique(v.begin(), v.end()), v.end());
```

Find an element:

```
if (find(v.begin(), v.end(), 20) != v.end())
   cout << "Found";
```

□ Vector of Pairs

```
vector<pair<int, int>> vp;
vp.push_back({1, 2});
vp.push_back(make_pair(3, 4));

for (auto p : vp)
   cout << p.first << " " << p.second << endl;
```

□ 2D Vectors (Vector of Vectors)

```
vector<vector<int>> matrix(3, vector<int>(4, 0));  // 3x4 matrix with 0s

matrix[1][2] = 5;
```

Input/output in 2D vector:

```
for (int i = 0; i < 3; i++)
   for (int j = 0; j < 4; j++)
      cin >> matrix[i][j];

for (auto row : matrix) {
```

```
    for (auto val : row)
        cout << val << " ";
    cout << "\n";
}
```

 Advantages of Vectors over Arrays

Feature	Vector	Array
Size	Dynamic	Fixed
Bounds Check	at() supports it	Not available
Functions	Rich STL support	None
Easy to insert	push_back(), insert()	Manual shifting
Safe & flexible	Yes	No

 When NOT to Use Vectors

- When fixed-size arrays are sufficient and performance is extremely critical (e.g., embedded systems).
- When you need raw pointers and memory control.

DAY-7:

Problem: Majority Element

 Problem Statement

Given an array of integers, **find the element that appears more than n / 2 times** (called the **majority element**). You can assume it always exists.

Name: Prathamesh Arvind Jadhav

Example:

Input: nums = [2,2,1,1,1,2,2]

Output: 2

Explanation: 2 appears 4 times out of 7, which is more than 7/2 = 3.5

Approach 1: Brute Force

Logic:

- Loop through every element.
- For each element, count how many times it appears in the array.
- If the count is greater than n/2, return that element.

Time Complexity: $O(n^2)$

Space Complexity: O(1)

Code with Explanation:

```
int majorityElement(vector<int>& nums) {
    int n = nums.size();

    // Outer loop: pick each element
    for (int i = 0; i < n; i++) {
        int count = 0;

        // Inner loop: count occurrences of nums[i]
        for (int j = 0; j < n; j++) {
            if (nums[j] == nums[i]) {
                count++;
```

```
            }
        }


        // If count exceeds n/2, we found the majority
        if (count > n / 2) {
            return nums[i];
        }
    }


    return -1; // This will never be reached since majority is guaranteed
}
```

Explanation:

- We compare each element with all others.
- If any element occurs more than n/2 times, we return

□ Approach 2: Better (Hash Map / Frequency Count)

Logic:

- Use a **hash map** to store frequency of each number.
- Traverse the array and increment frequency in map.
- If any element's frequency exceeds n/2, return it.

Time: O(n)

Space: O(n)

Code with Explanation:

```
#include <unordered_map>
```

Name: Prathamesh Arvind Jadhav

```
int majorityElement(vector<int>& nums) {
    unordered_map<int, int> freq;  // Map to store frequency
    int n = nums.size();

    // Count frequency of each number
    for (int num : nums) {
        freq[num]++;  // Increase count
        if (freq[num] > n / 2) {
            return num;  // As soon as one exceeds n/2, return it
        }
    }

    return -1; // This won't be executed (guaranteed majority)
}
```

Key Concepts:

- unordered_map stores each number with its frequency.
- We check while inserting if it exceeds n/2.

Dry Run:

nums = [2, 2, 1, 1, 1, 2, 2]

Map:

- 2 → 1

- 2 → 2

Name: Prathamesh Arvind Jadhav

- 1 → 1

- 1 → 2

- 1 → 3

- 2 → 3

- 2 → 4 → return 2

□ Approach 3: Optimal (Moore's Voting Algorithm)

Logic:

- This is a clever **voting algorithm**.
- Maintain:
  - candidate: potential majority
  - count: how confident we are about that candidate
- Rules:
  - If count == 0, choose new candidate.
  - If current element == candidate → count++
  - Else → count--

Time: O(n)

Space: O(1) (most optimal)

Code with Explanation:

```
int majorityElement(vector<int>& nums) {
    int count = 0;       // Keeps track of "votes"
    int candidate = 0;   // Stores current majority candidate

    for (int num : nums) {
        if (count == 0) {
```

```
            candidate = num;  // Choose new candidate
        }

        // If same as candidate, increase vote
        if (num == candidate) {
            count++;
        } else {
            count--;  // Else reduce vote
        }
    }

    return candidate;
}
```

Explanation:

Let's dry run nums = [2, 2, 1, 1, 1, 2, 2]:

Step	num	count	candidate	Action
1	2	0 → 1	2	New candidate 2
2	2	2	2	Same → count++
3	1	1	2	Not same → count--
4	1	0	2	Not same → count--
5	1	1	1	count==0 → new candidate = 1

Name: Prathamesh Arvind Jadhav

Step	num	count	candidate	Action
6	2	0	1	Not same → count--
7	2	1	2	count==0 → new candidate = 2

Final candidate is 2

Summary Table

Approach	Time	Space	Logic
Brute Force	$O(n^2)$	O(1)	Count each element with nested loop
Hash Map	O(n)	O(n)	Use map to store and check frequency
Moore's Voting	O(n)	O(1)	Cancel out non-majority elements cleverly

Problem: Maximum Subarray

 Problem Statement
Given an integer array nums, find the subarray with the largest sum, and return its sum.

Example:
Input: nums = [-2,1,-3,4,-1,2,1,-5,4]
Output: 6
Explanation: The subarray [4, -1, 2, 1] has the largest sum 6.

 Approach 1: Brute Force
Logic:

- Generate all possible subarrays.
- Calculate the sum of each subarray.
- Keep track of the maximum sum encountered.

Name: Prathamesh Arvind Jadhav

Time Complexity: $O(n^2)$ (because of nested loops to check all subarrays)
Space Complexity: O(1) (no extra space except variables)

Code with Explanation:

```
int maxSubArray(vector<int>& nums) {
    int n = nums.size();
    int max_sum = nums[0];

    // Outer loop picks starting point of subarray
    for (int i = 0; i < n; i++) {
        int current_sum = 0;

        // Inner loop picks ending point of subarray
        for (int j = i; j < n; j++) {
            current_sum += nums[j];  // Sum elements from i to j

            if (current_sum > max_sum) {
                max_sum = current_sum;  // Update max if current sum is larger
            }
        }
    }
    return max_sum;
}
```

Explanation:

- For every starting index i, we calculate sums of all subarrays starting at i.
- Track the maximum sum found so far.

 Approach 2: Optimal (Kadane's Algorithm)
Logic:

- Traverse the array, keeping track of the current subarray sum (current_sum).
- If current_sum becomes negative, reset it to zero (start fresh from next element).
- Keep track of the maximum sum found (max_sum) during the traversal.

Name: Prathamesh Arvind Jadhav

Time Complexity: O(n) (single pass)
Space Complexity: O(1) (only variables used)

Code with Explanation:

```
int maxSubArray(vector<int>& nums) {
    int max_sum = nums[0];
    int current_sum = 0;

    for (int num : nums) {
        current_sum += num;  // Add current element to current sum

        if (current_sum > max_sum) {
            max_sum = current_sum;  // Update max_sum if current_sum is greater
        }

        if (current_sum < 0) {
            current_sum = 0;  // Reset current_sum if it becomes negative
        }
    }
    return max_sum;
}
```

Explanation:

- Add elements one by one to current_sum.
- If at any point current_sum is less than zero, it means starting a new subarray from next element will yield better results, so reset current_sum.
- Keep updating max_sum with the highest sum encountered.

 Summary Table

Approach	Time Complexity	Space Complexity	Logic
Brute Force	$O(n^2)$	O(1)	Check all subarrays and calculate sum
Kadane's Algorithm	O(n)	O(1)	Dynamic sum tracking, reset if negative

Name: Prathamesh Arvind Jadhav

Problem: Two Sum

□ Problem Summary

Given:

- An array of integers nums
- An integer target

Goal:
Find two **distinct indices** i and j in nums such that:
nums[i] + nums[j] == target
Return [i, j] (any order is fine).

Constraint:

- Only one valid pair exists.
- You cannot use the same element twice.

□ Brute-Force Approach

Idea:

Check **every possible pair** of numbers in the array to see if they add up to the target.

□ Time Complexity:

- **$O(n^2)$** – two nested loops.

□ Space Complexity:

- **O(1)** – no extra space used.

CODE:

```
class Solution {
```

Name: Prathamesh Arvind Jadhav

```
public:
    vector<int> twoSum(vector<int>& nums, int target) {
        int n = nums.size();


        for(int i = 0; i < n; ++i) {
            for(int j = i + 1; j < n; ++j) {
                if(nums[i] + nums[j] == target) {
                    return {i, j};  // Return as a vector of indices
                }
            }
        }

        return {};  // Return empty vector if no solution found
    }
};
```

 Example Dry Run:

Input: nums = [2,7,11,15], target = 9

- i=0, j=1 → nums[0] + nums[1] = 2 + 7 = 9
  Return [0,1]

 Optimal Approach (Using HashMap)

Idea:

- Use a **dictionary (hash map)** to store the **complement** (target - current number) and index.

Name: Prathamesh Arvind Jadhav

- While iterating, check if the **current number** is already in the hash map → if yes, you've found the pair!

 Time Complexity:

- **O(n)** – single pass through the list.

 Space Complexity:

- **O(n)** – for storing elements in hash map.

CODE:

```
class Solution {
public:
    vector<int> twoSum(vector<int>& nums, int target) {
        unordered_map<int, int> numMap;  // key: number, value: index

        for(int i = 0; i < nums.size(); ++i) {
            int complement = target - nums[i];

            // Check if complement already exists in the map
            if(numMap.find(complement) != numMap.end()) {
                return {numMap[complement], i};
            }

            // Store current number with its index
            numMap[nums[i]] = i;
        }
```

```
        return {};  // Return empty vector if no solution found (per constraints, won't happen)
    }
};
```

 Example:

Input: nums = [2, 7, 11, 15], target = 9
Output: [0, 1]

In the optimal approach:

- 2 is stored at index 0
- On seeing 7, we look for 9 - 7 = 2 → found in map → return {0, 1}

 Summary

Approach	Time Complexity	Space Complexity	Suitable For
Brute Force	$O(n^2)$	O(1)	Small input sizes
Optimal (Hash)	O(n)	O(n)	Large input sizes

DAY-8:

Time and Space Complexity

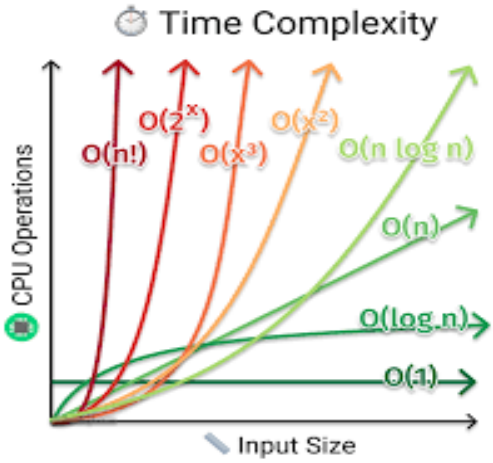
---

1. Introduction to Complexity Analysis

In **competitive programming**, understanding how fast and memory-efficient your code is, becomes **critical**. Time and Space complexity helps you **predict performance**.

Name: Prathamesh Arvind Jadhav

- **Time Complexity**: Measures how the runtime of an algorithm increases with input size n.
- **Space Complexity**: Measures how much **extra memory** an algorithm uses as input size n increases.



---

2. Time Complexity

□ 2.1 Definition

Time Complexity is the **amount of time** taken by an algorithm to run, as a function of the length of the input.

□ 2.2 Notations

- **Big O (O)**: Worst-case
- **Big Omega (Ω)**: Best-case
- **Big Theta (Θ)**: Average-case (tight bound)

For CP, we mostly focus on **Big O**.

□ 2.3 Common Time Complexities

Time Complexity	Description	Example Algorithm
O(1)	Constant Time	Accessing array element

Name: Prathamesh Arvind Jadhav

Time Complexity	Description	Example Algorithm
O(log n)	Logarithmic	Binary Search
O(n)	Linear	Linear Search
O(n log n)	Linearithmic	Merge Sort, Heap Sort
$O(n^2)$	Quadratic	Bubble Sort, Insertion Sort
$O(n^3)$	Cubic	Matrix Multiplication
$O(2^n)$	Exponential	Recursion with 2 branches
O(n!)	Factorial	Backtracking (e.g., permutations)

□ 2.4 Time Complexity of Loops

Single Loop

```
for(int i = 0; i < n; i++) {
   // constant work
}
```

Time Complexity: O(n)

Nested Loop

```
for(int i = 0; i < n; i++) {
   for(int j = 0; j < n; j++) {
      // constant work
   }
}
```

Time Complexity: $O(n^2)$

Logarithmic Loop

```
for(int i = 1; i < n; i *= 2) {
   // constant work
}
```

Name: Prathamesh Arvind Jadhav

Time Complexity: O(log n)

□ 2.5 Time Complexity of Recursion

Use **Recurrence Relation**.

Example:

```
int fib(int n) {
   if (n <= 1) return n;
   return fib(n-1) + fib(n-2);
}
```

- Recurrence: $T(n) = T(n-1) + T(n-2)$
- Time Complexity: $O(2^n)$

□ 2.6 Estimating Time Limit in Competitive Coding

□ **Rule of Thumb**:
1 second = about **10^8 operations**

Input size n	**Acceptable Complexity**
≤ 10	O(n!), $O(2^n)$
≤ 100	$O(n^3)$
≤ 1,000	$O(n^2)$
≤ 100,000	O(n log n)
≤ 10^7	O(n)
≤ 10^8	O(1), O(log n)

Name: Prathamesh Arvind Jadhav

3. Space Complexity

□ 3.1 Definition

Space complexity is the **total memory space required** by an algorithm with respect to input size n.

Includes:

- Input space
- Auxiliary space (temporary variables, recursion stack)

□ 3.2 Common Space Complexities

Space Complexity	Example
O(1)	In-place algorithms
O(n)	Arrays, Hash Maps
O(log n)	Binary Search recursion
$O(n^2)$	2D arrays

□ 3.3 Space in Recursion

Each recursive call adds a **stack frame**.

Example:

```
void recurse(int n) {
   if(n == 0) return;
   recurse(n - 1);
}
```

- **Time Complexity**: O(n)
- **Space Complexity**: O(n) (due to call stack)

Name: Prathamesh Arvind Jadhav

□ 4. Tips for Competitive Programming

1. Always consider **worst-case time complexity**.
2. Avoid nested loops for large inputs.
3. Use efficient data structures (sets, maps, heaps).
4. Know how to estimate limits using **10^8 operations/sec** rule.
5. Use **iterative solutions** when recursion depth is large.
6. Optimize space using **in-place algorithms** where possible.
7. Understand sorting complexity — avoid $O(n^2)$ sorts unless n is small.

Problem : Best Time to Buy and Sell Stock

Problem Summary

You are given an array prices[] where prices[i] is the stock price on day i.
You must **buy before you sell**.
Goal: **Maximize profit** = prices[j] - prices[i] where j > i.

If no profit is possible, return 0.

---

1. Brute Force Approach

□ **Logic:**

- Try **all pairs** of days.
- For each i, try selling on every future day j.
- Calculate profit: prices[j] - prices[i].
- Track the **maximum profit**.

□ **Time Complexity: $O(n^2)$**

□ **Space Complexity: O(1)**

□ **Drawback:**

- Not feasible for large n (up to 10^5), leads to **TLE**.

Name: Prathamesh Arvind Jadhav

C++ Code:

```
class Solution {
public:
    int maxProfit(vector<int>& prices) {
        int n = prices.size();
        int maxProfit = 0;

        for (int i = 0; i < n; i++) {
            for (int j = i + 1; j < n; j++) {
                int profit = prices[j] - prices[i];
                maxProfit = max(maxProfit, profit);
            }
        }
        return maxProfit;
    }
};
```

---

2. Better Approach (Prefix Minimum Array)

 Logic:

- Keep track of **minimum price till index i** using a prefix min.
- At each index i, calculate:
  profit = prices[i] - minPriceSoFar
- Track max profit.

 Time Complexity: O(n)

 Space Complexity: O(n) (due to prefix array)

C++ Code:

```
class Solution {
public:
    int maxProfit(vector<int>& prices) {
        int n = prices.size();
        vector<int> minPrice(n);
        minPrice[0] = prices[0];
```

```
        for (int i = 1; i < n; i++) {
            minPrice[i] = min(minPrice[i - 1], prices[i]);
        }

        int maxProfit = 0;
        for (int i = 1; i < n; i++) {
            maxProfit = max(maxProfit, prices[i] - minPrice[i - 1]);
        }

        return maxProfit;
    }
};
```

---

3. Optimal Approach (Single Pass)

 Logic:

- Track the **minimum price so far** while iterating.
- At each day, calculate potential profit = current price - min so far.
- Update **max profit**.

 Time Complexity: O(n)

 Space Complexity: O(1)

Intuition:

No need to store prefix; just **track min and max profit on the fly**.

C++ Code:

```
class Solution {
public:
    int maxProfit(vector<int>& prices) {
        int minPrice = INT_MAX;
        int maxProfit = 0;

        for (int price : prices) {
            if (price < minPrice) {
                minPrice = price; // Buy at lower price
```

```
        } else {
            maxProfit = max(maxProfit, price - minPrice); // Sell at higher price
        }
    }

    return maxProfit;
  }
};
```

 Summary Table

Approach	Time	Space	Status
Brute Force	$O(n^2)$	O(1)	TLE
Better (Prefix)	O(n)	O(n)	Accept
Optimal (1-Pass)	O(n)	O(1)	Best

Problem: Pow(x, n)

Problem Statement

Implement pow(x, n), which calculates x^n.

- You may assume that $x \neq 0$ or $n > 0$.
- You must handle **negative exponents**.
- Range of n: from -2^{31} to $2^{31} - 1$.

---

1. Brute Force Approach

 Logic:

- Multiply x by itself n times.
- If n is negative, compute 1 / pow(x, -n).

Issues:

- Takes **O(n)** time — not acceptable for large n.

- May cause **TLE** when n is near $\pm 2^{31}$.

 Time Complexity: O(n)

 Space Complexity: O(1)

C++ Code:

```
class Solution {
public:
   double myPow(double x, int n) {
      long long power = abs((long long)n);
      double result = 1.0;

      for (long long i = 0; i < power; i++) {
         result *= x;
      }

      return n < 0 ? 1.0 / result : result;
   }
};
```

---

2. Better Approach (Iterative Fast Power)

 Logic:

- Use **binary exponentiation**:
  - Square the base and halve the power.
  - If n is odd, multiply the result once by x.

 Time Complexity: O(log n)

 Space Complexity: O(1)

C++ Code:

```
class Solution {
public:
   double myPow(double x, int n) {
      long long N = n;
```

```
        if (N < 0) {
            x = 1 / x;
            N = -N;
        }

        double result = 1.0;
        while (N > 0) {
            if (N % 2 == 1)
                result *= x;
            x *= x;
            N /= 2;
        }

        return result;
    }
};
```

 Summary Table

Approach	Time	Space	Status
Brute Force	O(n)	O(1)	TLE
Iterative Fast Power	O(log n)	O(1)	Best

DAY-9:

Problem: Container With Most Water

1. Problem Summary

You are given an array height[] of length n, where each element represents the height of a vertical line drawn at that index (like coordinates on the x-axis).

Goal:

Find **two lines** that, along with the x-axis, form a container such that the container holds the **most water**.

Name: Prathamesh Arvind Jadhav

□ Formula for Area:

Area between lines at index i and j is:

Area = min(height[i], height[j]) * (j - i)

---

2. Intuition

To maximize the area:

- You want a **large width** (i.e., distance between lines)
- And a **large minimum height** (because water is limited by the shorter wall)

But increasing width might decrease height and vice versa.
We need a strategy to balance both to find the best pair.

---

3. Logic & Approaches

Brute Force:

- Check **all pairs of lines**
- Compute area for each pair
- Return the **maximum** found

□ Two Pointer (Optimal):

- Start with two pointers: one at the beginning and one at the end
- Calculate area between the two lines
- Move the pointer pointing to the **shorter** line inward (in hopes of finding a taller one to increase area)

Why move the shorter one? Because moving the taller won't help if the short one is still limiting the height.

□ 4. Complexity

Approach	Time Complexity	Space Complexity
Brute Force	$O(n^2)$	O(1)
Two Pointers	O(n)	O(1)

---

5. Brute Force Solution

```
class Solution {
public:
    int maxArea(vector<int>& height) {
        int max_area = 0;
        int n = height.size();

        for (int i = 0; i < n; ++i) {
            for (int j = i + 1; j < n; ++j) {
                int h = min(height[i], height[j]);
                int w = j - i;
                int area = h * w;
                max_area = max(max_area, area);
            }
        }

        return max_area;
    }
};
```

 Explanation:

- Nested loop to try every possible pair (i, j)
- Compute area and update max_area

---

 6. Optimal Two-Pointer Solution

```
class Solution {
public:
    int maxArea(vector<int>& height) {
        int left = 0;
        int right = height.size() - 1;
```

```
        int max_area = 0;

        while (left < right) {
            int h = min(height[left], height[right]);
            int w = right - left;
            int area = h * w;
            max_area = max(max_area, area);

            // Move the pointer pointing to the shorter line
            if (height[left] < height[right]) {
                ++left;
            } else {
                --right;
            }
        }

        return max_area;
    }
};
```

 Explanation:

- Start with the widest container
- Gradually narrow it down while hoping to find taller walls
- Efficiently finds the max area in linear time

 Example Dry Run:

Input: height = [1,8,6,2,5,4,8,3,7]

Steps:

- left = 0, right = 8 → min(1,7) * 8 = 8
- Move left → left = 1, right = 8 → min(8,7) * 7 = 49
- Continue moving...

Final output: 49

DAY-10:

Name: Prathamesh Arvind Jadhav

Problem: Product of Array Except Self

Problem Restatement

Given an integer array nums, return an array answer such that answer[i] is equal to the product of all elements of nums except nums[i].

Constraints:

- No division allowed.
- Must run in O(n) time.
- Optimize for O(1) extra space (excluding the output array).

---

1. Brute Force Approach

Logic:

- For each element nums[i], multiply all elements except nums[i].
- This requires nested loops:
  - Outer loop picks element i.
  - Inner loop multiplies all elements except i.
- Time complexity: $O(n^2)$.
- Space complexity: O(1) ignoring output array.

Code (Brute Force):

```
class Solution {
public:
   vector<int> productExceptSelf(vector<int>& nums) {
      int n = nums.size();
      vector<int> answer(n, 1);

      for (int i = 0; i < n; i++) {
         int product = 1;
         for (int j = 0; j < n; j++) {
            if (i != j) {
               product *= nums[j];
```

```
            }
        }
        answer[i] = product;
    }

    return answer;
  }
};
```

---

2. Optimal Approach

Logic:

- We want to avoid division and achieve O(n) time.
- Idea: For each element at index i, product except self = product of all elements to the left of i * product of all elements to the right of i.

We can precompute:

- leftProducts[i] = product of all elements to the left of index i.
- rightProducts[i] = product of all elements to the right of index i.

Then:

answer[i] = leftProducts[i] * rightProducts[i]

Space optimization:

- Instead of storing two extra arrays, store the left product directly in the output array.
- Then iterate from the right to update the answer by multiplying with the running right product.

Steps:

1. Initialize answer array with answer[0] = 1 (no elements to left of 0).
2. Fill answer[i] with the product of all elements to the left of i.
3. Maintain a variable R to keep product of elements to the right while traversing from the right.
4. Multiply answer[i] with R and update R with nums[i].

Name: Prathamesh Arvind Jadhav

Time complexity: O(n)
Space complexity: O(1) (output array doesn't count as extra space)

Code (Optimal Approach):

```
class Solution {
public:
   vector<int> productExceptSelf(vector<int>& nums) {
      int n = nums.size();
      vector<int> answer(n, 1);

      // Step 1: Fill with left products
      for (int i = 1; i < n; i++) {
         answer[i] = answer[i - 1] * nums[i - 1];
      }

      // Step 2: Multiply with right products on the fly
      int R = 1; // Running product of elements to the right
      for (int i = n - 1; i >= 0; i--) {
         answer[i] = answer[i] * R;
         R *= nums[i];
      }

      return answer;
   }
};
```

Example Walkthrough:

For input [1, 2, 3, 4]

- Left products (stored in answer):
  - i=0 → 1
  - i=1 → 1 * nums[0] = 1 * 1 = 1
  - i=2 → 1 * 2 = 2
  - i=3 → 2 * 3 = 6
    answer after left pass = [1, 1, 2, 6]
- Right product R initialized to 1:

- i=3 → answer[3] = 6 * 1 = 6, R = 1 * 4 = 4
- i=2 → answer[2] = 2 * 4 = 8, R = 4 * 3 = 12
- i=1 → answer[1] = 1 * 12 = 12, R = 12 * 2 = 24
- i=0 → answer[0] = 1 * 24 = 24, R = 24 * 1 = 24

Final answer = [24, 12, 8, 6]

DAY-11:

□ Introduction to STL

The **Standard Template Library (STL)** is a collection of **template classes and functions** that provide **data structures** and **algorithms** for C++ programming. STL is crucial in competitive programming due to its efficiency and ease of use.

□ Components of STL

1. **Containers** – Store collections of objects.
2. **Iterators** – Point to memory addresses and traverse containers.
3. **Algorithms** – Provide functions for operations like searching, sorting, etc.
4. **Functors** – Objects that can be used as functions or function pointers.

1.Containers in STL – Full Detailed Notes

□ Sequence Containers (Linear Structures)

These maintain the ordering of elements as they were inserted. They're similar to arrays and linked lists.

---

1. vector

Name: Prathamesh Arvind Jadhav

Dynamic array with automatic resizing.

- **Random Access**: v[i] is O(1).
- **Insertion/Deletion at end**: O(1) amortized.
- **Insertion in middle/start**: O(n), as elements shift.

Common Operations:

```
vector<int> v = {1, 2, 3};
v.push_back(4);       // Adds 4 to the end
v.pop_back();         // Removes last element
v[0];                 // Access first element
v.size();             // Returns size
v.clear();            // Empties vector
v.insert(v.begin(), 5); // Insert 5 at beginning
```

Useful STL Functions with Vector:

```
sort(v.begin(), v.end());            // Sorts ascending
reverse(v.begin(), v.end());         // Reverses vector
binary_search(v.begin(), v.end(), x); // true/false
lower_bound(v.begin(), v.end(), x);   // first index ≥ x
upper_bound(v.begin(), v.end(), x);   // first index > x
```

Best use case: When you need dynamic array-like structure with random access and frequent appends.

---

2. deque (Double Ended Queue)

A **dynamic array** with fast **insertion/deletion** from both ends.

- No random access optimizations as in vector.
- Efficient for sliding window problems.

Example:

```
deque<int> dq;
dq.push_back(1);
dq.push_front(2);     // [2, 1]
dq.pop_back();
dq.pop_front();
```

Best use case: For problems like **sliding window maximum/minimum**, **monotonic queues**.

---

3. list (Doubly Linked List)

- Each node has a pointer to both previous and next nodes.
- No random access (list[i] not allowed).
- Insertion/deletion at any position in O(1) time **with iterator**.

Example:

```
list<int> lst = {1, 2, 3};
lst.push_front(0);     // [0, 1, 2, 3]
lst.push_back(4);      // [0, 1, 2, 3, 4]
auto it = lst.begin();
advance(it, 2);        // Move to 2nd index
lst.insert(it, 99);    // Insert 99 at 2nd index
```

Best use case: When **frequent insert/delete** in the middle of the container is needed.

---

4. forward_list (C++11) – Singly Linked List

- Less memory usage than list.
- Only forward traversal possible.

Example:

```
forward_list<int> fl = {1, 2, 3};
fl.push_front(0);     // [0, 1, 2, 3]
fl.pop_front();
```

Best use case: When you need a light-weight linked list with only forward traversal.

---

5. array (C++11)

- A wrapper around C-style fixed-size arrays with STL interface.
- Size must be known at compile time.

Example:

```
array<int, 5> arr = {1, 2, 3, 4, 5};
arr[0];        // Access elements like normal array
arr.size();    // Get size
```

Best use case: When size is fixed and performance is critical.

 Associative Containers (Sorted)

These store **key-based** values in **sorted order**. All operations are O(log n), implemented using **Red-Black Trees**.

---

1. set

- Stores unique keys.
- Automatically sorted.

Example:

```
set<int> s;
s.insert(3);
s.insert(1);
s.insert(1); // Won't insert again
// s = {1, 3}
```

 Search:

```
if (s.find(3) != s.end()) { /* found */ }
```

Best use case: Fast existence check, sorted data with unique elements.

---

2. multiset

- Allows **duplicate keys**.

- Also keeps them sorted.

Example:

```
multiset<int> ms = {1, 2, 2, 3};
ms.insert(2);
ms.erase(ms.find(2)); // Deletes one occurrence
```

Best use case: Counting elements with duplicate values in sorted order.

---

3. map

- Stores **key-value** pairs with **unique keys**.
- Automatically sorted by key.

Example:

```
map<string, int> m;
m["apple"] = 3;
m["banana"] = 5;
```

 Access:

```
m.count("apple"); // 1 if exists, 0 otherwise
m.find("banana"); // iterator to key-value pair
```

Best use case: Frequency maps, data with unique keys.

---

4. multimap

- Allows **duplicate keys**.
- Useful when multiple values are associated with one key.

Example:

```
multimap<string, int> mm;
mm.insert({"apple", 1});
mm.insert({"apple", 2});
```

Best use case: When a key maps to multiple values (like graph adjacency list).

Name: Prathamesh Arvind Jadhav

□ Unordered Associative Containers (Hash Tables)

- Provide **average O(1)** time for insertion, deletion, and lookup.
- **No ordering guaranteed**.
- Internally use **hash tables**.

Types:

- unordered_set
- unordered_map
- unordered_multiset
- unordered_multimap

---

unordered_map Example:

```
unordered_map<int, int> um;
um[1] = 2;
um[2] = 4;
```

Best use case: When **ordering doesn't matter**, and fast operations are needed (e.g., counting, hashing).

□ **Warning**: In rare cases of poor hash functions or collisions, time complexity can go to **O(n)**.

□ Container Adapters

These containers are **wrappers** around other containers and restrict operations to specific behavior.

---

1. stack – LIFO (Last-In First-Out)

Example:

```
stack<int> st;
st.push(1);
st.push(2);
st.top();   // 2
st.pop();   // Removes 2
```

Best use case: DFS, expression evaluation, undo operations.

---

2. queue – FIFO (First-In First-Out)

Example:

```
queue<int> q;
q.push(1);
q.push(2);
q.front();  // 1
q.pop();    // Removes 1
```

Best use case: BFS, scheduling.

---

3. priority_queue – Max Heap by Default

Max Heap:

```
priority_queue<int> pq;
pq.push(10);
pq.push(5);
pq.top(); // 10
pq.pop(); // Removes 10
```

Min Heap:

```
priority_queue<int, vector<int>, greater<int>> min_pq;
min_pq.push(10);
min_pq.push(5);
min_pq.top(); // 5
```

Best use case: Dijkstra’s algorithm, top-K problems, task scheduling.

Name: Prathamesh Arvind Jadhav

 Summary Table

Container Type	Container	Key Traits
Sequence	vector	Dynamic array, random access
Sequence	deque	Fast insert/delete at both ends
Sequence	list	Doubly linked list
Sequence	forward_list	Singly linked list (lightweight)
Sequence	array	Fixed-size array
Associative	set	Unique keys, sorted
Associative	multiset	Duplicate keys allowed
Associative	map	Key-value pairs, unique keys, sorted
Associative	multimap	Multiple values per key
Unordered	unordered_map	Key-value, unique keys, avg O(1) ops
Adapter	stack	LIFO
Adapter	queue	FIFO
Adapter	priority_queue	Max/Min heap

2.Iterators in C++ STL

 What are Iterators?

Name: Prathamesh Arvind Jadhav

Iterators are **abstract pointers** used to access elements inside STL containers. They provide a **unified way** to traverse through **different types of containers** (vector, set, map, etc.).

Think of them as **smart pointers** that work across **all STL containers**, even those that don't allow random access.

□ Why Use Iterators?

- Container-independent traversal (same syntax for vector, set, map, etc.)
- Necessary for algorithms like sort(), find(), lower_bound(), etc.
- Allows **element modification**, traversal, and **insertion at specific positions**.

□ Common Iterator Functions

Function	Description
begin()	Points to the first element
end()	Points to **one past** the last element
rbegin()	Reverse begin iterator
rend()	Reverse end iterator
cbegin()	Constant begin (read-only)
cend()	Constant end (read-only)
advance(it, n)	Moves iterator n steps forward
prev(it, n)	Moves iterator n steps backward
next(it, n)	Moves forward and returns the new iterator

Name: Prathamesh Arvind Jadhav

□ Example Usage (with vector):

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

int main() {
   vector<int> v = {10, 20, 30};

   // Traditional iterator
   vector<int>::iterator it;
   for (it = v.begin(); it != v.end(); ++it) {
      cout << *it << " "; // dereference to get value
   }

   // Modern C++: Use auto
   for (auto it = v.begin(); it != v.end(); ++it) {
      cout << *it << " ";
   }

   // Range-based for loop (alternative)
   for (int x : v) {
      cout << x << " ";
   }
}
```

□ Types of Iterators

STL defines 5 categories of iterators:

Type	Supports	Containers
Input Iterator	Read-only, single-pass	istream_iterator
Output Iterator	Write-only, single-pass	ostream_iterator
Forward Iterator	Multiple passes, read/write	forward_list

Name: Prathamesh Arvind Jadhav

Type	Supports	Containers
Bidirectional	Forward/backward, read/write	list, set, map
Random Access	+ Arithmetic ops (it + 1, it[i])	vector, deque, array

---

□ Reverse Iterators

Used to iterate **from end to beginning**.

```
vector<int> v = {1, 2, 3, 4};
for (auto rit = v.rbegin(); rit != v.rend(); ++rit) {
   cout << *rit << " "; // prints 4 3 2 1
}
```

□ Modifying Elements via Iterator

```
for (auto it = v.begin(); it != v.end(); ++it) {
   *it *= 2;  // doubles every element
}
```

□ Iterator in Associative Containers

Example with map:

```
map<string, int> m = {{"apple", 3}, {"banana", 5}};
for (auto it = m.begin(); it != m.end(); ++it) {
   cout << it->first << " " << it->second << endl;
}
```

Here, it->first is the key, it->second is the value.

Best Practices for Competitive Programming

- Always use auto unless you need an explicit type.

- Prefer range-based for loops if you don't need index access.
- Use rbegin()/rend() for reverse iteration.

□ Summary Cheatsheet

Operation	Syntax	Description
Traverse forward	for (auto it = c.begin(); it != c.end(); ++it)	From first to last element
Traverse reverse	for (auto it = c.rbegin(); it != c.rend(); ++it)	From last to first
Modify in-place	*it = *it + 1;	Modify element via dereference
Access key-value pair	it->first, it->second	Only for maps and multimaps
Insert with iterator	v.insert(pos, val)	Insert at pos in sequence
Move iterator forward	advance(it, n)	Advance iterator n steps

3.STL Algorithms (<algorithm>)

The <algorithm> header provides powerful, optimized functions to operate on ranges and containers. These are crucial in contests to reduce code length and improve efficiency.

□ Common Algorithms

sort(begin, end) – O(n log n)

Name: Prathamesh Arvind Jadhav

Sorts a container in ascending order.

sort(v.begin(), v.end());

reverse(begin, end)

Reverses the order of elements in the container.

reverse(v.begin(), v.end());

count(begin, end, value) – O(n)

Counts how many times value occurs in the range.

int freq = count(v.begin(), v.end(), 5);

find(begin, end, value) – O(n)

Returns an iterator to the **first occurrence** of value, or end() if not found.

```
auto it = find(v.begin(), v.end(), 10);
if (it != v.end()) cout << "Found!";
```

 Searching Algorithms

All work in **sorted containers**. Time: O(log n)

binary_search(begin, end, x)

Returns a **boolean** indicating if x exists.

```
if (binary_search(v.begin(), v.end(), 7))
   cout << "Exists";
```

lower_bound(begin, end, x)

Returns iterator to the **first element ≥ x**

auto it = lower_bound(v.begin(), v.end(), 5); // 5 or next greater

upper_bound(begin, end, x)

Returns iterator to the **first element > x**

auto it = upper_bound(v.begin(), v.end(), 5); // strictly greater

Difference between them:

Function	Return Meaning
lower_bound	First position ≥ x
upper_bound	First position > x
binary_search	True/False if x is present

 Permutations

next_permutation(begin, end)

Transforms to the **next lexicographical** permutation.

```
vector<int> v = {1, 2, 3};
do {
   for (int x : v) cout << x << " ";
   cout << endl;
} while (next_permutation(v.begin(), v.end()));
```

prev_permutation(begin, end)

Gives the **previous** lexicographical permutation.

 Min/Max Functions

min(a, b) / max(a, b)

Simple min/max between two values.

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

cout << min(a, b); // 3

min_element(begin, end) / max_element(begin, end)

Finds iterator to the **min/max element** in range.

auto it = min_element(v.begin(), v.end());
cout << *it;

 Additional Helpful Algorithms

Algorithm	Use
accumulate	Sum of elements
all_of	Check if all elements meet condition
any_of	Check if any element meets condition
none_of	Check if no element meets condition
unique	Remove duplicates from sorted range
rotate	Rotates elements
partition	Separates based on condition

Example (C++11):

accumulate(v.begin(), v.end(), 0); // Sum

 Summary Table – Algorithms

Function	Use Case
sort()	Sorting in ascending

Function	Use Case
reverse()	Reverse order
count()	Count frequency
find()	Find element
binary_search()	Fast existence check
lower_bound()	First ≥ value
upper_bound()	First > value
next_permutation()	Generate permutations
min_element()/max_element()	Min/Max in container

4.Functors (Function Objects)

 What is a Functor?

A **functor** is any object that can be **called like a function**. It is typically implemented by **overloading operator()** in a class or struct.

They are useful for:

- Custom sorting
- Passing custom logic to STL algorithms

 Functor Example: Custom Sort (Descending)

```
struct Compare {
   bool operator()(int a, int b) {
```

```
        return a > b; // Descending
    }
};

vector<int> v = {3, 1, 4, 2};
sort(v.begin(), v.end(), Compare()); // Use functor
```

Using Lambdas (Recommended in CP, C++11+)

A **lambda function** is an anonymous inline function object.

```
sort(v.begin(), v.end(), [](int a, int b) {
    return a > b; // Descending
});
```

Lambda with map, priority_queue:

Custom comparator for priority_queue (min-heap with pair):

```
priority_queue<pair<int, int>, vector<pair<int, int>>,
    function<bool(pair<int,int>, pair<int,int>)>> pq(
    [](pair<int,int> a, pair<int,int> b) {
        return a.second > b.second; // min by second
    }
);
```

 Lambda Syntax

```
 [ capture ] ( parameters ) -> return_type {
    // body
}
```

Example:

```
auto add = [](int a, int b) -> int {
    return a + b;
};
cout << add(3, 5); // 8
```

Name: Prathamesh Arvind Jadhav

□ Summary Table – Functors

Type	Example Use	Syntax
Functor	Custom class	struct C { bool operator()() {} };
Lambda	Quick inline comparator	[](int a, int b) { return a < b; }
Use Case	sort, priority_queue, custom behavior	Pass as comparator argument

---

Best Practices for Competitive Programming

- Use **lambda** for short comparators.
- Use **functors** only when the logic is large or reused.
- Always remember to **sort** before using binary_search, lower_bound, upper_bound.

DAY-12:

Linear Search -

What is Linear Search?

Linear Search (also called **Sequential Search**) is the simplest searching algorithm that checks each element in a list one by one until the desired element is found or the list ends.

□ Key Concept

- Traverse the array **from start to end**.
- Compare each element with the **target value**.
- If found, return the **index**.
- If not found after the whole array is checked, return **-1** or indicate **"not found"**.

Name: Prathamesh Arvind Jadhav

□ Algorithm Steps

1. Start from the **first element** of the array.
2. Compare the current element with the **target**.
3. If a match is found, return the **index**.
4. If no match is found, move to the **next element**.
5. Repeat until the **end** of the array.
6. If the end is reached without finding the element, return **-1**.

⧖ Time Complexity

Case	Time Complexity
Best Case	O(1)
Average Case	O(n)
Worst Case	O(n)

Best Case: Element is at the beginning.
Worst Case: Element is not in the array or at the end.

□ Space Complexity

- **O(1)** → No extra space is used except for variables.

□ When to Use?

- When the array is **unsorted**.
- When the array is **small**.
- Simplicity is more important than efficiency.

Name: Prathamesh Arvind Jadhav

Advantages

- Simple to implement.
- Works on **unsorted** data.
- No extra memory needed.

Disadvantages

- **Inefficient** for large datasets.
- Requires **O(n)** comparisons in worst case.

C++ Code Example

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

int linearSearch(int arr[], int n, int key) {
    for (int i = 0; i < n; i++) {
        if (arr[i] == key) {
            return i; // Element found at index i
        }
    }
    return -1; // Element not found
}

int main() {
    int arr[] = {10, 25, 30, 45, 60};
    int size = sizeof(arr) / sizeof(arr[0]);
    int key = 30;

    int result = linearSearch(arr, size, key);

    if (result != -1)
        cout << "Element found at index: " << result << endl;
```

```
    else
        cout << "Element not found." << endl;

    return 0;
}
```

□ Visualization Example

Array: [10, 25, 30, 45, 60]
Index: 0 1 2 3 4

Search for 30:
→ Check index 0 → 10 ≠ 30
→ Check index 1 → 25 ≠ 30
→ Check index 2 → 30 = 30 □ Found!

DAY-13: