Part01:

Github:
https://github.com/janaashraf888/rowadmasrtasks.git

Default Values of Array Elements in C#:

In C#, when an array is created, all its elements are automatically assigned a **default value** depending on the type of the array:

- **Numeric types** (int, float, double, long, etc.) → 0
- **Boolean type** (bool) → false
- **Character type** (char) → '\0' (null character)
- **Reference types** (string, objects, arrays, etc.) → null

This ensures that all elements have a predictable initial value even if you do not explicitly assign values to them.

Difference Between Array.Clone() and Array.Copy() in C#:

1. **Array.Clone()**
    - Creates a **shallow copy** of the original array.
    - Returns a **new array of the same type and length** with the same elements.
    - Changes to the elements of **reference types inside the array** will affect the original array (because it's shallow).
2. **Array.Copy()**
    - Copies elements from a **source array** to a **destination array**.
    - The destination array **must already exist**.
    - Can copy **all or part of the array**.

- Useful for selectively copying or merging arrays.

Difference Between GetLength() and Length for Multi-Dimensional Arrays in C#:

1. **Length**
    - Returns the **total number of elements** in the array, across **all dimensions**.
    - Example: a 3x4 array → Length = 12.
2. **GetLength(int dimension)**
    - Returns the **number of elements in a specific dimension**.
    - The dimension parameter is zero-based:
        - 0 → first dimension (rows)
        - 1 → second dimension (columns)
    - Example: a 3x4 array → GetLength(0) = 3, GetLength(1) = 4.

Difference Between Array.Copy() and Array.ConstrainedCopy() in C#:

1. **Array.Copy()**
    - Copies elements from a **source array** to a **destination array**.
    - If an **exception occurs during the copy** (e.g., index out of range), the destination array may be partially updated.
    - Does **not guarantee atomicity** — some elements may be copied before the error occurs.
2. **Array.ConstrainedCopy()**

- Also copies elements from a source array to a destination array.
- **Ensures atomicity**: if an exception occurs, **no changes are made** to the destination array.
- Provides a **safer copy** when exceptions could occur.

Why foreach Is Preferred for Read-Only Operations on Arrays in C#:

The foreach loop is preferred for **read-only operations** because:

1. It **automatically iterates** through all elements of the array without needing an index.
2. It **prevents accidental modification** of array elements, since the iteration variable is read-only.
3. It **makes code cleaner and less error-prone** compared to using a for loop with manual indexing.

Importance of Input Validation When Working with User Inputs:

Input validation is important because it ensures that the data entered by the user is **correct, safe, and expected**. Proper validation:

1. **Prevents errors and exceptions** – ensures the program does not crash due to invalid input.
2. **Maintains data integrity** – only valid and meaningful data is processed or stored.
3. **Enhances security** – helps protect against malicious input that could exploit the program.

4. **Improves user experience** – provides clear feedback when input is invalid.

Formatting the Output of a 2D Array for Better Readability:

To make a 2D array easier to read when printed:

1. **Use nested loops** – an outer loop for rows and an inner loop for columns.
2. **Separate elements with spaces or tabs** (" " or "\t") to align columns.
3. **Print a new line after each row** (Console.WriteLine()) so rows appear clearly.
4. Optionally, **add row/column headers** or labels to make the output more informative.

When to Prefer a switch Statement Over if-else in C#:

A switch statement is preferred when you need to **compare a single variable against multiple constant values**. It is especially useful because:

1. **Improves readability** – cleaner and easier to follow than a long chain of if-else if statements.
2. **Simplifies code maintenance** – adding or removing cases is easier.
3. **Can be more efficient** – some compilers optimize switch better than multiple if-else checks.

Time Complexity of Array.Sort() in C#:

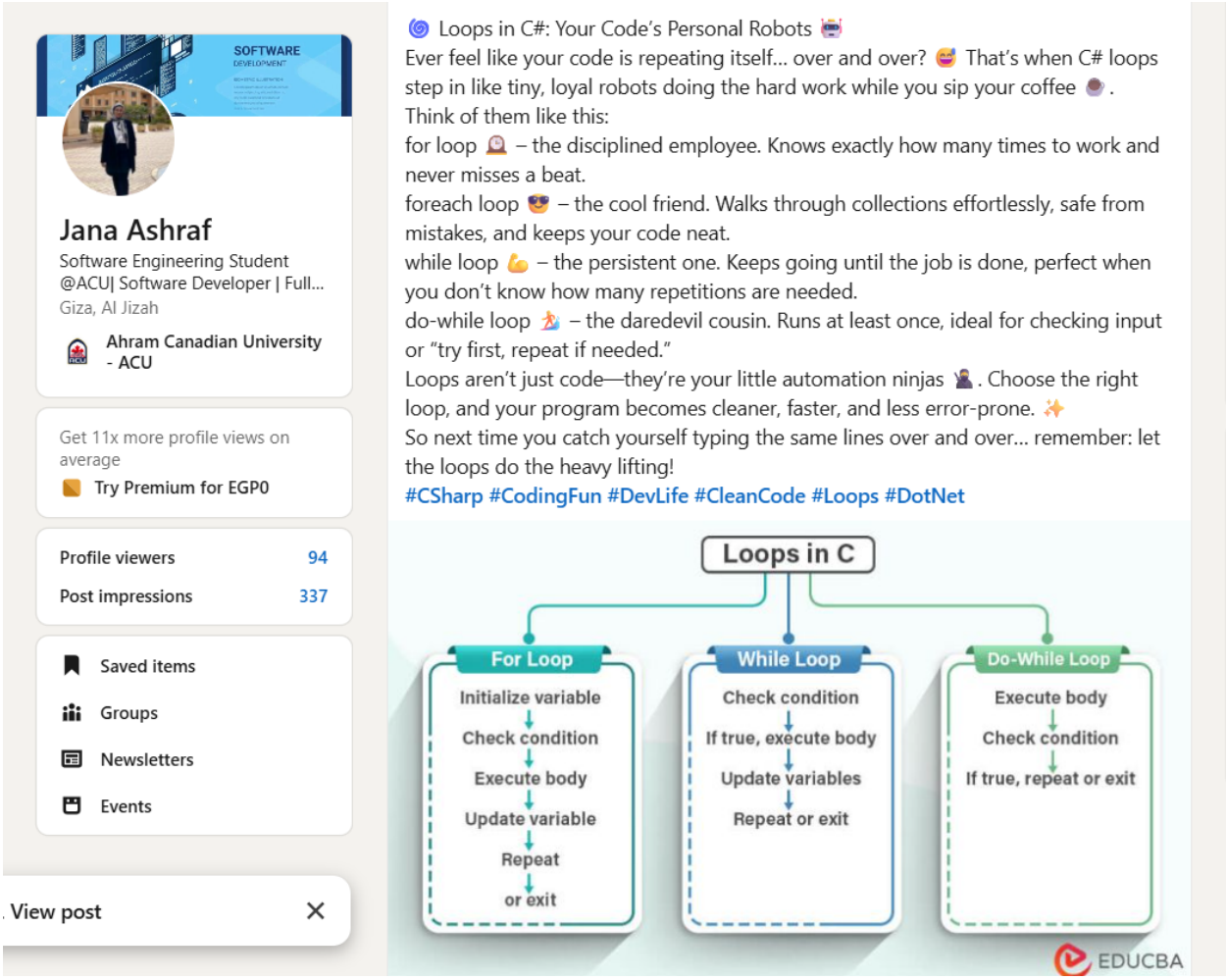
- The Array.Sort() method in C# uses the **Introspective Sort (Introsort)** algorithm, which is a **hybrid of QuickSort, HeapSort, and InsertionSort**.

- **Average case:** O(n log n) – very efficient for most arrays.
- **Worst case:** O(n log n) – guaranteed by switching to HeapSort if QuickSort recursion becomes too deep.
- **Best case:** O(n) – occurs when the array is already nearly sorted (InsertionSort is used for small partitions).

Efficiency of for vs foreach for Calculating the Sum of an Array in C#:

- **for loop** is generally **slightly more efficient** than foreach when calculating the sum of an array.
- Reason: for uses **direct index-based access**, which is faster for arrays.
- foreach internally uses an **enumerator**, which adds a tiny overhead, even though it is more readable.

Part02:



Behavior When the User Enters a Value Outside the Range 1–7:

If a program expects a number between **1 and 7** (for example, to select a day of the week) and the user enters a value **outside this range**, the program should handle it using **input validation**.

- Without validation, attempting to **convert the number to an enum or access an array** could cause a **runtime error or exception**.
- With proper validation, the program can **display an error message** (e.g., "Invalid input! Enter a number between 1 and 7") and **prompt the user to enter a valid value** again.

Part03:

1️⃣ Counting Systems: Decimal, Binary, Hexadecimal

- **Decimal (Base 10):** Uses digits 0–9; standard number system for humans.
- **Binary (Base 2):** Uses digits 0 and 1; standard for computers.
- **Hexadecimal (Base 16):** Uses digits 0–9 and letters A–F; often used to represent memory addresses or compact binary data.

2️⃣ Time Complexity

- **Time complexity** measures how the execution time of an algorithm grows with input size.
- Common notations include O(1) for constant time, O(n) for linear time, and $O(n^2)$ for quadratic time.
- It helps **compare efficiency** of algorithms and predict performance for large inputs.

3️⃣ Array of Objects: Extra Step

- Creating an array of objects **does not automatically create the objects themselves**.

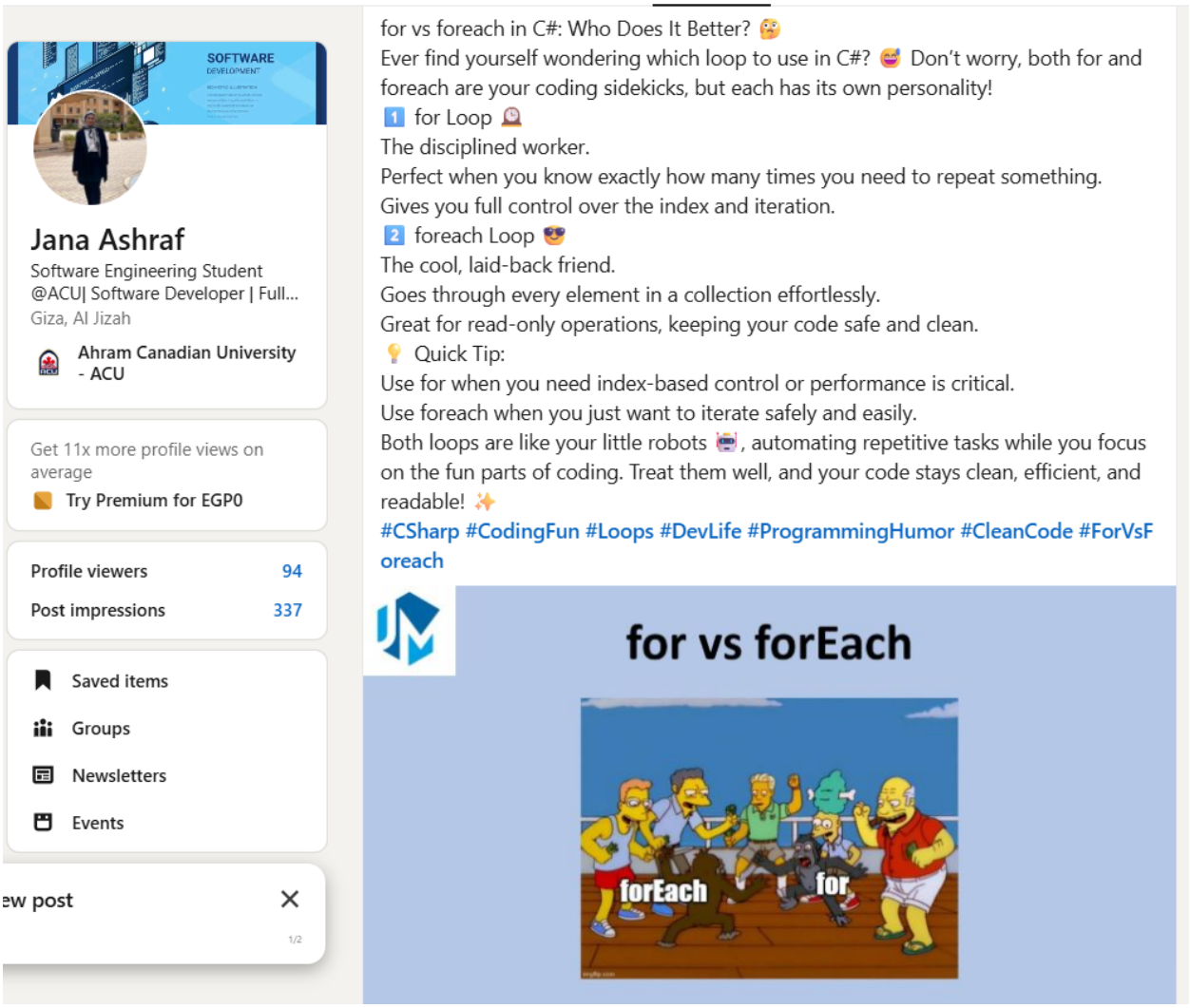
- An **extra step is required** to instantiate each object before using it.

4️⃣ Jagged Array

- A **jagged array** is an **array of arrays**, where each row can have a **different length**.
- It is used when rows of data have varying sizes.

5️⃣ Deep Copy with Array of Objects

- A **shallow copy** copies only references, so changes affect the original objects.
- A **deep copy** creates **new instances** of each object, so modifying one does **not affect the other**.
- Deep copies are necessary to avoid unintended side effects when working with object arrays.



Default Size of Stack and Heap in C# and Considerations

1. **Stack:**

- The stack is a **small, fast memory area** used for storing **local variables, method parameters, and function call information**.
- **Default size:** Usually **1 MB per thread** in .NET (can vary depending on OS and runtime settings).
- **Considerations:**
    - Limited in size → avoid allocating large objects on the stack.

- Fast allocation and deallocation (LIFO order).
- Thread-specific memory; each thread gets its own stack.

2. **Heap:**

- The heap is a **larger, slower memory area** used for **objects, reference types, and dynamic memory allocation**.
- **Default size:** Managed by the **.NET runtime's garbage collector**; grows as needed, limited by system memory.
- **Considerations:**
  - Slower than stack allocation.
  - Memory fragmentation is possible.
  - Requires garbage collection to free unused memory.
  - Suitable for large objects or objects needing longer lifetimes.

Time Complexity

Time complexity measures how the **execution time of an algorithm grows** as the size of the input increases. It is used to **analyze the efficiency** of algorithms and compare them.

- **Common notations:**
  - **O(1):** Constant time – execution does not depend on input size.
  - **O(n):** Linear time – execution grows proportionally with input size.
  - **O(n^2):** Quadratic time – execution grows with the square of the input size.