Question Answering Bonus

1. What's the default size of stack and heap and what are the considerations ?

The **stack** and **heap** are memory regions used for different purposes, and their default sizes depend on the runtime environment (e.g., .NET Framework, .NET Core, or .NET 5+).

Default Stack Size

- **Default Size**:
  - **In .NET (Windows):** The default stack size for a thread is **1 MB** for 32-bit processes and **4 MB** for 64-bit processes.
  - **In .NET Core/.NET 5+:** The default stack size is typically **1 MB** for the main thread, but it can vary slightly depending on the platform (e.g., Windows, Linux, macOS).
  - Each new thread created in a program gets its own stack, typically with the same default size (1 MB), though this can be customized when creating threads.
- **Purpose**: The stack stores value types (e.g., int, struct), method call frames, and local variables with short lifetimes. It operates in a last-in, first-out (LIFO) manner and is managed automatically.

Default Heap Size

- **Default Size**:
  - The heap size is not fixed; it's dynamically allocated by the .NET runtime's garbage collector (GC) and grows as needed, limited by available system memory.
  - In .NET, the heap is divided into:
    - **Small Object Heap (SOH)**: For objects < 85,000 bytes.
    - **Large Object Heap (LOH)**: For objects ≥ 85,000 bytes.

    - Initial heap size is small (e.g., a few MB), but it expands based on application demand, up to the system's virtual memory limit (e.g., ~2 GB for 32-bit processes, much larger for 64-bit).
- **Purpose**: The heap stores reference types (e.g., objects, arrays, strings) and is managed by the garbage collector, which allocates and deallocates memory dynamically.

Considerations for Stack and Heap

1. **Stack Considerations**:
    - **Fixed Size Limitation**: The stack's fixed size (e.g., 1 MB) means deep recursion or large local variables (e.g., large structs or arrays) can cause a StackOverflowException. Avoid excessive recursion or large stack-allocated data.
    - **Performance**: Stack allocation is fast because it's a simple pointer adjustment. Use value types and local variables for efficiency when possible.
    - **Threading**: Each thread has its own stack, so creating many threads increases memory usage (e.g., 100 threads × 1 MB = 100 MB). Consider thread pooling to manage stack memory.
    - **Customization**: You can set a custom stack size when creating a thread (e.g., new Thread(method, stackSizeInBytes)), but this is rarely needed in typical applications.
2. **Heap Considerations**:
    - **Dynamic Growth**: The heap grows dynamically, but excessive allocations (e.g., large objects or frequent allocations) can lead to memory fragmentation or increased GC pauses, impacting performance.
    - **Garbage Collection**: The GC manages heap memory, but frequent collections (especially for the LOH) can degrade performance. Minimize large object allocations and consider object pooling for reusable objects.
    - **Memory Limits**: In 32-bit processes, the heap is limited to ~2 GB (or less due to fragmentation). 64-bit processes have much

larger limits but can still exhaust system memory if mismanaged.
    - **Object Lifetime**: Long-lived objects (e.g., static objects) stay in memory longer, increasing heap usage. Be mindful of static references or caches that prevent GC.

2. What is time complexity ?

Time complexity measures the amount of time an algorithm takes to run as a function of the input size, typically expressed using Big O notation (e.g., O(n), O(n^2)). It describes how the runtime scales as the input grows, focusing on the worst-case or average-case performance.

Key Points

- **Purpose**: Indicates an algorithm's efficiency, helping compare different approaches for the same task.
- **Common Examples**:
    - **O(1)**: Constant time (e.g., accessing an array element by index).
    - **O(n)**: Linear time (e.g., iterating through an array once).
    - **O(n log n)**: Linearithmic time (e.g., efficient sorting like QuickSort).
    - **O(n^2)**: Quadratic time (e.g., nested loops for each element).
- **Factors**: Depends on operations like loops, recursion, or comparisons, not on hardware or exact execution time.
- **Why it matters**: Helps predict performance for large inputs, ensuring scalability in applications.

The time complexities ordered from **better** (faster, more efficient) to **worse** (slower, less efficient) as the input size n grows, are:

1. **O(1)**: Constant time – Execution time is fixed, regardless of input size.

2. **O(n)**: Linear time – Execution time grows linearly with input size.
3. **O(n log n)**: Linearithmic time – Execution time grows slightly faster than linear due to a logarithmic factor.
4. **O(n^2)**: Quadratic time – Execution time grows quadratically, becoming much slower for large inputs.

Why This Order?

- **O(1)** is the fastest because it takes the same time regardless of n.
- **O(n)** scales linearly, so doubling n n n doubles the runtime.
- **O(n log n)** grows faster than linear but much slower than quadratic, making it efficient for sorting or divide-and-conquer algorithms.
- **O(n^2)** is the slowest here, as the runtime squares with input size, making it impractical for large n n n.

For example, for n=1000:

- O(1): ~1 operation
- O(n): ~1000 operations
- O(n log n): ~10,000 operations (assuming log n ≈ 10)
- O(n^2): ~1,000,000 operations