Homework

Task.

- clone **`git clone https://github.com/sebastienros/memoryleak.git`** (great testing app by Sébastien Ros)
- run it (.\src\MemoryLeak\MemoryLeak):

```
dotnet run -c Release
```

- **`https://localhost:5001/`** should present a nice introspective graph about memory usage (Working Set), allocations, CPU and current Request Per Second (RPS)
- MemoryLeak exposes some REST endpoints for testing various memory-related scenarios, we will use **`\bigstring`** which just allocates and returns 10KB string. You can test it at **`https://localhost:5001/api/bigstring`**
- you can hit it by F5 many times to observe some memory usage change
- we will make a simple load test agains **`bistring`** endpoint using **`https://github.com/aliostad/SuperBenchmarker`** command-line tool (just download single EXE file from the repository)

```
.\sb.exe -y 100 -n 10000000 -c 64 -u http://localhost:5000/api/bigstring
```

- play with the concurrency (**`-c`**) and delay between calls (**`-y`**) params to put more or less allocation pressure on the GC
  - does it influence the client perspective? Superbenchmarker automatically opens metrics from the client side.

Task (cont.)

- change its GC modes via **`COMPlus_gcServer`** and **`COMPlus_gcConcurrent`** variables to observe more or less aggresive GC behaviour, e.g.:

```
$Env:COMPlus_gcServer=0
$Env:COMPlus_gcConcurrent=0
dotnet run -c Release
```

- play with **`GCHeapHardLimit`** and/or **`GCHeapCount`** (you can also use **`COMPlus_...`** variables) for the server GC to observe limited memory usage and resulting GC aggressiveness
  - *Note:* remember that when setting values via environment variables, hexadecimal values are expected. E.f. refer to Heap limit documentaton Tip for more details.
- (optional) Extend MemoryLeak to print GCMemoryInfo.TotalAvailableMemoryBytes and Environment.ProcessorCount next to printing GC Server/Workstation mode and observe how they change with respect to the above settings