Homework

Task 1. `memoryleak` app with LOH allocations

- we will be using the same **`memoryleak`** app so I just skip how it should be prepared.
- this time we will be using its **`/loh`** endpoint
- run the app and also start the load test:

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

- now, let's use **the new shiny toy** from Maoni (😍) - **`GCRealTimeMon`** available at https://github.com/Maoni0/realmon
- you can just install it as a global tool:

```
❯ dotnet tool install -g dotnet-gcmon
```

- and run it during our load test:

```
❯ dotnet-gcmon -n MemoryLeak
```

  Note: With the help of **`s`** you can print detailed info about the recent GC.
- what are the result? What GC are triggered and why? What are the generation sizes in time?

Task 1. `memoryleak` app with LOH allocations

- let's prepare custom **`dotnet-gcmon`** configuration file for displaying more LOH-related data - run

```
❯ dotnet-gcmon -g lohanalysis.yaml
```

- ... and select there **`type`**, **`gen`**, **`reason`**, **`peak size`**, **`promoted`**, **`LOH size`**, **`LOH survival rate`**, **`LOH frag ratio`**
- as you will see you can additionaly setup the heaps stats timer and filtering GCs based on pause duration - let's skip it for now
- use generated file to run the tool:

```
❯ dotnet-gcmon -c .\lohanalysis.yaml -n MemoryLeak
```

- what are the result? What's LOH fragmentation and survival rate?
- we know that those FullGCs are happening because of **`AllocLarge`**, which means - most probably - the LOH allocation budget has been exceeded. Let's confirm that by recording **`dotnet-trace`** session and dig in into *GCStats*. If so, what allocation budgets are calculated for LOH?

Task 2. `AssemblyLoadContext` and static fields

- use the course repository to pull **`.\Module07-Roots-Generations-And-Memory-Leaks\PluginApp`** application - it implements a very simple "plugins" functionality with the help of custom **`AssemblyLoadContext`**
- look around in the code. **`TestCommand`** contains **`static InternalIdentifier`** that we will investigate in memory. Because the main application loads the *TestPlugin* twice, it will be interesting to observe how it behaves.
- build the application **from the level of `sln` file** (because of dependencies between projects):

  ```
  ❯ dotnet build -c Release
  ```

  and then run it from the same folder:

  ```
  ❯ dotnet run -c Release --project .\WebWorkerApp\WebWorkerApp.csproj
  ```

- after seeing **`Processing...`**, take a memory dump with **`dotnet-dump`** tool
- open the dump with the help of **`dotnet-dump analyze`** and look what's keeping **`InternalIdentifier`** instance(s) alive. In which generations those instance(s) live?
- also use **`dumpalc`** (use **`help`**) for looking what information you can get about the corresponding assembly contextt

Task 2. `AssemblyLoadContext` and static fields

- convert our regular dump file into PerfView's Heap Snapshot by double clicking **`.dmp`** file from PerfView - *Collecting Memory Data* dialog should open for making conversion to **`gcDump`** format. Click *Dump GC Heap* there. A result file should be immediately opened. Look for **`InternalIdentifier`** in the *Heap Stacks* report.
- (optional) open the dump in Visual Studio to see how **`InternalIdentifier`** instances are presented there.