Benchmark tutorial

1. Create a project

⚠ If you already have a project, skip this step.

```
dotnet new console --name "MyProject"
```

2. Install BenchmarkDotNet

Install `BenchmarkDotNet` via the NuGet package:

```
// visual studio 2019
PM> Install-Package BenchmarkDotNet
```

```
// VSCode
dotnet add package BenchmarkDotNet
```

3. Design a Benchmark

We create a class to hold our benchmarks within it. This can contain any number of private methods and can include setup code within the constructor. Any code within the constructor is not included in the timing of the method. We can then create public methods and add the attribute of `[Benchmark]` to have them listed as items that should be measured and benchmarked.

In the following example, we compare *MD5* and *SHA256* cryptographic hash functions:

```
using System;
using System.Security.Cryptography;
using BenchmarkDotNet.Attributes;
using BenchmarkDotNet.Running;

namespace MyBenchmarks
{
    [MemoryDiagnoser] // adds memory information
    public class Md5VsSha256
    {
        private const int N = 10000;
        private readonly byte[] data;

        private readonly SHA256 sha256 = SHA256.Create();
        private readonly MD5 md5 = MD5.Create();

        public Md5VsSha256()
        {
            data = new byte[N];
            new Random(42).NextBytes(data);
```

```
        }

        [Benchmark]
        public byte[] Sha256() => sha256.ComputeHash(data);

        [Benchmark]
        public byte[] Md5() => md5.ComputeHash(data);
    }

    public class Program
    {
        public static void Main(string[] args)
        {
            var summary = BenchmarkRunner.Run<Md5VsSha256>();
        }
    }
}
```

Inside our main method of our console application, we use the `BenchmarkRunner` class to run our benchmark.

Now we can run the benchmarks with:

```
dotnet run -c RELEASE
```

4. View Results

The `BenchmarkRunner.Run<Md5VsSha256>()` call runs your benchmarks and print results to console output.

Results:

Method	Mean	Error	StdDev	Allocated
Sha256	44.01 us	0.784 us	0.733 us	112 B
Md5	19.12 us	0.221 us	0.207 us	80 B

- **Mean** : Arithmetic mean of all measurements
- **Error** : Half of 99.9% confidence interval
- **StdDev** : Standard deviation of all measurements
- **Allocated** : Allocated memory per single operation (managed only, inclusive, 1KB = 1024B)
- **1 us** : 1 Microsecond (0.000001 sec)

This simple table gives us a pretty good overview of the benchmark. Taking a look, it seems like **MD5** is about twice as fast of the **SHA256** method.

5. Analyze results

In your bin directory, you can find a lot of useful files with detailed information.

For example:

- Csv reports with raw data: Md5VsSha256-report.csv, Md5VsSha256-runs.csv
- Markdown reports: Md5VsSha256-report-default.md, Md5VsSha256-report-stackoverflow.md, Md5VsSha256-report-github.md

- Plain report and log: Md5VsSha256-report.txt, Md5VsSha256.log
- Plots (if you have installed R): Md5VsSha256-barplot.png, Md5VsSha256-boxplot.png, and so on.

6. Examples

6.1 Arrays vs Lists

This simple example will show us some implementations differences between an array and a list.

```
using System;
using System.Collections.Generic;
using System.Security.Cryptography;
using BenchmarkDotNet.Attributes;
using BenchmarkDotNet.Running;

namespace Benchmarker
{
    [MemoryDiagnoser]
    public class ListMemoryBenchmarker
    {
        [Params(1000, 1000000)]
        public int N;

        [Benchmark]
        public void Array()
        {
            var array = new int[N];
            for(int i = 0; i < N; i++)
            {
                array[i] = i;
            }
        }

        [Benchmark]
        public void ListA()
        {
            var list = new List<int>(N);
            for(int i = 0; i < N; i++)
            {
                list.Add(i);
            }
        }

        [Benchmark]
        public void ListB()
        {
            var list = new List<int>();
            for(int i = 0; i < N; i++)
            {
                list.Add(i);
            }
        }

        [Benchmark]
        public void HashSet()
        {
```

```
            var set = new HashSet<int>();
            for(int i = 0; i < N; i++)
            {
                set.Add(i);
            }
        }
    }

    public class Program
    {
        public static void Main(string[] args)
        {
            var summary = BenchmarkRunner.Run<ListMemoryBenchmarker>();
        }
    }
}
```

We have four benchmarks this time. Each adding elements to a different collection. The first collection is a basic array. The second collection is a list specifying an initial capacity. The third collection is a list with the default initial capacity. And the last collection is a HashSet. We are also adding a second value in the Params tag to run with our Benchmark. This will allow us to see what happens as higher values of N are used.

Results:

Method	N	Mean	Error	StdDev	Gen 0	Gen 1	Gen 2	Allocated
Array	1000	645.2 ns	7.41 ns	6.57 ns	1.2817	-	-	4 KB
ListA	1000	1,623.0 ns	32.43 ns	55.93 ns	1.2913	-	-	4 KB
ListB	1000	2,009.9 ns	25.50 ns	23.85 ns	2.6817	-	-	8 KB
HashSet	1000	13,112.0 ns	145.38 ns	121.40 ns	18.6310	-	-	57 KB
Array	1000000	2,119,791.2 ns	41,270.57 ns	53,663.40 ns	996.0938	996.0938	996.0938	3,906 KB
ListA	1000000	3,029,406.1 ns	38,858.12 ns	36,347.91 ns	996.0938	996.0938	996.0938	3,906 KB
ListB	1000000	5,031,434.4 ns	64,199.53 ns	60,052.28 ns	1992.1875	1992.1875	1992.1875	8,192 KB
HashSet	1000000	24,492,881.1 ns	488,367.82 ns	963,990.43 ns	1156.2500	1093.7500	1093.7500	42,101 KB

- **N** : Value of the 'N' parameter
- **Mean** : Arithmetic mean of all measurements
- **Error** : Half of 99.9% confidence interval
- **StdDev** : Standard deviation of all measurements
- **Gen 0** : GC Generation 0 collects per 1000 operations
- **Gen 1** : GC Generation 1 collects per 1000 operations
- **Gen 2** : GC Generation 2 collects per 1000 operations
- **Allocated** : Allocated memory per single operation (managed only, inclusive, 1KB = 1024B)
- **1 ns** : 1 Nanosecond (0.000000001 sec)

Conclusion: Examining the results, its easy to see that across the board using an `Array` gave us the best performance, followed by `ListA`, then `ListB`, the way behind is `HashSet`.

`Array` and `ListA` had the same memory footprint for both Ns. Time wise, `Array` ran faster than `ListA`, around twice as fast for 1000 and ~40% faster for 1,000,000.

6.2 Needle in a haystack

We are going to build up a large list in C# with random items within it, and place a "needle" right in the middle of the list. Then we will compare how doing `SingleOrDefault` on a list compares to `FirstOrDefault`.

```
using System;
using System.Collections.Generic;
using System.Linq;

using BenchmarkDotNet.Attributes;
using BenchmarkDotNet.Running;

namespace BenchmarkExample
{
    public class SingleVsFirst
    {
        private readonly List<string> _haystack = new List<string>();
        private readonly int _haystackSize = 1000000;
        private readonly string _needle = "needle";

        public SingleVsFirst()
        {
            //Add a large amount of items to our list.
            Enumerable.Range(1, _haystackSize).ToList().ForEach(x =>
_haystack.Add(x.ToString()));
            //Insert the needle right in the middle.
            _haystack.Insert(_haystackSize / 2, _needle);
        }

        [Benchmark]
        public string Single() => _haystack.SingleOrDefault(x => x == _needle);

        [Benchmark]
        public string First() => _haystack.FirstOrDefault(x => x == _needle);

    }

    class Program
    {
        static void Main(string[] args)
        {
            var summary = BenchmarkRunner.Run<SingleVsFirst>();
            Console.ReadLine();
        }
    }
}
```

Method	Mean	Error	StdDev	Allocated

Method	Mean	Error	StdDev	Allocated
Single	15.121 ms	0.1565 ms	0.1464 ms	104 B
First	7.402 ms	0.0950 ms	0.0842 ms	104 B

- **Mean** : Arithmetic mean of all measurements
- **Error** : Half of 99.9% confidence interval
- **StdDev** : Standard deviation of all measurements
- **Allocated** : Allocated memory per single operation (managed only, inclusive, 1KB = 1024B)
- **1 ms** : 1 Millisecond (0.001 sec)

Conclusion: So it looks like Single is twice as slow as First! If you understand what Single does under the hood, this is to be expected. When First finds an item, it immediately returns (After all, it only wants the "First" item). However when Single finds an item, it still needs to traverse the entire rest of the list because if there is more than one, it needs to throw an exception. This makes sense when we are placing the item in the middle of the list!

6.3 String vs Span

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Security.Cryptography;
using BenchmarkDotNet.Attributes;
using BenchmarkDotNet.Running;

namespace Bench
{
    [MemoryDiagnoser]
    public class StringVsSpan
    {
        private static readonly string _dateAsText = "10 02 2022";

        [Benchmark]
        public (int day, int month, int year) DateWithStringAndSubstring()
        {
            var dayAsText = _dateAsText.Substring(0,2);
            var monthAsText = _dateAsText.Substring(3, 2);
            var yearAsText = _dateAsText.Substring(6);

            var day = int.Parse(dayAsText);
            var month = int.Parse(monthAsText);
            var year = int.Parse(yearAsText);

            return (day, month, year);
        }

        [Benchmark]
        public (int day, int month, int year) DateWithStringAndSpan()
        {
            ReadOnlySpan<char> dateAsSpan = _dateAsText;

            var dayAsText = dateAsSpan.Slice(0,2);
            var monthAsText = dateAsSpan.Slice(3, 2);
            var yearAsText = dateAsSpan.Slice(6);
```

```
            var day = int.Parse(dayAsText);
            var month = int.Parse(monthAsText);
            var year = int.Parse(yearAsText);

            return (day, month, year);
        }

        public class Program {

            static void Main(string[] args)
            {
                BenchmarkRunner.Run<StringVsSpan>();
            }
        }
    }
}
```

Method	Mean	Error	StdDev	Gen 0	Allocated
DateWithStringAndSubstring	76.12 ns	0.877 ns	0.777 ns	0.0305	96 B
DateWithStringAndSpan	53.21 ns	0.525 ns	0.491 ns	-	-

- **Mean** : Arithmetic mean of all measurements
- **Error** : Half of 99.9% confidence interval
- **StdDev** : Standard deviation of all measurements
- **Gen 0** : GC Generation 0 collects per 1000 operations
- **Allocated** : Allocated memory per single operation (managed only, inclusive, 1KB = 1024B)
- **1 ns** : 1 Nanosecond (0.000000001 sec)

Conclusion: Spans are Faster and more efficient thant SubStrings.

7. Links

- BenchmarkDotNet
- Examples 1
- Examples 2
- En Castellano