Initialize lists size to improve performance

Some collections, like `List<T>`, have a predefined initial size.

Every time you add a new item to the collection, there are two scenarios:

1. the collection has free space, allocated but not yet populated, so adding an item is immediate;
2. the collection is already full: internally, .NET resizes the collection, so that the next time you add a new item, we fall back to option #1.

Clearly, the second approach has an impact on the overall performance. Can we prove it?

Here's a benchmark that you can run using BenchmarkDotNet:

```
[Params(2, 100, 1000, 10000, 100_000)]
public int Size;

[Benchmark]
public void SizeDefined()
{
    int itemsCount = Size;

    List<int> set = new List<int>(itemsCount);
    foreach (var i in Enumerable.Range(0, itemsCount))
    {
        set.Add(i);
    }
}

[Benchmark]
public void SizeNotDefined()
{
    int itemsCount = Size;

    List<int> set = new List<int>();
    foreach (var i in Enumerable.Range(0, itemsCount))
    {
        set.Add(i);
    }
}
```

Those two methods are almost identical: the only difference is that in one method we specify the initial size of the list: `new List<int>(itemsCount)`.

Have a look at the result of the benchmark run with .NET 7:

Method	Size	Mean	Error	StdDev	Median	Gen 0	Gen 1	Gen 2	Allocated
SizeDefined	2	49.50 ns	1.039 ns	1.678 ns	49.14 ns	0.0248	-	-	104 B
SizeNotDefined	2	63.66 ns	3.016 ns	8.507 ns	61.99 ns	0.0268	-	-	112 B
SizeDefined	100	798.44 ns	15.259 ns	32.847 ns	790.23 ns	0.1183	-	-	496 B
SizeNotDefined	100	1,057.29 ns	42.100 ns	121.469 ns	1,056.42 ns	0.2918	-	-	1224 B
SizeDefined	1000	9,180.34 ns	496.521 ns	1,400.446 ns	8,965.82 ns	0.9766	-	-	4096 B
SizeNotDefined	1000	9,720.66 ns	406.184 ns	1,184.857 ns	9,401.37 ns	2.0142	-	-	8464 B
SizeDefined	10000	104,645.87 ns	7,636.303 ns	22,395.954 ns	99,032.68 ns	9.5215	1.0986	-	40096 B
SizeNotDefined	10000	95,192.82 ns	4,341.040 ns	12,524.893 ns	92,824.50 ns	31.2500	-	-	131440 B
SizeDefined	100000	1,416,074.69 ns	55,800.034 ns	162,771.317 ns	1,402,166.02 ns	123.0469	123.0469	123.0469	400300 B

Method	Size	Mean	Error	StdDev	Median	Gen 0	Gen 1	Gen 2	Allocated
SizeNotDefined	100000	1,705,672.83 ns	67,032.839 ns	186,860.763 ns	1,621,602.73 ns	285.1563	285.1563	285.1563	1049485 B

Notice that, in general, they execute in a similar amount of time; for instance when running the same method with 100000 items, we have the same magnitude of time execution: 1,416,074.69 ns vs 1,705,672.83 ns.

The huge difference is with the allocated space: 400,300 B vs 1,049,485 B. Almost 2.5 times better!

Ok, it works. Next question: **How can we check a List capacity?**

We've just learned that capacity impacts the performance of a List.

How can you try it live? Easy: have a look at the `Capacity` property!

```
List<int> myList = new List<int>();

foreach (var element in Enumerable.Range(0,65))
{
    myList.Add(element);
    Console.WriteLine($"Items count: {myList.Count} - List capacity: {myList.Capacity}");
}
```

If you run this method, you'll see this output:

```
Items count: 1 - List capacity: 4
Items count: 2 - List capacity: 4
Items count: 3 - List capacity: 4
Items count: 4 - List capacity: 4
Items count: 5 - List capacity: 8
Items count: 6 - List capacity: 8
Items count: 7 - List capacity: 8
Items count: 8 - List capacity: 8
Items count: 9 - List capacity: 16
Items count: 10 - List capacity: 16
Items count: 11 - List capacity: 16
Items count: 12 - List capacity: 16
Items count: 13 - List capacity: 16
Items count: 14 - List capacity: 16
Items count: 15 - List capacity: 16
Items count: 16 - List capacity: 16
Items count: 17 - List capacity: 32
```

```
Items count: 18 - List capacity: 32
Items count: 19 - List capacity: 32
Items count: 20 - List capacity: 32
Items count: 21 - List capacity: 32
Items count: 22 - List capacity: 32
Items count: 23 - List capacity: 32
Items count: 24 - List capacity: 32
Items count: 25 - List capacity: 32
Items count: 26 - List capacity: 32
Items count: 27 - List capacity: 32
Items count: 28 - List capacity: 32
Items count: 29 - List capacity: 32
Items count: 30 - List capacity: 32
Items count: 31 - List capacity: 32
Items count: 32 - List capacity: 32
Items count: 33 - List capacity: 64
Items count: 34 - List capacity: 64
Items count: 35 - List capacity: 64
Items count: 36 - List capacity: 64
Items count: 37 - List capacity: 64
Items count: 38 - List capacity: 64
Items count: 39 - List capacity: 64
Items count: 40 - List capacity: 64
Items count: 41 - List capacity: 64
Items count: 42 - List capacity: 64
Items count: 43 - List capacity: 64
Items count: 44 - List capacity: 64
Items count: 45 - List capacity: 64
Items count: 46 - List capacity: 64
Items count: 47 - List capacity: 64
Items count: 48 - List capacity: 64
Items count: 49 - List capacity: 64
Items count: 50 - List capacity: 64
Items count: 51 - List capacity: 64
Items count: 52 - List capacity: 64
Items count: 53 - List capacity: 64
Items count: 54 - List capacity: 64
Items count: 55 - List capacity: 64
Items count: 56 - List capacity: 64
Items count: 57 - List capacity: 64
Items count: 58 - List capacity: 64
Items count: 59 - List capacity: 64
Items count: 60 - List capacity: 64
Items count: 61 - List capacity: 64
Items count: 62 - List capacity: 64
Items count: 63 - List capacity: 64
Items count: 64 - List capacity: 64
Items count: 65 - List capacity: 128
```

So, as you can see, **List capacity is doubled every time the current capacity is not enough**.