



VISUGXL 2024

A Season of Speed

Turning Puzzles
into C# Performance Wins

Michaël Hompus

▲ Architect @ Info Support

▲ Microsoft MVP

🦋 @eNeRGy164

🐙 /eNeRGy164



Menu

- 🎄 Advent of Code
- 📅 Puzzle
- 🕒 Measuring performance
- 📈 Improve the solution
- 🔗 All kinds of stuff
- 👉 To follow



Advent of code



Advent of code

- ▲ Created by Eric Wastl
- ▲ Annual coding challenge held in December
- ▲ Designed to improve programming skill through fun puzzles
- ▲ Open to programmers of all skill levels













Advent of code

- ▲ Every day you get a (unique) input file
- ▲ The daily puzzle is composed of 2 parts
- ▲ Part 2 unlocks when the correct answer for part 1 is submitted



Leaderboards & Scoring

- ▲ Scoring points is based on the amount of people on the leaderboard
- ▲ There is a global leaderboard for the top 100
 - ▲ The first person to get a star gets 100 points; the hundredth person gets 1 point

▲ Ch	1)	3257		xiaowuc1
▲ You	2)	3174		tckmn
▲ Ha	3)	2909		5space (AoC++)
▲ Head	4)	2486		nthistle (AoC++)
▲ This	5)	2484		jonathanpaulson
	6)	2476		Antonio Molero
	7)	2404		dan-simon
	8)	2370		bluepichu
	9)	2285		leijurv (AoC++)
	10)	2241		boboquack

- ▼ Channels
- # advent-of-code
- # advent-of-code-oplossingen
- # advent-of-code-score

-----Part 1-----				-----Part 2-----			
Day	Time	Rank	Score	Time	Rank	Score	
25	03:01:18	3111	0	03:01:22	2590	0	
24	01:36:50	2611	0	04:19:01	1939	0	
23	01:44:49	3403	0	02:20:10	1683	0	
22	02:05:04	2770	0	02:25:58	2369	0	
21	00:21:55	2245	0	02:28:06	995	0	
20	00:56:30	1404	0	01:36:46	1119	0	
19	00:51:42	3515	0	02:08:42	2664	0	
18	00:43:26	2470	0	07:01:07	6773	0	
17	03:08:04	3691	0	03:16:31	3229	0	
16	01:45:31	5449	0	02:09:55	5385	0	
15	00:10:19	3613	0	00:43:41	3724	0	
14	00:36:44	5307	0	03:07:02	6574	0	
13	00:42:09	3314	0	04:28:22	8769	0	
12	13:25:34	22532	0	15:14:30	11949	0	
11	00:24:53	2998	0	00:27:31	1843	0	
10	00:22:55	1964	0	02:14:13	2923	0	
9	00:26:02	4675	0	00:26:02	3796	0	
8	00:52:14	1273	0	00:52:14	4004	0	
7	01:01:24	4067	0	01:01:24	3856	0	
6	00:31:09	8020	0	00:31:09	7403	0	
5	13:27:44	3010	0	13:27:44	26061	0	
4	00:21:07	2302	0	00:21:07	2422	0	
3	00:26:25	1059	0	00:26:25	1057	0	
2	00:25:41	5896	0	00:30:04	5198	0	
1	00:05:21	2329	0	00:20:53	1881	0	

9) 5965 ★★★★★
10) 5750 ★★★★★

*Spoilers
ahead!*



2020 – Day 1 – Part 1

Before you leave, the Elves in accounting just need you to fix your expense report (your puzzle input); apparently, something isn't quite adding up.

Specifically, they need you to find the two entries that sum to 2020 and then multiply those two numbers together.

For example, suppose your expense report contained the following:

```
1721
979
366
299
675
1456
```

In this list, the two entries that sum to 2020 are 1721 and 299. Multiplying them together produces $1721 * 299 = 514579$, so the correct answer is 514579.

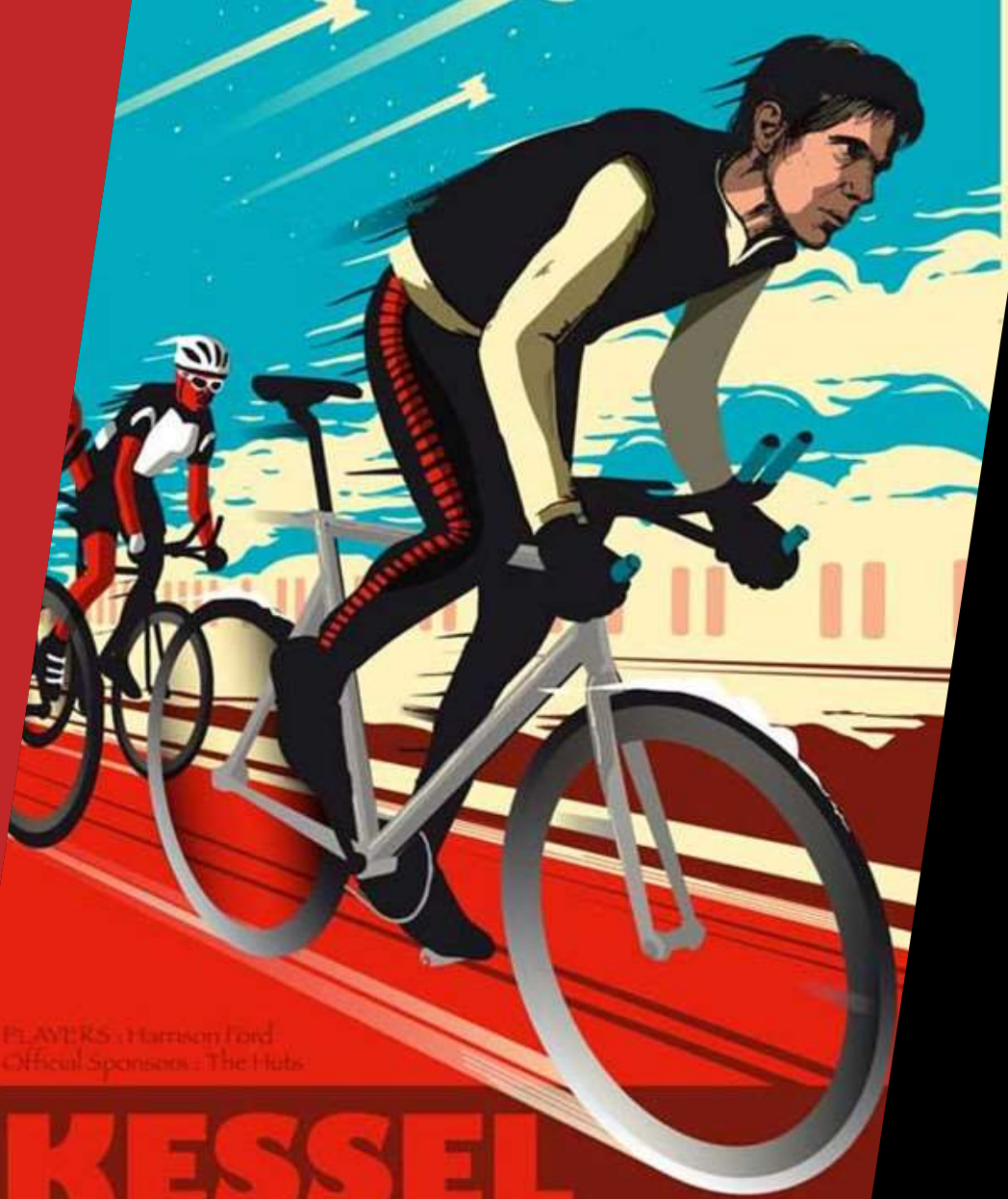
Of course, your expense report is much larger. Find the two entries that sum to 2020; what do you get if you multiply them together?



Solution-LINQ Query

```
var query = from i in input
             let x = Convert.ToInt32(i)
             from j in input
             let y = Convert.ToInt32(j)
             where x + y == 2020
             select x * y;

return query.First();
```



PLAYERS: Harrison Ford
Official Sponsors: The Hutts

**KESSEL
RUN**

= The Parsec challenge
Kessel - Si'Klaata Clus

Benchmarking

Measuring performance

```
var input = File.ReadAllLines("input.txt");

var stopwatch = Stopwatch.StartNew();

var part1 = 0;
var part2 = 0;

// Magic

stopwatch.Stop();
Console.WriteLine($"{stopwatch.ElapsedMilliseconds}ms ({stopwatch.ElapsedTicks}) ticks");
Console.WriteLine($"Part1: {part1}");
Console.WriteLine($"Part2: {part2}");
```

Time scale – seconds

1s

Time scale – milliseconds

0,001s



1 ms

Time scale – microseconds

0,000,001s



1 ms



1 μs

Time scale – nanoseconds

0,000,000,001s



1 ms



1 μs



1 ns

Time scale – ticks

0,000,000,001s



1 ms



1 μ s



1 tick



1 ns

BenchmarkDotNet

- ▲ **BenchmarkDotNet** helps you to transform methods into benchmarks, track their performance, and share reproducible measurement experiments

Example

```
BenchmarkRunner.Run<Benchmark>();
```

```
public class Benchmark
{
    private static readonly string[] input = File.ReadAllLines("input.txt");

    [Benchmark]
    public int LinqQuery()
    {
        var query = from i in input
                     let x = Convert.ToInt32(i)
                     from j in input
                     let y = Convert.ToInt32(j)
                     where x + y == 2020
                     select x * y;

        return query.First();
    }
}
```

Run (as release)

```
■ Michaël@Buran G:> eNeRGy164> AdventBenchmark  
> dotnet run -c Release -f net8.0
```

The image is a title card with a green wood grain background. The text is white with a black outline and is arranged in three lines. The first line reads "A LITTLE LONGER", the second line reads "THAN A Few", and the third line reads "MiNUTEs LaTER".

**A LITTLE LONGER
THAN A Few
MiNUTEs LaTER**

Results

```
// * Summary *
```

```
BenchmarkDotNet v0.14.1-nightly.20241115.196, Windows 11 (10.0.22631.4460/23H2/2023Update/SunValley3)
```

```
AMD Ryzen 5 7600 3.80GHz, 1 CPU, 12 logical and 6 physical cores
```

```
.NET SDK 9.0.100
```

```
[Host] : .NET 9.0.0 (9.0.24.52809), X64 RyuJIT AVX-512F+CD+BW+DQ+VL+VBMI  
.NET Framework 4.8.1 : .NET Framework 4.8.1 (4.8.9282.0), X64 RyuJIT VectorSize=256  
.NET 5.0 : .NET 5.0.17 (5.0.1722.21314), X64 RyuJIT AVX2  
.NET 6.0 : .NET 6.0.36 (6.0.3624.51421), X64 RyuJIT AVX2  
.NET 7.0 : .NET 7.0.20 (7.0.2024.26716), X64 RyuJIT AVX2  
.NET 8.0 : .NET 8.0.11 (8.0.1124.51707), X64 RyuJIT AVX-512F+CD+BW+DQ+VL+VBMI  
.NET 9.0 : .NET 9.0.0 (9.0.24.52809), X64 RyuJIT AVX-512F+CD+BW+DQ+VL+VBMI
```

Method	Job	Runtime	Median	Gen0	Allocated
LinqQuery	.NET Framework 4.8.1	.NET Framework 4.8.1	1,006.5 us	166.0156	1026.81 KB
LinqQuery	.NET 5.0	.NET 5.0	494.5 us	62.5000	1023.76 KB
LinqQuery	.NET 6.0	.NET 6.0	447.1 us	62.5000	1023.76 KB
LinqQuery	.NET 7.0	.NET 7.0	421.0 us	62.5000	1023.76 KB
LinqQuery	.NET 8.0	.NET 8.0	297.2 us	62.5000	1023.76 KB
LinqQuery	.NET 9.0	.NET 9.0	294.5 us	62.5000	1023.76 KB

Measurement – LINQ Query

Method	Mean	Ratio	Gen0	Gen1	Allocated	Alloc Ratio
LINQ Query	484.269 ns	1,000	834,961		1.048.328 B	1,000

Measurement – Part 1

0,000,484,269s



You
Are
here

*Let's
speed up!*



Solution-LINQ with indexes

```
var query = from i in Enumerable.Range(0, input.Length)
             let x = Convert.ToInt32(input[i])
             from j in Enumerable.Range(i + 1, input.Length - i - 1)
             let y = Convert.ToInt32(input[j])
             where x + y == 2020
             select x * y;

return query.First();
```

Measurement - LINQ w/ indexes

Method	Mean	Ratio	Gen0	Gen1	Allocated	Alloc Ratio
LINQ Query	484.269 ns	1,000	834,961		1.048.328 B	1,000
LINQ w/ indexes	362.067 ns	0,749	659,180		830.576 B	0,792

Solution-foreach

```
var numbers = input.Select(int.Parse).ToArray();
```

```
var result = 0;
```

```
var i = 0;
```

```
foreach (var x in numbers)
```

```
{
```

```
    foreach (var y in numbers.Skip(i))
```

```
    {
```

```
        if (x + y == 2020)
```

```
            result = x * y;
```

```
    }
```

```
    i++;
```

```
}
```

```
return result;
```

Measurement-foreach

Method	Mean	Ratio	Gen0	Gen1	Allocated	Alloc Ratio
LINQ Query	484.269 ns	1,000	834,961		1.048.328 B	1,000
LINQ w/ indexes	362.067 ns	0,749	659,180		830.576 B	0,792
foreach	136.976 ns	0,282	0,732		10.472 B	0,010

Solution-foreach with Range

```
var numbers = input.Select(int.Parse).ToArray();
```

```
var result = 0;
```

```
var i = 0;
```

```
foreach (var x in numbers)
```

```
{
```

```
    foreach (var y in numbers[i..])
```

```
    {
```

```
        if (x + y == 2020)
```

```
            result = x * y;
```

```
    }
```

```
    i++;
```

```
}
```

```
return result;
```

Measurement-foreach w/ Range

Method	Mean	Ratio	Gen0	Gen1	Allocated	Alloc Ratio
LINQ Query	484.269 ns	1,000	834,961		1.048.328 B	1,000
foreach with Skip	136.976 ns	0,282	0,732		10.472 B	0,010
foreach with Range	14.207 ns	0,029	68,817	0,0153	86.472 B	0,082

Solution-for

```
var numbers = input.Select(int.Parse).ToArray();

var result = 0;

for (var i = 0; i < numbers.Length; i++)
{
    for (var j = i + 1; j < numbers.Length; j++)
    {
        if (numbers[i] + numbers[j] == 2020)
        {
            result = numbers[i] * numbers[j];
        }
    }
}

return result;
```

Measurement-for

Method	Mean	Ratio	Gen0	Gen1	Allocated	Alloc Ratio
LINQ Query	484.269 ns	1,000	834,961		1.048.328 B	1,000
foreach with Range	14.207 ns	0,029	68,817	0,0153	86.472 B	0,082
for	8.913 ns	0,018	0.0610		872 B	0,001

Solution-goto

```
var numbers = input.Select(int.Parse).ToArray();

var result = 0;

for (var i = 0; i < numbers.Length; i++)
{
    for (var j = i + 1; j < numbers.Length; j++)
    {
        if (numbers[i] + numbers[j] == 2020)
        {
            result = numbers[i] * numbers[j];
            goto quit;
        }
    }
}

quit:
return result;
```

Measurement – goto

Method	Mean	Ratio	Gen0	Gen1	Allocated	Alloc Ratio
LINQ Query	484.269 ns	1,000	834,961		1.048.328 B	1,000
for	8.913 ns	0,018	0,0610		872 B	0,001
goto	5.874 ns	0,012	0,0687		872 B	0,001

Solution – intermediate variables

```
var numbers = input.Select(int.Parse).ToArray();
```

```
var result = 0;
```

```
for (var i = 0; i < numbers.Length; i++)  
{
```

```
    var a = numbers[i];
```

```
    for (var j = i + 1; j < numbers.Length; j++)  
    {
```

```
        var b = numbers[j];
```

```
        if (a + b == 2020)  
        {
```

```
            result = a * b;  
            goto quit;
```

```
        }
```

```
    }
```

```
}
```

```
quit:
```

```
return result;
```


Measurement – intermediate vars

Method	Mean	Ratio	Gen0	Gen1	Allocated	Alloc Ratio
LINQ Query	484.269 ns	1,000	834,961		1.048.328 B	1,000
goto	5.874 ns	0,012	0,0687		872 B	0,001
Intermediate variables	5.586 ns	0,012	0,0687		872 B	0,001

Solution – initialize array

```
// Old  
var numbers = input.Select(int.Parse).ToArray();
```

```
// New  
var numbers = new int[input.Length];  
for (var i = 0; i < input.Length; i++)  
{  
    numbers[i] = int.Parse(input[i]);  
}
```

Measurement – initialize array

Method	Mean	Ratio	Gen0	Gen1	Allocated	Alloc Ratio
LINQ Query	484.269 ns	1,000	834,961		1.048.328 B	1,000
Intermediate variables	5.586 ns	0,012	0,0687		872 B	0,001
Initialize Array	5.949 ns	0,012	0,0610		824 B	0,001

ASCII

- ▲ A **char** in dotnet is stored as a **ushort**
- ▲ By subtracting 48, you get the integer equivalent

Char	Value	Char	Value	Char	Value
(sp)	32	@	64	`	96
!	33	A	65	a	97
"	34	B	66	b	98
#	35	C	67	c	99
\$	36	D	68	d	100
%	37	E	69	e	101
&	38	F	70	f	102
'	39	G	71	g	103
(40	H	72	h	104
)	41	I	73	i	105
*	42	J	74	j	106
+	43	K	75	k	107
,	44	L	76	l	108
-	45	M	77	m	109
.	46	N	78	n	110
/	47	O	79	o	111
0	48	P	80	p	112
1	49	Q	81	q	113
2	50	R	82	r	114
3	51	S	83	s	115
4	52	T	84	t	116
5	53	U	85	u	117
6	54	V	86	v	118
7	55	W	87	w	119
8	56	X	88	x	120
9	57	Y	89	y	121
:	58	Z	90	z	122
;	59	[91	{	123
<	60	\	92		124
=	61]	93	}	125
>	62	^	94	~	126
?	63	_	95	(del)	127

Solution – Custom integer parsing

```
var numbers = new int[input.Length];
for (var i = 0; i < input.Length; i++)
{
    numbers[i] = ParseInt(input[i]);
}

int ParseInt(string input)
{
    var value = 0;

    for (var i = 0; i < input.Length; i++)
    {
        value = value * 10 + (input[i] - '0');
    }

    return value;
}
```

123

0 * 10 = 0

0 + 1 = 1

1 * 10 = 10

10 + 2 = 12

12 * 10 = 120

120 + 3 = 123

Measurement – Custom int parse

Method	Mean	Ratio	Gen0	Gen1	Allocated	Alloc Ratio
LINQ Query	484.269 ns	1,000	834,961		1.048.328 B	1,000
Initialize Array	5.949 ns	0,012	0,0610		824 B	0,001
Custom Integer Parser	4.432 ns	0,009	0,0610		824 B	0,001

Solution - ushort

```
var numbers = new ushort[input.Length];
for (var i = 0; i < input.Length; i++)
{
    numbers[i] = ParseShort(input[i]);
}

ushort ParseShort(string input)
{
    var value = 0;

    for (var i = 0; i < input.Length; i++)
    {
        value = value * 10 + (input[i] - '0');
    }

    return (ushort)value;
}
```

Measurement – ushort

Method	Mean	Ratio	Gen0	Gen1	Allocated	Alloc Ratio
LINQ Query	484.269 ns	1,000	834,961		1.048.328 B	1,000
Custom Integer Parser	4.432 ns	0,009	0,0610		824 B	0,001
Lower Memory	4.658 ns	0,010	0,0305		424 B	0,000

Solution-Pointers

```
int* numbers = stackalloc int[input.Length];
for (int i = 0; i < input.Length; i++)
{
    *(numbers + i) = ParseInt(input[i]);
}

for (int* a = numbers; a < numbers + input.Length; a++)
{
    for (int* b = a + 1; b < numbers + input.Length; b++)
    {
        if (*a + *b == 2020)
            return (*a) * (*b);
    }
}

return 0;
```

Measurement – Pointers

Method	Mean	Ratio	Gen0	Gen1	Allocated	Alloc Ratio
LINQ Query	484.269 ns	1,000	834,961		1.048.328 B	1,000
Custom Integer Parser	4.432 ns	0,009	0,0610		824 B	0,001
Pointers	5.708 ns	0,012				

Solution – Array references

```
var numbers = new int[input.Length];
for (var i = 0; i < input.Length; i++)
{
    numbers[i] = ParseInt(input[i]);
}

ref var numberRef = ref MemoryMarshal.GetArrayDataReference(numbers);

for (var i = 0; i < numbers.Length; i++)
{
    var a = Unsafe.Add(ref numberRef, i);

    for (var j = i + 1; j < numbers.Length; j++)
    {
        var b = Unsafe.Add(ref numberRef, j);

        if (a + b == 2020)
            return a * b;
    }
}

return 0;
```

Measurement – Array Refs

Method	Mean	Ratio	Gen0	Gen1	Allocated	Alloc Ratio
LINQ Query	484.269 ns	1,000	834,961		1.048.328 B	1,000
Custom Integer Parser	4.432 ns	0,009	0,0610		824 B	0,001
Refs	5.945 ns	0,012	0,0610		824 B	0,001

Solution–Two pointers technique

```
Array.Sort(numbers);

var left = 0;
var right = numbers.Length - 1;

while (left < right)
{
    var a = numbers[left];
    var b = numbers[right];

    var sum = a + b;
    if (sum == 2020)
        return a * b;
    else if (sum < 2020)
        left++;
    else
        right--;
}

return 0;
```

Measurement – Two pointers

Method	Mean	Ratio	Gen0	Gen1	Allocated	Alloc Ratio
LINQ Query	484.269 ns	1,000	834,961		1.048.328 B	1,000
Custom Integer Parser	4.432 ns	0,009	0,0610		824 B	0,001
Two pointers	3.059 ns	0,006	0,0687		872 B	0,001

Solution-Hashing

```
var numbers = new HashSet<int>(input.Length);  
for (var i = 0; i < input.Length; i++)  
{  
    numbers.Add(ParseInt(input[i]));  
}  
  
foreach (var number in numbers)  
{  
    var complement = 2020 - number;  
  
    if (numbers.Contains(complement))  
        return number * complement;  
}  
  
return 0;
```


Measurement – Hashing

Method	Mean	Ratio	Gen0	Gen1	Allocated	Alloc Ratio
LINQ Query	484.269 ns	1,000	834,961		1.048.328 B	1,000
Two pointers	3.059 ns	0,006	0,0687		872 B	0,001
Hashing	2.438 ns	0,005	0,3128		3.944 B	0,004

Solution – BitArray

```
var bitArray = new BitArray(2020 + 1);

for (var i = 0; i < input.Length; i++)
{
    var number = ParseInt(input[i]);
    var complement = 2020 - number;

    if (bitArray[complement])
    {
        return number * complement;
    }

    bitArray[number] = true;
}

return 0;
```

<https://github.com/dotnet/runtime/blob/main/src/libraries/System.Collections/src/System/Collections/BitArray.cs>

Measurement – Bit Array

Method	Mean	Ratio	Gen0	Gen1	Allocated	Alloc Ratio
LINQ Query	484.269 ns	1,000	834,961		1.048.328 B	1,000
Hashing	2.438 ns	0,005	0,3128		3.944 B	0,004
Bits	622 ns	0,001	0,0248		312 B	0,000

Measurement – Part 1

0,000,484,269s



You
Are
here

Measurement – Part 1

0,000,000,622s



You
Are
here

2020 – Day 1 – Part 2

--- Part Two ---

The Elves in accounting are thankful for your help; one of them even offers you a starfish coin they had left over from a past vacation. They offer you a second one if you can find **three** numbers in your expense report that meet the same criteria.

Using the above example again, the three entries that sum to `2020` are `979`, `366`, and `675`. Multiplying them together produces the answer, `241861950`.

In your expense report, what is the product of the three entries that sum to `2020`?

What a December,
huh?

Captain, it's Day 5



Solution-LINQ Query

```
var query = from i in input
             let x = Convert.ToInt32(i)
             from j in input
             let y = Convert.ToInt32(j)
             from k in input
             let z = Convert.ToInt32(k)
             where x + y + z == 2020
             select x * y * z;
```

Solution – Custom integer parser

```
for (var i = 0; i < numbers.Length; i++)
{
    var a = numbers[i];

    for (var j = i + 1; j < numbers.Length; j++)
    {
        var b = numbers[j];

        for (var k = j + 1; k < numbers.Length; k++)
        {
            var c = numbers[k];

            if (a + b + c == 2020)
                return a * b * c;
        }
    }
}

return 0;
```

Solution–Two pointers

```
Array.Sort(numbers);

for (var i = 0; i < numbers.Length - 2; i++)
{
    var a = numbers[i];

    var left = i + 1;
    var right = numbers.Length - 1;

    while (left < right)
    {
        var b = numbers[left];
        var c = numbers[right];

        ...
    }
}

return 0;
```

Measurements – Part 2

Method	Mean	Ratio	Gen0	Allocated	Alloc Ratio
LINQ Query	484.269 ns		834,961	1.048.328 B	
LINQ Query	17.115.944 ns	1,000	28,750,000	36.410.868 B	1,000
Custom Integer Parser	4.432 ns		0,061	824 B	
Custom Integer Parser	87.794 ns	0,005	0,061	824 B	0,000
Two pointers	3.059 ns		0,069	872 B	
Two pointers	2.995 ns	0,000	0,069	872 B	0,000

Why faster than dotnet itself?

- ▲ Not general purpose
- ▲ No out-of-range checks needed
- ▲ Limited input set (ASCII)
- ▲ Tailored methods for specific input (unsigned, signed)

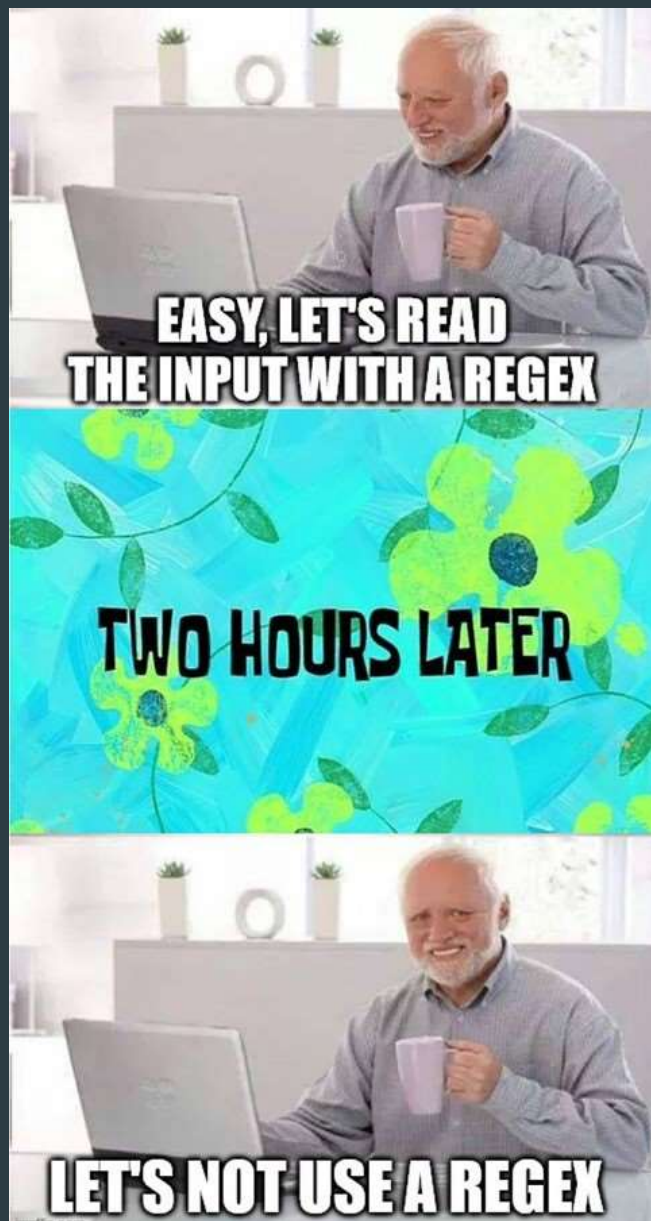
My custom methods

- ▲ IntParse
- ▲ SIntParse
- ▲ LongParse
- ▲ SLongParse
- ▲ IsAsciiDigit
- ▲ IsNotAsciiDigit
- ▲ Abs

*All kinds
of stuff...*



Input parsing



Input parsing

▲ Regex vs. String.Split vs. Span<T>

Method	Mean	Ratio	Gen0	Gen1	Allocated	Alloc Ratio
Regex	595,61 μs	1,00	136,718	16,60	1.683 KB	1,00
Generated Regex	483,82 μs	0,82	137,207	17,09	1.683 KB	1,00
string.Split	272,14 μs	0,46	68,359	8,30	840 KB	0,50
string.Split w/ char array	173,47 μs	0,29	45,654	5,61	560 KB	0,33
Span<T> w/ ref	42,00 μs	0,07	2,807		35 KB	0,02
Smart Span<T>	13,96 μs	0,02	2,853		35 KB	0,02

Test: Sum a list of **int**

Method	Mean	Ratio	Allocated	Alloc Ratio
IEnumerable.Sum(Func<T, int>)	411,69 µs	1,07	32 B	0,67
IEnumerable.Sum	385,08 µs	1,00	48 B	1,00
IEnumerable.Aggregate	375,12 µs	0,97	32 B	0,67
Array.ForEach	172,41 µs	0,45	72 B	1,50
foreach	91,41 µs	0,24		
while	89,78 µs	0,23		
for	88,71 µs	0,23		

Test: Walk through `int[512, 512]`

Test	Mean
for X, for Y	166,80 μ s
for Y, for X	135,23 μ s
foreach	130,99 μ s

Test: Serialize char[512, 512] *to* String

Test	Mean	Allocated
foreach	1.180,33 μ s	1.056.760 B
for Y, for X	1.115,98 μ s	1.056.724 B
string.Create	464,74 μ s	524,356 B
new string(MemoryMarshal.CreateSpan(...))	36,92 μ s	524,321 B

Test: Create and store in HashSet<T>

Test	Mean	Ratio	Allocated	Alloc Ratio
class with fields	891,0 µs	1,01	943,40 KB	1,00
sealed class	887,5 µs	1,00	943,40 KB	1,00
class	887,3 µs	1,00	943,40 KB	1,00
readonly struct with fields	865,1 µs	0,98	945,50 KB	1,00
readonly struct	848,8 µs	0,96	945,50 KB	1,00
struct	840,9 µs	0,95	945,50 KB	1,00
record class	651,1 µs	0,74	943,40 KB	1,00
sealed record class	634,1 µs	0,72	943,40 KB	1,00
record struct	522,5 µs	0,59	591,83 KB	0,63
readonly record struct	522,2 µs	0,59	591,83 KB	0,63

Know your list types

▲ Array

- 👍 Fixed size makes them memory efficient and fast access due to contiguous memory allocation
- 👎 Limited resizing capabilities

▲ Array[,]

- 👍 Ideal for representing data, like a 2D map
- 👎 More complex to manage and iterate over; resizing is not possible

▲ List<T>

- 👍 Dynamically resizes and has a lot of functionality out-of-the-box for searching, sorting, etc.
- 👎 Can be slower for certain operations due to internal resizing and non-contiguous data in memory

▲ Dictionary<K, V>

- 👍 Fast lookups by key, which can be more efficient than searching through a list
- 👎 Does not store elements in any particular order

Know your list types - part 2

▲ **HashSet<T>**

- 👍 Provides high-performance set operations and ensures that all elements are unique
- 👎 Does not store elements in any particular order

▲ **Queue<T>**

- 👍 First-In-First-Out (FIFO) collection management
- 👎 Only allows access to the element at the head of the queue

▲ **PriorityQueue<TElement, TPriority>**

- 👍 Extends Queue by allowing elements to be processed based on priority rather than insertion order
- 👎 Every element must have a priority

▲ **Stack<T>**

- 👍 Provides Last-In-First-Out (LIFO) collection management
- 👎 Only allows access to the element at the top of the element

More random tips & tricks

- ▲ Modulo (%) is great for cycling through a list without concern for overflows
- ▲ Use the **in** argument to prevent copies of value types
- ▲ **goto** is an easy way to jump out of multiple iterators at once
- ▲ Don't initialize **char** arrays with defaults; **\0** is perfectly fine as comparison
- ▲ Use **Range ^1** to get the last item of a **String**, or **Array**
- ▲ Initialize list types with the right capacity to prevent internal resizing
- ▲ If a **char** of a string uniquely identifies the whole string, use only the **char**
 - ▲ e.g., instead of comparing "**red**", "**green**", "**blue**" match **[0]** with **'r'**, **'g'**, **'b'**

Don't overthink it...

```
static int NumberOfDigits(in long number) ⇒ number switch
{
    < 10L ⇒ 1,
    < 100L ⇒ 2,
    < 1000L ⇒ 3,
    < 10000L ⇒ 4,
    < 100000L ⇒ 5,
    < 1000000L ⇒ 6,
    < 10000000L ⇒ 7,
    < 100000000L ⇒ 8,
    < 1000000000L ⇒ 9,
    < 10000000000L ⇒ 10,
    < 100000000000L ⇒ 11,
    < 1000000000000L ⇒ 12,
    < 10000000000000L ⇒ 13,
    < 100000000000000L ⇒ 14,
    < 1000000000000000L ⇒ 15,
    < 10000000000000000L ⇒ 16,
    < 100000000000000000L ⇒ 17,
    < 1000000000000000000L ⇒ 18,
    _ ⇒ 19
};
```




**LEAST
COMMON MULTIPLE**

DAY 8

**CHINESE
REMAINDER
THEOREM**

imgflip.com

Useful algorithms to know

- ▲ Dijkstra's Algorithm
- ▲ A* (A-star) Algorithm
- ▲ Manhattan Distance (Taxicab Distance or City Block Distance)
- ▲ Least Common Multiple (LCM)
- ▲ Greatest Common Divisor (GCD)
- ▲ Shoelace Formula (Gauss's area formula)
- ▲ Kargers algorithm for minimum cut

*Always
measure....*



Code & Benchmarks.....

▲ <https://github.com/eNeRGy164/AdventOfPerformance>

```
      |  
     \|/  
    --*--  
   >o<  
  >0<<<  
 >>o>>*  
>o<<<o<<<  
>>@>*<<0<<<  
>o>>@>>o>o<<  
>*>>*<o<@<o<<<<  
>o>o<<<0<*>>*>0<  
  _ _ | | _ _  
Advent of Code
```


To follow...

▲ Dotnetos

▲ <https://www.youtube.com/@Dotnetos>

▲ Konrad Kokosa

▲ <https://twitter.com/konradkokosa>

▲ <https://tooslowexception.com/>

▲ Bartosz Adamczewski

▲ <https://twitter.com/badamczewski01>

▲ <https://leveluppp.ghost.io/net-infographics/>

▲ Michal Strehovský

▲ <https://twitter.com/MStrehovsky>

▲ <https://migeel.sk/>

```
  |
  \|/
--*--
 >o<
>0<<<
>>o>>*<
>o<<<o<<<
>>@>*<<0<<<
>o>>@>>>o>o<<
>*>>*<o<@<o<<<<
>o>o<<<0<*>>*>>0<
  _ _ | | _ _
```

Advent of Code

Advent of Code

▲ <https://adventofcode.com/>



Advent of Code



imgflip.com

Thank you!