Multiples of 2, 3 and 5

Robert Santana

Smoking Gun Interactive

Abstract

This paper presents the solution to the problem of "Determine the nth multiple of only $2, 3, 5$". The exposed problem requires the computation of the n^{th} multiple of only $2, 3, 5$. The proposed solution solves a generalization of the stated problem to allow any list of numbers (besides 2, 3 and 5) by implementing a linear strategy that compute all n_i multiples $(1 \leq i \leq n^{th})$ and stops when the multiple at position n^{th} has been determined. The solution complexity for the stated problem is proven to be $T(n) = O(n^{th})$, $S(n) = O(n^{th})$. The paper concludes in the mention of points of improvements and suggestion for next steps.

Keywords: multiples, nth multiple, prime numbers, math

Multiples of 2, 3 and 5 – Problem Description

Consider a series in ascending order that only consists of numbers that can be factored by any combination of $2, 3$ and 5. e.g. $1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15$ ...

For example, the numbers 7 (prime), 13 (prime) or $14 = 2 * 7$ (7 is not a valid factor), are not in the above series.

They are not factorable by $2, 3, 5$.

The number 1 is included.

For example, the number in position 18 would be 30:

1. 1
2. 2
3. 3
4. $4 = 2 * 2$
5. 5
6. $6 = 2 * 3$
7. $8 = 2 * 2 * 2$
8. $9 = 3 * 3$
9. $10 = 2 * 5$
10. $12 = 2 * 2 * 3$
11. $15 = 3 * 5$
12. $16 = 2 * 2 * 2 * 2$
13. $18 = 2 * 3 * 3$
14. $20 = 2 * 2 * 5$
15. $24 = 2 * 2 * 2 * 3$
16. $25 = 5 * 5$
17. $27 = 3 * 3 * 3$
18. $30 = 2 * 3 * 5$

Question: Design an algorithm to find the number that occupies position 1500 in this series.

NOTE: the correct answer is 859963392, use this to verify your algorithm.

Delivery Method:

The solution is delivered under the Multiples/ directory.

The dist/ folder contains the deliverable objects, composed of:

- A Windows Console application (Multiples.UI.exe), ready to run. It executes the service for the stated factors $(2, 3, 5)$ and iterates indefinitely asking the user to enter the position they wish to calculate.
- The final signed version of this document (Multiples of 2, 3 and 5.pdf)

The src/ folder contains the source code used to build the solution:

- The implementation of the solution is delivered as a service located in a .NET Standard Library named /Multiples.Core.
- The service is presented to the user as a Windows Console Application named /Multiples.UI that executes the service for the stated factors $(2, 3, 5)$ and iterates indefinitely asking the user to enter the position they wish to calculate.
- Additionally, an xUnit Test project named /Multiples.Core.Tests is provided to test the service.
- The Word version of this document (Multiples of 2, 3 and 5.docx)

Proposed Solution

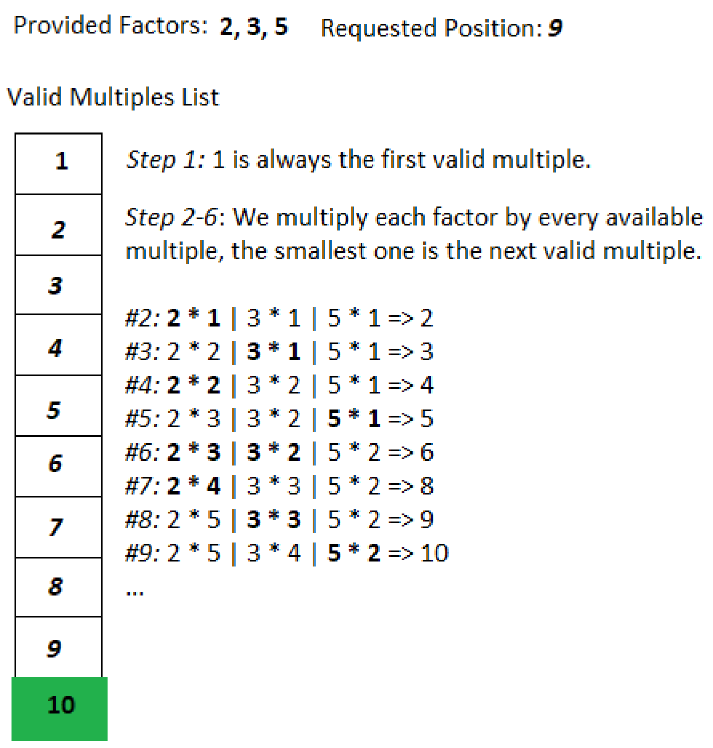
The proposed solution solves a generalization of the stated problem to allow any list of numbers (besides 2, 3 and 5). The solution computes all the valid multiples for the given factors until the multiple in the requested position is found. The operational complexity of the solution linearly depends on the position of the multiple to compute as well as the amount of numbers, although such could be considered constants for the problem in question.

The main concept of the problem is that every multiple of the list must be factored only by the provided factors, which means that if m is a valid multiple there must be a way to compose it as $m = \prod_{i=0}^{n} f_i^{F_i}$ where f_i is the i^{th} given factor and $0 \leq F_i \in N < \infty$.

Notice that finding a factorization that doesn't comply with the definition is not proof enough to rule out a number as a valid multiple. Ex.: With factors: 3, 4 and 8. 16 is a valid multiple given it can be factored as $16 = 4^2$ even when could be factored as $16 = 8 * 2$.

The outlined solution strategy is:

- The first valid multiple is always 1, by definition.
- Every provided number (hereon referred to as *factors*), is *always* a valid multiple.
- If m is a valid multiple, then $m' = f_i m$ is also a valid multiple.
- To determine the next multiple into the list, we take the smallest (not already computed) product between each $factor$ and an existing multiple in the list, starting by 1.
- We stop when the list is filled by the same number of elements as the requested position.
- The last element of the list is the desired multiple.



Provided Factors: **2, 3, 5** Requested Position: ***9***

Valid Multiples List

1 **2** **3** **4** **5** **6** **7** **8** **9** **10**

Step 1: 1 is always the first valid multiple.

Step 2-6: We multiply each factor by every available multiple, the smallest one is the next valid multiple.

#2: **2 * 1** | 3 * 1 | 5 * 1 => 2
#3: 2 * 2 | **3 * 1** | 5 * 1 => 3
#4: **2 * 2** | 3 * 2 | 5 * 1 => 4
#5: 2 * 3 | 3 * 2 | **5 * 1** => 5
#6: **2 * 3** | **3 * 2** | 5 * 2 => 6
#7: **2 * 4** | 3 * 3 | 5 * 2 => 8
#8: 2 * 5 | **3 * 3** | 5 * 2 => 9
#9: 2 * 5 | 3 * 4 | **5 * 2** => 10
…

To implement the strategy, we keep an internal state for each factor that stores a reference to the next already determined multiple and multiplies its values by the factor.

$$\{Factor = 5, \quad NextIndex = 10, \quad NextValue = Multiples[10] * 5\}$$

1 is always the first valid multiple, so the initial status of every factor is:

$$\{Factor = f_i, \quad NextIndex = 0, \quad NextValue = f_i\}$$

For every iteration, the next multiple into the list will be the smallest $F_i.NextValue$ among all the $factors$ that hasn't been added into the list yet. Ex.: We could have 2×5 and 5×2 if both, $2, 5$ are $factors$.

After a new multiple has been determined, the $NextIndex$ of the chosen factor is updated to point to the next multiple in the list $F_i.NextIndex = (F_i.NextIndex + 1)$

When $Multiples.Lenght = Position$ the value in $Multiples[Position - 1]$ is the desired result.

Proof of Correctness:

In order to prove the solution, let's first demonstrate the following 3 lemmas:

Lemma #1: "Every element in the list of Multiples is a valid multiple"

Note that a number only gets into the list if it can be expressed in the form of: $m = f_i m'$, where f_i is one of the provided factors and m' is a valid multiple.

We can prove this lemma by proving m is also a valid multiple, using the Principle of Mathematical Induction:

1. The 1st element of the list (1) is a valid multiple, following the problem definition.
2. We have that $m_j = f_i m_k$ with $k < j$ and $m_k < m_j$ given $f_i > 1$
3. Assuming m_k is a valid multiple, there's a way to compose it as:
   $m_k = f_1^{F_1} \times \ldots \times f_i^{F_i} \times \ldots f_n^{F_N}$, with $F_i \geq 0$,
   $m_j = f_i m_k = f_1^{F_1} \times \ldots \times f_i^{F_i+1} \times \ldots f_n^{F_N}$ which is also a valid multiple.

Lemma #2: "Every valid multiple, is contained in the list of Multiples"

Note that a number only gets into the list if it can be expressed in the form of: $m = f_i m'$, where f_i is one of the provided factors and m' is already in the list.

We can prove this lemma by proving every possible multiple can be generated following that formula and starting with only 1. We do this by using "Reductio ad absurdum":

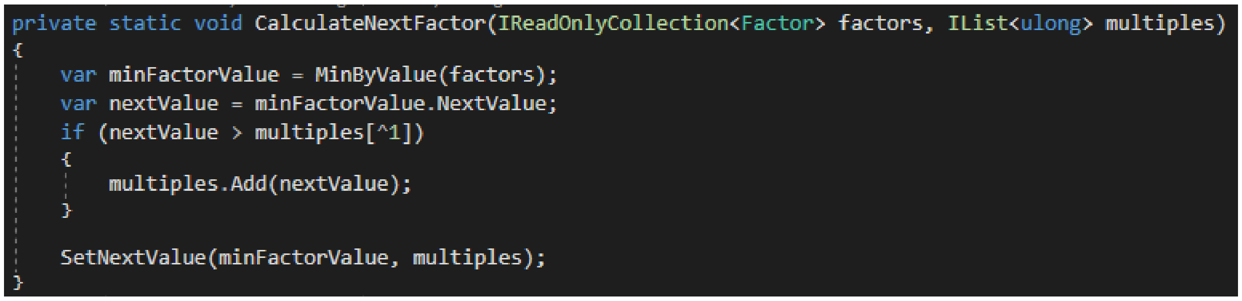
1. Let's take M as the smallest valid multiple of $Factors$ not included in $Multiples$.
2. We know that $M \notin \{1\} \cup Factors$ otherwise it would belong to $Multiples$.

- The first value of $Multiples$ is always 1.
- The first state of each factor is $\{ NextIndex = 0,\ NextValue = Factor \times Multiples[0] = Factor\}$
- Every time a value gets added into the list, the $NextIndex$ of its factor gets increased by one.
- Following the previous point and the fact that $Multiples$ is an ordered list, $NextValue$ is a *strictly monotone increasing* function, so eventually, $\forall(f \in Factors), f_i < F_i.NextValue$ and F_i will be added into the list.

3. M cannot be written as $M = f_j m_i$, where $m_i \in Multiples$, otherwise it would belong to $Multiples$:
   - $M > m_i$ given $f_j > 1$
   - $F_j.NextIndex$ traverses $Multiples$ in increments of 1, starting at 0.
   - $F_j.NextIndex$ will eventually become i, meaning that $F_j.NextValue = f_j m_i = M$.
   - Given $F.NextValue$ is a strictly *monotone increasing* function, $F_j.NextValue$ will eventually be the smallest value not yet added into the list, resulting into the addition of M to the list of $Multiples$.
4. After 4. we know that M is composed of at least $f_j, f_k \in Factors$, with possibly $j = k$:
5. We can then write say $M = f_j\left(f_0^{F^0} \times \ldots \times f_j^{F_{j-1}} \times \ldots f_n^{F_n}\right)$
6. Let's say $M' = f_0^{F^0} \times \ldots \times f_j^{F_{j-1}} \times \ldots f_n^{F_n}$, which, after 6. $M' > 1$ and a multiple of $Factors$.
7. Concluding $M = f_j M'$, where, M' is a multiple of $Factors$.
8. After 5. $M' \notin Multiples$ contradicting the hypothesis and proving this lemma.

Lemma #3: "Every element in the list is greater than the elements before it"

Note that a number only gets into the list if it is greater than the last element of the list.



```
private static void CalculateNextFactor(IReadOnlyCollection<Factor> factors, IList<ulong> multiples)
{
    var minFactorValue = MinByValue(factors);
    var nextValue = minFactorValue.NextValue;
    if (nextValue > multiples[^1])
    {
        multiples.Add(nextValue);
    }

    SetNextValue(minFactorValue, multiples);
}
```

Which clearly proves that, when an item is added into the list, it is the greatest element in the list so far.

After execution, the nth in the list is the requested multiple:

1. Multiples contains every possible valid multiple of Factors
2. $\forall(0 \leq i < j < \infty)$ Multiples[i] < Multiples[j]

3. <u>$\text{Multiples}[\text{position} - 1]$ is the valid multiple in the given position.</u>

Solution Complexity:

Spatial Complexity: $\boldsymbol{S(n)} = \mathbf{O}\big(\mathbf{Max}(\mathbf{position}, |\mathbf{Factors}|)\big)$

Proof:

1. We need to compute the list of every multiple until the list size is the same as the required position. At the end of the algorithm $Multiples.Lenght = position$.
2. We know that $sizeof(Multiples) = c_1 \times position$, where c_1 is the constant size of m.
3. We know that $sizeof(Factors) = c_2 \times |Factors|$, where c_2 is the constant size of each F.
4. Every other memory consumed is constant regardless the input so it can be represented as Q
5. We have that: $S(n) = \mathrm{c}_1 \times position + c_2 \times |Factors| + Q = \mathrm{O}(position) + \mathrm{O}(|Factors|) = \mathbf{O}\big(\mathbf{Max}(\mathbf{position}, |\mathbf{factors}|)\big)$

Time Complexity: $\boldsymbol{T(n)} = \mathbf{O}(|\mathbf{Factors}| \times \ \mathbf{position})$

Note: For the scope of the problem, $\mathbf{T(n)} = \mathbf{O}(|\{\mathbf{2},\mathbf{3},\mathbf{5}\}| \times \mathbf{position}) = \mathbf{O}(\mathbf{position})$

Proof:

1. The execution stops when $|Multiples| = position$.
2. Every iteration we increase a $F_i.NextIndex$ by one.
3. $F_i.NextIndex$ will always refer to an existing index in the list:
   - Let's consider the step prior to $F_i.NextIndex$ being increased.
   - If $F_i.NextIndex = |Multiples| - 1$ implies $F_i.NextValue = f_i \times Multiples[NextIndex] > Multiples[NextIndex]$ so a new item gets added to the list and $F_i.NextIndex + 1 = [Multiples]$ implying that $F_i.NextIndex = [Multiples] - 1$
   - If $F_i.NextIndex < |Multiples| - 1$ implies $F_i.NextIndex + 1 \leq |Multiples| - 1$
4. After 1., 2. and 3. the maximum amount of iterations will be $|Factors| \times position$.
5. Every iteration performs a constant number of operations c.
6. As consequence $T(n) = c \times |Factors| \times position = \mathbf{O}(|\boldsymbol{Factors}| \times \boldsymbol{position})$

Conclusion:

I couldn't prove $T(n) = \Omega(position)$ meaning there's the possibility of a better solution to the problem in the realm of $\Omega(\log(position))$ or lower. Same reasoning applies to $S(n)$.

Considering the scope of the problem with $Factors = \{2, 3, 5\}$, given that $T(n) = O(position)$ and $S(n) = O(position)$ the main computation limitation becomes the maximum possible representable multiple, which in the chosen framework is the 64-bit unsigned integer primitive type, which can hold up to the value $18{,}446{,}744{,}073{,}709{,}551{,}615$, which makes 13282 as the biggest possible position to compute. In order to compute larger positions, we would have to resort to the use of $BigNumbers$ possibly making the algorithm impractical.

Considering the relatively smaller maximum (13282) input size, both, the practical $S(n) \sim 105KB$ and $T(n)$ are small enough as to be considered acceptable solution for every scenario other than those that require the maximum possible optimizations.

As consequence, I made the decision to prioritize time to market, considering that further effort investment wouldn't result in a much higher ROI under most real-life scenarios.

Further research is recommended to either proof $T(n), S(n) = \Theta(position)$ or develop a more efficient solution if such investment is deemed required.

[illegible]

cn=Roberto Santana Perdomo, email=rsantanap@outlook.com, c=UY
2020.07.04 12:01:55 -03'00'