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 or 5”. It is divided on several sections, first, the problem is introduced, then the proposed solution is explained and proven. I conclude by highlighting points of improvements.

Keywords: multiples, nth multiple, prime numbers

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 extensively test the provided 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 be applied for any given list of factors (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.

The core idea of the solution 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 $0 \leq F_i \in N < \infty$.

Example:

1. With factors: 2, 3 and 5. 10 is a valid multiple given than $10 = 2 * 5$. But 14 is not because the only way to factor it out is as $14 = 2 * 7$ and 7 is not part of the given list.
2. With factors: 3, 4 and 8. 16 is a valid multiple given it can be factored as $16 = 4^2$ even when it can also be factored as $16 = 8 * 2$

The first multiple will always be 1 and it cannot be part of the $Factors$ list.

The 2nd multiple is the smaller provided factor.

To determine the next multiples, we ponder on the idea that if m is a valid multiple then $m' = m * f_i$ is also a valid multiple and we compute the list of every possible multiple by multiplying each of the factors by the previously determined multiples.

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

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

At the start, the smallest factor is always the 2nd multiple, so it's initial state is: $\{ Factor = SmallestFactor,\ NextIndex = 1,\ NextValue = SmallestFactor^2\}$ while the other factors are initialized to $\{ Factor = f_i,\ NextIndex = 0,\ NextValue = f_i\}$

For every iteration, the next multiple in the list is the smaller $F_i.NextValue$.

After a new multiple has been determine, the $NextIndex$ of the chosen factor (that with the smallest $NextValue$) gets updated as:

1. $NextIndex = NextIndex + 1$ if $NextIndex\ != 0$
2. $NextIndex = Multiples.Count - 1$ otherwise

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

The computed values remain stored so successive executions of the solution won't recompute already determined multiples.

Proof of Correctness:

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

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

We can proof this lemma using the Principle of Mathematic Induction:

1. The 1st multiple (1) is a valid multiple, following the problem definition.
2. The 2nd multiple is always the smallest factor and $f_i = f_i$, so it's a valid multiple.
3. We have that $m_j = f_i * m_{k<j}$ and assuming $m_{k<j}$ is a valid multiple, then m_j is also a valid multiple, which proofs this lemma.

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

We can proof this lemma using "Reductio ad absurdum":

1. Let's take the smallest M which is a valid multiple of $Factors$ but not included in $Multiples$.
2. There's a way to write it as $M = \prod_{i=0}^{n} f_i^{F_i}$ where f_i is the i^{th} given factor and $0 \leq F_i \in N < \infty$ by definition of a valid multiple.
3. We know that $M \notin \{1\} \cup Factors$ otherwise it would belong to $Multiples$.
4. M cannot be written as $M = f_j * m_i$, where $m_i \in Multiples$, otherwise it would belong to $Multiples$.
5. After 3. we know that M is composed of at least $f_j, f_k \in Factors$.
6. After 5. we know that $M = f_j * \left(f_0^{F^0} * \dots * f_j^{F_{j-1}} * \dots f_n^{F_n}\right)$.
7. Let's say $M' = f_0^{F^0} * \dots * f_j^{F_{j-1}} * \dots f_n^{F_n}$, which, after 5. $M' > 1$ and a multiple of $Factors$.
8. After 6. $M = f_j M'$, where, because of 7. M' is a multiple of $Factors$ contradicting 4 and proving this lemma.

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

We can proof this lemma by using "Reductio ad absurdum".

1. Let's take $m_j \in Multiples$ as the smallest multiple with an $m_i \in Multiples \ \wedge i < j$
2. We know that:
   a. $m_j = f_a m_k$, with $1 \leq k < j$
   b. $m_i = f_b m_l$, with $1 \leq l < i < j$
   c. And, $f_a m_k < f_b m_l$
3. When each value was added to the list, its factor status was:
   a. m_i: $F_b = \{ Factor = f_b,\ NextIndex = l, NextValue = f_b m_l\}$
   b. m_j: $F_a = \{ Factor = f_a, NextIndex = k, NextValue = f_a m_k\}$
4. m_i was added to the list before m_j by hypothesis.
5. Let's call $o = F_a.NextIndex$, then $o < k$:
   a. If $f_b = f_a$ then $m_i = f_a m_l$, with $l \neq k$, otherwise $m_i = m_j$. Furthermore $l < k$ because $NextIndex$ is a monotonous increasing function.
   b. If $f_b \neq f_a$ then $f_b m_l < f_a m_o$ by solution definition implying that $o < k$ given that $NextIndex$ is a monotonous increasing function and $f_b m_l > f_a m_k$
6. By hypothesis $m_k > m_o$, otherwise m_j wouldn't be the first multiple lesser than m_i. So:
   a. $m_k = m_o + c$
   b. $m_i \leq f_a * m_o$ implies $m_i \leq f_a * (m_o + c)$ implies $m_i \leq f_a * m_k$ implies $m_i \leq m_j$ which contradicts the hypothesis and proves the lemma.

Proof of Correctness:

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

Solution Complexity:

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

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 $sizeof(Multiples) = c_1 * position$, where c_1 is the constant size of m.
3. We know that $sizeof(Factors) = c_2 * |Factors|$, 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: $S(n) = c_1 * position + c_2 * |Factors| + Q = \Theta(position) + \Theta(|Factors|) = \mathbf{\Theta}(\mathbf{Max}(\mathbf{position}, |\mathbf{factors}|))$

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

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

Proof:

1. The execution stops when $|Multiples| = position$.
2. Every iteration we increase one $F_i.NextIndex$ 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 $F_i.NextValue = f_i * Multiples[NextIndex] > Multiples[NextIndex]$ 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 $|Factors| * position$.
5. Every iteration performs a constant number of operations c.
6. As consequence $T(n) = c * |Factors| * position = \mathbf{O(|Factors| * position)}$

Conclusion:

I couldn't proof $T(n) = \Omega(position)$ 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 $T(n) = \mathrm{O}(position)$ and $S(n) = \Theta(position)$ 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., being

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.

cn=Roberto Santana Perdomo, email=rsantanap@outlook.com, c=UY
2020.07.01 21:41:37 -03'00'