A Social Media Combinatorial Problem

Kedar Mhaswade
kedar.mhaswade@gmail.com

Abstract

This article discusses an algorithmic approach and a computer program based on it to solve an elementary combinatorial problem popular on social media. Its intended audience are high school students with some aptitude for abstract thinking and an interest in problem-solving.

1 Introduction

Consider the following puzzle[1]:

Puzzle 1 *How can you make* 24 *using the numbers* $1, 3, 4,$ *and* 6 *exactly once each, and the four arithmetic operations* $(+, -, \times, \div)$ *any number of times*[a]*?*

[a] Using a number formed by concatenating two or more numbers (e.g. 13) is disallowed

A ready-made answer is probably just an Internet search away. Just for fun, however, one should find such an arithmetic expression without searching it on the Web. Give it a try before you read further. Don't worry, the author took quite some time to find it.

1.1 Initial Impressions

This puzzle is surprisingly challenging, especially if one is not well-versed with solving such puzzles. We take arithmetic for granted, but the four basic operations combine with four small numbers to render it tantalizingly difficult. This is so even if you could somehow solve it.

In our attempts, we may start with some heuristics: If we can make an expression using 1, 3, and 4 that evaluates to 4, then we can make 24 by multiplying it by 6. But now we have a different puzzle that is just as tricky.

We may try parenthesizing parts of shorter expressions and combining their results (e.g. $((6 + 4) - 1) \times 3 = 27$). After toiling for a while we might *luckily* find the magic expression.

But we may not. What if no such expression exists? Can we tell if one exists at all? If more than one solution exist, can we find them all? Can we even tell if we have examined *all* expressions possible?

We keep *feeling* inadequate as we continue searching ...

[1] I first heard it from Sunil Singh, a mathematics educator

2 An Algorithmic Approach to Searching

As you keep searching, you may feel frustrated. After all, what we do is form expressions of four small numbers and four familiar arithmetic signs and evaluate them on the fly. It gets quite mechanical and hence boring after a short while. But we tend not to give up for a perceived shame grips us. After all, we have been seeing such numbers and $+, -, \times, \div$ since first grade!

Even after eventually finding the expression, we keep wondering if we weren't lucky. In case we are about to abandon the search, we seem to lack the confidence that we have indeed examined every expression.

Maybe we have picked up some unhelpful compliance in our childhood and carried them along unknowingly. Perhaps we should develop a different perspective. But, most of all, we feel the need of an assistance from a machine that carries out calculations correctly and, well, mechanically. It's a solace that just such an accomplice exists: A computer!

A computer is astoundingly fast at performing known calculations precisely and, at the same time, naturally incapable of possessing intelligence or originality (even when there is a tremendous pressure on us from modern Artificial Intelligence techniques on proving otherwise) in devising *abstractions*, the basis of computing. Alan Perlis once said [3] that, like a few other things, a computer, by definition, is *explicitly programmed*.

Typically, a computer depends on us to tell it something (worthwhile) to do. And it needs precise, formal instructions. Or, more factually, we have got to believe that it follows our instructions to the letter. If the outcome of those instructions is different from what we expect, then we must doubt our instructions and it is up to us to modify them. We must not doubt if the computer followed them dutifully and precisely. To our dismay or to our pleasant surprise, "AI-powered" computers have started showing signs of a startling but *disembodied* intelligence, but we will save that topic for later.

Clearly, we need an *algorithm* to solve every problem we delegate to the computer. Perhaps surprisingly, however, the idea of an algorithm does not require a computer. According to Donald Knuth [1], Euclid described the first "computer algorithm", known as Euclid's GCD algorithm, 2300 years ago!

So, how do we fit this puzzle within the confines of the strict formalism that a computer demands? If only we could get rid of those pesky parentheses and mnemonic-laden[2] ways of remembering the so-called *operator precedence* ...

2.1 Some Formal Background

To develop a perspective, we need to precisely define *expression*, *operator*, and *operand*. An operator has a symbol and describes an operation on a number of operands which are just numbers in our case. Every operation *takes* a number of operands and *produces* a result (or results) that is, in many ways, like its operands–again, numbers in our case. Depending upon how many operands the operation takes, an operator can be *unary* (takes one operand), *binary* (takes two operands), etc. This property of an operator is called its *arity*.

We learn in grade school that unary operator symbols are written either before

[2] Remember "BODMAS"?

the numbers (e.g. $\sqrt{8}$ to denote the square root of 8) or after the numbers (e.g. 3! to denote the factorial of 3), whereas the binary operator symbols are written between them ($2+3=5$). For binary operators, such an expression is called an *infix* expression.

The four basic arithmetic operations–addition, subtraction, multiplication, division–are all binary. They take two numbers and produce a number. That helps us combine them sequentially to form expressions. Each expression can be *evaluated* to produce a value which is a number in our case.

Expressions such as $2+3+8+7$ achieve brevity[3] which aids comprehension. However, ambiguity arises when different operators appear in the same expression: $1+2\times3$ because we don't know whether to add add first and then multiply (producing 9) or vice versa (producing 7). To address the ambiguity, we may make a rule like: *Do the operations in the order of their appearance.*

Instead, we use parentheses to disambiguate: $(1+2)\times3$ evaluates to 9 and $1+(2\times3)$ to 7. If we do not want to introduce parentheses, however, we need to define *precedence* of operators. This is where the mnemonics like "BODMAS" originate. Infix notation is clearly a rules-heavy way to denote and evaluate expressions. But is that the only way?

2.2 Postfix Expressions and the Stack

As students, we should develop the habit of *meaningfully challenging the status quo.* History of mathematics and other sciences teaches us about people who dared to explore and we learn a great deal about trailblazing from them.

The Polish logician Jan Łukasiewicz[4] proposed a different idea for the notation[2]: **Place the symbol after (or before) the operands**. We call such an expression a *postfix*[5] (or *prefix*) expression. Thus, the postfix expression corresponding to the first expression above is $1\ 2+3\times$ and that to the second is $1\ 2\ 3\times+$.

To evaluate expressions, we propose the idea of a *stack.* A stack is an abstract structure that resembles a stack of dishes we often see at restaurants. If we only allow adding and removing a dish to and from the top of the stack then it makes a dish added last to be removed first. This view makes stack a linear queue whose elements are added and removed only from one of its two ends–the top[6].

Two operations on a stack are essential: 1) `push element`, which pushes the given element on the stack, and 2) `pop` which simply removes the element at the top of the stack, and returns its value. Doing the `push` operation once increments the *size* of the stack whereas doing the `pop` operation once decrements it.

If you have never encountered a stack before, you may not be impressed with the immense power its simplicity offers to *model* a number of practical situations. But that is the usefulness of abstractions: It doesn't matter to the stack what it is employed to store (dishes, objects, or numbers). It provides a structure to store it and the two essential operations: `push` and `pop` defined above. Optionally, we may want to know if the stack is empty (in which case we can't `pop` an element) or the stack is full (which can happen in practice where resources are limited and we can't `push` an element).

[3] Imagine the plight of reading $2+3;5+8;13+7$ instead

[4] Pronounced *yahn woo-ka-shay-vitch*

[5] It's also called the *Reverse Polish notation*

[6] A stack is therefore called a LIFO–Last In First Out–queue

Perhaps you are thinking of postfix expressions as an unnecessary digression. You may be wondering why a postfix expression like: $1\ 2\ 3\ \times\ +$ is more convenient than the corresponding infix expression: $1 + (2 \times 3)$.

And one might even successfully argue that infix expressions are easier to read. But isn't readability a question of getting used to? Had we been reading postfix expressions since grade school, we might have become more comfortable with them. Is there another benefit?

It turns out that postfix expressions need no operator precedence rules, and, as a result, they do not need parenthesizing expressions to enforce precedence. A given postfix expression can be evaluated *mechanically* (with the help of the stack)! This bonus becomes apparent when we study the *algorithmic* nature of evaluating postfix expressions.

We start with a postfix expression. For the sake of our algorithm we read it a *token* at a time. A token is either an operand or an operator. Our operators are all binary. However, the scheme can be easily extended to other operators (e.g. unary). We use the stack only to store operands and results of operations. Thus, ours is a stack of numbers. Here is our Algorithm[1].

Algorithm 1: Evaluate any postfix expression with binary operators

```
1:  S ← Stack.New
2:  T ← [t1 t2 ...]
3:  while T has tokens do                    ▷ Start scanning the expression from left
4:      t ← T.next
5:      if t is an operand then
6:          S.push(t)
7:      else if t is an operator then                          ▷ defensive programming
8:          o2 ← S.pop()                                       ▷ Pop two operands
9:          o1 ← S.pop()                          ▷ Order matters; pop second operand first
                                  ▷ It's an error if S doesn't have enough operands to pop
10:         r ← operator(od1, od2)                             ▷ e.g. 2 + 3 and r ← 5
11:         S.push(r)
12:     else
13:         return Error                                       ▷ Invalid postfix expression?
14: if S.empty()? then
15:     return Error                                           ▷ S must have the result
16: else
17:     return S.pop()
```

2.3 A Helpful Insight?

Infix and postfix (or prefix), after all they are just notations. And it so happens that one can deterministically translate a valid infix expression into postfix and vice versa. However, removing the need for parentheses and providing a mechanical way of evaluating are strengths of the postfix expression which the author used to address our puzzle 1. Let's list the helpful facts that we know so far. They may be disparate at the moment,

but they may align themselves in a sort of coherent whole later:

> We want to employ a reliable method to carry out the *exhaustive* search for all expressions involving given numbers $(1, 3, 4, 6)$ and (binary) arithmetic operators $(+, -, \times, \div)$. Can postfix notation help?
>
> Since there are four operands and each binary operator takes two operands and produces a result (which may then pair up with next operand), we must have exactly three binary operators in an expression with seven tokens.

We have four operands that can appear in any order in a postfix expression. Since we must have exactly three binary operators to operate on them, there are seven places each of which is occupied either by an operator or an operand.

Here is a possible arrangement:

Fig 1: A Valid Arrangement

Luckily, this is a valid postfix expression that evaluates to 1. We can trace the run of our algorithm [1] on this expression to confirm the result.

Here is another possible arrangement:

Fig 2: An Invalid Arrangement

We are unlucky in this case because this arrangement is not a valid postfix expression; it does not evaluate to anything. Expectedly, our algorithm [1] returns an error in this case. But a computer doesn't care and perhaps neither should we, because we can ignore this arrangement as a "non expression" and move on to examining the next one. It would be nice to know a priori that such an expression is invalid (without actually evaluating it with our algorithm [1]), but that is not necessary. Human mind may *immediately* discard all the expressions like [2] that begin with an operator (because there is nothing to operate it on). But a computer algorithm can sometimes beat that ingenuity with shear speed of (possibly needless) calculation. This does not mean we write sloppy algorithms. In fact, we should always look for opportunities to optimize algorithms once their correctness is established.

2.4 The Algorithm

Perhaps we now feel closer to describing a foolproof[7] method of solving our puzzle to the computer. Our search has got to be exhaustive; we must not miss any (valid) arrangement of four operands and three binary operators. We evaluate each expression. If an expression evaluates to our target number (24), we have succeeded. If there is no

[7] Being foolproof is required because a computer is going to carry it out

such expression, the puzzle is unsolvable. An algorithm seems to emerge. We just need to describe the process precisely ...

We are given a set of operands, but where they appear in the seven locations (their *order*) matters. This is a archetypal situation leading us to a fascinating area of mathematics called Combinatorics. Specifically, we can arrange k objects in n locations[8] in ${}_nP_k = \frac{n!}{(n-k)!}$ ways. In our case, the number of such *ordered arrangements* of four operands in seven locations $= \frac{7!}{3!} = 7 \times 6 \times 5 \times 4 = 840$. Here are some of the arrangements:

1	3	4	6			
1	3	6	4			
...	...	...	...	...	...	...
1	4		3		6	
...	...	...	...	...	...	...
		1	3		6	4
...	...	...	...	...	...	...
			6	4	3	1

Fig 3: All Permutations of Four in Seven

These are *all* the possible ways in which we can arrange members of the set of operands $\{1, 3, 4, 6\}$ in seven places and there are exactly 840 such arrangements. Some arrangements will invariably lead to invalid postfix expressions (e.g. the permutations in red in Fig [3]) after we place the operator symbols in remaining places.

How do we place operator symbols? Well, we have three places and we want to place three operators in them. However, this time repetition is allowed because we can choose the same operator more than once. Such arrangements where members of an n-set are placed in k places (where selecting the same set member is allowed) is called k-tuples [4] of the n-set. The number of k-tuples is n^k. In our case $k = 3, n = 4, n^k = 64$. We have 64 3-tuples of 4 operators to consider.

Let's demonstrate the placement of all operator 3-tuples with one arrangement of operands, [1,4,␣,3,␣,6,␣], (shown in green) from Figure 3.

1	4	+	3	+	6	+	$= 14$
1	4	+	3	+	6	−	$= 2$
1	4	+	3	+	6	×	$= 48$
1	4	+	3	+	6	÷	$= 1.\overline{333}$

[8] Each such arrangement is called a *Permutation*

1	4	−	3	+	6	+	$= 6$
1	4	−	3	+	6	−	$= -6$
...	...	...	...	...	...	...	...
1	4	÷	3	÷	6	×	$= 0.5$
1	4	÷	3	÷	6	÷	$= 0.13\overline{888}$

Fig 4: All Postfix Expressions for the Operand Permutation $[1, 4, 3, 6]$

An upper bound on all 7-token postfix expressions thus formed is: $64 \times 840 = 53760$. We examine every expression and, if it is valid, check the result it produces. Some *optimizations* can reduce the number of expressions we need to evaluate. Here is our algorithm [2]:

Algorithm 2: Examine All Postfix Expressions with Given Operands and Binary Operators

```
g ← 24                                          ▷ Target result
P ← perms()                                     ▷ P ← [P_1, P_2, ..., P_840]
T ← tuples()                                    ▷ [t_1, t_2, ..., t_64]
for all p ∈ P do
    for all t ∈ T do
        e ← expr(p, t)          ▷ expression from given permutation and tuple
        r ← eval(e)                             ▷ evaluate using algorithm [1]
        if r = g then
            Output: e
```

2.5 An Implementation

3 Summary and Conclusion

4 Answer to the Puzzle

$$((4 \div 3) - 1) \div 9 \quad \mathbf{1}$$

References

[1] Donald Knuth. *The Art of Computer Programming. Fundamental Algorithms*. Vol. 1. Addison-Wesley Professional, 1997 (cit. on p. 2).

[2] Hamblin C. L. "Translation to and from Polish Notation". *The Computer Journal* 5.3 (1962), pp. 210–213. DOI: 10.1093/comjnl/5.3.210 (cit. on p. 3).

[3] Alan Perlis. *Foreword to Structure and Interpretation of Computer Programs*. 1984. URL: https://sourceacademy.org/sicpjs/foreword84 (visited on 02/09/2025) (cit. on p. 2).

[4] *Permutation*. Feb. 12, 2025. URL: https://en.wikipedia.org/wiki/Permutation#Permutations_with_repetition (cit. on p. 6).