# WHOLE NUMBER ARITHMETIC (+, −, x, ÷) IN THE ICONIC CALCULATOR

William Bricken
June 2012

Boundary integers are collections of indistinguishable units within a common container. These tally collections can be reduced by grouping to a depth-notation form that supports any base system.  Base-2 and base-10 are illustrated.

---

## BOUNDARY RULES

| | | |
|---|---|---|
| ● ... ● = (●) | GROUP/unGROUP | defines base |
| (A)(B) = (A B) | MERGE/unMERGE | manages base |
| ○ ● = | CANCEL | negative/positive numbers |
| ( ) = | EMPTY | deletes empty containers |

| | |
|---|---|
| [A] | highlight for prepare to unGROUP |
| (A}{B) | highlight for prepare to unMERGE |

## TRANSCRIBE FROM STANDARD ARITHMETIC

| | |
|---|---|
| A ± B ==> A B | PUT TOGETHER (Additive Principle) |
| A x B ==> [B ● A] | SUBSTITUTE B for ● in A (Mult Principle) |
| A ÷ B ==> [● B A] | SUBSTITUTE ● for B in A |

| | |
|---|---|
| * | highlight for substitutions in multiply/divide |
| [X Y Z] | shorthand notation for Substitute X for Y in Z. |

## SPECIFICS FOR TALLYS AND GROUPS

| | |
|---|---|
| ●● = (●) | GROUP base 2 |
| ●●●●●●●●●● = (●) | GROUP base 10 |

| | |
|---|---|
| C = A B | SPLIT/unSPLIT (sum less than base) |
| −C = −A −B | SPLIT/unSPLIT (sum less than base) |

| | |
|---|---|
| 0 ==> | |
| 1 ==> | ● |
| 2 ==> | ●● = (●) |
| 3 ==> | ●●● = (●) ● |
| 4 ==> | ●●●● = (●)(●) = (●●) = ((●)) |

| | |
|---|---|
| −1 ==> | ◊ |
| −2 ==> | ◊◊ = (◊) |
| −3 ==> | ◊◊◊ = (◊) ◊ |
| −4 ==> | ◊◊◊◊ = (◊)(◊) = (◊◊) = ((◊)) |

---

Capital Letter Notation: The capital letters in MERGE stand in place of any unit or bounded form, so long as one exists. Within the boundary rules, variables cannot be void. A capital letter used during transcription (not transformation) can stand in place of any unit or bounded form, or it can be void since void is substituted for conventional zero during transcription.

## Model

The model is that arithmetic operations are easy/trivial. Maintaining a base system takes effort. The base system is a grouping mechanism that collects groups of a specified size and puts them into a container. The triviality of arithmetic is expressed by three principles:

ADDITIVE PRINCIPLE: A sum looks like its parts.

SUBTRACTIVE PRINCIPLE: Subtraction cancels polar units.

MULTIPLICATIVE PRINCIPLE: Multiplication is substitution of groups for units.

## ARITHMETIC TRANSCRIPTION

Base1 is tally arithmetic with no grouping. In base1, the engine uses three rules:

A + B ==> A B Put into Same Container

Addition/Subtraction is transcribed into PUT INTO SAME CONTAINER. PUT INTO SAME CONTAINER is more of a parsing step than a rule. “A” and “B” stand in place of any collection of dots/marks or nothing at all. PUT works for any kind of form, including positive and negative units and boundary integers. The container is often implicit as the display space framed by the typographical page or by the indentation of a figure.

○ ● ==> Cancel Opposites

CANCEL OPPOSITES achieves subtraction. It’s in boundary notation cause “N + -N” is not quite right.

A x B ==> Substitute B for each unit in A

Using the shorthand notation for substitution,

a x b ==> [b • a] Substitute

a ÷ b ==> [• b a] Substitute

SUBSTITUTE achieves multiplication and division.

These transcription rules are more like physical interpretations than arbitrary manipulations. The convert the abstract operations of arithmetic into physically realizable actions.

## BASE SYSTEM RULES

In any base other than 1 (eg BASE 2 and BASE 10dots) two additional “clean up” rules implement and manage depth-value notation.

● ... ● ==> (●) Group

GROUP achieves construction of a base. It can occur anywhere there is a sufficiently large group. The ellipsis stands in place of the number of units required to form a base group. So

● ● ==> (●) Group Base2

● ● ● ● ● ● ● ● ● ● ==> (●) Group Base10

Boundary forms accommodate any base, and also support mixed bases within the same form (this occurs in division).

(A)(B) ==> (A B) Merge Boundaries

MERGE BOUNDARIES implements depth-value bookkeeping by maintaining the right order of magnitude (defined by the base) for each unit. MERGE is also called COMMON BOUNDARIES CANCEL. This rule triggers whenever two bounded objects share the same container, at any depth of nesting. “A” and ”B” stand for any content but not no content. In boundary arithmetic, there are no “empty” containers.

Almost all of the computational work is in maintaining a base system. For hand manipulation of base10 dots, students need to know how to make groups of ten. This requires five pattern rules. The shape of dot configurations is an open design question.

● ●●●●●●●●● ==> (●)
●● ●●●●●●●● ==> (●)
●●● ●●●●●●● ==> (●)
●●●● ●●●●●● ==> (●)
●●●●● ●●●●● ==> (●)

## Addition/Subtraction of Digits

When common digits are introduced, they come with a price. Digits permit groups of units to the symbolized abstractly, and any symbolic abstraction comes with a load on memory. In particular, digits require memorization of the digit addition and multiplication tables. In conventional arithmetic, these are 10x10 tables, with many symmetries to reduce the number of entries at the cost of memorizing other abstractions.

The representation used by the Iconic Calculator makes many of these symmetries invisible.

```
-- since forms are in space, there is no commutativity     (100 -> 55)
-- since there is no zero, there are no add-zero rules     ( 55 -> 45)
-- with the GROUP operation, no additions are more than 10 ( 45 -> 25)
```

For example, 8  9  =  8  2  7  =  (1) 7

To use digits, students need to know 25 addition facts.

```
1+1  1+2  1+3  1+4  1+5  1+6  1+7  1+8  1+9
     2+2  2+3  2+4  2+5  2+6  2+7  2+8
          3+3  3+4  3+5  3+6  3+7
               4+4  4+5  4+6
                    5+5
```

These 25 facts subdivide into

```
--  5 add-to-ten facts  (along the right side)
--  9 add-one facts     (along the top)
-- 11 digit split/unsplit facts
```

The other memory intensive skill is management of place value notation. This is fully taken care of by MERGE BOUNDARIES and GROUP into tens.

For example, 57+89 = 146

```
(5) 7 (8) 9           transcribe
(5)( 8) 7 9           linear
(5  8 ) 7  9          merge
(5 5 3) 7 3 6         split       optionally (3 2 8) 6 1 9
((1) 3) (1) 6         group
((1) 3   1) 6         merge
((1) 4    ) 6         unsplit
```

Note that “carrying” is GROUP followed by MERGE. Each container space is independent of the others, so that all operations within specific containers can occur in parallel.

Subtraction requires splitting digits to match positives and negatives in each space. This results in containers that may contain positives mixed with containers that contain negatives. A “borrowing” transformation is need to reduce the mixed notation. Borrowing is running MERGE and GROUPS backwards, i.e. UNMERGE and UNGROUP. The “10” shows up as two numerals, one of which CANCELS the outer negatives. Eg,

```
(  4  )
(3   1)               split
(3) (1)               unmerge
(3) 10                ungroup
```

Example, 84 – 19:

```
(8) 4 (–1) –9         transcribe
(8) (–1) 4 –9         linear
(8   –1) 4 –9         merge
(7 1 –1) 4 –4 –5      split to cancel x 2
(7     )      –5      cancel
(6   1 )      –5      split
(6)  (1)      –5      unmerge
(6)  5 5      –5      ungroup
(6)  5                cancel
```

## Multiplication via Substitute

Multiplication requires collections to be replicated. This is simple in Base-1 and requires management in other bases. Base-10-unit examples:

```
3x2   •• x •••         [•••  •  ••] ==> ••• •••

6x4   •••••• x ••••    [••••  •  ••••••] ==> •••• •••• ••••
                                             •••• •••• ••••
                                         ==> (•)••(•)•• ==> (••)••••
```

A base-2 example:

```
5: ((•))•        7: ((•)•)•

5*7:  [7 • 5]     ((7)) 7          mixed notation
7*5:  [5 • 7]     ((5) 5) 5        mixed notation
```

G = group base-2     M = merge     S = substitute     L = linear artifact

5*7: substitute 7 for • in 5

```
((    *    ))     *              [((•)•)•   •   ((•))•]
(( ((•)•)• }} {{•)•)•            S
(( ((•)•)•      •)•)•            Mx2
(( ((•)•}  {•)   )•)•            G
(( ((•)•    •)   )•)•            M
(( ((•}  {•) )   )•)•            G
(( ((•    •) )   )•)•            M
(( ((  (•) ) )   )•)•            G
```

7*5: substitute 5 for • in 7

```
((    *   )    *    )    *       [((•))•   •   ((•)•)•]
(( ((•))• } {(•) )• } {( •) )•   S
(( ((•))•    (•) )•    ( •) )•   Mx2
(( ((•)}     {•)•}     { •)•)•   L
(( ((•)       •)•       •)•)•    Mx2   identical to line 3 of 5*7
(( ((•)       •}   {•)    )•)•   G
(( ((•)       •     •)    )•)•   M
(( ((•}        {•)    )   )•)•   G
(( ((•          •)    )   )•)•   M
(( ((    (•)    )     )   )•)•   G
```

A student needs to memorize 36 digit multiplication rules (given 0 and 1 are trivial and commutative symmetry)

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 2x2 | 2x3 | 2x4 | 2x5 | 2x6 | 2x7 | 2x8 | 2x9 |
| | 3x3 | 3x4 | 3x5 | 3x6 | 3x7 | 3x8 | 3x9 |
| | | 4x4 | 4x5 | 4x6 | 4x7 | 4x8 | 4x9 |
| | | | 5x5 | 5x6 | 5x7 | 5x8 | 5x9 |
| | | | | 6x6 | 6x7 | 6x8 | 6x9 |
| | | | | | 7x7 | 7x8 | 7x9 |
| | | | | | | 8x8 | 8x9 |
| | | | | | | | 9x9 |

## Multi-digit Multiplication

Depth-value handles numbers greater than 10. Eg 43x26

```
[(••)••••••  •  (••••)•••]  ==>

((••)•••••• (••)•••••• (••)•••••• (••)••••••) (••)•••••• (••)•••••• (••)••••••
```

```
((●●●●●●●●) ●●●●●●●●●● ●●●●●●●●●● ●●●●) (●●●●●●) ●●●●●●●●●● ●●●●●●●●
((●●●●●●●●) (●)        (●)        ●●●●) (●●●●●●) (●)        ●●●●●●●●
((●●●●●●●● ● ●) ●●●● ●●●●●● ●) ●●●●●●●●
((     (●)    )     (●)     ●) ●●●●●●●●
((     (●)           ●)     ●) ●●●●●●●●  ==>   (((●) ●) ●) ●●●●●●●●
```

This, of course, is pretty horrible, and digits work more nicely. The overhead again is memorizing digit multiplication rules.

Here in mixed notation, four replicates of (2)6 replace the four units represented by the numeral 4. Three replicates replace the 3.

```
43x26:      [(2)6  1  (4)3]  ==>  [(2)6   1   (1 1 1 1)1 1 1]

      (   *     *     *     *   )   *     *     *
      ( (2)6  (2)6  (2)6  (2)6  ) (2)6  (2)6  (2)6    subst
      ( (2}{2}{2}{2) 6  6   6  6) (2}{2}{2) 6  6  6   linear
      ( (2  2  2  2) 6  6   6  6) (2  2  2) 6  6  6   mergex5
      ( (     8    ) 6 4 2 2 4 6) (2  2  2) 6 4 2 6   split, unsplit
      ( (     8    ) (1) 2 2 (1)) (2  2  2) (1) 2 6   groupx3
      ( (     8    ) (1) 2 2 (1)) (2  2  2) (1)  8    unsplit
      ( (     8    } {1} {1) 2 2} {2  2  2} {1)  8    linear
      ( (     8       1   1) 2 2   2  2  2   1)  8    mergex4
      ( (        (1)       }      {1)        1)  8    groupx2
      ( (        (1)              1)        1)  8    merge
      (((1) 1) 1) 8
```

This method does not require multiplication rules, but of course you may have to generate up to nine replicates at each depth. An alternative is to use multiplication rules. The numeral being substituted into then becomes an operator. Below the “times” concept represented by “x” is made explicit in a mixed notation:

```
[(2)6  1  (4x*)3x*]

      ( 4x (  2)   6 ) 3x (  2)    6            mixed
      (    (4x2) 4x6 )    (3x2)  3x6            mixed
      (    ( 8 }{2)4 }    { 6 } {1)8            mult
      (    ( 8   2)4        6    1)8            Mx3
      (    (  (1) }    {1)       1)8            Gx2
      (    (  (1)       1)       1)8            merge
```

The difference between making replicates and using digit multiplication rules goes away quickly after the first few steps. Multiplication can be propagated into depth notation in one step. Every digit in each level of the insert multiplies every digit in each level of the insertee. Below, the insertee is highlighted with larger font.

```
((4) 3) 8  x  ((6) 9) 2  ==>  [((6)9)2   1   ((4)3)8]

(( (( *x4)  *x4) *x4 ) (( *x3)  *x3) *x3 ) (( *x8)  *x8)  *x8    subst
(( (( 6x4)  9x4) 2x4 ) (( 6x3)  9x3) 2x3 ) (( 6x8)  9x8)  2x8    subst
(( (((2)4} {3)6)   8 } {((1)8) (2)7)   6 } {((4)8} {7)2) (1)6    mult
(( (((2)4   3)6)   8    ((1)8   2)7)   6    ((4)8   7)2   1)6    Mx4
```

The above step is partial cause typing in a line does not clearly expose all the MERGING boundaries. More completely, there are eight MERGES that would occur in one step. Two linear tidy steps expose the remaining four concurrent merges:

```
(( (((2) 4 3) 6} {(1) 8 2) 8 7} {(4) 8 7    6 2 1) 6        linear tidy
(( (((2) 4 3) 6   (1) 8 2) 8 7   (4) 8 7  ) 6 2 1) 6        Mx2
(( (((2) 4 3}   {1) 6 8 2} {4)   8 7 8 7  ) 6 2 1) 6        linear tidy
(( (((2) 4 3     1) 6 8 2   4)   8 7 8 7  ) 6 2 1) 6        Mx2
(( (((2) 4 3     1) 6 4 8 2  ) 5 5 3 7 3 7) 6 2 1) 6        split **
(( (((2) 4 3     1} {1} {1)  } {1} {1} {1)) 6 2 1) 6        Gx5
(( (((2) 4 3     1   1   1)     1   1   1)) 6 2 1) 6        Mx5
(( (((2}      {1)         )         3    ))   9  ) 6        G, unsplit
(( (((2        1)         )         3    ))   9  ) 6        M
(( (((     3    )         )         3    ))   9  ) 6        unsplit

((((3)) 3)) 9) 6    or   303096
```

Split and unsplit can occur concurrently, but the unsplit might have to change later due to new merges. So I postponed the unsplit above until all groupings were done. So NOT

```
(( (((2) 4 3 1) 6 4 8 2) 5 5 3 7 3 7) 6 2 1 ) 6             split **
(( (((2)   8  ) 6 4 8 2) 5 5 3 7 3 7)   9   ) 6             split, unsplit
```

This is a design decision that occurs frequently. Either display the minimal number of steps by postponing UNSPLIT, or display the minimal form and (occasionally) have to add additional undo steps.

## Division

Division occurs by reverse Substitution.

Examples:

35 ÷ 7: substitute * for 7 in 35

```
(((((  •  )  ) ))•)•         [*   ((•)•)•    (((((•))))•)•]
(((( •    •   ) ))•)•        uG
(((( •)( •    ) ))•)•        uM
((((•)  •  •   ))•)•         uG
```

```
((((•)  •)(•   ))•)•         uM
((((•)  •)  • • )•)•         uG
((     *      • )•)•         subst
((     *  ))((• )•)•         uMx2
((     *  ))    *            subst
```

Similarly 35 ÷ 5:  substitute * for 5 in 35

```
((( ((   •   )       )))•)•   [*   ((•))•    (((((•))))•)•]
((( ( •     •        )))•)•   uG
((( ( •))(( •        )))•)•   uMx2
((( ( •))( •       • ))•)•   uG
((( ( •))( •  )   ( • ))•)•   uM
((( ( •)) •  •    ( • ))•)•   uG
((      *    •    ( • ))•)•   subst
((      *)  (•)  (( • ))•)•   uMx2        **
((      *)) (( • )) • (•))•   linear
((      *))    *      (•))•   subst
((      *))    *   )( (•))•   uM
((      *))    *   )   *      subst
```

** Note that UNMERGE could be applied four times here, resulting in an opportunistic match that shortens the number of steps.

```
((      *    •    ( • ))•)•   subst
((      *))((•))((( • ))•)•   uMx4        **
((      *))((( • ))•)((•))•   linear
((      *)}{    *   )  *      substx2
((      *)      *   )  *      M
```

The general principle is that opportunistic substitution may reduce the number of steps, but the final MERGE that is required has degraded the regularity of the recursive algorithm.

## Digit Division

Division remains the same, except that pattern-identification requires knowledge of digit multiplication rules.  Patterns should always be identified from the deepest space first.  Eg   76 ÷ 4 = 19

```
(      7           )  6
(4        3        ) 4 2           splitx2
(*        3        ) * 2           subst  **
(*}{      3        ] * 2           unmerge
(*)[      3        ] * 2
(*) 4 4 4 4 4 4 4 2  * 2           ungroup
```

```
(*) * * * * * * * 2  * 2           subst
(*) ********      2    2           linear
(*) ********        4             unsplit
(*) ********        *             subst
(1) 9
```

The identification of the “4” in the shallowest depth is optional.  Pattern identification can be opportunistic and in parallel, and I *think* none of the steps will ever need to be reversed.  The design goal may be maximal parallelism, or perhaps better “followability”.  The choice between parallel transformations and sequential “pedagogical” transformations occurs often.

## Long Division

Here’s a more complicated long division, with a remainder.

```
iG   identify bounded digit to ungroup
uG#  ungroup in terms of #
uM   unmerge to separate split digits ready to ungroup
S    split digits to access parts ready to unmerge
sub  substitute
L    linear artifact
```

```
     407139 ÷ 756  =  538  r411          [*   ((7)5)6   (((((4))7)1)3)9]

(( (([       4        ]     )                   7      )  1) 3) 9          iG

(( (( 7 7 7 7 7       5     )                   7      )  1) 3) 9          uG7
(( (( 7 7 7 7 7   3      2  )               5      2   )  1) 3) 9          S
(( (( 7 7 7 7 7} {3} {   2  )               5      2   )  1) 3) 9          uMx2
(( (( 7 7 7 7 7) (3) [   2  ]               5      2   )  1) 3) 9          iG

(( (( 7 7 7 7 7) (3)  5 5 5 5               5      2   )  1) 3) 9          uG5
(( (( 7 7 7 7 7) 5 5 5 5 5   (     3        )      2   )  1) 3) 9          L
(( (( 7 7 7 7 7) 5 5 5 5 5} {(     3        )} {   2   )  1) 3) 9          uMx2
(( (( 7 7 7 7 7) 5 5 5 5 5) ((     3        )) [   2   ]  1) 3) 9          iG

(( (( 7 7 7 7 7) 5 5 5 5 5) ((     3        ))  6 6 6 2   1) 3) 9          uG6
(( (( 7 7 7 7 7) 5 5 5 5 5) ((2        1    ))  6 6 6   3  ) 3) 9          S
(( (( 7 7 7 7 7) 5 5 5 5 5) ((2}   {   1    ))  6 6 6   3  ) 3) 9          uM
(( (( 7 7 7 7 7) 5 5 5 5 5) ((2)   [   1    ])  6 6 6   3  ) 3) 9          iG

(( (( 7 7 7 7 7) 5 5 5 5 5) ((2)    9     1 )  6 6 6   3  ) 3) 9          uG1
(( (( 7 7 7 7 7) 5 5 5 5 5) ((2)    9 } { 1 )  6 6 6   3  ) 3) 9          uM
(( (( 7 7 7 7 7) 5 5 5 5 5) ((2)    9 ) [ 1 ]  6 6 6   3  ) 3) 9          iG
```

```
(( (( 7 7 7 7 7) 5 5 5 5 5) ((2)    9 )  6 4   6 6 6   3  ) 3) 9           uG6
(( (( 7 7 7 7 7) 5 5 5 5 5) ((2)    9 )  6 6   6 6 6   1  ) 3) 9           S
(( (( 7 7 7 7 7) 5 5 5 5 5) 6 6 6 6 6 ((2) 9)          1  ) 3) 9           L
((***** ((  2  ) 9             )    1   )  3) 9                             sub
((5}   {((  2  ) 9             )    1   )  3) 9                             uM
((5)   (([  2  ] 9             )    1   )  3) 9                             iG

((5)   (( 7 7 6  9             )    1   )  3) 9                             uG7
((5)   (( 7 7 7 7            1 )    1   )  3) 9                             S
((5)   (( 7 7 7 7        } { 1 )    1   )  3) 9                             uM
((5)   (( 7 7 7     7    ) [ 1 ]    1   )  3) 9                             iG

((5)   (( 7 7 7     7    )  5 5     1   )  3) 9                             uG5
((5)   (( 7 7 7   6    1 )  5 5     1   )  3) 9                             S
((5)   (( 7 7 7} {6} { 1 )  5 5     1   )  3) 9                             uMx2
((5)   (( 7 7 7) (6) [ 1 ]  5 5     1   )  3) 9                             iG

((5)   (( 7 7 7) (6)  5 5   5 5     1   )  3) 9                             uG5
((5)   (( 7 7 7) (6)  5 5   5    6      )  3) 9                             S
((5)   (( 7 7 7) 5 5 5   (6)     6      )  3) 9                             L
((5)   (( 7 7 7) 5 5 5} {(6)     6      )  3) 9                             uM
((5)   (( 7 7 7) 5 5 5) ((6) 4      2   )  3) 9                             S
((5)   (( 7 7 7) 5 5 5) ((6) 4} {   2   )  3) 9                             uM
((5)   (( 7 7 7) 5 5 5) ((6) 4) [   2   ]  3) 9                             iG

((5)   (( 7 7 7) 5 5 5) ((6) 4)  6 6 6 2   3) 9                             uG6
((5)   (( 7 7 7) 5 5 5) ((6) 4)  6 6 6   5  ) 9                             S
((5)   (( 7 7 7) 5 5 5) 6 6 6  ((6) 4)   5  ) 9                             L

((5) ***  ((       6       )  4                  )         5       ) 9   sub
((5) 3}  {((       6       )  4                  )         5       ) 9   uM
((5) 3)  (([       6       ]  4                  )         5       ) 9   iG

((5) 3)  (( 7 7 7 7 7 7 7 7 4   4                )         5       ) 9   uG7
((5) 3)  (( 7 7 7 7 7 7 7 7 7                  1 )         5       ) 9   S
((5) 3)  (( 7 7 7 7 7 7 7 7 7              } { 1 )         5       ) 9   uM
((5) 3)  (( 7 7 7 7 7 7 7 7       7        ) [ 1 ]         5       ) 9   iG

((5) 3)  (( 7 7 7 7 7 7 7 7       7        )  5 5          5       ) 9   uG5
((5) 3)  (( 7 7 7 7 7 7 7 7   4       3    )  5 5          5       ) 9   S
((5) 3)  (( 7 7 7 7 7 7 7 7} {4} {    3    )  5 5          5       ) 9   uMx2
((5) 3)  (( 7 7 7 7 7 7 7 7) (4) [    3    ]  5 5          5       ) 9   iG

((5) 3)  (( 7 7 7 7 7 7 7 7) (4)  5 5 5 5 5   5 5          5       ) 9   uG5
((5) 3)  (( 7 7 7 7 7 7 7 7) 5 5 5 5 5 5 5 5   (4)         5       ) 9   L
```

```
((5) 3)  (( 7 7 7 7 7 7 7 7) 5 5 5 5 5 5 5 5} {(4)} {        5        ) 9   uMx2
((5) 3)  (( 7 7 7 7 7 7 7 7) 5 5 5 5 5 5 5 5) ((4)) [        5        ] 9   iG

((5) 3)  (( 7 7 7 7 7 7 7 7) 5 5 5 5 5 5 5 5) ((4))  6 6 6 6 6 6 6 6 2  9   uG6
((5) 3)  (( 7 7 7 7 7 7 7 7) 5 5 5 5 5 5 5 5) ((4))  6 6 6 6 6 6 6 6 2 8 1  S
((5) 3)  (( 7 7 7 7 7 7 7 7) 5 5 5 5 5 5 5 5) 6 6 6 6 6 6 6 6  ((4)) 2 8 1  L
((5) 3)  ********  ((4)) 2 8 1                                              sub
((5) 3) 8          ((4)) (1) 1                                              G
((5) 3) 8          ((4)   1) 1                                              M

((5) 3) 8   r ((4) 1) 1                                                     done

538   r 411
```

Here's an incomplete (rough!) recursive scheme for long division

```
GENERIC DIVISION

GET-T1
      Go to deepest space to count number of front target digit T1
      If there is one or more
            then setD=number-of-times-T1-is-available
            else FRONT-IDENTIFY
      GET-Tn

      FRONT-IDENTIFY
            Identify a digit to decompose via Ungroup
            UnGroup and decompose wrt T1
            setD=number-of-times-T1-is-available

GET-Tn
      Go to next shallower space
      Get D copies of Tn
      If shallowest space
            then FINISH-UP
            elseif D copies are available
                  then rearrange digits groups
                  else IDENTIFY
      GET-Tn-next-recur


      IDENTIFY
            Identify a digit to decompose via Ungroup
            UnGroup and decompose wrt Tn
            Rearrange digit groupings to get D copies of Tn
```