

# Search for and classification of Lychrel numbers

Lychrel numbers with respect to bases that are powers of two are well known, e.g. 10 101 00 in base 2 or 10 323 00 in base 4.

We analyzed the proof of their lychrel property, and used this knowledge to design an algorithm that searches similar lychrel numbers, by constructing lychrel candidates rather than just checking all numbers by brute force. This method is able to find such numbers for base 11 and 26 within seconds on a personal computer.

Furthermore, we performed an extensive search for lychrel numbers with respect to base 4. We present a classification scheme that contains most of the results, as well as some “sporadic” solutions.

This document relies on color vision. This document (LaTeX, PDF, Haskell program) is released into the public domain.

## 1. Introduction

**Terminology** See [1]. “Lychrel number” is used synonymously for “number that can be proven to be a lychrel number”, and “to prove the lychrel property of a number” is shortened to “to prove a number”.

**Discovery of (provable) lychrel numbers** The first idea for finding lychrel numbers might be considering base  $2 = 2^1$  and just applying the reverse-and-add algorithm to every number, beginning by 1, and ending by the first number where the reverse-and-add algorithm seemingly doesn’t terminate. This is easily done by hand, one has to check only 22 numbers, the first “lychrel candidate” is 10110. Applying 10 iterations to 10110 yields the calculation on the right, wherefrom the pattern

it.2:	1010100	+	0010101	it.6:	101101000	+	000101101
			1101001				110010101
			+ 1001011				+ 101010011
it.0:	10110	+	01101	it.4:	10110100	+	00101101
			100011				1011101000
it.1:	100011	+	110001	it.5:	11100001	+	101000101
			1010100				1010001011
it.2:	1010100	+	10000111	it.6:	101101000	+	1010001011
			101101000	it.10:	1011010000	+	1011010000

$$\text{iteration } 2 + 4n: \quad 10 \text{ (n 1s)} \quad 101 \text{ (n 0s)} \quad 00$$

emerges. It’s easy to prove that this pattern is indeed true, and that the numbers in all iterations between are not palindromic. Hence 10 101 00 is a lychrel number (we give a detailed proof for a similar number later).

**Generalizing to other bases** With some trial-and-error one can generalize this base-2 lychrel number to other bases that are powers of two, namely for base  $b = 2^m$  the number

$$1 \ 0 \ 2^{m-1} \ 2^{m-2} \ 2^{m-1} \ 0 \ 0.$$

**Outline of this paper** This raises two questions:

- (1) Are there any lychrel numbers whose proof work different?
- (2) Are there any lychrel numbers with respect to other bases?

Which can be answered using the following search methods:

- (1) Just do the same as in the paragraph “discovery of (provable) lychrel numbers”: Fix a base  $b$  (e.g.  $b = 4$ ), and for every number between 1 and an upper bound, execute some iterations, search for patterns, and if interesting enough, note the number down. In order to accelerate the process, use a computer and the method described at [2].
- (2) The brute force method in (1) is too inefficient to be applied on bases that are not powers of two. But some analysis of the proof of 10 101 00 leads to a set of constraints which are satisfied by all numbers that can be proved in a similar manner. Utilizing these rules, it is possible to develop a much more efficient search algorithm.

The ideas for such an algorithm (i.e. the constraints and how to use them) are presented in section 2, illustrated by the examples 10 323 00 (base 4) and 1N5ELA6C P6E7 0D59ME5N (base 26). A Haskell implementation is given in appendix A.

Leading to the following results:

- (1) There are many numbers that can be proved in a similar way, and they fit into a nice classification scheme, presented in section 3. The section is written for base 4, but can be generalized to any base that is a power of two. Furthermore, there are indeed some numbers that can be proved in a different way (at least in base 4), see section 3.5.
- (2) Two results, in base 11 and in base 26 are mentioned in section 2.7. If the reader is interested in more such numbers, they are invited to compute them using the above mentioned Haskell program.

## 2. Searching for lychrel numbers (any base)

### 2.1. Motivation

The introduction just presented a sketch of a proof for the lychrel property of 10 101 00 in base 2. We know prove this somewhat more rigorous for the number 10 323 00 in base 4.

Using induction:

- Induction hypothesis: For every  $n \in \mathbb{N}_0$ , the number 10 323 00 is after  $6n$  iterations of the form 10  $\underbrace{3}_{n}$  323  $\underbrace{0}_{n}$  00, and is not a palindrome during any of these iterations.
- Base case  $n = 0$ : Clear.
- Ind. step  $n \rightsquigarrow n + 1$ : We write  $\bar{0}$  and  $\bar{3}$  instead of  $\underbrace{0}_n$  resp.  $\underbrace{3}_n$ , and calculate the next 6 iterations, see fig. 1. Thus we arrived at 10  $\underbrace{\bar{3}}_{n+1}$  323  $\underbrace{\bar{0}}_{n+1}$  00, and in none of the iterations the number is a palindrome.

$$\begin{array}{r}
 1\ 0\ \bar{3}\ 3\ 2\ 3\ \bar{0}\ 0\ 0 \\
 + \quad \bar{0}\ \bar{0}\ \bar{0}\ \bar{3}\ 2\ 1\ \bar{3}\ \bar{0}\ \bar{0}\ 1 \\
 \hline
 1\ 1\ \bar{0}\ \bar{3}\ 1\ 2\ \bar{3}\ \bar{0}\ 1 \\
 + \quad \bar{0}\ \bar{0}\ \bar{1}\ \bar{1}\ \bar{1}\ \bar{1}\ \bar{3}\ \bar{0}\ \bar{0}\ 1 \\
 \hline
 1\ 2\ \bar{0}\ \bar{3}\ 2\ 1\ \bar{3}\ \bar{0}\ \bar{0}\ 1 \\
 + \quad \bar{1}\ \bar{2}\ \bar{0}\ \bar{3}\ 1\ \bar{3}\ \bar{0}\ \bar{1}\ \bar{2} \\
 \hline
 1\ 0\ \bar{3}\ \bar{3}\ 3\ 2\ 3\ \bar{0}\ 0\ 0 \\
 = 1\ 0\ \bar{3}\ \bar{3}\ 3\ 2\ 3\ \bar{0}\ 0\ 0
 \end{array}$$

Figure 1: Calculation for the proof on the left.

### 2.2. Basic observations

**Computation of the iterations** We think of numbers as digit strings, and iterations as digit-wise addition, which is performed from right to left, and adds a digit to the digit that “matches” this digit upon reversing, plus taking carries into account (fig. 2).

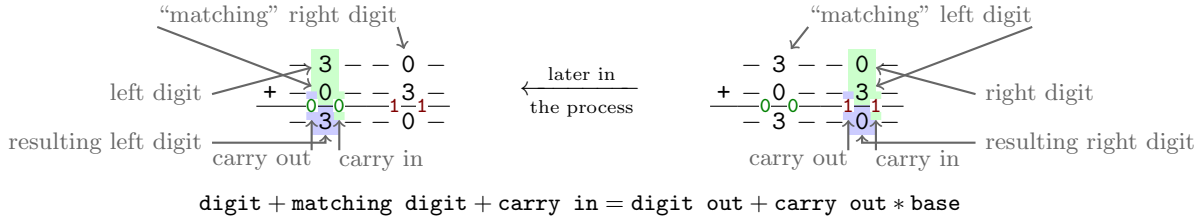


Figure 2: Computation of an iteration = digit-wise addition (from right to left) of digits and “matching” digits + taking carries into account. (The “-”s represent the other digits of the number.)

This point of view allows us to handle digit strings with arbitrary length, i.e. we always know that [cf. fig. 3], no matter how many 0’s resp. 3’s there are in the “-”-part (as long as it’s as many 0’s on the left as 3’s on the right).

$$\begin{array}{r}
 -\ 0\ .\ 0\ -\ -\ 3\ .\ 3\ - \\
 + \quad \bar{0}\ \bar{0}\ \bar{0}\ \bar{0}\ \bar{1}\ \bar{1}\ \bar{1}\ \bar{1} \\
 \hline
 -\ 3\ .\ 3\ -\ -\ 0\ .\ 0\ - \\
 \uparrow \quad \uparrow \quad \uparrow \quad \uparrow \quad \uparrow \\
 \text{same length} \quad \text{same length}
 \end{array}$$

Figure 3: Handling arbitrarily long digit strings.

**Constraints** We are now able carry out the calculation in fig. 1, but we still need to check whether the inductive step “works”,

$$10\ \underbrace{3}_{n}\ 323\ \underbrace{0}_{n}\ 00 \xrightarrow{6\ \text{iterations}} 10\ \underbrace{3}_{n+1}\ 323\ \underbrace{0}_{n+1}\ 00.$$

That is, after 6 iterations, the number looks almost the same, but some parts are shifted (cf. fig. 4). So, if we want a number, whose Lychrel-property can be proven in a similar way as for 10 323 00 above, this constraint must be satisfied.

### 2.3. Idea

As stated in the introduction, the goal is to find numbers that can be proved in a similar way as 10 323 00 in section 2.1. Of course there is the naive brute force approach, but there is also a more sophisticated way: By observing yet more constraints (cf. section 2.6), it is possible to “construct lychrel candidates from outside to inside” rather than just checking every number. For the sake of simplicity, we restrict ourselves to numbers of even length  $2n$ .

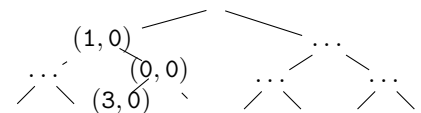


Figure 5: Illustration for explanation in the “more sophisticated approach” column.

$$\begin{array}{r}
 (1\ 0\ \bar{3}\ 3\ 2\ 3\ \bar{0}\ 0\ 0) \\
 \vdots \\
 = (1\ 0\ \bar{3}\ \bar{3}\ 3\ 2\ 3\ \bar{0}\ 0\ 0)
 \end{array}$$

Figure 4: “Proof works” = number looks almost the same after 6 iterations, but some parts are shifted.

## naive approach

First “generate” whole number, then check whole number, as in fig. 1. Repeat for all numbers from 0 to  $b^{2n} - 1$ . In pseudocode-speak:

```
for integer i=0,...,b^(2n)-1
  represent i wrt base b as digit string s
  calculate next x steps, result is s'
  check wheter s,s' satisfy the constraint
```

*Example:*

Number to be checked: 1033323000

Computation: 
$$\begin{array}{r} 1033323000 \\ : \\ = 103333230000 \end{array} \quad \text{ok } \checkmark$$
  
→ This is a number that we’re searching for.

Number to be checked: 1023323000

Computation: 
$$\begin{array}{r} 1023323000 \\ : \\ = 113212131131 \end{array} \quad \text{fail } \times$$
  
→ This is not a number that we’re searching for.

## more sophisticated approach

Generate number “from outside to inside”, digit pair by digit pair. Then compute the  $x$  iterations just for this digit pair, and then check whether this digit pair plus known rest of the number (that is, everything “outside” of the digit pair) satisfy the constraint, see example below. Because there is sometimes more than one possible digit pair, this will result in a tree of digit pairs instead of a single number (cf. fig. 5). “Flattening” this tree yields all found numbers (cf. section 2.5 for details).

*Example:* (where gray = not important for computation; and “\_” = digits that are not yet determined)

Already given digit pairs: (1, 0), (0, 0)

Digit pair to be checked: (3, 0)

Computation: 
$$\begin{array}{r} 103\_ \_ \_ 000 \\ : \\ = 103\_ \_ \_ 000 \end{array} \quad \text{ok } \checkmark$$
  
→ Continue to search for numbers of the form 103.000, i.e. with already given digit pairs (1, 0), (0, 0), (3, 0).

Already given digit pairs: (1, 0), (0, 0)

Digit pair to be checked: (2, 0)

Computation: 
$$\begin{array}{r} 102\_ \_ \_ 000 \\ : \\ = 103\_ \_ \_ 100 \end{array} \quad \text{fail } \times$$
  
→ The numbers that we’re searching for cannot be of the form 102.000.

*Remark:* Actually, it is somewhat more difficult, in particular it is not possible to calculate a digit pair without knowing the others (carries!). The example is meant to convey the idea, not being precise (the numbers are just made up). For details, see the following sections.

## 2.4. Terminology

As mentioned in the remark, it is not possible to calculate iterations for a isolated digit pair. The “carry in” might be not known, the “carry out” might be given. We introduce color codes to indicate which category a value belongs to.

	input of calculation	output of calculation	
known before calculation	no problem	check if $\text{result of calc.} = \text{known value}$	red : “carry in unknown”
unknown before calculation	try out all possibilities	no problem	yellow : “carry out given”
			green : cf. fig. 2
			blue : cf. fig. 2

Figure 6: Left: color codes. Right: color code examples (remark: in the naive approach, we have only green/blue, cf. fig. 2).

Not every digit in fig. 1 is treated equally, some get shifted, others not. Hence, we divide the computation into several parts:

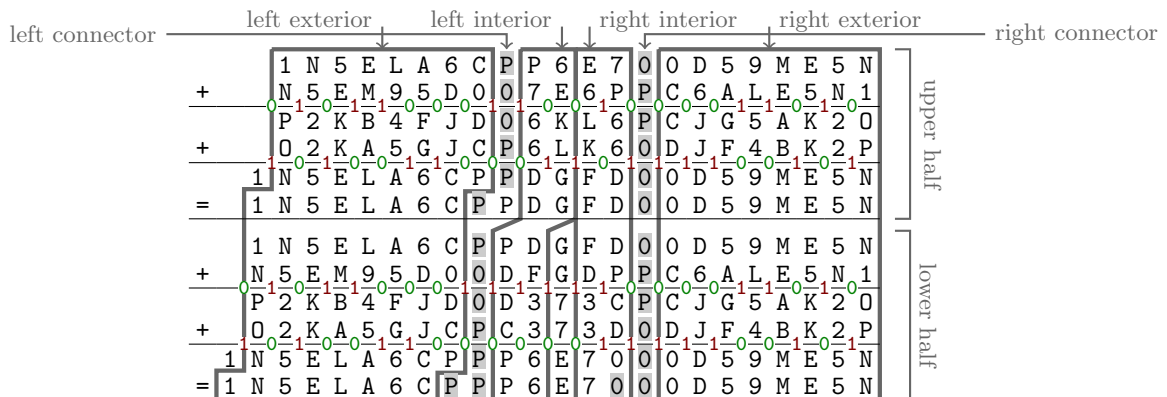


Figure 7: Dividing the computation for the inductive step into several parts.

## 2.5. Outline of the algorithm

Step	exterior	interior
1. Generate all possible roots,	i.e. all possible outermost digit pairs (rule ExtHead).	i.e. all possible innermost digit pairs (rule IntHead).
2. Build up tree recursively (cf. section 2.3), up to a specified depth, Additionally, check for every node whether it can connect to a connector,	from outside to inside (rule ExtRec). (rule ExtLast).	from inside to outside (rule IntRec). (rule IntLast).
3. Flatten tree: turn the tree of digit pairs into a list of digit strings that serve as exterior/interior part of a lychrel number (cf. fig. 8).		

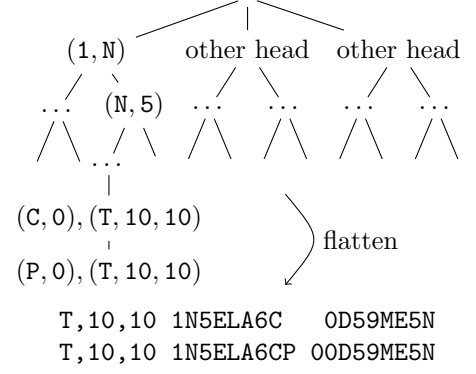
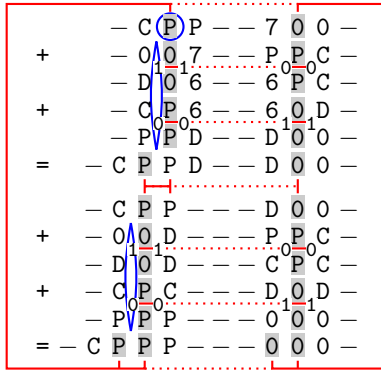


Figure 8: Tree of exterior part (base 26), depth 9.

## 2.6. More sophisticated observations

Connecting part:

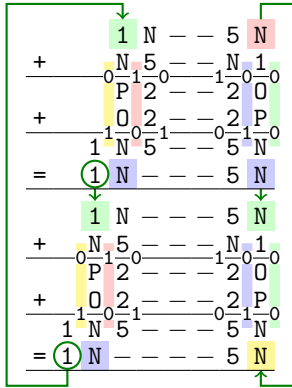


Connectors have a very simple structure.

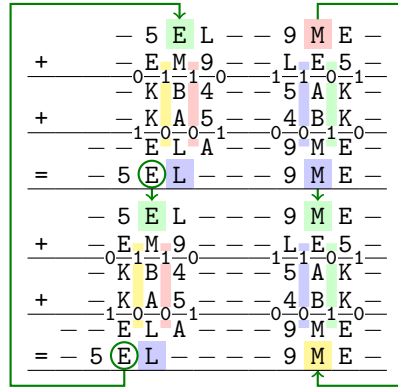
- Connected by red solid line: digits/carries are the same.
  - Connected by red dotted line: digits/carries are inverses to each other.
- Therefore, a connector is uniquely determined by the digits/carries that are encircled in blue. Furthermore, the encircled digit must be 0 or base – 1. By shortening base – 1 to T and 0 to F, the connector on the right is denoted by

$$\overbrace{\text{digit top left}}^{\text{T}}, \quad \overbrace{\text{carry upper half}}^{10}, \quad \overbrace{\text{carry lower half}}^{10}.$$

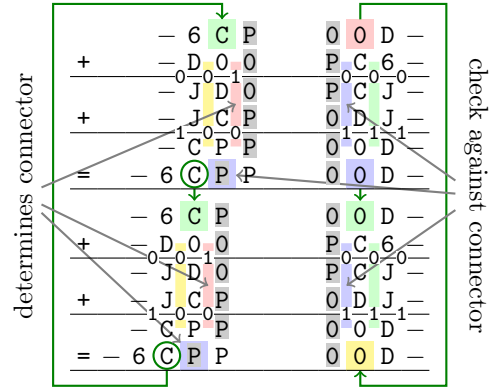
Exterior part:



rule ExtHead

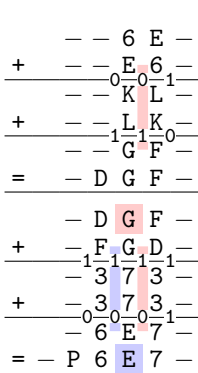


rule ExtRec

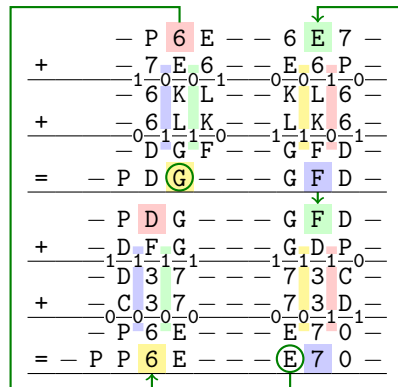


rule ExtLast

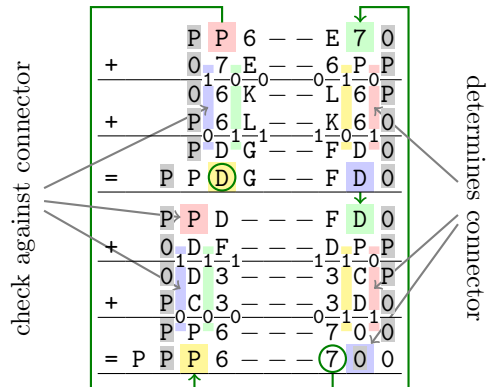
Interior part:



rule IntHead



rule IntRec



rule IntLast

Figure 9: Lychrel numbers in base 26 (above) and base 11 (below).

### 3. Classification (base 4)

#### 3.1. Motivation

**Observations** As mentioned in the introduction, the brute force search “fix base 4, and for every number up to  $4^{26}$ , perform some iterations, search for patterns, and, if interesting enough, note the number down” yields a lot of lychrel numbers, more precisely

- (1) many numbers that can be proven in a similar way as 10 323 00,
- (2) some numbers that can be proven in a different way.

Numbers of the form (2) are presented in section 3.5, in the following we focus on number of the form (1), which are presented in section 3.4. All these numbers (“all” in terms of found by the above mentioned search) share the following properties:

- The exterior part is always 10 ... 00.
- Some patterns seem to reoccur in the interior part, cf. fig. 10.

**Investigations** In order to investigate these patterns, we have to view the digit strings that make up the patterns *locally*, i.e. on their own, independent of the number they are contained in. In section 2.2 we presented a technique to handle digits indepently of (the length of) the remaining number. Recapitulate:

- A left and a right digit that match upon reversing serve as “input”.
- In order to compute the resulting left and right digit, we only need to know carry in/out instead of the whole number.
- We cannot do that for all 6 iterations, because after 3 iterations, the number grows, so we have to take shifts into account.

We now generalize this horizontally (digit strings instead of single digits) and vertically (several iterations instead of a single operations), the result is called a *block*.

How many digits make up a block (horizontally)? How many iterations make up a block (vertically)?

*Horizontal:* We group the digits in such a way that they can be classified nicely.

*Vertical:* 3 iterations, because this is the maximal number of iterations that we can handle without shifting in between.

#### 3.2. Full-blocks

**Definition of a block** See fig. 12. A *half block* consists of the following data, *each for left and right half*:

- digit string (“number in”),
- carry in & carry out for 3 iterations,
- how to shift digits after 3 iterations (“modification”, “mod.”),
- resulting digit string, after 3 iterations + mod. (“number out”).

A *full block* consists of two half blocks (upper and lower) that fit, i.e.:

- “number out”s of upper half = “number in”s of lower half, and
- “number in”s of upper half = “number out”s of lower half.

A *central half block* is a half block with

- for data  $\in \{\text{num in/out, carry in/out, modification}\}$ , it holds that (left data) = (right data).

**Constructing lychrel numbers out of blocks** Lychrel numbers can be build out of blocks as follows:

- One of the two *not blocks* outside (cf. fig. 13).
- A central block inside.
- Finitely many full-blocks in between, whereupon carries and modifications must “fit together” (cf. fig. 12).

10 323 00,  
10 33 3330000 323 0100023 00 00,  
10 323 010 323 00,  
10 3323 0010 3323 00, ...

Figure 10: A few results from the base 4 search.

Figure 11: Comp. for 10 3323 0010 3323 00, some ways to group digits are highlighted.

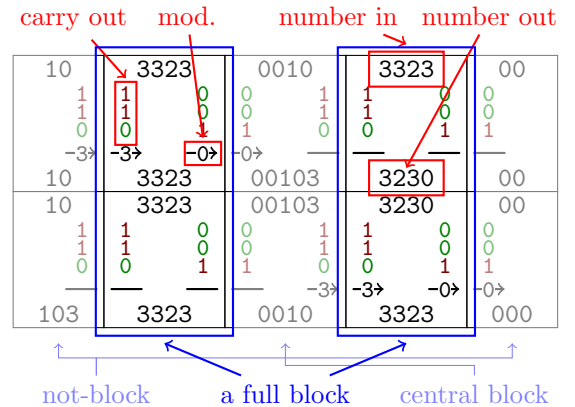


Figure 12: A possible way to deconstruct 10 3323 0010 3323 00 into blocks.

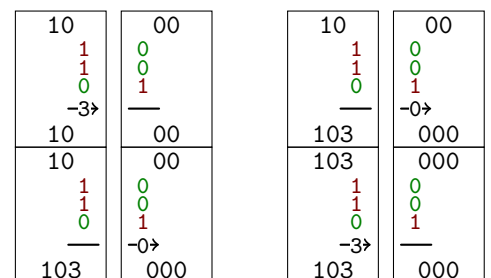


Figure 13: The two not-blocks.



Given a Lychrel number (that is made out of blocks), there are often many possible ways to deconstruct it into blocks. Example 10 3323 0010 3323 00 (where an “initial grouping of digits” is already given, indicated by spaces): During the first six iterations (fig. 11), there are several ways to group the digits (indicated by digits with two background colors, e.g. white and magenta or magenta and yellow). There is only one constraint, namely that for each half block the numbers of left and right half have the same length. Additionally, we restrict ourselves to the case, that at most one digit can be shifted, and if so only left-to-right. This leads to *four* possible ways to deconstruct 10 3323 0010 3323 00 into blocks (fig. 14).

10 1 1 0 -3→	3323 1 1 0 -3→	0010 0 0 1 -0→	3323 1 1 0 -3→	00 0 0 1 -0→
10 1 1 0 -3→	3323 1 1 0 -3→	00103 0 0 1 -0→	3230 1 1 0 -3→	00 0 0 1 -0→
103 1 1 0 -3→	3323 1 1 0 -3→	0010 0 0 1 -0→	3323 1 1 0 -3→	000 0 0 1 -0→

10 1 1 0 -3→	3323 1 1 0 -3→	0010 0 0 1 -0→	3323 1 1 0 -3→	00 0 0 1 -0→
10 1 1 0 -3→	33230 1 1 0 -3→	010 0 0 1 -0→	33230 1 1 0 -3→	00 0 0 1 -0→
103 1 1 0 -3→	3323 1 1 0 -3→	0010 0 0 1 -0→	3323 1 1 0 -3→	000 0 0 1 -0→

10 1 1 0 -3→	3323 1 1 0 -3→	010 0 0 1 -0→	3323 1 1 0 -3→	00 0 0 1 -0→
103 1 1 0 -3→	3230 1 1 0 -3→	010 0 0 1 -0→	3323 1 1 0 -3→	000 0 0 1 -0→
103 1 1 0 -3→	3230 1 1 0 -3→	010 0 0 1 -0→	3323 1 1 0 -3→	000 0 0 1 -0→

10 1 1 0 -3→	3323 1 1 0 -3→	0010 0 0 1 -0→	3323 1 1 0 -3→	00 0 0 1 -0→
103 1 1 0 -3→	323 1 1 0 -3→	00103 0 0 1 -0→	323 1 1 0 -3→	000 0 0 1 -0→
103 1 1 0 -3→	323 1 1 0 -3→	00103 0 0 1 -0→	323 1 1 0 -3→	000 0 0 1 -0→

Figure 14: The four possible deconstructions of 10 3323 0010 3323 00 into blocks corresponding to fig. 11.

This diversity of blocks (although an initial grouping of digits is already given!) is quite a mess. We need a better notation. For this purpose, we focus on half blocks instead of full blocks, and introduce a simplified notation.

### 3.3. Half-blocks

**Notational remark** When working with half blocks, shorten (left number in) =  $d_{la}$ , (left number out) =  $d_{lb}$ , (left carry out) =  $c_{ll}$ , (left carry in) =  $c_{lr}$ , and analogously for the right part (a/b/l/r = above/below/left/right).

**Carry signature** Shorten the carry  $(1, 1, 0)$  by 1 and  $(0, 0, 1)$  by 0. For a half block, write the *carry signature* as follows:

carry sig.:	0-0 1-1	1-1 0-0	0-1 0-1	1-0 1-0
signature for:	$\begin{array}{c} d_{la} \\ 0\ 0 \\ 0\ 0 \\ 1\ 1 \\ d_{lb} \end{array}$	$\begin{array}{c} d_{ra} \\ 1\ 1 \\ 1\ 1 \\ 0\ 0 \\ d_{rb} \end{array}$	$\begin{array}{c} d_{la} \\ 0\ 0 \\ 1\ 1 \\ 1\ 0 \\ d_{lb} \end{array}$	$\begin{array}{c} d_{ra} \\ 0\ 0 \\ 0\ 1 \\ 1\ 0 \\ d_{rb} \end{array}$

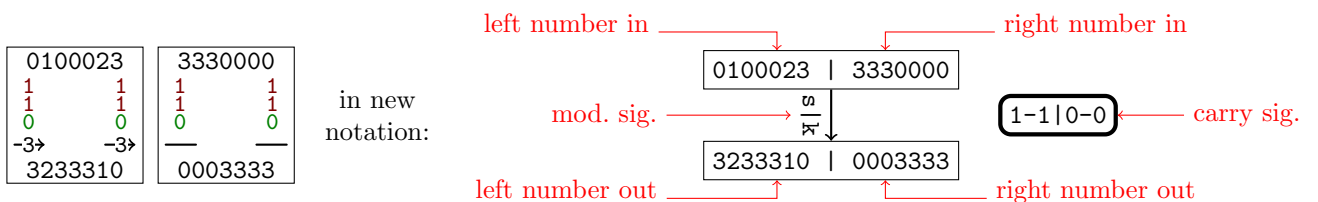
**Modification signature** Similarly, introduce a modification signature for half blocks (a=attach, r=remove, s=shift, k=keep):

mod. sig.:	a a	k k	k s
signature for:	$\begin{array}{c} d_{la} \\ c_{ll} \quad c_{lr} \\ -t_{ll} \rightarrow \\ d_{lb} \end{array}$	$\begin{array}{c} d_{ra} \\ c_{rl} \quad c_{rr} \\ -t_{rl} \rightarrow \\ d_{rb} \end{array}$	$\begin{array}{c} d_{la} \\ c_{ll} \quad c_{lr} \\ -t_{ll} \rightarrow \\ d_{lb} \end{array}$

mod. sig.:	r r	s s	s k
signature for:	$\begin{array}{c} d_{la} \\ c_{rl} \quad c_{rr} \\ -t_{lr} \rightarrow \\ d_{lb} \end{array}$	$\begin{array}{c} d_{ra} \\ c_{rl} \quad c_{rr} \\ -t_{rl} \rightarrow \\ d_{rb} \end{array}$	$\begin{array}{c} d_{la} \\ c_{ll} \quad c_{lr} \\ -t_{ll} \rightarrow \\ d_{lb} \end{array}$

where  $t_i = \begin{cases} 0 & \text{for } c_i = (0, 0, 1) \\ 3 & \text{for } c_i = (1, 1, 0) \end{cases}$   
for  $i \in \{ll, lr, rl, rr\}$

**Input/output digit string** Given “left/right number in” and a carry signature, “left/right number out” are uniquely determined by the modification signature. Thus, notate a half block as follows:



### 3.4. Classification scheme

This section presents all results of the brute force base 4 search that have the form (1) (cf. section 3.1). Instead of giving all found numbers, only the half blocks are given. From these half blocks all found numbers can be constructed, as described in sections 3.2 and 3.3. Altogether, 14 “minimal” half blocks were found, which can be extended by adding 0s and 3s. They fit into a nice classification scheme, which is shown in fig. 17. Figure 16 shows how to read this classification scheme. As example, here’s how to extract 10 3323 0010 3323 00 from this classification scheme:

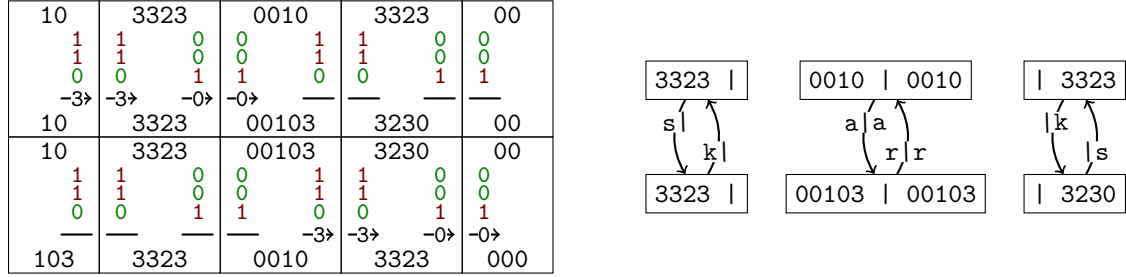


Figure 15: The number 10 3323 0010 3323 00; left: constructed out of full blocks, right: constructed out of half blocks.

And now for the classification:

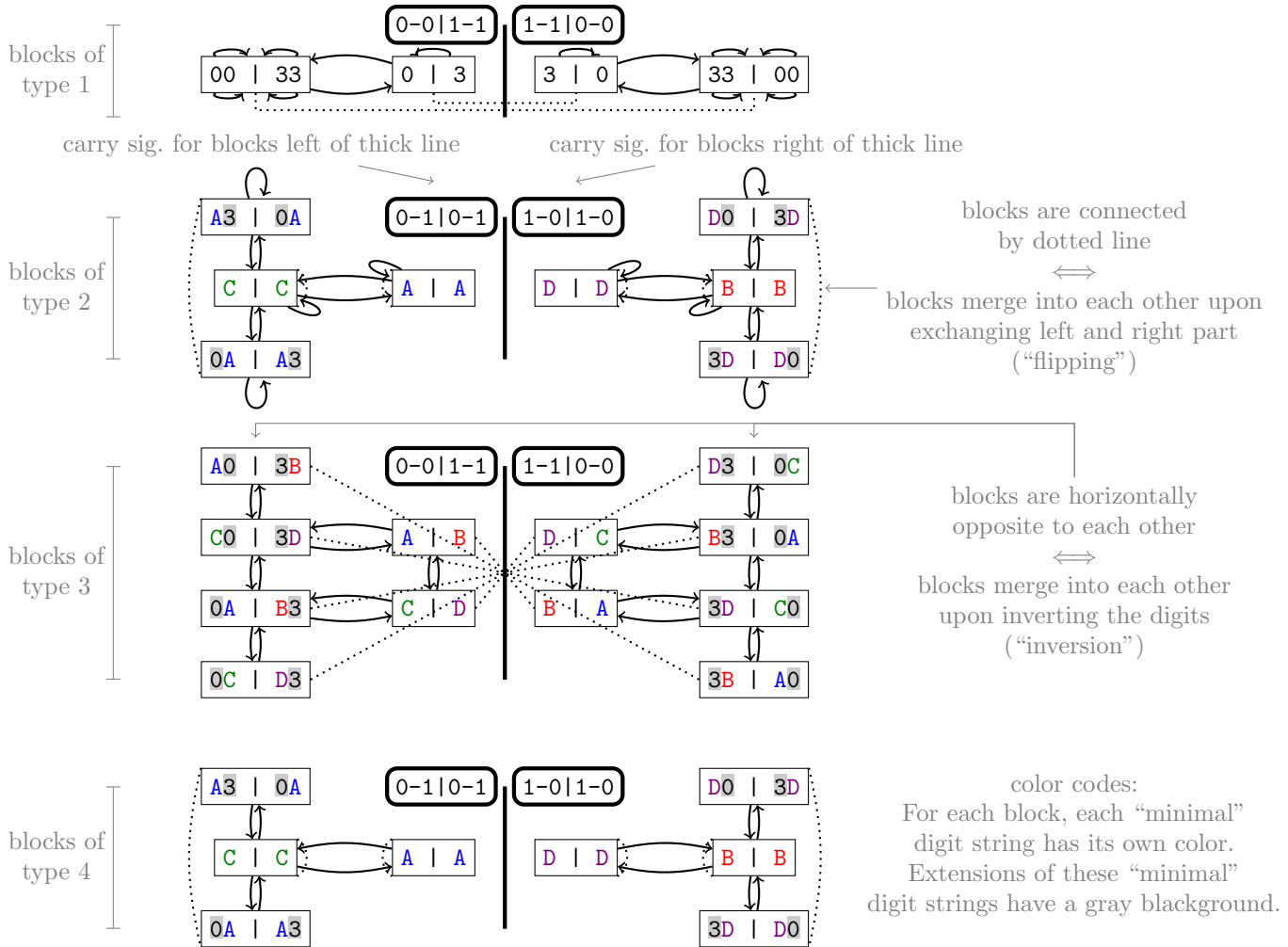


Figure 16: All blocks of type 1/2/3/4 with up to 3/5/8/13 digits – legend.

### 3.5. Sporadic solutions

This section presents all results of the brute force base 4 search that have the form (2) (cf. section 3.1). These are (where  $k, x \in \mathbb{N}_0$  and  $n$  is increasing by 3 after 6 iterations):

$$10002003 \underset{k}{0} 0221 \underset{n}{2} 3221 \underset{k}{3} 33101333, \quad 10002003 010113 \underset{k}{3} 3112 \underset{n}{1} 0112 \underset{k}{0} 002100 33101333.$$



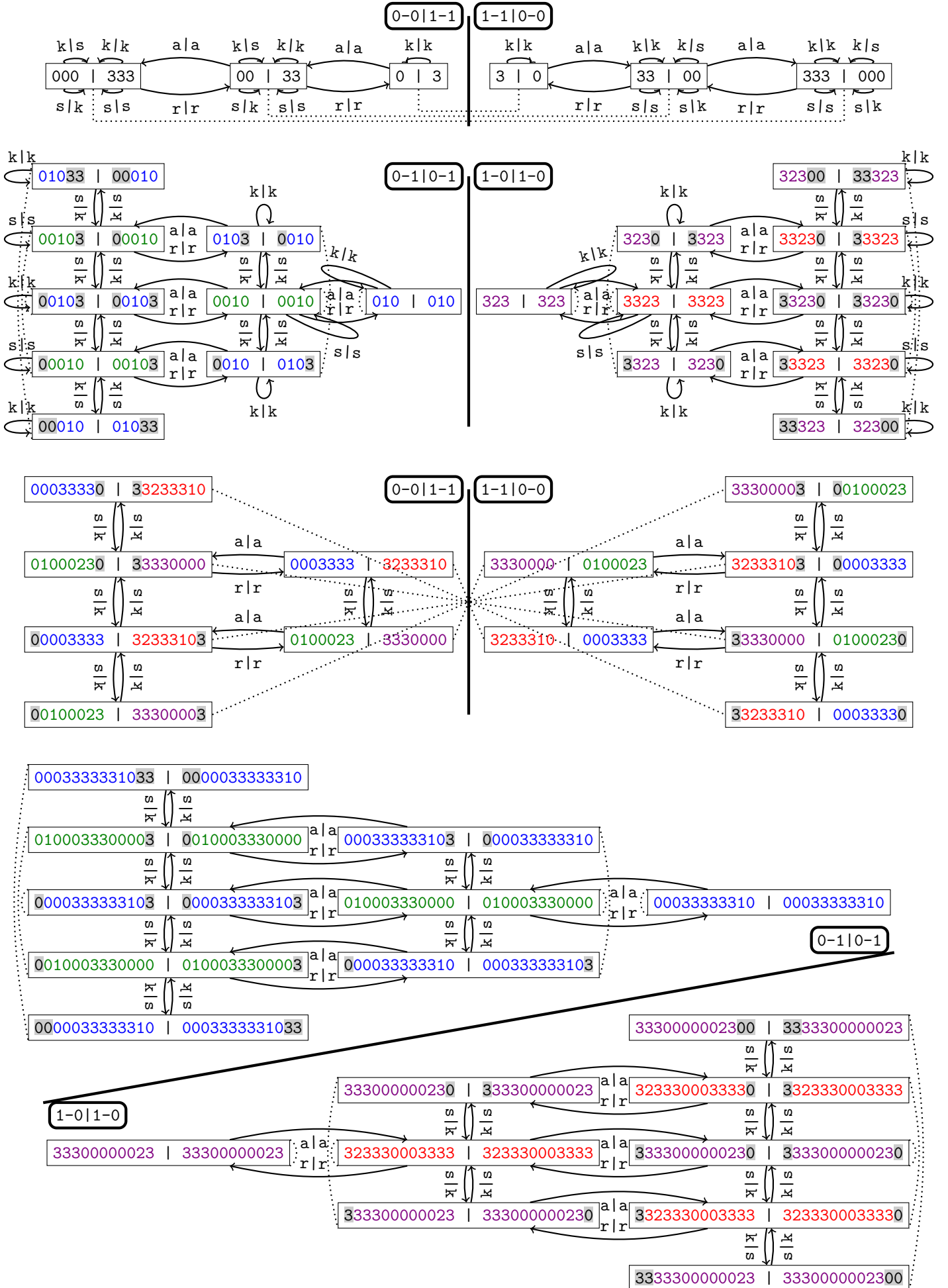


Figure 17: All blocks of type 1/2/3/4 with up to 3/5/8/13 digits – data.

## A. Haskell implementation of the search algorithm

```

import Data.Maybe
import Data.Char      -- for conversion Int -> Char
import System.Environment -- for command line arguments

{- auxiliary stuff -}

type B = Int      -- base
type D = Int      -- digit in {0,...,base-1}
type C = Int      -- carry in {0,1}
type CS = [Int]   -- list of carries

-- converts an Int to its binary representation; example: convIntToCs 5 13 = [1,0,1,1,0]
convIntToCs :: Int -> Int -> CS
convIntToCs l n = [if even $ n `div` 2i then 0 else 1 | i <- [0..l-1]]

-- converts binary representation of an integer to an Int; example: carrytoint [1,0,1,1,0] = 13
convCsToInt :: CS -> Int
convCsToInt xs = fst $ foldl (\ (n,i) x -> (n+x*2i,i+1)) (0,0) xs

-- inverts a list of carries; example: carryinv [1,0,1,1,0] = [0,1,0,0,1]
invCarry :: CS -> CS
invCarry xs = map (1-) xs

-- inverts a digit wrt a base; example: digitinv 10 2 = 8
invDigit :: B -> D -> D
invDigit base x = base-1-x

-- generates list of all possible carry lists with a given length
genCs :: Int -> [CS]
genCs l = map (convIntToCs l) [0..2l-1]

-- adds two digits and a carry and outputs this sum as (digit of sum,carry of sum), wrt a base;
-- example: addDW 10 (6,7,1) = (4,1), because 6+7+1 = 4 + 1*10; addDW = "add digit-wise"
addDW :: B -> (D,D,C) -> (D,C)
addDW base (a,b,c) = ((a+b+c) `rem` base, (a+b+c) `div` base)

-- adds a digit pair
addDP :: B -> ((D,D),(C,C)) -> ((D,D),(C,C))
addDP base ((dl,dr),(cl,cr)) = ((dl',dr'),(cl',cr'))
  where (dl',cl') = addDW base (dl,dr,cl); (dr',cr') = addDW base (dl,dr,cr)

{- types -}

data Side = Exterior | Interior deriving (Eq,Show,Read)
data BlkArg = BA {dlaHA :: D, drbHA :: D, clrHA :: CS, crrHA :: CS} deriving (Eq,Show)
data BlkVal = BV {dlbHV :: D, drbHV :: D, clbHV :: CS, clbHV :: CS} deriving (Eq,Show)
data HalfIn = HI {dlaHI :: D, draHI :: D, cliHI :: CS, criHI :: CS} deriving (Eq,Show)
data HalfOut = HO {dlbHO :: D, drbHO :: D, cloHO :: CS, croHO :: CS} deriving (Eq,Show)
data ExtHalfIn = EHI {dlaEI :: D, draEI :: D, clbEI :: CS, crrEI :: CS} deriving (Eq,Show)
data ExtHalfOut = EHO {dlbEO :: D, drbEO :: D, clbEO :: CS, clbEO :: CS} deriving (Eq,Show)
data IntHalfIn = IHI {dlaII :: D, draII :: D, clbII :: CS, clbII :: CS} deriving (Eq,Show)
data IntHalfOut = IHO {dlbIO :: D, drbIO :: D, clbIO :: CS, crrIO :: CS} deriving (Eq,Show)
data ExtFullIn = EFI {adlaEI :: D, bdlaEI :: D, acllEI :: CS, acrrEI :: CS, bcllEI :: CS, bcrrEI :: CS} deriving (Eq,Show)
data IntFullIn = IFI {adralI :: D, adblII :: D, aclrII :: CS, acrlII :: CS, bclrII :: CS, bcrIII :: CS} deriving (Eq,Show)
data FullInfo = Info Side ((HalfIn,HalfOut),(HalfIn,HalfOut)) deriving (Eq,Show)
data ConnIn = ConnIn {adlCI :: D, mdlCI :: D, aclCI :: CS, acrCI :: CS, bclCI :: CS, bcrCI :: CS} deriving (Eq,Show)
data ConnOut = ConnOut {adlCO :: Bool, aclCO :: CS, bclCO :: CS} deriving (Eq,Show)
data FullOut = FullOut (D,D) (Maybe ConnOut)

side :: (Either ExtFullIn IntFullIn) -> Side
side (Left _) = Exterior
side (Right _) = Interior

len :: (Either ExtHalfIn IntHalfIn) -> Int
len (Left (EHI _ _ clb crr)) = length clb
len (Right (IHI _ _ clb crr)) = length clb

{- basic calculations -}

calcDP :: B -> BlkArg -> BlkVal
calcDP base (BA dla dra clr crr) = BV dlb drb clb crr
  where computation = tail $ scanl (\ (d,_) c -> addDP base (d,c)) ((dla,dra),(0,0)) (zip clr crr)
        (dlb,drb) = fst . last $ computation
        (clb,crr) = unzip . snd . unzip $ computation

calcMiddle :: B -> (D,CS) -> (D,CS)
calcMiddle base (da,cr) = (db,cl)
  where computation = tail $ scanl (\ (d,_) c -> addDW base (d,d,c)) (da,0) cr
        db = fst . last $ computation
        cl = snd . unzip $ computation

calcConn :: B -> D -> CS -> CS -> (D,D)
calcConn base adl acl bcl = (fst a, fst b)
  where a = foldl (\ d c -> fst $ addDP base (d,c)) (adl, adr) (zip acl acr)
        bdl = fst a
        b = foldl (\ d c -> fst $ addDP base (d,c)) (bdl, bdr) (zip acl acr)
        adr = invDigit base adl; acr = invCarry acl;
        bdr = invDigit base bdl; bcr = invCarry bcl;

{- build and flatten tree -}

```

```

checkConn :: B -> ConnIn -> Maybe ConnOut
checkConn base (ConnIn adl mdl acl acr bcl bcr) = if initok && carryok && digitok
  then Just $ ConnOut (adl /= 0) acl bcl
  else Nothing
  where
    (mdl', adl') = calcConn base adl acl bcl
    initok = adl == 0 || adl == base-1
    digitok = adl == adl' && mdl == mdl'
    carryok = acl == invCarry acr && bcl == invCarry bcr

{- half block -}

halfInToArg :: CS -> (Either ExtHalfIn IntHalfIn) -> BlkArg
halfInToArg clr' (Left (EHI dla dra _ crr))
  = BA dla dra clr' crr
halfInToArg crr' (Right (IHI dla dra clr _))
  = BA dla dra clr crr'

halfValToOut :: CS -> (Either ExtHalfIn IntHalfIn) -> BlkVal -> Maybe HalfOut
halfValToOut clr' (Left (EHI _ _ cil' _)) (BV dlb drb cll crl)
  = if cil' == cll then Just $ H0 dlb drb cll crr' else Nothing
halfValToOut crr' (Right (IHI _ _ _ crr')) (BV dlb drb cll crl)
  = if crr' == crr then Just $ H0 dlb drb cll crr' else Nothing

halfInToIn :: (Either ExtHalfIn IntHalfIn) -> HalfIn
halfInToIn (Left (EHI dla dra cll crr))
  = HI dla dra cll crr
halfInToIn (Right (IHI dla dra clr crr))
  = HI dla dra clr crr

calcHalf :: B -> (Either ExtHalfIn IntHalfIn) -> [(HalfIn, HalfOut)]
calcHalf base x = zip (repeat $ halfInToIn x) (catMaybes $ map (f x) (genCs $ len x))
  where f x cs = (halfValToOut cs x) (calcDP base $ halfInToArg cs x)

{- full block -}

fullInA :: B -> (Either ExtFullIn IntFullIn) -> [(Either ExtHalfIn IntHalfIn)]
fullInA (Left (EFI adla _ acll acrr _))
  = [Left (EHI adla adra acll acrr) | adra <- [0..base-1]]
fullInA (Right (IFI adra _ acrr acrl _))
  = [Right (IHI adra adra acrr acrl) | adra <- [0..base-1]]

fullInB :: B -> (Either ExtFullIn IntFullIn) -> HalfOut -> [(Either ExtHalfIn IntHalfIn)]
fullInB (Left (EFI _ bdla _ _ bcll bcrr)) (H0 _ bdra _ _)
  = [Left (EHI bdla bdra bcll bcrr)]
fullInB (Right (IFI _ _ _ bclr bcrl)) (H0 _ bdra _ _)
  = [Right (IHI bdla bdra bclr bcrl) | bdla <- [0..base-1]]

fullFilter :: (Either ExtFullIn IntFullIn) -> FullInfo -> Bool
fullFilter (Left _)
  (Info - ((HI _ adra _ _), (HI _ adra _ _))) = adra == bdrb
fullFilter (Right (IFI _ adlb' _ _ _)) (Info - ((HI adla _ _ _), (HI adlb _ _ _)))
  = adla == bdlb && adlb' == adlb

calcFull :: B -> (Either ExtFullIn IntFullIn) -> [FullInfo]
calcFull base x = filter (fullFilter x) $ map (Info (side x)) bas
  where
    bas = concatMap (calcHalf base) $ fullInA base x
    bas = concat [ zip (repeat a) (concatMap (calcHalf base) $ fullInB base x (snd a)) | a <- as ]

fullToIn :: FullInfo -> (Either ExtFullIn IntFullIn)
fullToIn (Info Exterior ((_, H0 adlb _ acl acr), (HI bdla _ _ _)))
  = Left (EFI bdla adlb acl acr bcl bcr)
fullToIn (Info Interior ((_, H0 _ _ acl acr), (HI bdla _ _ _)))
  = Right (IFI bdrb bdla acl acr bcl bcr)

fullToConn :: B -> FullInfo -> ConnIn
fullToConn base (Info Exterior ((_, H0 adlb _ acl acr), (HI bdla _ _ _)))
  = ConnIn adl mdl acl acr bcl bcr
fullToConn base (Info Interior ((_, H0 _ _ acl acr), (HI bdla _ _ _)))
  = ConnIn adl mdl acl acr bcl bcr
  where mdl = bdla; adl = invDigit base bdrb;

fullToDP :: FullInfo -> (D,D)
fullToDP (Info - ((HI adla adra _ _ _), (HI _ _ _)))
  = (adla, adra)

calcVertical :: B -> (Either ExtFullIn IntFullIn) -> [(Either ExtFullIn IntFullIn, FullOut)]
calcVertical base x = [ (fullToIn y, FullOut (fullToDP y) ((checkConn base) $ y)) | y <- ys ]
  where ys = calcFull base x

```

```

data Tree = Node FullOut [Tree]
type NumPart = [(D,D)]

calcHorizontal :: B -> Int -> Either ExtFullIn IntFullIn -> [Tree]
calcHorizontal base 0 x = []
calcHorizontal base depth x = [ Node res (calcHorizontal base (depth-1) cont) | (cont,res) <- (calcVertical base x) ]

flatten :: NumPart -> Tree -> [(NumPart,ConnOut)]
flatten dps (Node (FullOut dp conn) trees) = if isJust conn then endshere : endslater else endslater
  where endshere = (dps ++ [dp], fromJust conn)
        endslater = concatMap (flatten (dps ++ [dp])) trees

genHeads :: B -> Int -> Int -> Side -> [Either ExtFullIn IntFullIn]
genHeads base pa pb Exterior = [Left (EFI 1 1 ac1l acrr bc1l bcurr)]
  where (ac1l,acrr) = ((replicate (pa-1) 0) ++ [1], replicate pa 0)
        (bc1l,bcurr) = ((replicate (pb-1) 0) ++ [1], replicate pb 0)
genHeads base pa pb Interior = [ f d ac bc | d<-[0..base-1], ac<-(genCs pa), bc<-(genCs pb) ]
  where f :: D -> CS -> CS -> (Either ExtFullIn IntFullIn)
        f adlb ac bcr1 = Right (IFI adra adlb ac ac bclr bcr1)
        where (adra,bclr) = calcMiddle base (adlb,bcr1)

resultTree :: B -> Int -> Int -> Int -> Side -> [Tree]
resultTree base pa pb depth side = trees
  where heads = genHeads base pa pb side
        trees = concatMap (calcHorizontal base depth) heads

resultList :: B -> Int -> Int -> Int -> Side -> [(NumPart,ConnOut)]
resultList base pa pb depth side = concatMap (flatten []) $ resultTree base pa pb depth side

{- shows -}

showD :: D -> String -- shows a digit: 0 is "0", 10 is "A", 36 is "a", out of range is "@"
showD d = [chr $ if d<0 || 62<=d then 64 else if d<10 then d+48 else if d<36 then d+55 else d+61]
showCS :: CS -> String -- shows a list of carries; example: showCS [1,0,1,1,0] = 10110
showCS = foldl (\ s c -> s ++ (show c)) ""

showConn :: ConnOut -> String
showConn (ConnOut adl ac1 bcl) = (if adl then "T" else "F") ++ ":" ++ (showCS ac1) ++ ":" ++ (showCS ac1)

showNumPart :: Side -> Int -> NumPart -> String
showNumPart Interior len dps = space ++ str ++ space
  where str = foldl (\s (l,r) -> (showD l) ++ s ++ (showD r)) " " dps
        space = replicate (len-length dps) ' '
showNumPart Exterior len dps = lstr ++ space ++ rstr
  where (lstr,rstr) = foldl (\ (ls,rs) (l,r) -> (ls ++ (showD l), (showD r) ++ rs)) ("","") dps
        space = (replicate (2*(len-length dps)+1) ' ')

showResult :: Side -> Int -> (NumPart,ConnOut) -> String
showResult side len (dps,conn) = (showConn conn) ++ " " ++ (showNumPart side len dps)

showTree :: Tree -> String
showTree (Node node subtrees) = "(" ++ (showNode node) ++ " -> " ++ (showSubtrees subtrees) ++ ")"
  where showNode (FullOut (dl,dr) conn) = (showD dl) ++ (if isJust conn then "+" else "-") ++ (showD dr)
        showSubtrees ts = (foldl (\ str t -> str ++ (showTree t)) "" ts)

{- main -}

main' :: [String] -> String
main' args
  | result == "Number" = unlines . map (showResult side depth) $ resultList base pa pb depth side
  | result == "Tree" = unlines . map (showTree) $ resultTree base pa pb depth side
  | otherwise = ""
  where result = read $ args!!0 :: String
        base = read $ args!!1 :: Int
        pa = read $ args!!2 :: Int
        pb = read $ args!!3 :: Int
        depth = read $ args!!4 :: Int
        side = read $ args!!5 :: Side

main = do
  args <- getArgs
  if length args == 6
  then do putStr $ " " --main' args
  else do putStr $ "command line arguments:" ++ "\n result, \\\\"Number\\\\" or \\\\"Tree\\\\"
        putStr $ "\n base" ++ "\n # iterations in upper part" ++ "\n # iterations in lower part"
        putStr $ "\n depth = max length / 2" ++ "\n side, Exterior or Interior" ++ "\n"
        putStrLn $ "example (where search is the filename of the program)"
        putStrLn $ " search \\\\"Number\\\\" \\\\"26\\\\" \\\\"2\\\\" \\\\"2\\\\" \\\\"9\\\\" \\\\"Exterior\\\\"

```

## References

- [1] <http://p196.org/definitions.html>
- [2] [http://p196.org/identifying\\_lychrels.html](http://p196.org/identifying_lychrels.html)