

Tracing Pseudo-code

A *trace* of the function listed below is shown at right to illustrate the layout of a trace table. The lines that will be executed in succession given the input data are shown in the first column. The function call (line [0]) assigns a value to the input variables, in this case just n . The various columns of the trace table show all variables in the workspace of the function and in every row the current values of those variables are shown after the identified line of code is executed. (Comment lines have been removed.)

```
[0] bits = di2b1 (n)
[1] bits ← ∅
[2] p ← floor(log(n)/log(2))
[3] for k (0 to p)
    [3.1] if n ≥ (2^(p-k))
        [3.1.1] bits ← bits , 1
        [3.1.2] n ← n - 2^(p-k)
    else
        [3.1.3] bits ← bits , 0
```

The code converts a positive decimal integer to binary form.

In the session slides we have seen that the same algorithm could be extended to convert fractional numbers as well and that the branching control could be dispensed with by simply concatenating the result of the binary test $n \geq 2^{p-k}$ to the list of bits. However, it is only intended for instructional purposes.

```
[0] bits = di2b2 (n)
[1] bits ← ∅
[2] while n > 0
    [2.1] q ← floor(n/2)
    [2.2] r ← n - 2×q
    [2.3] bits ← r,bits
    [2.4] n ← q
```

The textbook suggests one algorithm for converting integer parts to binary and another algorithm for dealing with fractional parts. Pseudo code for this algorithm is shown at left.

Construct a trace table for the execution of `di2b2(41)` and explain what would happen if the binary test at [2] were to be replaced with $n \geq 0$.

Trace table.

Line	n	bits	p	k	$n \geq 2^{p-k}$
[0]	41	-	-	-	-
[1]	41	∅	-	-	-
[2]	41	∅	5	-	-
[3]	41	∅	5	0	-
[3.1]	41	∅	5	0	1
[3.1.1]	41	1	5	0	1
[3.1.2]	9	1	5	0	1
[3]	9	1	5	1	1
[3.1]	9	1	5	1	0
[3.1.3]	9	1,0	5	1	0
[3]	9	1,0	5	2	0
[3.1]	9	1,0	5	2	1
[3.1.1]	9	1,0,1	5	2	1
[3.1.2]	1	1,0,1	5	2	1
[3]	1	1,0,1	5	3	1
[3.1]	1	1,0,1	5	3	0
[3.1.3]	1	1,0,1,0	5	3	0
[3]	1	1,0,1,0	5	4	0
[3.1]	1	1,0,1,0	5	4	0
[3.1.3]	1	1,0,1,0,0	5	4	0
[3]	1	1,0,1,0,0	5	5	0
[3.1]	1	1,0,1,0,0	5	5	1
[3.1.1]	1	1,0,1,0,0,1	5	5	1
[3.1.2]	0	1,0,1,0,0,1	5	5	1

Pseudo Code for Data Analysis

Write pseudo code for the following functions.

- (i) The function is called `sum`; its input is a list of numbers and its output is the sum of all the numbers in the list
- (ii) Function, `mean` has a list of numbers as its input and returns the arithmetic mean of the numbers in the list
- (iii) `dfm` is a function that computes the deviations of individual entries of a list from its mean value.
- (iv) The average of the squares of the deviations of the entries of a list from the mean of the list is computed by the function `msd`
- (v) The *standard deviation* of a list of numbers, is computed by taking the square root of the mean of the squares of the deviations of list entries from the mean value of the list.

Of course it is assumed that you can use all the arithmetic operators that may be needed and that they work pair-wise on lists, eg. `3 4 5 * 2 7 9` returns `6 28 45` and with scalar extension, ie. `3 4 5 * 8` returns `24 32 40`. Once you know the syntax for a particular function, you will know how to call it, if required, inside the code for another function. While you would not be able to test any code in practice until the codes for all the functions used have been successfully written, this does not mean that you have to work on these individual codes from the bottom up – that is, in the order shown above. You can also work from the top down and write the definition for the last function – the one for the *standard deviation* – before any of the functions that it calls upon have been written, **provided**, of course, that you know the syntax for calling these functions.

- So, **the first job** is to write syntax lines for each of the functions required.

Besides the arithmetic functions you may assume that the following list functions (ie. functions whose domains, or inputs, are lists of numbers or characters) are available:

`length(L)` returns the number of entries in a list, L of numbers or characters.

`floor(L)` returns a list of the biggest integers below each of the corresponding entries of the list

`integers(L)` returns the list of integers from the first entry of L to the last (or between them if they are not integers).

`first(L)` returns the first entry of L.

`behead(L)` returns the list L with the first entry removed.

`last(L)` returns the last entry of L.

`curtail(L)` returns the list L with the last entry removed.

`L , M` returns the list consisting of the entries of L followed by the entries of M.

Function `,` is sometimes called *catenate*. It has an in-fix syntax like `+` or `*`.

For simplicity we will use \emptyset to represent any **empty** list whether it be numeric or character.

Integers in a computer

With 12-bits for storing integers find:

- (a) the largest and smallest integers that can be represented?
 - (b) the computer representations for -1 , -27 , -200 , 37
 - (c) the integers represented by 111101011001 , 110110100011 and 000011111110
-