Monaco

A program to answer probability questions, especially from dice and card games.

Second Tutorial

Introduction

This is the second tutorial for the program called **Monaco** that can be used to answer questions that can arise in a wide range of games and other places. This tutorial assumes familiarity with the material in the first tutorial. Its purpose is to introduce more features of the program that allow you to solve more problems, or to simplify the solutions of some problems.

This tutorial is less structured around problems than the first tutorial, but there are still several problems described that are used to motivate the inclusion of some of the new features.

When producing exact or approximate results we refer to *exact mode* or *approximate mode*. A *term* refers to anything in an expression that evaluates to an integer (a number) or a list.

Option Variants

The first tutorial simplified describing program options by indicating they start with a `-` sign. However, there are also options that start with a `+` sign, each a variant of the corresponding option starting with a `-` sign. For example, there is a variant `+statistics` of `-statistics` and a variant `+table` of `-table`; the output from these two new options is described in the following section. Usually, the `-` and `+` forms of an option cannot be used together, but an exception is that the options `-table` and `+table` can be used together.

The option `+all` is as the option `-all`, but using `+statistics` rather than `-statistics`.

Alternative Dice

As described in the first tutorial, terms such as `sorted4from{2,3,3,4,4,5}` allow the use of dice with arbitrary face numbering. In approximate mode, or when the order of the dice matters, `4d{2,3,3,4,4,5}` can be used instead, or for one die `d{2,3,3,4,4,5}`. In either of these new terms the dice faces do not need to be constant; for example, you can use `d{d2,d3}` or `2d{d2,d3}`; using the latter term, the selection from `d2` and `d3` is made twice.

The sorted term can be used to demonstrate the output of the options `+statistics` and `+table`. Using `+statistics`, the output from `sum(sorted4from{2,3,3,4,4,5})` is:

```
Number of evaluations      = 35
Number of results          = 1296
Mean                       = 14
Standard deviation         = 1.91485
Minimum result             = 8
Maximum result             = 20
Median                     = 14
Mode                       = 14
```

This is as the output from `-statistics`, except for the added last two lines. Note that the number of evaluations is lower than for `sorted4d6`, where it is 126; the program can take advantage of the repeated dice face values. This distribution has a unique median and mode, which because the distribution is symmetric are equal. When the median and mode are not unique, for example using 3 dice instead of 4, the last two lines become, for that example:

```
Median                     = 10.5 = 21/2
Modes                      = 10, 11
```

The output from `+table`, for the example with four dice, is:

```
 N    P(N)      P(<=N)      P(>=N)
 8   1/1296     1/1296        1
 9    1/162      1/144     1295/1296
10    2/81      41/1296     143/144
11    7/108    125/1296    1255/1296
12    10/81     95/432     1171/1296
13   29/162    517/1296     337/432
14   131/648   779/1296    779/1296
15   29/162     337/432    517/1296
16    10/81    1171/1296    95/432
17    7/108    1255/1296   125/1296
18    2/81      143/144     41/1296
19    1/162    1295/1296     1/144
20   1/1296        1        1/1296
```

This output, i.e. using the option `+table`, is usually only possible in exact mode.

Constants

Sometimes you want to set a list once and never change it. For example, consider the problem of finding the distribution of the length of the longest suit in any player's hand playing *bridge*, in which a standard deck of cards is equally divided among four players. There are too many ways to shuffle all the cards and distribute them to use exact mode, so we use approximate mode. We number the 52 cards by suit (0 to 3) and shuffle them, then the first 13 cards are the first player's hand and so on. Rather than dividing the cards into separate lists, we keep them in one list and add 0, 4, 8 or 12 to the cards that belong to each of the four players.

This can be done by `v0:=sequence52/13;count_mode(shuffle(v0)+v0*4)`, using the list `v0` twice. The function `shuffle` performs a complete permutation of a list, here of `v0`.

The distribution of that expression is not the point of this example, but for possible interest, here is the output from the option `-histogram`, for ten million evaluations:

```
 4 -  292777 ~ 0.0292777  [0.0291734, 0.0293824]
 5 - 4020711 ~ 0.402071   [0.401767, 0.402375]
 6 - 4225035 ~ 0.422503   [0.422197, 0.42281]
 7 - 1262658 ~ 0.126266   [0.12606, 0.126472]
 8 -  183236 ~ 0.0183236  [0.0182407, 0.0184069]
 9 -   14882 ~ 0.0014882  [0.0014645, 0.00151229]
10 -     684 ~ 6.84e-05   [6.34591e-05, 7.3725e-05]
11 -      17 ~ 1.7e-06    [1.03953e-06, 2.74462e-06]
```

Note that ten million results is not enough to see a 12 or 13 card suit. It is possible to produce that distribution exactly using the program, but how to do this is not straightforward and is beyond the scope of this tutorial.

But in that example, all we do to `v0` is to set it once, to a fixed list. We thus do not need it to be a variable. We can instead use the *list constant* `u0`, here simply replacing all three uses of `v0` by `u0`. Why should we do that? In this case it probably does not matter, but in some cases evaluation can be more efficient using constants, and there are also some places, as will be described, where a constant list can be used but a variable list cannot be used.

There are also further list constants `u1` to `u9` and the integer constants `c0` to `c9`.

The term that is used to initialise a constant can include other constants. For example, after defining the constants `c0` and `u0`, the constant `c1` could be defined by `c1:=c0*sum(u0)`.

Initialisation as an Option

In the last example, we can simply replace `v0` by `u0` because then `u0` is initialised first. Unlike variables, which can be assigned to (almost) anywhere, constants can only be initialised at the start of an expression. But this is not the only way in which constants can be initialised.

We can instead move the initialisation of a constant to an option. In the last example we can remove the initial `u0:=sequence52/13;` and add the option `-u0 sequence52/13`, before the (main, see below) expression as usual. There are two new things to note here:

- We now have an option that consists of more than one parameter. In all such cases – whether initialising a constant or otherwise – it is only the first of the parameters that comprise an option that always starts with `-` or `+`, and the number of parameters in the option can always be determined from the first parameter of the option.
- In that option, the second parameter is an expression. Unlike previous expressions, it is a *list expression*, whose result is a list. Because it is initialising a list constant, as indicated by the `-u0`, the program knows that what follows is a list expression. The program always knows if an expression in an option is an integer or list expression.

When there is any possibility of confusion, we refer to the only expression there has been before this point as the *main expression*, but continue to just say the expression when that is clear. Because it is initialising a constant, the expression in the option `-u0` does not have the flexibility of the main expression; in particular it cannot include randomness or use variables.

Similarly, the constants `u1` to `u9` and `c0` to `c9` can be set using the options `-u1` to `-u9` and `-c0` to `-c9`. Another form of initialisation, of the randomness pool, that can be replaced is that `pool_nset[`*m*`]of[`*n*`]` can be replaced by the option `-pool` *m n* for constant *m* and *n*. Like the term that it replaces, but unlike most options, `-pool` can be used more than once.

There is usually no critical reason to pick either the use of the option `-u0` or the initialisation of `u0` in the main expression. One often useful convention is to use options for parameters of the problem at hand, and to use initialisation in the main expression for derived constants. A case where it is necessary to use an option to initialise a constant is when using that constant in another option, such as in the option `-eval` that is described later in this tutorial.

Initial Values of Variables

The initial value of an integer or list variable can be set using an option. For example, if an expression starts `r0:=2` then that term can be replaced by the option `-r0 2`, or if it starts `v0:=sequence4` then that term can be replaced by `-v0 sequence4`. Initialisation can also use expressions such as in `-r0 sum(v0+sequence[c0])`, assuming that `c0` is already defined and `v0` is already initialised. However, do not try initialising a random value of `r0` as the effect – not described in this tutorial – is unlikely to be as you expect or want.

List Lengths

The first tutorial describes how the lengths of `v0` to `v9` are `s0` to `s9`, which we now can see to be another form of constant. Similarly, the lengths of `u0` to `u9` are `k0` to `k9`. But while all that can be done with `k0` to `k9` is to use them as those lengths, that is not all that can be done with `s0` to `s9`. These constants can be initialised in either of the two ways that `c0` to `c9` can be initialised. For example, we can either use the option `-s0 4` or use `s0:=4` at the start of the main expression. Either of these then sets the length of `v0`, in those examples to 4.

If, for example, `s0` is used to set the length of `v0`, then `v0` can be immediately used with its initial value being a list of `s0` zero-valued elements. Alternatively, `v0` can then be set by, for example, `v0:=sequence` – the length of a term such as `sequence` can be omitted if it can be deduced – because we know the length of `v0`, and the length of `sequence` must match.

We can also use constants to set the length of a list term such as `sequence`. For example, after `-c0 52` we can use `sequence[c0]` in any expression. We can even use a constant expression in such cases; for example, if `c1` and `u0` have been initialised then we can use a list such as `sequence[c1*k0]`. As list lengths are fixed, we can only use constants, not variables. There is no significant inefficiency in using constants, including using calculated constants such as `c1*k0`, all such constants are calculated once per run, not per evaluation.

We thus can parameterise the example of the bridge hands to use a deck containing `c0` cards of each of `c1` suits, shared out between `c1` players by first initialising the problem parameters `c0` and `c1`. For example, this can be by using the options `-c0 13` and `-c1 4`, and then using the main expression `u0:=sequence[c0*c1]/c0;count_mode(shuffle(u0)+u0*c1)`.

Multiple Constants and Variables

We can simplify a list of integer variables such as `{r0,r1,r1}` to, in that case, `r011`. When used in a term such as `v0:=r011` this is just a notational convenience. However, if there are no repetitions among the variables, then we can use a term such as `r021:=v0` to set `r0`, `r2` and `r1`, in that order, from the elements of `v0`, which must have length 3. An example where this is useful to significantly simplify an expression is described in a later section.

The ability to use a term such as `r011` as a list of length 3 does not extend to single variables such as `r0`, those are handled – in a more flexible way – as described in the following section.

This notation can be used with constants as well as variables. For example, `c012` is the list `{c0,c1,c2}`. It can also be used to initialise constants and variables; for example, after the option `-c012 rsequence3` , `c0` is 2, `c1` is 1 and `c2` is 0. The `3` on `rsequence3` can be omitted; such list lengths can often be omitted and deduced by the program, but setting or combining with a constant or variable of known length is the only case considered here. Similarly, in the option `-c012 4` the `4` is a list whose is length deduced as 3, and `c0`, `c1` and `c2` all equal 4. Lists such as `s012` and `k012` can be used; the former can also be initialised.

For lists, this multi-digit notation joins lists, so, for example, `v012` is `v0#v1#v2`, and similarly for constant lists such as `u012`. Options such as `-v012` can be used to initialise variable lists, but only if their lengths (or, sometimes, all but one of their lengths) are already known.

Type Conversions

Contrary to the usual expression rules that require an integer, not a list, when an integer is expected, we can use a list as an integer, but only when the list is a single or multiple constant or variable, for example `u0` or `v12`. The integer value is the sum of the elements in the list. For example, if `v0` is {2,3,5}, then after `r0:=v0`, `r0` is 10.

Similarly, we saw in the first tutorial that we can use a number as a list, for example as `v0+1`. This also applies to a single constant or variable; for example, we can use `v0+c0` or `u0*r0`. The `c0` and `r0` (and the 1 in the previous example) are converted to lists of unknown length, the length being deduced as that needed to be added to `v0` or `u0`, i.e. as `s0` or `k0`.

As described in the first tutorial, any integer term can be put in `[]` and used as a list, with an optional length in `[]` following it. Alternatively, any list term can be put in `[]` and used as an integer term, but this is as its length, not as its sum. As the length is fixed, that value is usually not of interest, rather this is usually a way to use a list function with a wanted side-effect when an integer term is expected. The term is thus usually followed by `;` and an integer term.

Timing

The example above distributing the bridge deck, using ten million evaluations, takes about 7 seconds to run on my computer; it might take more or less time on yours.

To determine that, I added the option `+time` and got – the first time I used it – the output:

```
Time           = 7 seconds.
Processor time = 6.75626 seconds.
```

Of course each time I run this, the latter figure changes, as sometimes might the former figure. The first figure is the real time taken, although owing to how it is measured it might not be the exact nearest whole number of seconds. The meaning of the second figure might depend on your computer, but usually it will be as described and similar to the first figure – differences are most likely if your computer is doing significant other things at the same time. Incidentally, if I use `v0` rather than `u0` I get a slightly larger (by about 0.2 second) processor time.

There is also an option `-time` (there is always a `-` option if there is a `+` option). The difference is that `-time` only includes main expression evaluation time, while `+time` includes any initialisation time. In most cases there is no significant difference between these times.

However, these times are only available after the run has finished; before then you might start a run and during those seven seconds wonder how long this run will take – or even more so if you tried a hundred million evaluations and were waiting over a minute.

To get an earlier idea of the time needed, add the option `-progress`. During the course of the run, when it is, for example, about 23% complete this will have produced the output:

```
012345678901234567890123
```

From this you can assess how long the run is likely to take. In this approximate mode example you can adjust the run length to fit how much time you want to allow. In exact mode runs you might have less flexibility, but you can decide whether the run is practical. In particular if output is stuck on a single `0` – which is output before the first evaluation – it is unlikely to be practical.

The complete output from the option `-progress` usually looks like:

```
012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789
01234567890123456789
```

The split onto two lines is real, not just here to avoid having to reduce the font size even further. This is because some output – including that from `-progress` – is limited to a width that is, by default, 80 characters. You can increase that width, for example to 101 characters (the minimum that avoids the split for `-progress` output), by using the option `-wrap 101`.

Counting Evaluations and Results

There is another way to try to assess how long a run might take. If we consider the parameterised bridge hands example, using the constants `c0` and `c1` initialised to 13 and 4, then in exact mode we get the error message `Error: Too large for exact mode`. But suppose we reduced `c0` and `c1` to where we no longer get that, for example all the way to `-c0 6` and `-c1 3`, could we use exact mode now? No, this run is still too long, it gets stuck on the single `0` output from `-progress` that is produced before any evaluation of the main expression. To find out why, we can use the option `-numbers`. This produces the number of results – and evaluations when different – from `-statistics`, but before evaluation starts.

The output from the option `-numbers` in this case is:

```
Number of results          = 6402373705728000
```

That is far too large a number to even consider getting a result from. However, we can get an exact answer in this case by replacing `shuffle(u0)` by `permute[k0]from[u0]`. This term

is the permutation companion to the similar combination term described in the first tutorial. Because this permutation length is the same length as the list length, a convenient use of the constant `k0`, the permutation term shuffles all of the cards, but is more efficient than `shuffle`.

Using that permutation term we now have the `-numbers` output:

```
Number of evaluations      = 17153136
Number of results          = 6402373705728000
```

We also have the, not very useful, `-table` output:

```
N     P(N)       P(<=N)       P(>=N)
2  0.0424995   0.0424995  1
3  0.569459    0.611958   0.9575
4  0.354163    0.966121   0.388042
5  0.0333952   0.999516   0.0338793
6  0.00048411  1          0.00048411
```

Seventeen million evaluations – the figure that defines the work needed – is practical, and takes about 5 seconds on my computer. That increased efficiency (by a factor of nearly four hundred million) is because the permutation term can intelligently use that the list `u0` contains repetitions, but `shuffle` does not. But using the permutation term is still inefficient, it orders the cards in each hand, and we do not care about that. We cannot make this example much larger, or possibly any larger at all, and remain practical. The previously indicated solution to this problem overcomes that limitation, but needs more material than this tutorial can cover.

New Operators

The first tutorial introduced the functions `list_eq` and `list_ne` to compare lists, advising you not to use the operators `==` or `!=` that are used to compare integers. However, you can compare lists with `==` or `!=`, but the results are not overall comparisons of the lists, but lists formed from element by element comparisons of the lists. This also applies to the operators `<`, `<=`, `>` and `>=`. So, for example, `{1,2,3}<{3,2,1}` is {1,0,0}.

Alternative Function Notation

Using the function `min` described in the first tutorial, consider the example `min({r0,d6})`. That can instead be more succinctly written as `min{r0,d6}`, and similarly for any other integer function, or any list function, of a single list, in particular `max`.

Additional Functions

The first tutorial described some of the most useful functions to use in expressions; this section and the following two sections describe some more functions. You might choose to skip these three sections on first reading and return to them when a function you do not recognise is used, but these sections also include functions that are otherwise not used in this tutorial.

Three integer functions that can be applied to an integer *n* are `abs(`*n*`)`, the absolute value of *n*, i.e. *n* if *n* ≥ 0 or -*n* if *n* < 0, `sign(`*n*`)`, the sign of *n*, i.e. 1 if n > 0, 0 if *n* = 0 or -1 if *n* < 0, and `boolean(`*n*`)`, the logical value of *n*, i.e. 1 if *n* ≠ 0 or 0 if *n* = 0.

There are also list functions `abs`, `sign` and `boolean` that applied to a list *v* apply the similarly named integer function to each element of *v*, producing a list of the same length. This means that there are, for example, two functions named `abs`. This does not cause a problem because when analysing an expression, the program always “knows” the *type* (integer or list) of every term, starting by knowing the type of every expression, and thus it always picks the right function. It then knows what types each function’s argument(s) are, and continues accordingly.

The integer `number_combin(n,m)` is the number of combinations of *m* items from *n* items, often written ${}^{n}C_{m}$; for example, `number_combin(5,3)` is 10. Similarly, the integer `number_permut(n,m)` counts permutations; for example, `number_permut(5,3)` is 60.

The list function `unit£(n)` is a list with length `£` whose elements are all zero, except the *n*-th element, which is one. For example, `unit4(1)` is {0,1,0,0}. Just use `unit4` for {1,0,0,0}. The integer value, "wraps around", `unit4(-1)` and `unit4(7)` are both the same as `unit4(3)`.

The six integer functions `count_eq` etc. described in the first tutorial compare a list *v* to an integer *n*. There are also six list functions with the same names, with two list arguments *v* and *u*; these do not need to have the same length. The resulting list has the same length as *u*. If the *k*-th element of *u* is *n*, then the *k*-th element, for example, the list function `count_ne(v,u)` is, using the integer function, `count_ne(v,n)`. *v* is only evaluated once.

There is also a list-only variant `count_seq£` of the list function `count_eq`, where, as in the first tutorial, `£` represents a constant integer, including the new constant cases described above using `[]`. `count_seq£(v)` is equivalent to `count_eq(v,sequence£)`.

There is a set of six integer functions `find_eq` etc., with the same arguments as `count_eq` etc., whose result is the number of the first element in the list that satisfies the indicated condition. For example, if `v0` is {2,3,5,7} then `find_gt(v0,3)` is 2. If there is no such element then the result is the length of the list, here 4. There are also functions `rfind_eq` etc. that find the last such element, with result the element number or -1 if there is no such element.

Similarly, there are functions `find_min`, `find_max`, `find_mode_min` and `find_mode_max` that find the element number of the first element of their single list argument that is equal to the result of `min`, `max`, `mode_min` or `mode_max` applied to the same list. Note that these can always find such an element. There are also `rfind` functions to find the last such element.

The functions `unique_min` and `unique_max` have the result whether their single list argument has a single minimum or maximum element, respectively.

Another set of six functions is `replace_eq` etc. For example, `replace_lt(v,m,n)` replaces all elements of *v* that are less than *m* by *n*, where replace means create a new list that is *v* with those changes, and that is the result of the function. For example, `replace_lt(sequence5,2,-1)` is the list {-1,-1,2,3,4}.

Another list function that acts as a multiple version of the similarly named integer function is `get`. The *k*-th element of `get(u,v)`, where *u* and *v* are lists, not necessarily the same length, is `get(n,v)`, where *n* is the *k*-th element of *u*. The length of this list is the length of *u*.

The first tutorial introduced the functions `head£` and `tail£` that produce a sub-list of length `£` from their list argument *v*. However, it did not describe what happens when `£` is greater than the length of *v*. In this case the resulting list still has length `£`, it is *v* with additional zeros added. So if *v* is {1,2,3} then `head5(v)` is {1,2,3,0,0} and `tail5(v)` is {0,0,1,2,3}. Using zeros when outside a list is also how another function `mid£` works. `mid£` can take a sub-list from anywhere in a list, not necessarily at a fixed point, thus if we consider `mid3(r0,{1,2,3,4,5})` the result is, for example, {0,0,1}, {2,3,4} or {4,5,0} if `r0` is -2, 1 or 3, respectively.

The function `copy£(v)` creates a list that is `£` copies of *v* joined together. For example `copy2{2,3,5}` is {2,3,5,2,3,5}. The related `duplicate2{2,3,5}` is {2,2,3,3,5,5}.

The functions `sigma` and `delta` convert a list to another list of the same length. They are inverse functions: either followed by the other takes you back to the original list. Both leave the first element of the list unchanged; after that `sigma`'s result contains the cumulative sums of its argument's elements, while `delta`'s result contains the differences of its argument's elements. For example, `sigma{2,3,5,7}` is {2,5,10,17} and `delta{2,3,5,7}` is {2,1,2,2}.

Two lists, *u* and *v*, with equal length, can be combined as a *dot product* by summing the products of their corresponding elements using `dot(u,v)`, which is thus equivalent to `sum(u*v)`. For example, `dot({0,1,2,3},{2,3,5,7})` is (0×2)+(1×3)+(2×5)+(3×7) = 34.

With two lists *u* and *v*, which do not need to have the same length, we sometimes want to know how many numbers are in both lists. This is `overlap(u,v)`. If a number appears more than once in *u* and/or *v*, its contribution to the result is the minimum number of times it appears in either *u* or *v*. For example, `overlap({1,2,3,3},{2,3,3,3,4,4})` is 3.

Two functions that are related to the function `counts` introduced in the first tutorial are `occurs` and `pattern`. `occurs(v)` creates a list that replaces each element of the list *v* by its occurrence number in *v*, counting from zero. For example, `occurs{1,3,1,2,1,3,1}` is {0,0,1,0,2,1,3}. `pattern(v)` replaces each occurrence of the *k*-th different number in the list *v* by *k*, counting from zero. For example, `pattern{1,3,1,2,1,3,1}` is {0,1,0,2,0,1,0}.

The integer function `select(v)` selects a random element from the list *v*, for example after `v0:={2,3,5,7}`, `select(v0)` is 2, 3, 5 or 7, each with probability ¼. The list function `select£(v)` creates a list of `£` such selections. For example, with the same v0, there are 64 possible results from `select3(v0)`, from {2,2,2,2} to {7,7,7,7}. When applied directly to a braced list, for example, `select{2,3,5,7}` is equivalent to `d{2,3,5,7}` and similarly `select3{2,3,5,7}` is equivalent to `3d{2,3,5,7}`.

Output Functions

The first tutorial emphasised three forms of question that the program is designed to answer, using standard output options, but those are not the only forms of problem that the program can handle. Another form of problem is “find all the instances of this that satisfy that property”. To do this, the program can loop over the instances of this, sometimes one per evaluation of the main expression, sometimes in a loop within an expression.

An example – but there are much better ways to do this – is to search for Pythagorean triples (*a*, *b*, *c*) of integers that are all greater than zero and such that $a^2 + b^2 = c^2$. This is of course not a problem from a game, but we can still use the program to solve it. We also limit what we are interested in to triples having no common factor. Assuming that the triple is `{r0,r1,r2}`, i.e. `r012`, the required test for a Pythagorean triple is `r0*r0+r1*r1==r2*r2`, and the required test for no common factor is `hcf(r012)==1`, where `hcf` is a function to find the highest common factor of the elements of its list argument; it has a companion `lcm` to find the least common multiple of the elements of a list. We combine the two tests with `&` and use a further `&` to only output the list when both tests succeed, using the integer function `lwrite`, here `lwrite(r012)`. To output an integer we would use the integer function `write`.

The easiest way to set `r012` is to use exact mode and a sorted list `r012:=sorted3d[c0]`, where `c0` is the largest number in any triple. Note that this is the promised case where that would be considerably more awkward to write without the `r012` notation. Also note that exact mode turns what would be a random selection in approximate mode into a loop, here over ordered triples (in the required order here). As we are not reporting statistics, that this is a weighted term, and the weights it produces, can be ignored. The combined expression is thus:

```
r012:=sorted3d[c0];r0*r0+r1*r1==r2*r2&hcf(r012)==1&lwrite(r012)
```

If we let `c0` be `25`, producing short output to report here, that output is:

```
{3,4,5}
{5,12,13}
{8,15,17}
{7,24,25}
```

Each list is on a separate line because each output is from a separate evaluation of the main expression. However, output from each evaluation using `lwrite` and other output functions is on the same line, unless the function `nline` is used to force a new line. Another useful function is `space` to insert a space; for example, we could replace the use of `lwrite` in that expression by `(write(r0);space;write(r1);space;write(r2))` to get the output:

```
3 4 5
5 12 13
8 15 17
7 24 25
```

if this is preferred. To output, for example, two spaces use `space(2)`. To output a rational number, say `r0` divided by `r1`, in lowest terms, use `write_ratio{r0,r1}` – or in that particular example, `write_ratio(r01)`. If you want that ratio output as a real number, use `write_real` instead of `write_ratio`. If you want text output, for example `Number`, then use either `swrite("Number")` or `swrite('Number')` as is more convenient to you.

Returning to the Pythagorean triples, setting `c0` as 1000 requires 167,167,000 evaluations and produces 158 triples, up to {372,925,997}. That is a reasonable limit for this example.

Additional Loop Functions

The first loop function introduced in this second tutorial is `until`, a companion to the loop function `while` described in the first tutorial. `while`'s properties include that it tests first, so might loop zero times, and it loops while a condition is true. `until`'s corresponding properties are that it tests last, so always loops at least once, and it loops until a condition – its second argument – is true (i.e. while a condition is false).

`until(`*m*`,`*n*`)` is thus equivalent to:

(a) Introduce a temporary variable *k*, its initial value is never used.

(b) Evaluate *m* and set *k* to its value.

(c) Evaluate *n*; if it is true then finish with result *k*, otherwise continue.

(d) Go back to (b).

The first tutorial described a loop function `rloop$(`*n*`)`, where `$` is a digit from `0` to `9`. Among its other properties are that the result of the loop is the last value of *n*. Sometimes all we want from a loop is to evaluate *n*, we do not need a final result, in which case that function is fine.

But sometimes we want more. One way to use, for example, `rloop0` is to sum *m* values of *n*, which it is assumed uses `r0`, as, for example, `r1:=0;rloop0(`*m*`,r1+=`*n*`)`. The final sum is then available both as the result of `rloop0` and in `r1`. But we can do better than that. We can instead simply use `rsum0(`*m*`,`*n*`)`, or if we still want `r1`, we can use `r1:=rsum0(`*m*`,`*n*`)`.

Two other common ways to combine *m* values of *n* in such a loop are to find their minimum or maximum values, and for those we can use `rmin0(`*m*`,`*n*`)` and `rmax0(`*m*`,`*n*`)`. Better yet, while when using `rloop$` and `rsum$` the final value of `r$` is that of *m*, but when using `rmin$` and `rmax$` the final value of `r$` is its value where the minimum or maximum value of *n* was produced – or if that value was produced more than once, when it was first produced.

Sometimes, what you want to loop over is not a sequence 0, 1, 2, ... but the elements of a list. The functions `rvloop$`, `rvsum$`, `rvmin$` and `rvmax$` are like `rloop$`, `rsum$`, `rmin$` and `rmax$`, but replace their first integer argument by a list, and loop `r$` over the elements of the list. When finished they set `r$` in the same way as `rloop$` etc. do. Thus, for example, `rvmax0({2,-4,3,4),r0*r0)` is 16, and sets `r0` to 1.

Distributions

Sometimes it is convenient to use a *distribution*, a list *v* that contains as its element *k* the relative probability of the result *k*. A sample from that distribution is given by `distribute(v)`. For example, `distribute{2,1,3}` has probability ⅓ of 0, ⅙ of 1 and ½ of 2.

To be usable in exact mode, *v* must be constant. In this case we can use, for example, `distribute(u0)`, but, in the same way that `sorted3d6` is better than `3d6`, a better term to use is `select_by[u0]`, so the equivalent of `distribute{2,1,3}` is `select_by{2,1,3}`.

To get multiple results from a distribution we can use the numbered list function of the same name, thus, for example, `distribute4{2,1,3}` has 81 possible results such as {2,0,1,0}, that particular result having probability (½)(⅓)(⅙)(⅓) = 1/108. In exact mode prefer, for example, using `selection4by{2,1,3}`. Using constants we can use, for example, `selection[c0]by[u0]`, corresponding to `distribute[c0](u0)`. If the order of the results does not matter then we can use, for example, `sorted[c0]by[u0]`. There is also an unsorted `selection[c0]from[u0]` that corresponds to `select[c0](u0)`. Note that `from` indicates a list, `by` indicates a distribution. The unsorted terms require fewer evaluations than results if the list selected from, or defined by a distribution, has any repetitions in it.

For a distribution *v* over the values in the list *u* rather than over 0, 1, ... use the integer or list function `get`. For a single value use `get(distribute(v),u)`, or for a constant list *v* use `get(select_by[v],u)`. For a list of `£` values use `get(distribute£(v),u)`, or for a constant list *v* use `get(selection£by[v],u)`. *u* does not need to be constant.

List Results

A bridge problem that we can solve exactly is one in which given a bridge hand where we only care about suits, what is the distribution of the lengths of those suits, i.e. the numbers of cards in the suits, without caring which suit is which, from the most balanced hand {4,3,3,3} to the most extreme hand {13,0,0,0}. Which distribution is most common?

We can produce the required lists using the term:

```
rsort(count_seq4(combine13from[sequence52%4]))
```

`count_seq4` is one of the new functions described above, here counting the numbers of, 0s, 1s, 2s and 3s in its argument, the hand, as its four elements. We use `rsort` because we do not care which suit is which and we want the results in the form indicated above. But now we want to know the statistics of each possible list result, and we need a new feature to do that.

This feature is in two parts. First, we record that list as a *list result*, producing the expression:

```
list_result(rsort(count_seq4(combine13from[sequence52%4])))
```

Second, we report those statistics using either of the options `-lists` or `+lists`, which order the results differently. In both cases the output is longer than is reasonable to present here, so an ellipsis … in the following output replaces 33 other lists.

Using `-lists` the output is:

```
Number      -           39
{4,3,3,3}   -  66905856160 ~ 0.105361    = 836323202/7937669495
{4,4,3,2}   - 136852887600 ~ 0.215512    = 342132219/1587533899
{4,4,4,1}   -  19007345500 ~ 0.0299322   = 190073455/6350135596
{5,3,3,2}   -  98534079072 ~ 0.155168    = 6158379942/39688347475
{5,4,2,2}   -  67182326640 ~ 0.105797    = 839779083/7937669495
...
```

```
{13,0,0,0} -            4 ~ 6.29908e-12 = 1/158753389900
Total      - 635013559600
```

That number of evaluations would be impractical, but the `-numbers` output tells us how efficient `combine13from[sequence52%4])` is, that output being:

```
Number of evaluations      = 560
Number of results          = 635013559600
```

The `-lists` output is sorted according to the numbers in the lists. The alternative `+lists` sorts the lists by probability, most likely first, and output from it, again with 33 omitted lines, is:

```
Number     -           39
{4,4,3,2}  - 136852887600 ~ 0.215512    = 342132219/1587533899
{5,3,3,2}  -  98534079072 ~ 0.155168    = 6158379942/39688347475
{5,4,3,1}  -  82111732560 ~ 0.129307    = 1026396657/7937669495
{5,4,2,2}  -  67182326640 ~ 0.105797    = 839779083/7937669495
{4,3,3,3}  -  66905856160 ~ 0.105361    = 836323202/7937669495
...
{13,0,0,0} -            4 ~ 6.29908e-12 = 1/158753389900
Total      - 635013559600
```

We can see that the most balanced hand, {4,3,3,3} is only the fifth most common hand.

Multiple Results

We can also use list results, recorded in the same way, to collect more than one set of results at a time. For example, suppose that we have four players playing a game, each with different dice. Player A has a d10, player B has d8+1, player C has d6+2 and player D has d4+3, which all have the same average. They all roll together and each player scores one point for each other player with a lower roll. What are their mean scores across all possible rolls?

We can create their scores as the list `v0:={d10,d8+1,d6+2,d4+3}`, then get their scores and record them as a list result using `list_result(count_lt(v0,v0))`. Then we can get the means and other statistics using the option `+list_stats`, whose exact mode output is:

```
Number of list results     = 1920
List means                 = {1.35,1.325,1.30417,1.30417}
                           = {27/20,53/40,313/240,313/240}
List standard deviations   = {1.27083,1.16842,1.04442,0.928251}
List minima                = {0,0,0,0}
List maxima                = {3,3,3,3}
List medians               = {1,1,1,1}
List modes                 = {0,0,1,1}
```

If we instead had used `-list_stats` then we would not get the last two lines of output.

Although the program does not directly support lists of rational or real numbers, this output uses a similar notation to that of lists of integers to present the multiple means and other values. Note that is strictly only as output, you cannot use those notations in an expression.

The player with the best average score is player A, followed by player B, but players C and D have the same average score. We might want to know which is most likely to get to a target score first by repeating and accumulating scores until one player reaches a target score. Unfortunately we cannot do that exactly, even for a very low target score, by modifying this example, as the players might all roll the same number and score zero each, which could make the game last for an indefinite length of time. So we now switch to approximate mode.

We also need to handle the case when two or more players reach the target score together. So we instead say that a player wins with at least the target score and a greater score than any other player. Note that this can mean that a player who was not one of the first to reach the target score can win (which we consider here as a game rule). So setting a target score of `c0`, where we use `-c0 10` in the example below, we can define most of our expression as:

```
until(v0:={d10,d8+1,d6+2,d4+3};v1+=count_lt(v0,v0),max(v1)>=c0&unique_max(v1))
```

`until`, `unique_max`, and the list function `count_lt` are new functions described above.

We can now finish that expression by adding `list_result(count_gt(v1,v1)==0)` at the end. Note that the comparison `==0` between the result of the list function `count_gt`, and a constant integer produces a list of those comparisons, as required. We can now use the option `-list_stats` with the option `-probability` to get the output, for ten million evaluations:

```
Number of list results     = 10000000
List false result numbers  = {6806637,7241833,7773019,8178511}
List true result numbers   = {3193363,2758167,2226981,1821489}
List probabilities         = {0.319336,0.275817,0.222698,0.182149}
List 95% conf intervals    = {[0.319047,0.319625],[0.27554,0.276094],
                              [0.22244,0.222956],[0.18191,0.182388]}
```

Note that – for reasons beyond the scope of this tutorial – we cannot simply say that because the 95% confidence intervals do not overlap, we have 95% confidence that the players will win more often from A down to D. However, these intervals are sufficiently separated that we can be all but certain this is the case. Although their mean one round scores are the same, player C is more likely to win this game than player D is.

Errors

The errors described in the first tutorial are ones that cause a program run to fail. However, there is a second kind of error, ones that happen during evaluation of the main expression. (Errors of this sort evaluating other expressions usually cause the run to fail.) For example, if we use the expression `d10/dz6` then one in six evaluations attempt to divide by zero, and that is an error. The program checks for that case, and other common cases, before they happen. This does not cause the run to fail, instead, an error is a special case of a result. Errors are excluded from program statistics, but their statistics are collected and included in a separate report from `-statistics` or `+statistics`. The former output for that example is:

```
Number of evaluations      = 60
Number of results          = 50
Mean                       = 2.24 = 56/25
Standard deviation         = 2.35423
Minimum result             = 0
Maximum result             = 10

Number of errors           = 10
Proportion of errors       = 0.166667 = 1/6
```

Note that the mean is calculated over the 50 results, not the 60 evaluations.

Errors are usually slow, so you should either not use them, or use them sparingly; as a rule of thumb, no more than one percent of your results should be errors. If errors should not occur and you want an error to cause an immediate run failure, add the option `-finish`. If you do this in exact mode then no final output is produced, as it would not be meaningful. For the same effect in approximate mode also add the option `-fatal`.

Games with States – Introducing Markov Chains

A wide range of games can be represented by what is known as a *Markov chain*, and some results from these games can be determined exactly, even when the game might last for an indefinite length, and cannot be directly solved in exact mode by implementing it. This includes some cases previously suggested to not be practical in exact mode (but not all of them).

A Markov chain is a process with states, transitions between states, and rules such as that those probabilities not changing, which the remaining examples in this tutorial demonstrate. How the functions that are described work is beyond the scope of this tutorial, and it is even not necessary to know that these are called Markov chains unless you want to dig deeper.

The examples that follow use a kind of Markov chain called an *absorbing Markov chain*, and the program has some functions specifically for this type of chain. There are other kinds of Markov chains, and the program has functions for use in other cases. But absorbing Markov chains are typical in games, because they can be considered to eventually finish, which is usually what we want for a game, and in cases such as the following example with a winner.

Consider a game between two players. Player A, who goes first, has a probability of winning each point of *p*, player B has a probability of winning each point of *q*. The game is played in rounds until one player is in the lead by two points won. This is a simplification of games such as *lawn tennis*, where *p*+*q* = 1 and there is also a requirement to have won at least four points to win the game. That results in a bigger version of this problem, but here the smaller problem is more convenient as a demonstration, although we do extend it in another way. We assume that all points are independent – this is, at best, only an approximation in tennis, but might be suitable for a game such as a tennis simulation using dice.

This game can be considered to be in one of five *states*: level (where we start), player A is winning, player B is winning, player A has won, player B has won. Each point moves us from one state to another state, until we reach a winning state, where we remain. For example, in the level state there is a probability of *p* of moving to the player A is winning state and a probability *q* of moving to the player B is winning state. In tennis it is impossible to stay in any state, other than a winning state, but to make this example more interesting, we assume a probability *r* of staying in the same state, other than a winning state, with *p*+*q*+*r* = 1. In either case, the game could last for an indefinite number of points, although it will always finish.

We can summarise this game by a *matrix* (formally, a *transition matrix*) which is just a grid of numbers, where the entry in row *i*, column *j* of the matrix is the probability, when in state *i*, of the next point taking us to state *j*, including the case that *j* = *i*. For the states in the order listed above, numbered from the top left down (*i*) and across (*j*), the matrix for this game – the large parentheses are the standard matrix notation, which will also be useful below – is:

$$\begin{pmatrix} r & p & q & 0 & 0 \\ q & r & 0 & p & 0 \\ p & 0 & r & 0 & q \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Note that once in either of the last two *absorbing states* we never leave it, the game is over.

Now we assume that *p*, *q* and *r* are rational, given – with a common denominator *n* – by m_p / n, m_q / n and m_r / n, respectively. We are allowed, by matrix rules, to move the common factor 1/*n* outside the matrix parentheses, and the matrix is now:

$$\frac{1}{n}\begin{pmatrix} m_r & m_p & m_q & 0 & 0 \\ m_q & m_r & 0 & m_p & 0 \\ m_p & 0 & m_r & 0 & m_q \\ 0 & 0 & 0 & n & 0 \\ 0 & 0 & 0 & 0 & n \end{pmatrix}$$

Games with States – Probability of Winning

We need to represent the final matrix from the previous section, which apart from the 1/*n* is just integers, by a list of integers. We do that by ignoring the 1/*n* – why we can do that is explained below – and reading off the matrix row by row. To make that clearer, when entering the matrix we can use extra braces, as shown below. We can replace *n*, m_p, m_q and m_r by `c0` to `c3`, defined by options, which, since we must have $n = m_p + m_q + m_r$, can include either `-c0 c1+c2+c3` or `-c0 c123`. The matrix is thus represented by the constant list `u0`, using:

```
-u0 {{c3,c1,c2,0,0},{c2,c3,0,c1,0},{c1,0,c3,0,c2},{0,0,0,c0,0},{0,0,0,0,c0}}
```

We can ignore the factor 1/*n*, because the sum of each row of a matrix such as this is always 1, so in the functions that use such matrices, the program can work out what *n* must be from `u0` without the need for us to tell it. Here all row sums are the same, but that is not necessary.

Now we consider how we are going to use that list. The first thing to note is that we have states that the process will always end up in one of. This is what makes this an absorbing Markov chain. You can test whether any `u0` represents an absorbing Markov chain by whether `cmat_absorb(u0)` is true (it is).

The main function we need, applied to `u0`, is `mmat_absorb_states(u0)`, a list that we will denote as *list*. *list* represents another matrix whose entry in row *i*, column *j* is the probability of finishing in state *j* if we start in state *i*. But we know we start in the first state, which as usual we number as state 0, so we only need the first row of that matrix, which is `dmat_row(0,`*list*`)`, and we only need the last two elements in that list, the rest being zero, the list `tail2(dmat_row(0,`*list*`))`, which we here let `u1` be, using the option `-u1`.

All of this has no random factors in it, and all we need to do is to report the two probabilities whose relative values are in `u1`. There is no output option that just produces that. However, there are two functions – related to, but different to, the previously described `write_ratio` and `write_real` – that can be used. We could use one evaluation of a main expression including those functions. But instead, we use a new option `-eval`. This option – which we can use more than once, unlike most options – includes an integer expression that is evaluated once, before the main expression (but after all other options described here, so for clarity include it, or them, last). We now do not need a main expression, and we can simply omit it.

So how do we output the probabilities we want from `u1`? We have two functions that can do this, either `write_dist(u1)` or `write_rdist(u1)`, we here use both of them, separated by a space, using `-eval write_dist(u1);space;write_rdist(u1)`. We can, in other problems, use these functions on lists of any length, not just two.

For example, using `-c1 3 -c2 2 -c3 1`, the output – reported as if these were lists of rational numbers and real numbers, similar to the output from `-list_stats` shown in an earlier section – is:

```
{9/13,4/13} {0.692308,0.307692}
```

So, player A wins with probability 9/13, player B wins with probability 4/13; see the output directly above for the real versions of those numbers.

But suppose you wanted to check that. You could do so by playing the game. That cannot be done in exact mode, because the game can last indefinitely long, but here is a version of the game that can be run in approximate mode, based on that each round you could roll a d6, 1-3 is player A wins a point, 4-5 is player B wins a point, 6 is neither win a point. It is easiest to track a score difference `r0`, +1 is player A is leading, -1 is player B leading.

```
until(r1:=d6;r1<4?r0+=1:r1<6&(r0-=1),abs(r0)>1);r0>0
```

We can run this as usual, which we do for ten million evaluations. Here use the option `+statistics` rather than `-statistics`, and also add the option `-fraction 100`; the effects of both of which are explained after the output:

```
Number of results          = 10000000
Number of false results    = 3076558
Number of true results     = 6923442
Probability of false       = 0.307656 ~ 4/13
Probability of true        = 0.692344 ~ 9/13
False 95% conf interval    = [0.30737, 0.307942]
True 95% conf interval     = [0.692058, 0.69263]
```

The effect of `+statistics` when also using `-probability` is to fully report the true and false probabilities, not just the probability of being true. Without the option `-fraction` we would, just considering the true probability, just get the value 0.692344, which we probably would not recognise. The option `-fraction 100` causes an added report of the closest rational numbers with denominators up to 100. Here that gives what are actually the correct values, but that cannot be guaranteed – and will never work when the denominator is not simple. But it gives us a possible hypothesis as to what the exact value might be if it is simple.

So, in conclusion, the game has increased player A's margin from winning 60% of conclusive points to winning 69.2% of games.

Games with States – More Results

We can analyse absorbing Markov chains further. With `u0` as in the previous section, we consider the list `u1` created using the other main function for absorbing Markov chains, as `-u1 mmat_absorb_visits(u0)`. We can report that list as a matrix of integers by using the function `dmat_write(u1)`, producing the output:

```
{{150,90,60,0,0},{60,114,24,0,0},{90,54,114,0,0},{0,0,0,65,0},{0,0,0,0,65}}
```

This is an upscaled version of a matrix M whose entry m_{ij}, the j-th element of the i-th row, is the mean number of times that state j is visited if starting in state i. But we need to know the scaling factor, an integer k, and instead we use `dmat_write(u1,k)`. With the correct k – which is 65, how to derive that follows – the output, here split onto two lines, is:

```
{{30/13,18/13,12/13,0,0},{12/13,114/65,24/65,0,0},
{18/13,54/65,114/65,0,0},{0,0,0,1,0},{0,0,0,0,1}}
```

As defined here, and as implemented by `mmat_absorb_visits`, we only enter an absorbing state once, and that is the key to finding k. If, as here, we know that the last state is an absorbing state, we can use `last(u1)`. In general we can use `mmat_absorb_scale(u1)`.

We also know that we start in state 0, so we are interested in the m_{ij} with $i = 0$. We can extract that row as the list `u2`, set to `dmat_row(0,u1)`, still needing to be divided by k. The mean number of points that the game will last is given by `sum(u2)`, or `u2`, divided by k, which we can report by using either `write_ratio` or `write_real`, as previously described. For this example, `write_real{u2,mmat_absorb_scale(u1)}` produces the output `4.61538`.

In some cases we want more. Consider a game in which we roll a die, totalling the values of the rolls, until we roll a 3, which we do not count. We use state 0 to represent the start state and state $n > 0$ to represent having just rolled n. State 3 is thus an absorbing state. Our mean score (because we can always sum mean values, even when not independent) is the sum of n (except 3) times the mean number of times in the state n. Letting the game matrix be `u0`, as above, we also derive `u1` and `u2` from `u0` as above. We also let `u3` be `{0,1,2,0,4,5,6}`.

The matrix for this game, as noted used to set `u0`, is:

$$\frac{1}{6}\begin{pmatrix} 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 6 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix}$$

The mean score is `dot(u3,u2)` divided by `mmat_absorb_scale(u1)`, and equals 18.

Summary of New Features

The following lists include all of the new features in this tutorial. `£` is an unsigned number or a constant integer term in `[]`. `c$`, including in `-c$`, refers to any of `c0` to `c9`, and similarly `k$`, `r$`, `s$`, `u$` and `v$`. `c$..$` refers to lists such as `c012`, and similarly for other constants and variables. In the function list, `§` stands for any of `eq`, `ne`, `lt`, `le`, `gt` or `ge`. In the weighted term list, `§` stands for a list of constant integer terms in `{}` or a constant list term in `[]`.

The new features are:

- Options: `+all`, `-eval`, `-fatal`, `-finish`, `-fraction`, `-list_stats`, `+list_stats`, `-lists`, `+lists`, `-numbers`, `-pool`, `-progress`, `+statistics`, `+table`, `-time`, `+time`, `-wrap`, `-c$`, `-r$`, `-s$`, `-u$`, `-v$`, `-c$..$`, `-r$..$`, `-s$..$`, `-u$..$`, `-v$..$`.
- Constants: `c$`, `k$`, `s$`, `u$`, `c$..$`, `k$..$`, `s$..$`, `u$..$`.
- Variables: `r$..$`, `v$..$`.
- Dice terms: `d{...}`, `£d{...}`.
- Weighted terms: `permute£from§`, `select_by§`, `selection£by§`, `selection£from§`, `sorted£by§`.
- Functions: `abs` (integer and list), `boolean` (integer and list), `cmat_absorb`, `copy£`, `count_§` (list), `count_seq£`, `delta`, `distribute`, `distribute£`, `dmat_row`, `dmat_write`, `dot`, `duplicate£`, `find_§`, `find_max`, `find_min`, `find_mode_max`, `find_mode_min`, `get` (list), `hcf`, `lcm`, `list_result`, `lwrite`, `mid£`, `mmat_absorb_scale`, `mmat_absorb_states`, `mmat_absorb_visits`, `nline`, `number_combin`, `number_permut`, `occurs`, `overlap`, `pattern`, `replace_§`, `rfind_§`, `rfind_max`, `rfind_min`, `rfind_mode_max`, `rfind_mode_min`, `rmax$`, `rmin$`, `rsum$`, `rvloop$`, `rvmax$`, `rvmin$`, `rvsum$`, `select`, `select£`, `shuffle`, `sigma`, `sign` (integer and list), `space`, `swrite`, `unique_max`, `unique_min`, `unit£`, `until`, `write`, `write_dist`, `write_ratio`, `write_rdist`, `write_real`.
- Operators: ==, !=, <, <=, >, >= for lists.
- Using a list term in `[]` when an integer term is expected, with value its length. Using extra braces `{}` when entering a matrix as a list.

Final Note