A Collection of Dice Problems

Solutions and Illustrations Using the Program *Monaco*

This document describes how the program *Monaco*[1] can be used to address the problems in the document *A Collection of Dice Problems* by Matthew M. Conroy, available at `https://www.madandmoonly.com/doctormatt/mathematics/dice1.pdf`. This document considers the (as of writing, most recent) version of that document dated 22nd September 2023.

Most of this document assumes significant familiarity with Monaco. Without that, this document can only be used as an indication as to what sorts of dice problems Monaco can solve – see the later section that summarises which problems Monaco can and cannot address (over half completely, most to at least some degree). Monaco can also handle non-dice problems, in particular problems using cards, but no such problems are included here**.**

The purpose of this document is to investigate the applicability of Monaco to these programs. In some problems Monaco is an ideal tool for the task, in others it is less so, and in a few it cannot be used at all. The best that can be done – or at least the best this author can do – is presented, whether a full solution, a partial solution, an illustration of the problem, or simply no solution. In many cases – even cases in which Monaco is entirely satisfactory – there may be or are better approaches than using Monaco, but these have not been considered.

References in this document to "the given solution" or "the given solutions" refer to the solutions to the problems that make up most of the referenced problem document. However, the solutions in this document – although not always the parameterisation of those solutions, in order to produce comparable results – were created independently from those solutions.

Monaco's single biggest limitation, commented on when relevant, is that it addresses single problems, it cannot produce general solutions where one or more parameters in the problem are retained in the solution – although sometimes, especially for problems with only one such parameter, its solutions can suggest hypotheses as to what such a general solution might be. Most problems are handled in a way that makes changing the problem parameters – especially the numbers and sizes of dice – easy to do.

In order to save space, actual Monaco output – most often from the option `-statistics`, but also from output functions in an expression – is usually not reproduced here, but is instead extracted from. The use of the option `-statistics` as the source of quoted results is not commented on in the solutions here. Similarly, the option `-probability`, often required to produce probability output in an appropriate form, is only mentioned when it is also needed to convert a numerical result from an expression to a logical value.

Unless noted otherwise, Monaco's exact mode, specified by the option `-exact`, is used to produce these results. (The alternative, approximate mode, requires no option.) In some cases exact answers are produced without a main expression by using the option `-eval`; in this case the option `-exact` is not required.

[1] See `http://www.mnemosyne.uk/monaco`. The results in this document were created using version 2.29 of the program. However, the solutions presented here were created before (although they have been re-run since) the addition of Markov reward processes and Markov decision processes to the program, the use of which could simplify some of the later solutions.

To distinguish exact results from approximate results, as well as pointing the latter out, when expressed in decimal form only the latter are qualified using “about”. Exact results are accurate to the precision quoted when presented in decimal form, or are actually exact when presented as a rational number. The precision of an exact value presented as a decimal value could be increased by using the option `-precision`, but that has not been done here.

A summary section at the end of this document shows the parameters of the runs used – other than those runs that just change parameter values, which can be created by simple modification of the given parameters, as is noted in each case. These parameters, in particular the expressions that they contain, are not fully explained in this document – more so in some cases than in others – as to do so would significantly increase its length.

Standard Dice

1. This problem is too simple to need Monaco, but as a warm-up to the following problems this problem can be represented by a Markov chain with two states – a start state (0) and a finish state (1) with transition matrix:

   $$\frac{1}{6}\begin{pmatrix}5 & 1\\ 0 & 6\end{pmatrix}$$

   This transition matrix – an example of a stochastic matrix – can be represented by either of the lists `-u0 {{5,1},{0,6}}` or `-u0 {{5,1},{0,1}}`. The former approach is used here and in most later problems.

   We want to know the mean time – i.e. the mean number of transitions – until entering the finish state. Using that we know that the start state is the first state, and that the finish state is the last state, we can determine this value from the matrix u1 that is defined by `-u1 mmat_absorb_visits(u0)`. The mean time is then the ratio c0 divided by c1, where those two constants are set by `-c0 sum(dmat_row(0,u1))` and by `-c1 last(u1)`. We can output the mean time as a rational number using `write_ratio(c01)`, or as a real number using `rwrite_ratio(c01)`, where c01 is {c0,c1}. (In this problem, and those immediately following, outputting both is unnecessary, as the solution is an integer, but this will not be the case in some later problems.) Putting all of that together using the option `-eval` – no main expression is needed – the result is 6, as in the given solution.

2. The Markov chain from the previous problem can now be expanded to one with three states: the start state and also the state after rolling a 1 to 5 (state 0), the state after rolling a single 6 (state 1), and the finish state after rolling two 6s (state 2). The transition matrix, is now given by `-u0 {{5,1,0},{5,0,1},{0,0,6}}`. The rest of the solution is unchanged from the previous problem, and the result is 42, as the given solution.

3. The Markov chain from the previous problem can now be modified so that the finish state is after rolling a 6 then a 5. It is now possible to stay in state 1 after rolling a 6, and the transition matrix is thus given by `-u0 {{5,1,0},{4,1,1},{0,0,1}}`. The rest of the solution is unchanged from the previous two problems and the result is 36, as the given solution.

4. To solve this problem we can use a Markov chain with eight states: start, after 1 to 6 but not finished, and finish. Otherwise the solution is as the last three problems. The transition matrix for the first part of the problem is:

$$\frac{1}{6}\begin{pmatrix} 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 & 2 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 & 2 \\ 0 & 1 & 1 & 0 & 1 & 0 & 1 & 2 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 & 2 \\ 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 6 \end{pmatrix}$$

and for the second part of the problem is:

$$\frac{1}{6}\begin{pmatrix} 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 2 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 3 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 & 3 \\ 0 & 1 & 1 & 0 & 0 & 0 & 1 & 3 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 & 3 \\ 0 & 1 & 1 & 1 & 1 & 0 & 0 & 2 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 6 \end{pmatrix}$$

These are used to define `u0`. The results are 239/51 and 377/115, respectively, as in the given solution.

Note that it is possible to simplify this; in both cases the states 1 and 6 are equivalent, as are the states 2 and 5, and the states 3 and 4. Using those states in that order between the start and finish state, the transition matrices simplify to:

$$\frac{1}{6}\begin{pmatrix} 0 & 2 & 2 & 2 & 0 \\ 0 & 2 & 1 & 2 & 1 \\ 0 & 1 & 2 & 1 & 2 \\ 0 & 2 & 1 & 1 & 2 \\ 0 & 0 & 0 & 0 & 6 \end{pmatrix}$$

and:

$$\frac{1}{6}\begin{pmatrix} 0 & 2 & 2 & 2 & 0 \\ 0 & 1 & 1 & 2 & 2 \\ 0 & 1 & 1 & 1 & 3 \\ 0 & 2 & 1 & 0 & 3 \\ 0 & 0 & 0 & 0 & 6 \end{pmatrix}$$

with, necessarily, the same results.

5. In common with other similar problems that follow, Monaco cannot solve problems for a general *n*. Instead we can solve the problem for *n* = 6, 7, … in turn. We use `-c0 n` to parameterise the problem, and the main expression `count_diff(sorted[c0]d6)==6`. Results for *n* = 6 to *n* = 12 are 5/324, 35/648, 665/5832, 245/1296, 38045/139968, 99715/279936, 1654565/3779136.

The given solution is equivalent to a summation, rather than a simple closed form; examples given include the above results and continue to $n = 20$, plus a decimal value for $n = 36$. We can replicate these results, although the case $n = 36$ requires 128 bit numbers (a separate compilation of the program).

6. We return to the Markov chains of problems 1 to 4, this time with seven states: no sequence yet started, and most recent sequences 1, 12, 123, 1234, 12345 and the finish state 123456. The transition matrix is:

$$\frac{1}{6}\begin{pmatrix} 5 & 1 & 0 & 0 & 0 & 0 & 0 \\ 4 & 1 & 1 & 0 & 0 & 0 & 0 \\ 4 & 1 & 0 & 1 & 0 & 0 & 0 \\ 4 & 1 & 0 & 0 & 1 & 0 & 0 \\ 4 & 1 & 0 & 0 & 0 & 1 & 0 \\ 4 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 6 \end{pmatrix}$$

Repeating the calculation in problems 1 to 4, to find the mean number of transitions required to achieve the objective, the result is 46656. However, that is not the problem here, which is to determine the probability whether, after n transitions, the objective has been reached, i.e. the Markov chain is in the final, absorbing, state. If the transition matrix is P, this probability is the top right element of P^n, the probability of being in the last state when starting in the first state. We can calculate this for small values of n, but are limited by only being able to perform this calculation with rational numbers, using limited precision integers, rather than using real numbers. This does however mean that results are exact.

Producing results for increasing n, from $n = 1$, one per evaluation, using the option `-noreset` can be achieved by using the expression:

```
v0:=mmat_mult(u0,v0);write(number+1);space(2);
rwrite([dmat_get(0,6,v0)]/[first(dmat_rsum(v0))])
```

To avoid an unwanted warning message that is due to the option `-noreset` or to the function number, the option `-nowarnings` can also be used.

With the usual default of 64 bit integers, the maximum possible value of n, and hence the number of error-free evaluations to use, is 24, where the probability is 0.000407194. With 128 bit integers this maximum increases to $n = 49$, where the probability is 0.000942732; with the largest possible integer size, 1024 bits, this maximum increases to $n = 395$, where the probability is 0.00832518. The maximum given solution that can also be implemented is $n = 200$, and the two results agree to the precision quoted by the given solution.

As an alternative, without such a limit for n, we can directly implement the behaviour described in the problem using that Markov chain, but in approximate mode. We parameterise the problem with `-c2` n and then, relying on that the initial value of `r0` is 0, can use the expression `do[c2](r0:=distribute(dmat_row(r0,u0)))==6`. This updates the state `r0` by c2 transitions, and then checks if it has reached the final state. For a million evaluations for $n = 5000$, that probability is about 0.101, with a small uncertainty in the final digit. We can speed thus up slightly (by about 40% on my computer) by dispensing with the Markov chain and implementing the problem directly,

retaining only c2, and stopping as soon as the required event occurs, by using the expression `until(r2:=d6;r1:=r2==r1+1?r1+1:r2==1,r1==6|incr0==c2);r1==6`. The estimated probability is statistically equivalent but different, but again is about 0.101, with a similar uncertainty.

These probabilities are consistent with the given solution in this case of 0.1015397384. More results would be needed to have confidence in the third significant figure as the simulation gives a result of about 0.101, as against an exact result that is closer to 0.102. Ten million results would be better; however, the two sets of one million results took nearly three minutes and about two minutes, respectively, on my computer. The time required increases approximately linearly with c2 (slightly slower in the latter case if the probability is larger) as well as with the number of evaluations, and thus reproducing the given solution for the largest values of *n* it includes is not practical using this approach.

7. We modify the Markov chain approach to the previous problem so that the states 1 to 5 are having 1 to 5 different rolls in the latest 1 to 5 rolls. The transition matrix is:

$$\frac{1}{6}\begin{pmatrix} 0 & 6 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 5 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 4 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 3 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 2 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 6 \end{pmatrix}$$

This has a much lower mean number of transactions, 416/5. The same limits on *n* apply as in the previous problem, but the corresponding probabilities are higher. For example with *n* = 20 the probability is 0.177598, which matches the given solution. The highest value of a given solution that we can match is *n* = 364, with probability 0.990043, evidently chosen to represent the 99th percentile result – the probability for *n* = 363 is 0.989914. As this is high enough for greatest interest, the approximate approach to the previous problem is not considered here.

8. This is another example where the best we can do is to produce results for selected values of *m* and *n* rather than producing a general solution. Using the options `-c0` *n* and `-c1` *m* to parameterise the problem, this result is the logical value of the expression `overlap(sorted[c0]d6,sorted[c1]d6)`, where we use the option `-probability` to make those results logical. Some results that can be produced are:

m = 1, *n* = 1: `1/6 ~ 0.166667`

m = 2, *n* = 1: `11/36 ~ 0.30555`

m = 2, *n* = 2: `37/72 ~ 0.513889`

m = 6, *n* = 6: `120194317/120932352 ~ 0.993897`

These all agree with the given solutions. (The other given solutions can also be produced.)

9. We return to Markov chains, with a similar set of states to problem 7, but with the transition matrix, the only difference from that previous problem, now being:

$$\frac{1}{6}\begin{pmatrix} 0 & 6 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 5 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 4 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 4 & 2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 5 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 6 \end{pmatrix}$$

This form of transition matrix , where we only ever transition to the same or the next state, means that we do not need to a general approach to analysing Markov chains, but if we do handle this in that way, as previously, then we get the solution 147/10, which agrees with the given solution.

To implement an *n*-sided die we either have to create a new transition matrix for each *n* ourselves, or write an expression (not the main expression) to create a suitable transition matrix given *n*. The latter is better, and can be used for larger values of *n*, so it is used here.

Using `-c0 n` and `-c1 c0+1`, and changing the previous c0 and c1 to c2 and c3, a suitable definition for the matrix u0 is `-u0 ccreate_seq[c1*c1](f0(l/c1,l%c1))`, where f0 is defined by `-f0 p1==p0?p1:p1==p0+1?c1-p1:0`.

With *n* = 6 this, necessarily, gives the same result as above. The given solution does not provide results for other values of *n*, other than by evaluating the given sum of *n* terms. However, it gives an asymptotic result for large *n*, as *n* log *n*. For *n* = 100 the result of the above, using 256 bit integers, is 518.738 – which can also be confirmed to be the scaled sum of the harmonic series in the given solution – but 100 log 100 is 460.517, so there is still some way to go to be close to convergence. Thus while we can handle specific values of *n*, we cannot usefully handle the asymptotic case.

10. This problem can be solved as the previous problem for the case *n* = 6, although the Markov chain required is significantly larger. It needs a state for how many faces have been seen two and one times so far, with the sum of those numbers being six or less. These can be ordered – from start state to finish state – as (0,0), (0,1), … (0,6), (1,0), (1,1), … (1,5), … , (5,0), (1,5), (6,0), a total of 7+6+…+2+1 = 28 states.

At this point we again do not want to write out that transition matrix by hand, and then create a list with $28^2 = 784$ elements. Again, we can write an expression to calculate that transition matrix for us. We can generalise this to using c0 for the number of faces, here we want the option `-c0 6`, so that we can – although this is not reported here – verify the expression for a small number of faces. The number of states is then the constant `-c1 (c0+1)*(c0+2)/2`.

We construct this matrix as the list v0, setting its initial size using `-s0 c1*c1`. We then rely on that v0 is initialised to empty. We then can loop over the state (r0,r1), which can be shown to have state number `r0*(2*c0+3-r0)/2+r1`. It is convenient to define the function `-f0 p0*(2*c0+3-p0)/2+p1`, so that the state (r0,r1) has state number `f0(r0,r1)`.

Next we consider the three states that most states (r0,r1) can transition to. There are r0 ways to stay in the state (r0,r1), r1 ways to transition to the state (r0+1,r1-1) and c0-r0-r1 ways to transition to the state (r0,r1+1). There are some exceptional cases where those transitions cannot occur, but they all evaluate as zero ways to happen. However as this zero could overwrite another element – because Monaco uses cyclic indexing – we must prevent that case from occurring.

We now need the element number in v0 of the transition from state *m* to state *n*, which is c1**m*+*n*. We thus define the function `-f1 c1*f0(p0,p1)+f0(p2,p3)`, so that the transition from state (r0,r1) to state (*p*,*q*) is added by setting element f1(r0,r1,*p*,*q*) of v0. For each (r0,r1) we thus use the three terms vset0(f1(r0,r1,r0,r1),r0), vset0(f1(r0,r1,r0+1,r1-1),r1) and vset0(f1(r0,r1,r0,r1+1),c0-r0-r1), but each only when not setting a zero value. We can achieve this by defining the function `-f2 p4&vset0(f1(p0,p1,p2,p3),p4)`, so that those terms can be replaced by f2(r0,r1,r0,r1,r0), f2(r0,r1,r0+1,r1-1,r1) and f2(r0,r1,r0,r1+1,c0-r0-r1).

We can loop the ; separated three latter terms above, written here as *term*, through all values of (r0,r1) by using rloop0(c0+1,rloop1(c0+1-r0,*term*)). We put that all together, including with the Markov chain analysis, as shown at the end of this document.

The result is 390968681/16200000, or 24.1339, which matches the given solution.

The given solution continues by considering the probability of reaching the finish state after various numbers of rolls. This can be done as in problem 6, with the same constraints on numbers of evaluations (rolls) as that problem. However, in this case 256 bits suffices to reproduce all of the given solutions.

11. We can solve this problem by starting with the Markov chain in problem 9, and using the square of its transition matrix, which can be calculated for us rather than be constructed by hand by using `-u1 mmat_mult(u0,u0)`. Other than now replacing u0 by u1, and u1 by u2, the solution proceeds as problem 9, and is 70219/9240 or 7.59946, which agrees with the given solution.

    Copying the previous approach to add the probabilities for each number of rolls only gives the probabilities of reaching the finish state in less than or equal to the given number of transitions, and only as a real probability. This solution could be modified to produce the additional results, but that has not been done here. The given solutions up to 24 rolls require 128 bit arithmetic, because with two dice the number of rolls that can be implemented is halved. The results here match the given solution.

    The additional question posed with the given solutions can be answered using the cube or higher power of the original transition matrix, which can again be calculated for us.

12. This is another problem where we can only give results for selected values of *n*. With `-c0` *n*, the number required is count_diff(sorted[c0]d6) – which is fast enough to avoid introducing some even faster possible alternatives that could be used here. Some examples are *n* = 1, mean = 1; *n* = 2, mean = 11/6; *n* = 6, mean = 3.99061; *n* = 24, mean = 5.92453; n = 48, mean = 5.99905. The last of those requires 128 bit integers. All agree with the given solution, which – showing the greater value of analysis – has the simple result that the mean is $6 - 6(⅚)^n$.

13. As usual, we can only produce solutions for specific values of *n*. Using `-c0 n`, a suitable expression is `max(sorted[c0]d6)`, or equivalently `last(sorted[c0]d6)`, and the distribution can be reported using the option `-histogram`. For example, with *n* = 3 the relative probabilities of results of 1 to 6 are 1, 7, 19, 37, 61, 91. These match the given solution of being $k^n - (k-1)^n$ for $n = 3$ and $k = 1, \ldots, 6$.

14. This is a problem that requires finding an ideal algorithm, which is usually beyond the capability of Monaco. However, in this case the decision process is sufficiently straightforward that the problem can be solved, exactly – although the solution only uses `-eval` options, no main expression, and thus does not actually use exact mode.

    We start by defining `c0` as the size of the dice used, here 6, and `c1` as the maximum number of rolls. Note that it does not matter how many rolls we started with, only how many are left, and so we can solve for all numbers of rolls of interest in a single run. Here, in order to cover the given solutions, we let `c1` be 50. This requires us to use 128 bit integers.

    We require two lists, `v0` and `v1`, each of which contains an entry for each number of rolls from 1 to `c1`, and each possible roll, from 0 to `c0-1` (we correct for that dice are actually 1 to `c1` later). We organise these so that the `c0` entries for each number of rolls are consecutive. Entries in `v0` are logical, 1 meaning roll again, 0 meaning stop. Entries in `v1` are rational, the mean value after rolling that value (assuming ideal play after that roll). Thus `s0` is `c0*c1` and `s1` is `2*s0`. However, for convenience later we define the function `f0` as `c0*p0+p1`, the entry number for `p0` die rolls and (0-based) die value `p1`; we also use this function with a default second argument zero for the position of all information for die roll `p0`, and set `s0` to `f0(c1)`. The all zero elements of `v0` are what we require for the final setting when we have used all our rolls, we must stop.

    Rather than setting `s1` we set start by setting `v1` to `copy[s0](rnull)`, invalid rational numbers (which will never be used, but this provides error checking in case they accidentally are used). However, we also need to set the final `s0` values in `v1` to the mean value after a final roll, i.e. to that roll. We do so with the first of three `-eval` options, `rloop1(c0,f1(c1-1,r1,ratio(r1+1)))`, using a second function `f1` defined as `[ratio_vset1(f0(p0,p1),q2)]`, which is used again below. This sets the mean value after die roll `p0` is `p1` (both 0-based) to the rational number `q2`.

    We use one other function, `g0`, defined as `ratio_lmean(mid[2*c0](2*f0(p0),v1))`; this is the rational mean value for die roll `p0`, averaging over the possible `c0` rolls.

    Our second -eval expression, split onto two lines for convenience is:

    ```
    rloop0(c1-1,r0:=c1-2-r0;v2:=g0(r0+1);rloop1(c0,v3:=ratio(r1+1);
    ratio_gt(v2,v3)?vset0(f0(r0,r1),1);f1(r0,r1,v2):f1(r0,r1,v3)))
    ```

    Breaking that down, it consists of a single loop of loop `r0` over each die roll, working backwards of the form `rloop0(c1-1,...)`, which loops `r0` from 0 to `c1-2`, inclusive, but then `r0:=c1-2-r0` at the start of the ... converts that into looping `r0` from `c1-2` to 0, inclusive. Note that the mean value after the last die roll, `c1-1`, was initialised by the first `-eval` expression.

    We then – after a detail to be described – loop `r1` over the `c0` possible 0-based die rolls. In each case we compare `v2`, the mean value if we continue, with `v3`, the mean value if

we stop. But as v2 does not depend on the die roll, we put it outside the loop. Its value is the mean value after the next die roll – i.e. die roll r0+1 – averaged over all die rolls, which we defined *g0* to provide. v3, inside the r1 loop, is simply the die roll, where we use r1+1 to restore the 1-based dice. We then compare v2 and v3 using ratio_gt(v2,v3), but also setting the relevant element of v0, using f1, to indicate that we continue when v2 is greater. (We do not need to set v0 when we stop, it was initialised, by default, to all zeros.)

Finally, our third `-eval` expression produces output, in two parts, with a blank line separating them. The first part is, split on two lines for convenience:

```
rloop0(c1,v2:=g0(r0);write(c1-r0);space(2);
rwrite_ratio(v2);space(2);write_ratio(v2);nline)
```

We loop through v1, using the function *g0*, reporting the mean value after that die roll. We label that line by how many die rolls we had left before making that roll. Thus, for example, one line of that output is:

```
3   4.66667   14/3
```

This says that with three die rolls our mean score is 14/3, which – like the other figures it produces – agrees with the given solution.

The second part, again split on two lines for convenience, is:

```
rloop0(c1-1-r0,write(r0+1);space(2);
lwrite(mid[c0](f0(r0),v0)?0:sequence+1);nline)
```

This time we report the number of remaining dice rolls, so the corresponding line to the above is:

```
2   {0,0,0,0,5,6}
```

This says that with two remaining dice rolls (i.e. after the first roll when we started with three dice rolls) we should stop if the roll was 5 or 6, continue otherwise. The results agree with the given solution: always stop with a 6, stop with a 5 if 4 or fewer die rolls remaining, stop with a 4 with 1 die roll remaining, only stop with 1 to 3 if there are no remaining die rolls and you must stop.

15. As previous cases, we need to run for individual values of *n* dice, which number we set using `-c0` *n*. An expression that checks whether those dice include a 6 is any_eq(sorted[c0]d6,6). Trial and error shows that the probability with *n* = 16 is 0.945912 and with *n* = 17 is 0.954927, so the solution is 17 dice, which agrees with the given solution. The given solution goes on to discuss other probabilities, with could be answered in the same way, although with more than 24 dice (and no more than 49 dice) the 128 bit version of the program would be needed.

    This requires separate runs for each value of *n*. We can avoid that by instead using the term v0:=order_dz_dist6(number+1,number), which evaluates v0 as the relative probabilities that the largest of 1, 2, ... dice are the values 1 to 6. To avoid an unwanted warning message that is due to using the function number, the option `-nowarnings` can also be used.

The required probabilities can be output using `rwrite_ratio{last(v0),v0}`. The maximum number of evaluations that is useful is 24 for 64 bit integers, or 49 for 128 bit integers. (It is also convenient to output `number+1` to make reading the results clearer.) Note that these are exact results (the probabilities can also be output as rational numbers if preferred) despite using the program in approximate mode.

16. The first part of this problem can be tackled in the same way as the first approach to the previous problem (the second approach is not applicable) by replacing the expression `any_eq(sorted[c0]d6,6)` by the expression `overlap(sorted[c0]d6,{1,2})==2`, and again using trial and error. The option `-probability` is needed to convert the overlap value to a logical value. The probability with $n = 20$ is 0.948133, and with $n = 21$ is 0.956727, so the solution is 21. We proceed similarly for the second part of this problem replacing `{1,2}` by `{1,2,3}` and `==2` by `==3`. We can just manage to use the standard 64 bit version of the program as the probability with $n = 22$ is 0.946059, and with $n = 23$ is 0.954982, so the solution is 23. For the third part of this problem we could modify the expression similarly, but a more compact solution is `count_diff(sorted[c0]d6)==6`. We now do need to use the 128 bit version of the program, and the probability with $n = 26$ is 0.947983, and with $n = 27$ is 0.956586, so the solution is 27. These three solutions (21, 23 and 27) all agree with the given solution.

17. This is another problem like the previous problem where we run for different numbers of dice – here we use m dice, because n is otherwise used in the problem – and by trial and error find the maximum. We assume here, without proof, that the probability as a function of m is convex and this therefore must work.

    With `-c0` m, a main expression to find the probability for the first part of the problem, looking for a single 6, is `count_eq(sorted[c0]d6,6)==1`. We find that the probability with $m = 4$ is 0.385802, with $m = 5$ is 0.401878, with $m = 6$ is 0.401878, and with $m = 7$ is 0.390714. The best number of dice is thus 5 or 6. (We know that probabilities are multiples of 6^{-m} and thus those two probabilities must be exactly equal, or we could output rational values to show that.)

    With a target of two 6s, we just change `==1` to `==2`. We now find that the probability with $m = 10$ is 0.29071, with $m = 11$ is 0.296094, with $m = 12$ is 0.296094, and with $m = 13$ is 0.291607. The best number of dice is thus 11 or 12.

    As usual, we cannot solve for an indefinite number n of 6s wanted, we can only try example values. We here use `-c1` n and replace `==1` in the original expression by `==c1`. We here try only $n = 6$, and we need to use the 128 bit version of the program for the numbers of dice needed. However, we can also note that the given solution opts out of providing a proof for a general n. For $n = 6$, the probability with $m = 34$ is 0.174867, with $m = 35$ is 0.175872, with $m = 36$ is 0.175872, and with $m = 37$ is 0.174927. The best number of dice is 35 or 36, as the given claimed solution that this should be $6n-1$ or $6n$.

18. The underlying problem here is another Markov chain that we only need to transition a known number of times, set by `-c0 100`. We are looking for a run of sizes, set by `-c1 10`. We start with a state relative probability vector initialised by `-v1 unit[c1]`. For a transition matrix `v0`, with length initialised by `-s0 s1*s1`, defined so that all rows sum to 6, we can update v1 once using `v1:=mat_mult(v1,v0)`, and update it completely using `do[c0](v1:=mat_mult(v1,v0))`, used as a `-eval` expression. Final output can be

another `-eval` option, using an expression that can be composed from `r0:=last(v1);r1:=pow(6,c0)`, `write_ratio(r01)` and `rwrite(r0/r1)`, the first of which extracts the relative probability of the final absorbing state of the Markov chain and the required normalising factor, the other two of which can be separated with `nline`.

There remains only to set the Markov matrix `v0`. It is sufficient to set non-zero elements because v0 is initialised to all zero elements, and these non-zero elements can be set by using `cloop[c1](dmat_vset0(l,0,5);dmat_vset0(l,l+1,1))` for transitions from a transient state and `dmat_vset0(c1,c1,6)` for the absorbing state.

The final output contains the ratio of two very large integers, matching the given solution, and the ratio 1.2569e-06 that also matches the given solution once divided out.

19. This is another problem that can be solved using a Markov chain, although the number of states increases rapidly with the number of repetitions required. This is thus not the best approach to the problem. However, to compare solutions, the first part of the problem is solved here using both a Markov chain and an alternative approach.

    For the first part of the problem, we can use a Markov chain with 8 states. States 0 to 6 are the number of faces that have appeared once so far; the final state is the final absorbing state for one face appearing twice. The transition matrix is:

$$\frac{1}{6}\begin{pmatrix} 0 & 6 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 5 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 4 & 0 & 0 & 0 & 2 \\ 0 & 0 & 0 & 0 & 3 & 0 & 0 & 3 \\ 0 & 0 & 0 & 0 & 0 & 2 & 0 & 4 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 5 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 6 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 6 \end{pmatrix}$$

    Assuming that this is assigned to `u0`, the required result can be determined as for earlier problems to give the result 1223/324 or 3.77469, which agrees with the given solution.

    However, in this case there are a maximum of seven die rolls, and the last of those is unnecessary as it always succeeds. We could thus simply use the expression `find_eq(occurs(6d6),1)+1`, the 1 and +1 because of the zero-based indexing used, and the mean is 1223/324 or 3.77469, as before.

    However, this solution also does not scale well, as it rolls unnecessary dice. Instead we can roll only dice as needed. We can use the randomness pool in the expression `s0:=6;xuntil6(r0:=pool_dz(6);incr1,ince0==2,incr1)` with the initial pool set using the option `-pool 6 6`. (The second `incr1` in the expression could be replaced by 7, but it is convenient to use it to minimise the changes needed for the second part of the problem.) The result is again 1223/324 or 3.77469.

    We can modify this to handle needing a number to appear three times by replacing `xuntil6` with `xuntil12` and `==2` with `==3`, with the pool option becoming `-pool 12 6`. However, although this larger problem is easier to write an expression for than using a Markov chain, this solution requires 69056916 evaluations of the expression – although that is better than the 2176782336 results from an unpooled 12d6, and is a practical run on my computer. The result is 4084571/559872 or 7.29554, which matches the given solution.

The given solution additionally asks about needing four or five matching rolls, without providing solutions, but these cases are not practical using this approach. The Markov chain approach, writing code to create the transition matrix, might be possible, but the most prctical solution would be to return to the approach that for four matching rolls becomes `find_eq(occurs(18d6),3)+1` and use approximate mode. For ten million results this has a mean of about 11.21. However, as noted, we have no given solution to compare this with. Five matching rolls would proceed similarly.

20 This problem can be solved using the simple expression `sorted(10d6)`. This needs the option `-probability` to convert its result to a logical value, and the required probability is 1001/20155392 or 4.96641×10^{-5}, matching the given solution. The other results given in the solutions can also be produced in this way, although above ten dice they can become slow – ten dice takes about 6.5 seconds on my computer, and each extra die increases that time by about a factor of 6, so to confirm the comment about 13 dice in the given solutions would take over 20 minutes on my computer – which is possible but has not been done here.

That approach, while easy to write, rolls many unnecessary dice. Rolling dice one at a time we can stop as soon as a die is less than the previous die. Rolling a variable number of dice can use the randomness pool. Using `c0` for the number of dice, set using `-c0 10` for the first solution, we initialise the pool using `-pool c0 6`, and then the expression `until(r0:=r1;r1:=pool_d(6),r1<r0|incr2==c0);r1>=r0`. The resulting probability is unchanged, but takes only about 15 milliseconds on my computer. Even 13 dice takes only 50 milliseconds on my computer, and gives a probability of 119/181398528 – which as noted in the given solution is less than one in a million, being 6.56014×10^{-7}.

21. This is another problem that can be straightforwardly solved, by using the expression `list_eq(sorted2d6,sorted2d6)`, with the other problems referred to using the expressions `list_eq(sorted3d6,sorted3d6)` and `list_eq(sorted6d6,sorted6d6)`. The probabilities of the three cases are 11/216 or 0.0509259, 83/3888 or 0.0213477, and 737353/181398528 or 0.00406482. These all agree with the given solutions.

22. This problem is also straightforwardly solved by comparing the probabilities from `count_eq(sorted6d6,6)>=1`, `count_eq(sorted12d6,6)>=2` and `count_eq(sorted18d6,6)>=3`. These are 0.665102, 0.618667 and 0.597346, so rolling six dice is best. These results agree with the given solution.

    It is possible to speed these results up significantly, because we do not care about the distribution of the results 1 to 5. But even the slowest expression above is effectively instantaneous (at least on my computer) and is clearer than any alternative, and thus we do not need a faster solution.

23. This is another Markov chain problem, but with a transition matrix with order n+1 for states with n, …, 0 remaining dice. For example with n = 4 the transition matrix is:

$$\frac{1}{1296}\begin{pmatrix} 625 & 500 & 150 & 20 & 1 \\ 0 & 750 & 450 & 90 & 6 \\ 0 & 0 & 900 & 360 & 36 \\ 0 & 0 & 0 & 1080 & 216 \\ 0 & 0 & 0 & 0 & 1296 \end{pmatrix}$$

However, determining even one such transition matrix by hand is tedious, especially as *n* increases. We thus want the program to create the transition matrix u0 for a given number of dice c0, which we set using -c0 *n* as usual. This can be achieved by defining u0 by -u0 ccreate_mat[c0+1](l0<=l1?f0(c0-l0,c0-l1):0), where, noting that we do not need to normalise each row of the transition matrix by the same factor, f0 can be defined by -f0 pow(5,p1)*number_combin(p0,p1). With *n* = 4 we have u0 = {{625,500,150,20,1},{0,125,75,15,1},{0,0,25,10,1},{0,0,0,5,1},{0,0,0,0,1}}.

Given u0, we can then find the mean as in previous Markov chain problems. As usual we can only do so for specific values of *n*. For the above example with *n* = 4, the mean is 728256/61061 or 11.9267, which agrees with that given solution. With 64 bit integers, the maximum value of *n* for which the mean can be calculated exactly is *n* = 7, giving the result 3736509841479198/253815850752893 or 14.7213, which agrees with the given solution. It would be possible to use larger integers to produce results for larger *n*, but the rational solution is not very useful. Instead, still only using 64 bit integers, we can get the required mean result as a real value as mmat_absorb_time(u0,unit). This could introduce rounding errors using real numbers, but this is not considered here – it is not expected to be significant. For *n* = 7 the mean is again 14.7213; for *n* = 10 the mean is 16.5648 and for *n* = 20 the mean is 20.2329, which both agree with the given solution. For *n* = 30, *n* = 40, and n = 50 the mean cannot be determined even by this method with 64 bit integers, but can be determined by using 128 bit integers, with means 22.4118, 23.967 and 25.1773, which agree with the given solutions.

It is not possible to confirm the indicated limiting behaviour, except to note that these results are consistent with it. It is possible to go beyond the case *n* = 50, in due course using yet larger integers. With 1024 bit arithmetic, a mean of 36.4659 can be determined for *n* = 395, the largest value possible; it is unknown what, if any, rounding errors have occurred by this point, but again these are not expected to be significant.

However, we can take the ratio of the mean divided by log *n* to get, for the above means, *n* = 4: 10.62, *n* = 7: 7.565, *n* = 10: 7.194, *n* = 20: 6.754, *n* = 30: 6.589, *n* = 40: 6.497, *n* = 50: 6.436, *n* = 395: 6.099. The limiting value cannot be even plausibly guessed from these values. However, as these values appear to be reducing with increasing *n*, that the behaviour of the mean is O(log *n*) is highly believable – although whether that bound could be tightened is unknown.

24. Defining c0 as *k*, and noting that we do not care about the specific values 1 to 6 – or equivalently 0 to 5 – the required probability can be determined by using the expression same(pattern_dist[6*c0]rand6). For *k* = 1 to *k* = 4, the maximum possible using 64 bit integers, the probabilities are 0.0154321, 0.00343829, 0.00135117 and 0.000685186. We then borrow from the given solution – we have no other reason to choose this – to multiply these by $k^{5/2}$. That is only an integer for square values of *k*, and the easiest approach to scaling probabiliies, using the option +pscale, can only scale by an integer. Alternatives are possible, for example using the option -output, but here we just consider the cases *k* = 1, *k* = 4, *k* = 9 and *k* = 16 (the latter two requiring 256 bit and 1024 bit integers) for which the scaling value is c0*c0*sqrt(c0) and the scaled values are 0.0154321, 0.0219259, 0.0234519 and 0.0240123. The given solution says that this tends to $\frac{\sqrt{6}}{(2\pi)^{5/2}}$, or 0.0247529, as $k \to \infty$, and those values might be tending towards that value. However, the best we can say with any confidence is that this is plausible.

Note that the last of these figures takes about four minutes on my computer.

25. In considering this problem, we will use rolls of 0 to 5 rather than 1 to 6 for convenience. Although we will only use 6 sided dice, for clarity we will let $n = 6$ for the number of faces of the die. Also we will let $m = 2^n = 64$ be the number of possible sets of differences that have occurred so far. We assume that n is even in all that follows.

    This problem can be solved using a Markov chain with $N = ½n(m-1)+2$ states. State 0 is the state where no dice have been rolled and thus no differences are recorded. State $N-1$ is the state where all six differences have been recorded, regardless of the final roll.

    The other states are a combination of the n possible rolls and the $m-1$ possible sets of differences, other than all differences, reduced by a factor of 2 as the states of just having rolled k and just having rolled $(n-1)-k$ are exactly equivalent. We let the set of differences be represented by the value K, from 0 to $m-2$, inclusive, and the last die roll, or its equivalent, be k, from 0 to $½n-1$, inclusive. These are represented by state $½nK+k+1$. Note that many of these states are impossible, but this does not matter, it just means that there are no sequences of transitions into those states from the starting state. We use a representation of the set of differences such that bit d is set if difference d has been observed, i.e. the number representing the set (we only ever need these binary numbers) is bitwise or-ed with 2^d.

    We thus can start with a transition matrix that is all zeros, and then for each state, other than the final state, i.e. each transient state, there are n possible transitions that create up to n nonzero values in the corresponding row of the matrix. This is "up to" because more than one die roll from a state can end up in the same state. Thus for each transient state there will be n increments of an element in its row, for the n die rolls 0 to $n-1$. Note that this means that our transition matrix will use a list variable, `v0`, rather than a constant list.

    If we start in state 0, with die roll r, no difference is recorded, and we thus create the new state with difference set 0 and die roll $f(r)$, where $f(r)$ is r if $r < ½n$, or $(n-1)-r$ if $r \geq ½n$, i.e. state $f(r)+1$. We can simplify this case to only considering die rolls $r < ½n$ and adding 2 (any value greater than zero works, but this keeps all row sums equal to n) to state $r+1$.

    If, for $0 \leq s < ½n(m-1)$ we start in transient state $s+1$, first we determine k and K from $s+1 = ½nK+k+1$, i.e. $s = ½nK+k$. Then we have a new difference of d = abs(r-k), and a new state L that is the bitwise or of K and 2^d, as noted above. Then the new state, whose transition probability is incremented is $½nL+f(r)+1$, unless this would equal or exceed the number of states, in which case the last state is used, as it represents all final dice values. Note that this makes the new state $\min(½nL+f(r),½n(m-1))+1$.

    We complete the transition matrix with the final element being any value greater than zero, n is convenient to make all row sums equal to n.

    We start with the constant definitions `-c0 n`, `-c1 c0/2`, `-c2 c1*(pow(2,c0)-1)`, `-c3 c2+2` and `-s0 c3*c3`, and the function definition `-f0 p0<c1?p0:c0-1-p0`.

    The final state setting is simply `vset0(-1,c0)`.

    The initial state settings can be implemented by `rloop0(c1,incr0;e0:=2)`.

For the other states we can use, here on two lines for convenience:

```
rloop1(c2,rloop0(c0,r2:=mat_elem[c3](r1+1,
min{c2,c1*bitor{r1/c1,pow(2,abs(r0-r1%c1))}+f0(r0)}+1);ince02))
```

We then use `v1:=mmat_absorb_visits(v0)` and proceed as previous similar problems. With n = 6, the mean is 25.8484, or a rational fraction exactly matching the given solution. This exact solution requires 128 bit integers.

26. This can be solved as another Markov chain, with the states 0 to 6 being the number of kept dice. The transition probabilities are not straightforward to calculate, and represent the most difficult part of the problem.

    The solution here uses three `-eval` options. These could be combined, but this is easier to implement and describe.

    The first `-eval` option determines the transition matrix `v0`, rather than the previous `u0`, and is described below.

    The second `-eval` option analyses `v0` as in previous problems, except using `v3` rather than `u1` (`v1` and `v2` are used in the first `-eval` expression below) and `r0` and `r1` rather than `c0` and `c1`. (`r0` and `r1` are also used in the first `-eval` expression but the variables can be reused as their earlier values have no effect.) The second `-eval` expression is then `v3:=mmat_absorb_visits(v0);r0:=sum(dmat_row(0,v3));r1:=last(v3)`.

    The third `-eval` option outputs the required result similarly to previous problems using the expression `write_ratio(r01);space(2);rwrite_ratio(r01)`.

    The first `-eval` expression uses the function `f0` to determine from its parameter `q0` how many dice should be kept and is specified by the expression `count_eq(counts(q0),1)`. Also before the first `-eval` expression it is convenient to use `-c0 6` for the standard die size and the number of dice, `-c1 c0+1` for the number of states, `-s0 c1*c1` for the number of elements in the transition matrix `v0` and `-s1 c0`, needed for the list `v1` that is used. (Also used is `v2`, but its length can be determined by list length deduction from `v2:=v1` in the expression.)

    It is convenient to use dice numbered 0 to 5 rather than 1 to 6, and `v0` loops through all possible rolls of six dice using `vmask_loop1(c0,`*term*$_1$`)`. *term*$_1$ first sets `v2:=v1` and loops through each possible number `r0` of kept dice, the row number in the transition matrix, so *term*$_1$ is `v2:=v1;rloop0(c1,` *term*$_2$`)`. *term*$_2$ keeps the first `r0` dice in `v2`, without loss of generality to 0 to `r0-1` using `w20:=sequence`. `r1` is then set to the number of dice now to be kept using `r1:=f0(v2)`. This is the column number in the transition matrix, and the element number in the list representing the transition matrix is `r2:=c1*r0+r1`. We then update the transition matrix list `v0` element `r2` using `ince02`. Those four terms, semicolon separated, comprise *term*$_2$.

    The output from that run is 1692288/54575 and 31.0085, which matches the given solution. Given that we now have the transition matrix `v0`, the other results in the given solutions – not requested in the problem statement – could also be determined, but this has not been done here.

27. This is another example of a problem that Monaco can only produce results for specific values of *n* and *s*, although it is possible that from such results an overall result might be hypothesised.

    We can determine the result for c0 and c1 equal to *n* and *s*, respectively using product(count_seq[c1](pattern_list[c0]rand[c1])).

    Starting with standard dice, *s* = 6, the result is obviously zero if *n* < 6. For *n* = 6, 7, 8, 9, 10 the mean value is 5/324, 35/324, 35/81, 35/27, 175/54. For *s* = 4 and *n* = 4, 5, 6, 7, 8 the mean value is 3/32, 15/32, 45/32, 105/32, 105/16. These probabilities must be integral multiples of $s^{-(n-1)}$, and those multiples are, for s = 6: 120, 5040, 120960, 2177280, 32659200 and for *s* = 4 those multiples are: 6, 120, 1440, 13440, 107520. These are all multiples of their first term, which is – as can be easily seen – equal to (*s*-1)!, which when removed leaves for *s* = 6: 1, 42, 1008, 18144, 272160 and for *s* = 4: 1, 20, 240, 2240, 17920. The OEIS (http://oeis.org) tells us that those are ${}^{n}C_6 \times 6^{n-6}$ and ${}^{n}C_4 \times 4^{n-4}$. We can thus hypothesise that that the probability is $[(s\text{-}1)! \times {}^{n}C_s \times s^{n-s}] / s^{n-1}$, which simplifies to $s! \, {}^{n}C_s / s^s$ or ${}^{n}P_s / s^s$. As an example to test this *n* = 16, *s* = 10 has result from the use of Monaco as 1135134/390625, and the same result from that hypothesised formula, which can also be calculated using Monaco. That formula is the given solution.

Dice Sums

28. The example in the problem is easily checked with two runs, 3d6==14 and 5d6==14, which both produce the probability 5/72, matching the given solution.

    For a more general case we compare c0 c1-sided dice and c2 c3-sided dice. This problem is only concerned with the case that c1 = c3, but we consider the general case as we will also use it in the following problem.

    We start with the distribution of c0 c1-sided 0-based dice, sum_dz_dist[c0][c1] which we set as u0. Similarly u1 is sum_dz_dist[c0][c1]. The corresponding distributions of standard 1-based dice would have c0 and c1 zeros prepended. We do not need to create those lists, but it is convenient to designate those as *u* and *v* in order to create the term we need.

    We need to loop over the corresponding elements of *u* and *v* to determine if their probabilities are equal. These are scaled differently, so we need to multiply each by the sum of the other, or by the integers u1 and u0, respectively. However, we can limit the range of elements of *u* and *v* that we need to loop over both to ignore elements where either is zero, and to only consider the shorter of the two lists *u* and *v*. This is from element c4, inclusive, to element c5 of the lists, where c4 is max(c02) and c5 is min(c02+k01).

    We thus use the loop rloop0(c5-c4,r0+=c4,...) and compare elements r0 of *u* and *v*, or elements r1:=r0-c0 and r2:=r0-c2 of u0 and u1, or i01 and i12, after scaling these by the integers u1 and u0, as noted above. If these are equal, i.e. if i01*u1==i12*u0, then we output the element number, i.e. r0. In order that we can differentiate outputs, we precede that with output of the list c0123, and in order to report the common probability we follow that with write_ratio{i01,u0}. To avoid impossible cases, we make the whole loop conditional on c5>c4.

In the indicated case (i.e. `-c0 3 -c1 6 -c2 5 -c3 6`) the output is, as expected, 14. An exhaustive search – using Unix shell script variables to set c0, c1, c2 and c3 equal to c1 – over 2 to 10 dice with sizes 2 to 20, produced (in a few seconds on my computer) the output:

```
{2,3,3,3}  5  2/9
{2,9,4,9}  15  4/81
{2,20,3,20}  27  7/200
{3,4,4,4}  9  5/32
{3,6,5,6}  14  5/72
{4,3,6,3}  10  10/81
{7,2,9,2}  12  21/128
```

These are the six given solutions, plus one additional solution, the last one above – thus answering the give question as to whether there are any other solutions.

29. This is as the previous problem, except with the numbers of dice both equal to 2 rather than the sizes of dice being equal. For the specific problem, 2d5==9 and 2d10==9, both have probability 2/25, matching the given solution.

    Searching for additional solutions is as the previous example, except that we let c0 and c2 both equal 2 and let c1 and c3 be different. As we are only varying two values we can allow larger dice sizes and do so up to 100.

    The solutions – more than the given solutions, which are all included – are:

```
{2,5,2,10}  9  2/25
{2,5,2,15}  10  1/25
{2,10,2,20}  17  1/25
{2,10,2,30}  19  1/50
{2,13,2,65}  26  1/169
{2,15,2,30}  25  2/75
{2,15,2,45}  28  1/75
{2,17,2,68}  33  2/289
{2,20,2,40}  33  1/50
{2,20,2,60}  37  1/100
{2,25,2,50}  41  2/125
{2,25,2,75}  46  1/125
{2,26,2,39}  37  4/169
{2,30,2,60}  49  1/75
{2,30,2,90}  55  1/150
{2,35,2,70}  57  2/175
{2,40,2,80}  65  1/100
{2,45,2,90}  73  2/225
{2,50,2,100}  81  1/125
{2,51,2,85}  76  3/289
{2,52,2,78}  73  2/169
{2,75,2,100}  97  6/625
```

    The given solution is superior in that it proves that there are infinitely many solutions, which we cannot do.

30. As usual, we are limited to considering specific values of *n*. with the usual `-c0 n`, a simple expression that can be used is `sum(tail3(sorted[c0]d6))==18`. With *n* = 4 the probability is 7/432, with *n* = 5 the probability is 23/648, which both agree with the given solution.

    Other solutions that might be faster are possible, but the solution above is easy and fast enough.

31. This is the list `0[3]#max_dz_dist[3][6](4)` – the initial `0[3]` correcting from dz6s to d6s. That list can be output using the function `lwrite` in a `-eval` expression. The list is {0,0,0,1,4,10,21,38,62,91,122,148,167,172,160,131,94,54,21}. This is as the given solution, except that is given from a value of 1, not 0.

32. This problem can be solved for *n* = 6 with the expression `dot(sorted3d6,{1,1,-1})==0`, with probability 5/24, matching the given solution. Other values of *n* can be considered singly as usual, replacing the 6 by *n*, or in general using `-c0 n` and the expression `dot(sorted3d[c0],{1,1,-1})==0`.

33. Assuming that $1 \le k \le n$, we can define this problem with a three state Markov chain where state 0 is the initial state, state 1 is after a roll of 1 to *k*−1, and state 2 is after a roll of *k* to *n*. The transition matrix is:

    $$\frac{1}{n}\begin{pmatrix} 0 & k-1 & n-k+1 \\ 0 & k-1 & n-k+1 \\ 0 & 0 & n \end{pmatrix}$$

    This can be analysed as usual, giving a mean number of transitions until reaching the final state of μ. The last of these transitions is due to a die roll with a mean value of $\frac{1}{2}(k+n)$, the other transitions are due to a die roll with a mean value of $\frac{1}{2}k$. Because rolls are independent, and mean values can be summed even when not independent, the overall mean value is $\frac{1}{2}((\mu-1)k+(k+n)) = \frac{1}{2}(\mu k+n)$.

    Using `-c0 n`, `-c1 k` and `-u0 {{0,c1-1,c0-c1+1},{0,c1-1,c0-c1+1},{0,0,c0}}`, then the mean number of transitions until reaching an absorbing state, i.e. until a roll of *k* or greater, is `mmat_absorb_visits(u0)`, which is set as u1. We then determine c2 and c3 from u1 in the way that r1 and r2 are deduced from v3 in problem 26. Thus μ is the rational value c23, and the required mean value $\frac{1}{2}(\mu k+n)$ is the rational value c45 that is given by `-c45 ratio_add(ratio_mult(c23,{c1,2}),{c0,2})`.

    For example, with *n* = 6, *k* = 5, the mean result is 21/2. The given mean result is:

    $$\frac{n(n+1)}{2(n-k+1)}$$

    which for that example is also 21/2, i.e. the results agree.

    Other examples could be implemented, but instead here we consider that the argument above about manipulation of mean values – although accurate – might not be convincing. So instead here for comparison the actual process is directly modelled in approximate mode by `until(r0+=r1:=d[c0],r1>=c1)`. The mean from that, over ten million results, is about 10.50, with a small uncertainty in the last digit, matching the given solution.

34. This problem can, in principle, be solved using a Markov chain similarly to problem 25, except that there is no need to include the last die roll in the state, and that the die roll

distribution is – because it is the sum of dice rolls – not uniform. We can define c0 as the number of dice, and c1 as the number of dice sides. We use 0-based dice rather than 1-based dice for convenience, and the dice distribution is sum_dz_dist[c0][c1], which we set as u0. State *n* is that we have seen the die rolls indicated by *n* as a bit pattern, and the number of bit patterns is pow(2,k0), which we set as c2. Then the transition matrix v0 has size s0 equal to c2*c2.

We can set the elements of v0 using the -eval expression:

```
rloop0(c2,rloop1(k0,r2:=bitor{r0,bitshift(1,r1)};r3:=mat_elem[c2](r0,r2);
e03+=i01))
```

Here, r0 loops over the states, r1 loops over the die rolls, r2 is the new state number, and r3 is the element number in v0 for the transition, incremented using e03+=i01, as more than one transition can use the same element.

As usual we then, in two more -eval statements, set v1 to mmat_absorb_visits(v0) and output the required mean using r0:=sum(dmat_row(0,v1));r1:=last(v1) and then use write_ratio and rwrite_ratio to output the rational mean r01.

For c0 equal to 2 and c1 equal to 6, the first posed problem, this is practical, although slow; it takes about three minutes on my computer. The results – which only require standard 64 bit integers – is 769767316159/12574325400 or 61.2174, which exactly matches the given solution.

Unfortunately, this is not practical for three dice, as the number of states increases from 2^{11} = 2048 to 2^{16} = 65536, and even linear scaling – which is not the case – would make this impractical. Instead we turn to an approximate solution. Using c0 and c1 as above, we record which numbers have been rolled in v0, with length s0 set to c0*(c1-1)+1. Using 0-based dice, we use the expression until(r0:=c0@dz6;e0:=1;incr1,all(v0)).

Using ten million evaluations of that expression, repeating the above case with two dice the mean is about 61.2, agreeing with the exact solution. For three dice the mean is about 338. This agrees with the given solution to the indicated precision.

35. We can, as usual, only consider specific values of *n*, here used as -c0 *n*. There is a weighted term that directly implements this problem and can be used in the expression find_eq(until_sum[c0]d6). Using that expression, the mean values for some values of *n* are: *n* = 5, 2401/1296 = 1.85262; *n* = 10, 33495175/10077696 = 3.32369; *n* = 20, 6.1902; *n* = 50, 14.7619, *n* = 100, 29.047. Values above *n* = 24 require larger integers, and *n* = 100 has reached the point where it requires 1209319 evaluations of the expression to compute that result. These results all agree with the given solutions. The asymptotic solution cannot be derived.

36. As usual we can only solve for specific values of *n*, here used as -c0 *n*. The solution can use a Markov chain with *n*+1 states. In order to keep the usual pattern of the absorbing state being the final state, state 0 is the start state, states 1 to *n*-1 have remainder 1 and state *n* has remainder 0.

However, we start by ignoring termination, and the transition matrix for $n = 4$ is:

$$\frac{1}{6}\begin{pmatrix} 0 & 2 & 2 & 1 & 1 \\ 0 & 1 & 2 & 2 & 1 \\ 0 & 1 & 1 & 2 & 2 \\ 0 & 2 & 1 & 1 & 2 \\ 0 & 2 & 2 & 1 & 1 \end{pmatrix}$$

and for $n = 8$ is:

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

With the final state set to be absorbing, these matrices can be created as u0 by using `-u0 ccreate_mat[c0+1](l1?l0<c0?(5-(l1-l0-1)%c0+c0)/c0:l1==c0:0)`.

[To arrive at the critical part of that formula, `(5-(l1-l0-1)%c0+c0)/c0`, note that must work for l0 from 0 to c0-1 and l1 from 0 to c0. It is convenient to replace the d6 rolled by dz6+1, and add the 1 to the remainder before transition, so that is l0+1, modulo c0, and the remainder after transition is l1, modulo c0, so the required dz6 roll, modulo c0, required is (l1-l0-1)%c0. The number of dz6 rolls that equal that value *k*, modulo c0, considering the cases where c0 ≷ 6, is (5-*k*+c0)/c0. Note that the simplification (5-*k*)/c0+1 only works if all possible *k* satisfy *k* ≤ 5, which is only the case if c0 ≤ 6.]

Using the usual analysis based on that u0 for $n = 4$ and $n = 8$ gives means results of 4 and 8, as the problem requires.

The given solution asks a followup question without an answer: what are necessary conditions on the values of a die so that *n* is the expected number of rolls until the sum is a multiple of *n*? We cannot answer that question directly, but we can investigate other dice.

To do that, we use u0 for the dice face values, and the example `-u0 {2,3,3,4,4,5}`. We use v0 rather than u0 as the transition matrix, v1 rather than u1, and r0 and r1 as the numerator and denominator of the mean determined from v1. v1, r0 and r1 must be set using `-eval` rather than by using `-v1 -r0` and `-r1`. To set v0., for convenience we use `-c1 c0+1` and then we can set `-s0 c1*c1`. To set v0 we use `-eval` with expression `rloop0(c0,rvloop1(u0,r2:=c1*r0+((r0+r1)%c0?:c0);ince02));r0:=-1;e0:=1`. Note that the operation *k*%c0?:c0 is *k* modulo c0 in the range 1, …, c0. The ratio of r0 to r1 has mean result 4, showing that this is a die with the same property. Experimentation (see the summary section below) shows that for a d6 all of whose sides are from 1 to 6, the only die without this property for $n = 4$ is one with all sides equal to 4, which obviously requires a single roll, and there are no such dice for $n = 8$.

37. This is as problem 35, except rolling an *n*-sided die rather than a d6, and, using `-c0 n` the required expression thus becomes `find_eq(until_sum[c0]d[c0])`. Some results are: *n* = 5, mean is 1296/625 = 2.0736; *n* = 10, mean is 2.35795; *n* = 20, mean is 2.52695 (requires 128 bit integers). These match the given solutions; the greater given solutions (the next is *n* = 100) are not possible using this approach with any available integer size.

38. Letting *x* be `c0`, we can modify the solution to problem 35 to `sum(until_sum[c0]d6)==c0`. Example solutions are: *x* = 5, probability is 0.308771; *x* = 10, probability is 0.289288; *x* = 20, probability is 0.285621; *x* = 50, probability is 0.285714 (requiring greater than 128 bit integers). It is plausible that this heading for a limit of about 0.286, but no more can be said using this approach. Using approximate mode, with the expression `until(r0+=d6,r0>=c0)==c0`, we need a hundred million evaluations rather than ten million to have confidence in the third decimal place, and some confidence in the fourth. For *x* = 50 the probability is about 0.2857, with uncertainty in the fourth figure. For *x* = 100 the probability is about 0.2858, with uncertainty in the fourth figure.

    Those results match the given solution – except those only go up to *x* = 20. The given solution provides the asymptotic solution as being 2/7 or 0.285714. Answering the final open question in the given solution section is not possible.

    A better approach to this problem for some values of *x* is using a Markov chain with two absorbing states, and thus *x*+2 states in total, looking for the relative probabilities of being absorbed by those two states using the function `mmat_absorb_states`. Letting `c1` be `c0+2`, the required transition matrix `v0` can be calculated by using the expression `rloop0(c0,rloop1(6,r2:=c1*r0+(r0+r1+1?<c0+1);ince02));r0:=-1;e0:=1;r0:=-c1-2;e0:=1,` and `r01:=tail2(dmat_row(0,mmat_absorb_states(v0)));rwrite_ratio{r0,r01}`, which produces the probability 0.285714 for both *x* = 50 and *x* = 100 to normal precision. The difference can be seen by increasing the output precision, with `-precision 10` the probabilities are 0.2857143001 and 0.2857142857. An asymptotic solution of 2/7 is not proven, but looks likely, based on these later results, which agree to six significant figures.

39. This is a fairly straightforward modification of the previous problem. We only use the Markov chain approach here. Considering the range of values to be tested for to be *y* values, and adding `-c1 y`, so that the previous `c1` becomes `c2`, and defining state `c0` as the values *x*, inclusive, to *x*+*y*, exclusive, and the final state `c0+1` as the values from *x*+*y*, inclusive, then the core of the loop evaluating `v0`, the calculation of `r2`, becomes `r2:=r0+r1+1;r2:=c2*r0+(r2<c0?r2:r2<c01?c0:c0+1)`.

    With *y* = 2, i.e. the probability of reaching *x* or x+1, results are: *x* = 5, probability 0.617541; *x* = 10, probability 0.534467; *x* = 20, probability 0.523986; *x* = 50, probability 0.52381; *x* = 100, probability 0.52381. The latter two results require larger than 128 bit integers. The given solutions do not contain these values, only the asymptotic probability 11/21 = 0.5238095, which these results are consistent with. For greater values of *y* we here just use *x* = 100, with probabilities from *y* = 3 to *y* = 6 of 0.714286, 0.857143, 0.952381 and 1. These are all close to the given asymptotic results of 5/7 = 0.7142857, 6/7 = 0.857129, 20/21 = 0.9523810 and 1.

40. A `d6` has distribution `u0:={0,1,1,1,1,1,1}`. Rolling that many d6s has distribution `u1:=accum_dist(u0,u0)`, and we can output its mean, median and mode as

`write_ratio{dot(u1,sequence),u1}`, `write(find_ge(sigma(u1),u1/2))` and `write(find_max(u1))`. The results are mean 49/4, median 12, mode 6.

Now rolling that many d6s, this has distribution `u2:=accum_dist(u1,u0)` and its statistics can be output accordingly using `u2` rather than `u1`. However, this requires larger than 64 bit integers – 128 bits is sufficient. The mean, median and mode are 343/8, 41 and 20.

Both sets of results agree with the given results.

41. Although this is a variant of the first part of the previous problem, it is easier to solve it by direct implementation. The same might be true of the first part of the previous problem, but due to the second part – which has no equivalent here – the approach there is used. However, if we consider the previous first part, it could be implemented as `d6@d6` – but that cannot be used in exact mode. Instead we could use the randomness pool, and the expression `d6@pool_d(6)`, the initialising the pool with `-pool 6 6`. Here we can use the expression `(r0:=d6)@((r1:=pool_d(6))>=r0?r1:0)`. The mean is 133/18 = 7.38889, the median is 6 and the mode is 6. These all agree with the given solution.

42. The expected value for *n* dice is *n* times the expected value with one die. This is an open-ended distribution, but we can determine its mean value using a Markov chain, where if we let the die have *m* sides then the transition matrix is:

$$\frac{1}{m}\begin{pmatrix}1 & m-1\\ 0 & m\end{pmatrix}$$

If we let `c0` be *m*, and let this matrix be `u0`, it can be set as `{{1,c0-1},{0,c0}}`. The matrix `u1` defined by `mmat_absorb_visits(u0)` is, when normalised, the mean number of visits from each state to each other state, where the normalising factor is `c1`, which can be defined by `dmat_diag(mmat_absorb_diag(u0),u1)`. (We could simplify this as we know which is the absorbing state.) The mean value of the repeated rolls of the die is then `c0` times the mean number of visits to the first state, where we also start, plus the mean of the values 1, ..., `c0-1` for the one transition to the second state, on rolling anything but a `c0`. That mean is, as a rational number, `{c0,2}`, and thus the overall mean is the rational number `ratio_add({first(u1),c1},{c0,2})` which we define as `u2`. Finally we output `u2` as rational and real using `write_ratio` and `rwrite_ratio`. The result for *m* = 6 is 21/5 or 4.2, as the given solution. For *n* dice the mean is 4.2*n*.

For the distribution we need another approach, and we use that although the distribution is open-ended, we can cap the number of re-rolls and still get some values of the distribution exactly. We set parameters of `c0` as the number of dice, `c1` as the size of the dice, and we could use `c3` as the maximum number of re-rolls, per die. However, to compare with the given result, we instead use `c2` such that we want the exact distribution of results up to `c2`, exclusive, and the probability of a result of `c2` or more. We thus derive `c3` from the other parameters, which can be as `(c2-c0+1)/c1`. We will determine the required distribution as a number of occurrences, which could be by rolling `(c3+1)*c0` dice, requiring a normalisation factor of `pow(c1,(c3+1)*c0)`, which we set as `c4`. The maximum possible result we can have from one die is `(c3+1)*c1+1`, which we set `s0` and `s1` to, as we need the two list variables `v0` and `v1` to be of that length.

The list variables `v0` and `v1` contain two parts of the distribution for a single die, `v0` is for when the last die roll was not equal to `c1` and is not re-rolled; `v1` is for when the last die roll was equal to `c1` and is re-rolled (unless the rolling limit has been reached). We set `u0`

and u1 to those distributions for a single roll, as {0}#1[c1-1]#{0} and 0[c1]#{1}, respectively. v0 and v1 are initialised to those lists, but with their set lengths, thus as head(u0) and head(u1), respectively. (The lengths of the head functions are deduced.)

The distributions v0 and v1 are then updated c3 times, using do[c3] in a first -eval option. First v0 is multiplied by c1 as those results are not affected by this die roll (which is not actually made in this case). Then it is updated by the die rolls applied to v1 that stop there. Including another use of head to manage the length, this is adding head(sum_dist(v1,u0)) to v0. Similarly v1 is replaced by head(sum_dist(v1,u1))).

A second -eval option creates v2, the distribution of c0 summed dice using the distribution v0+v1 of one die, as v2:=multi_dist[c0](v0+v1), then v3 extracts the part of this of interest – and which is exactly correct despite the capping of the number of rolls – as v3:=head[c2](v2).

The third and final -eval option is output. First we output v3 as rational and real lists, using lwrite and rlwrite, in each case normalised using c4. Then we output the probability of a result of c2 or greater, the integer v2-v3, again normalised using c4 and as rational or real using write_ratio and rwrite_ratio.

This is all put together as shown in the summary section below.

For the given example, c0 is 5, c1 is 6 and c2 is 20. The probability of a result of 20 or more is 48601/93312 or 0.520844, which matches the given result. The rational distribution is:

```
{0,0,0,0,0,1/7776,5/7776,5/2592,35/7776,35/3888,121/7776,
1115/46656,1555/46656,665/15552,2365/46656,659/11664,2795/46656
5695/93312,5635/93312,5495/93312}
```

These exactly match the given solution's generating function coefficients.

43. The number of dice rolled is indefinite, and so this problem is not suitable for a general exact solution, although it can be handled as a Markov chain for small values of *x*. The easiest way to define the transition matrix is to have 2^x states, where state *k* records, in bits 0, 1, ... whether a 1, 2, ... has occurred. We start in state 0, and state 2^x-1 is absorbing. For example for $x = 3$, the transition matrix is:

$$\frac{1}{6}\begin{pmatrix} 3 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 3 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 4 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 3 & 0 & 0 & 0 & 3 \\ 0 & 0 & 0 & 0 & 4 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 4 & 0 & 2 \\ 0 & 0 & 0 & 0 & 0 & 0 & 5 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 6 \end{pmatrix}$$

Processed in the usual way, see earlier problems using Markov chains, this has a mean number of dice rolled of 139/18, or 7.72222, which agrees with the given solution.

That version of this matrix was created by hand. It could have been created using -c0 3 for *x*, -s0 pow(4,c0) for the matrix order, the user-defined function f0 defined by bitor{p0,bitshift(1,p1-1,c0),bitshift(p0,p1,c0)} that updates a state p0 to the

new state with a die roll p1, and the expression, using the option `-eval` for convenience, `cloop[pow(2,c0)](cloop6(cset(f0(l0,l1+1),dmat_vset0(l0,l2,dmat_get(l0,l2,v0)+1))))`.

Note that because this calculates the transition matrix as v0 rather than u0 (using u0 would be possible, but needing extra steps) the previous Markov chain processing using u1, c0 and c1 is replaced by v1, r0 and r1, and the options to set those must become `-eval` options for the order of expression evaluation to be correct.

For *x* = 4 the mean is 9139/1152 = 7.93316.
For *x* = 5 the mean is 28669967/3600000 = 7.96388.
For *x* = 6 the mean is 777101609/97200000 = 7.99487.
For *x* = 7 the mean is 2341848577/291600000 = 8.03103.
For *x* = 8 the mean is 883143607/109350000 = 8.0763.
For *x* = 9 the mean is 42538515011/5248800000 = 8.10443.
For *x* = 10 the mean is 256722093191/31492800000 = 8.15177.
For x = 11 the mean is 1550818181021/188956800000 = 8.20726.

These all agree with the given solution, and all need no more than 64 bit arithmetic. However, the last of these took about four minutes on my computer, and the time taken is climbing by almost an order of magnitude for each larger value of x, so – like the given solution – this is a good place to stop.

The additional results given , not requested in the problem, could easily be obtained by a process similar to problem 18, but this has not been done here.

44. The indefinite number of rolls makes this unsuited for an exact solution, and so we try approximate mode, using the function `-f0 pow(sqrt(p0),2)==r0` to test for squareness, and then the expression `until(incr1,f0(r0+=d6))`. Over a hundred million results the mean is about 7.08. This agrees with, but is rather less accurate than, the given solution (which is accurate to 50 significant figures, starting with 7.07976).

45. This is as the previous problem, replacing f0 by `is_prime`. The result is about 2.429. (Greater precision is possible here because primes are denser than squares and thus there are fewer very long runs that increase the variance of the results. This is evidenced by that these hundred million results have a longest run of 68 rolls, whereas the previous problem had a run of 1065 rolls. The result again agrees with the given solution, although the last digit is very slightly not the best, the accurate mean being 2.428498.

    For composite numbers, use f0 defined by `-f0 r9:=p0;r9>1&!is_prime(r9)`. The approximate solution is about 2.128. The given solution observes that the number of rolls has a maximum, and an exact solution is possible. Replacing d6 by `pool_d(6)`, we need a pool defined by `-pool` *n* `6` for a suitable *n*. The given solution suggests a value, but any *n* greater than or equal to the needed value works, and then can be reduced to find the minimum, which is 26, matching the given solution. The mean value is then 605134982368031963 47/28430288029929701376, or 2.12849, matching the given solution.

46. We can write a first digit function as `-f0 r0:=p0;while(r0>=10,r0/=10);r0`. Then we can use the expression `f0(product(sorted2d6))==1` for two dice and similarly for other numbers of dice.

For example, for two dice the probability is 1/3 or 0.333333, for three dice it is 65/216 or 0.300926, for ten dice it is 18271529/60466176 or 0.302178.

However, the "what is going on?" part of the problem cannot be answered using Monaco.

Non-Standard Dice

47. This is a problem where we can at best, just illustrate the issue.

    For example, an alternative to d6, one that can only be used in approximate mode, is `floor(6*uniform)+1`. A slightly weighted alternative is `floor(5.8*uniform)+1`, with a reduced probability of a 6. We can roll two of these as `repeat2{floor(5.8*uniform)+1}` and then check for doubles as `same(repeat2{floor(5.8*uniform)+1})`. Using ten million evaluations the probability is 0.16757, with a 95% confidence interval of [0.167339, 0.167802], which the unweighted probability of ⅙ = 0.166667 is well outside. Thus this particular weighting has, as expected, increased the probability of doubles.

48. Monaco is not useful to address this problem. At best it can verify a solution derived by other means, but if the required weighting is not rational – as here it is not – then even that can only be approximate.

    The given solution only provides one case that can be so handled, using the expression:

    ```
    do_sum2(real_dist(0.38332764225046719182826783 97,
    0.14694008131330216014651697 41,0.11265238984384009963206170 58,
    0.10795693748179925338483297 81,0.11587205926654175072308109 30,
    0.13325088984404954428523940 95))+2
    ```

    The +2 handles the correction between 0-based and 1-based dice. The `-histogram` output using `-range 2 6` over a billion results (taking less than three minutes on my computer) is:

    ```
    2 - 146923840 ~ 0.146924  [0.146902, 0.146946]
    3 - 112642072 ~ 0.112642  [0.112622, 0.112662]
    4 - 107972277 ~ 0.107972  [0.107953, 0.107992]
    5 - 115881576 ~ 0.115882  [0.115862, 0.115901]
    6 - 133248905 ~ 0.133249  [0.133228, 0.13327]
    ```

    These figures, which can be quoted as about 0.1469, 0.1126, 0.1080, 0.1159 and 0.1332, with some uncertainty in the final figures, mostly agree with the probabilities used, which to the same precision are 0.1469, 0.1127, 0.1080, 0.1159 and 0.1333.

49. We can find such dice by brute force. We could look for dice with faces in the range 1 to c0, but because the program uses zero-based indexing it is easier to use dice numbered 0 to c0-1. Those dice are v0:=sorted6dz[c0] and similarly for v1. Note that although this looks like we are numbering the dice at random, in exact mode we are actually considering all such possible dice, as required. Collecting statistics is not meaningful here, rather we output v0+1 and v1+1 – restoring the offset to report one-based dice – when the dice are suitable. Although we use weighted terms in order to only consider each possible die pair once, the weighting of the results is irrelevant. To only report any dice pair once, we also only consider cases in which v0 is earlier than v1 in list order.

Thus the expression takes the form – split on two lines due to length, but only here:

```
v0:=sorted6dz[c0];v1:=sorted6dz[c0];list_le(v0,v1)&(condition)
&(lwrite(v0+1);space(2);lwrite(v1+1))
```

condition is to be true when the two dice match the required condition, and can be `v2:=outer_sum(v0,v1);v3:=count_seq(v2);list_eq(v3,sequence6#rsequence5+1)`

It is clear that `c0` must be at least 6, but that it is pointless it being above 11. With value 6 or 7 the only solution found is two standard dice {1,2,3,4,5,6}. With any value from 8 to 11 one other solution is found, {1,2,2,3,3,4} and {1,3,4,5,6,8}. This matches the given solution.

50. Monaco cannot be used to solve this problem. It can be used to verify the given solution – but only approximately. First note that this means that the limited precision given solution is more than precise enough, any discrepancy with the ideal weightings is small compared to even the largest practical approximate solution. The approximate results used here are for ten million evaluations.

    We create the required die rolls using the sum of two uses of the variadic integer function `real_dist`, with arguments the given lists of weights – repeated in the summary section below – plus two to correct from 0-based to 1-based dice. We can see the distribution using `-histogram`, but rather than repeat that – which by default in every case includes the expected probability – here we can use the option `+chi` to investigate the fit to the expected distribution. Letting `c0` be the number of dice sides $n = 10$, the two parameters of `+chi` are the expected die rolls are `sequence[2*c0-1]+2`, and the relative probabilities `(sequence[c0]+1)#(rsequence[c0-1]+1)`. The default result is that the hypothesis is not rejected at the 80% level, the lowest level that the program implements. That is the best indication of a likely fit as is possible – and with ten million results that is a good indication of a fit.

51. Monaco is not useful to address this problem. At best it could verify a solution derived by other means, but the point of this problem is there are no solutions, and there is no possibility of a search to confirm that.

52. For A beats B this is tested by the simple expression `d{2,2,4,4,9,9}>d{1,1,6,6,8,8}` and has probability 5/9. The other two cases have similar results – agreeing with the given solution. Discussion is beyond the scope of this or any similar program.

    Considering the followup questions in the given solution, we could attempt to find other sets of dice – and hence bound the values required – using brute force, but this would be anywhere from time consuming to impractical depending on the search space.

53. We can output all such single dice by first creating all possible dice, similarly to problem 49 but for a single die `v0`, and creating all of its sums as `v1`, again similar to that problem. Then we can get that distribution as `v2:=count_seq11(v1)` and output `v0+1`, the un-

offset die, if `all(v2)` is true. In order to investigate the distribution we also output `sort(v2)` in this case. See the summary section below for the combined expression.

This finds eleven such dice: {1,1,2,3,5,6}, {1,2,2,3,5,6}, {1,2,3,3,5,6}, {1,1,2,4,5,6}, {1,2,2,4,5,6}, {1,2,3,4,5,6}, {1,2,4,4,5,6}, {1,2,3,5,5,6}, {1,2,4,5,5,6}, {1,2,3,5,6,6}, {1,2,4,5,6,6}. None has a distribution that any meaning of "more uniform" could apply to.

Switching to two dice, making the obvious adjustments to the expression, there are 3601 pairs of dice. Rather than output these we can count them by just having the result as whether the dice match the test and using the option `-output %#ny`, which ignores weights The number of pairs of dice can be reduced to 1806 by removing duplicate cases are using `list_le(v0,v1)` before calculating what is now `v2`, the 36 results.

Looking for a more uniform distribution of results, we can reduce this number further we by excluding cases where any result can occur more than 6 times, using `max(v3)<=6`, v3 being the relative probabilities of the 11 possible results. This reduces the number to 489. If we instead make that `max(v3)<=5` then this reduces the number to 24 pairs. These have some claim to being more uniform, although the final solution we select below is actually none of these. The first of these is {1,1,2,5,5,6} and {1,1,3,3,4,6} with sorted distribution {1,1,2,2,4,4,4,4,5,5}. None have `max(v4)<=4`.

To go further, we then need to consider what "more uniform" means. Here we assume it means, in particular, either reducing the incidence of the most common result, or increasing that of the least common result, i.e. minimising `max(v3)-min(v3)`. We no longer require that `max(v3)<=5` (we will necessarily have `max(v3)<=6`).

To show any improvement on two standard dice we start with `max(v3)-min(v3)<5`. That reduces us to 57 solutions, so we try filtering further to `max(v3)-min(v3)<4`, which reduces us to a single solution, {1,1,1,6,6,6} and {1,2,3,4,5,6}, with sorted distribution {3,3,3,3,3,3,3,3,3,3,6}. It is readily seen (by symmetry) that the single spike is for a result of 7. This we thus claim is the most uniform result pair of dice.

The given solution adopts a different definition of "most uniform", as mean squared error from a unform distribution. It finds the same 11 single dice and also agrees that the standard die gives the most uniform results. For different dice it also agrees with the above selection.

54. The probability that the sum of two standard dice is prime is easily tested using `is_prime(2d6)`, and has probability 5/12. (This is not listed in the summary section below as it is not part of the actual problem – this probability is provided in the problem.)

    We cannot solve this problem definitively for any possible die face values. Instead we look for dice with face values from 1 to `c0`, and increase `c0` to see how the probability increases. To do this we need to use the evaluations of the main expression for different dice, and compute the relative probability for a pair of the chosen dice in each evaluation. Given a die `v0`, that relative probability (of the sum of the two dice being prime) is determined by first letting `v1:=outer_sum(v0,v0)`, the set of all 36 totals, and then determining the number of prime results from `r1:=vcount10(is_prime(e10))`

    We can create a suitable `v0`, with no repeated faces, as `combine6from[sequence[c0]+1]`. We limit the cases that we are interested in to when `r1>=c1`, for values of `c1` that start at

15 (the value of r1 for a standard die) and can be increased as we use larger values of c0. The output of interest is v0 and r1.

With c0 = 6 we, necessarily, just get a standard die. Improvements, including the maximum value of c1 that produces results, are: c0 = 7, c1 =15 (3 solutions); c0 = 8, c1 = 15 (5 solutions), c0 = 9, c1 = 17 (1 solution); c0 = 10, c1 = 19 (1 solution). The latter is the die {1,2,3,4,9,10}. No better dice were found up to c0 = 40.

The given solution shows that 19 is the maximum possible – which our result suggests but cannot prove. The same example die is quoted.

55. With an additional parameter for the dice size, this could simply follow the previous solution, but note that for all results to be prime, hence odd (except 2, which cannot occur in a maximal solution) we must have all odd numbers on one die, here v0, and all even numbers on the other die, here v1. So we have three parameters: c0, the die size *n*, c1 such that 2*c1 is the maximum allowed on either die, and c2, the number of solutions to find – while we know we want c2 = c0*c1, it is convenient to include it so we can see how we are making progress. The expression is a modification of that in the previous problem, including output of v0 and v1; for details see the summary section below.

    For *n* = 2 we immediately get the solution {1,3} and {2,4}. For *n* = 3 we need to reach c1 = 5 to get the solution {1,3,9} and {2,4,10}. For *n* = 4 we need to reach c1 = 16 to get four solutions, the one that minimises the maximum value required is {1,7,13,31} and {6,10,16,30}. (The other three require values of 32.) For *n* = 5 we need to reach c1 = 23 to get four solutions, the two that minimise the maximum value required being {3,9,15,39,45} and {2,8,14,28,44}, and {3,9,15,29,45} and {2,8,14,38,44}. (The other two require values of 46.) Note that this finds a second solution that is not in the given solutions. This takes about 14 minutes on my computer, and we cannot find any complete solutions for larger values of *n* in a reasonable amount of time. (We could verify the given solutions for *n* = 6 and *n* = 7, but this has not been done here.)

    The given solution suggests some generalisations of this problem, but these have not been considered here.

56. We cannot fully solve this problem, but it is possible to test small numbers of dice with limited values of dice faces that do not repeat on a die to see if any solutions can be found. We let s0 and s1 be the dice sizes – there is no need for them to be equal, although the given problem assumes that they are – and let the dice faces be from 0 to c0, inclusive; for examples here we use `-c0 30`. This means that the maximum total possible is 60, so we need Fibonacci numbers up to 60, which – including zero but no duplicate values – can be defined by `-u0 {0}#recurrence9({1,2},{1,1})` – or this could be just be by `-u0 {0,1,2,3,5,8,13,21,34,55}`.

    We then can loop through the required faces on the first die represented by list v0 using `v0:=combine[s0]from[sequence[c0+1]]` and similarly for v1 using s1. The set of possible sums is `v2:=outer_sum(v0,v1)`, which contains only Fibonacci numbers if `sum_eq(v2,u0)==s2`, and if so the lists v0, v1 and v2 can be output by using the term `vwrite0;space;vwrite1;space;vspace2`. It is also useful to add `-numbers` to count the number of evaluations that are looped through in exact mode.

    With two 2-sided dice, requiring 216225 evaluations, within the given limits, there are 45 solutions, from {0,1} and {0,1} making {0,1,1,2} to {4,25} and {9,30} making {13,34,34,55}.

With two 3-sided dice, requiring 20205025 evaluations, there are – as the given solution indicates – no solutions in the given range.

However, there are also solutions – from 2090175 evaluations – with one each 2-sided and 3-sided dice. Within the given range only six of them, all with the 2-sided die having the form $\{0,F_n\}$, where F_n is the n-th Fibonacci number, and the 3-sided die has the form $\{0,F_{n-1},F_{n+1}\}$. It is straightforward to see – without using the program – why all such lists are solutions, but not if these are the only solutions, however n others have been found. A 2-sided die and a 4-sided die have no solutions in the given range.

Games with Dice

57. This is an absorbing Markov chain with four states: player A to play, player B to play, player A wins, player B wins. The transition matrix is:

$$\frac{1}{36}\begin{pmatrix} 0 & 31 & 5 & 0 \\ 30 & 0 & 0 & 6 \\ 0 & 0 & 0 & 36 \\ 0 & 0 & 0 & 36 \end{pmatrix}$$

Setting `u0` to the relative probabilities in this matrix as usual, the required probabilities can be determined from the matrix `mmat_absorb_states(u0)`; in particular, since we start in the first state, if we let `c01` be `tail2(dmat_row(0,mmat_absorb_states(u0)))` then the two relative probabilities of winning are `c0` and `c1`, and we can use `write_ratio{c0,c01}` and `write_ratio{c1,c01}` to report them.

The results are that player A has a probability of winning of 30/61, and player B has a probability of winning of 31/61. The game is almost fair, but not completely so. These results agree with the given solution.

58. All we can do for this problem is either to verify a solution found otherwise, or to search through possible results to see if a solution can be found. To do the latter we move all of the work into the main expression and replace `u0` by `v0`, `c0` by `r0` and `c1` by `r1`, looking for a solution with `r0==r1`.

If now the probabilities of player A and player B winning on their turns are m/d and n/d, respectively, then the transition matrix is:

$$\frac{1}{d}\begin{pmatrix} 0 & d-m & m & 0 \\ d-n & 0 & 0 & n \\ 0 & 0 & 0 & d \\ 0 & 0 & 0 & d \end{pmatrix}$$

Note that we can usually – and always for rolling 2d6 – construct sets of winning rolls to produce any given m and n.

If we let `c0` be d, then we can let `c1` and `c2` be the maximum values of m and n considered. It is easiest to consider all possible values of m and n, for which we use `r8` and `r9`, but immediately reject with no further calculation any cases for which $m \geq n$. There is no point in considering cases for which `c1` > $d/2$ or `c2` > d. If possible, and in these cases it is, we let `c1` and `c2` be $d/2$ and d.

For $d = 36$, there are no solutions other than the cases $m = 9$ and $n = 12$, $m = 12$ and $n = 18$, and the trivial case $m = 18$ and $n = 36$. The first two of these can be reduced to

cases for *d* = 12 and *d* = 6, respectively. If we consider rolling three dice with *d* = 216 then there is a fourth solution with *m* = 24 and *n* = 27, which can only be reduced to a case for *d* = 72. Keeping it as a sum of three dice solution, *m* = 24 can be realised by, for example, rolling 4 or 8, while *n* = 27 can be realised by rolling 10.

Moving away from standard dice, using *d* = 100 (which would include any interesting cases for e.g. *d* = 10 or *d* = 20) there is the case *m* = 20 and *n* = 25, which can be reduced to *d* = 20, *m* = 4, *n* = 5.

These results for standard dice agree with the given solution (although that chooses the equally likely 11 rather than 10 in the three dice case). The given solution does not consider non-standard dice.

59. This is a modification of problem 57 that needs an extra state, because the starting state is not the same as the usual player A's turn state. (There are other ways to define states that also work.) Although it only happens on the third turn, we here keep the usual player A start state before the player B start state, and the transition matrix is:

$$\frac{1}{1296}\begin{pmatrix} 0 & 0 & 1116 & 180 & 0 \\ 0 & 0 & 961 & 335 & 0 \\ 0 & 300 & 0 & 0 & 996 \\ 0 & 0 & 0 & 0 & 1296 \\ 0 & 0 & 0 & 0 & 1296 \end{pmatrix}$$

We then analyse this exactly as problem 57 to get the probabilities of A and B winning, these being 10355/22631 and 12276/22631, which matches the given solution.

60. We can, as usual, only investigate single values of *m* and *n*. However, if we start with `-c0` *m* and `-c1` *n*, then the basic probabilities can be seen in `-histogram` output for `d[c0]<=>d[c1]`, which is 1 if player 1 wins, -1 if player 2 wins, 0 if a tie.

All the following example results are for *m* = 10, *n* = 6. In the basic case, player 1 wins with probability 13/20, player 2 wins with probability 1/4 and a tie has probability 1/10.

The easiest way to continue until not tied is to give a tie a weight of zero, so the expression becomes `r0:=d[c0]<=>d[c1];weight(r0!=0);r0`. If wanting to convert that to probability output, rather than the -1 and 1 output needing a histogram, use the expression `r0:=d[c0]<=>d[c1];weight(r0!=0);r0>0`, where true is player 1 wins and false is player 2 wins, with probabilities 13/18 and 5/18, respectively.

If a tie is a win for player 1, use `d[c0]>=d[c1]`, with example probabilities ¾ for a player 1 win, ¼ for a player 2 win. If a tie is a win for player 2, use `d[c0]>d[c1]`, with example probabilities 13/20 for a player 1 win, 7/20 for a player 2 win.

The given solution is general. For the first scenario they are player 1 wins with probability $1 - \frac{n+1}{2m}$, player 2 wins with probability $\frac{n-1}{2m}$, and a tie has probability $\frac{1}{m}$. For our example these are 13/20, 5/20 = 1/4 and 1/10. For the second scenario they are player 1 wins with probability $1 - \frac{n-1}{2(m-1)}$ and player 2 wins with probability $\frac{n-1}{2(m-1)}$. For our example these are 13/18 and 5/18. For the third scenario they are player 1 wins with probability $1 - \frac{n-1}{2m}$ and player 2 wins with probability $\frac{n-1}{2m}$. For our example these are 15/20 = 3/4 and 5/20 = 1/4. For the fourth scenario they are player 1 wins with probability $1 - \frac{n+1}{2m}$ and player 2 wins with probability $\frac{n+1}{2m}$. For our example these are 13/20 and 7/20. The solutions thus agree.

61. This is another Markov chain problem. Letting c0 be the number of tokens each player starts with, it is convenient to have states 0 to 2*c0, inclusive, for the number of tokens that player B has. The absorbing states are then 0 (player A wins) and 2*c0 (player B wins) and the starting state is state c0. To create the transition matrix note that the three die roll has 216 possibilities, which we set as c1, a roll of 11 (player A winning) has 27 possibilities, which we set as c2, and a roll of 14 (player B winning) has 15 possibilities, which we set as c3.

    The transition matrix is then created as u0, which can be defined by the expression ccreate_mat[c4](l0==0|l0==2*c0?l1==l0?c1:0:l1==l0-1?c2:l1==l0+1?c3:l1==l0?c1-c23:0). We then define u1 almost as usual, as dmat_row(c0,mmat_absorb_states(u0)), where c0 here is the starting row number. We then have the relative probabilities of player A and player B winning as first(u1) and last(u1), respectively. We can assign these to c4 and c5 and produce player A's winning probability as usual.

    This requires more than 128 bit integers, but the maximum 1024 bits suffices. Player A's probability of winning is 282429536481/282673677106 or 0.999136, which matches the given solution in this case. (The given solution is a general solution, which as usual we can only implement instances of.)

62. There are two possibilities here: player A goes first and player B goes first.

    We start with player A goes first. Player B would win on his tenth turn, so player A has ten turns to win, and his score after ten turns is 30d6. (He might win earlier than his tenth turn, but that does not matter.) This requires 128 bit integers. We thus want to know the probability that 30d6 ≥ 100, or more conveniently 30dz6 ≥ 70. The distribution of 30dz6 is given by v0:=multi_dist30(1[6]), which is a list with length 151. The required probability is the ratio of the sum of the elements of this list from number 70 onwards to the sum of all of the elements of this list. We can define this as the rational number, represented by a list with length 2, v1:={sum(tail81(v0)),v0}, and then we can output this in rational and real forms as write_ratio(v1) and rwrite_ratio(v1). The latter is 0.72089, which matches the given solution, as does the rational value.

    If instead player B goes first, this reduces player A to rolling 27 dice and we replace v0 and v1 by v2 and v3, modified accordingly – v1 uses a tail length of 63 – with real result 0.287482, again both rational and real results matching the given solution.

    We get the probability of player A winning if the start player is chosen randomly using v4:=ratio_idiv(ratio_add(v1,v3),2).

    The given solution discusses other game limits, but these have not been considered here.

63. This is a Markov chain problem. Here we construct the transition matrix by hand, so we can consider the states between the start state and two final states (lose and win in that order for convenience) as the possible point values 4, 5, 6, 8, 9 and 10. The transition matrix is then:

$$\frac{1}{36}\begin{pmatrix} 0 & 3 & 4 & 5 & 5 & 4 & 3 & 4 & 8 \\ 0 & 27 & 0 & 0 & 0 & 0 & 0 & 6 & 3 \\ 0 & 0 & 26 & 0 & 0 & 0 & 0 & 6 & 4 \\ 0 & 0 & 0 & 25 & 0 & 0 & 0 & 6 & 5 \\ 0 & 0 & 0 & 0 & 25 & 0 & 0 & 6 & 5 \\ 0 & 0 & 0 & 0 & 0 & 26 & 0 & 6 & 4 \\ 0 & 0 & 0 & 0 & 0 & 0 & 27 & 6 & 3 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 36 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 36 \end{pmatrix}$$

The analysis is as problem 57, adjusted to that we want the probability of the final state. For details see the summary section below. The probability of winning is 244/495 = 0.492929, which is as the given solution.

64. We start with the usual `-c0 n`, and then construct the transition matrix, similar to that in the previous problem, in two stages. First we construct a list `u0` that is the first row of the transition matrix, first defining the number of states `c1` as `2*c0-3`, and then defining `u0` as `ccreate_seq[c1](l==0?0:l==c1-1?c0+2:l==c1-2?4:l+2<c0?l+2:c1-l)`.

    Next we create the transition matrix, now `u1`, from `u0` as (split on two lines for clarity):

    ```
    ccreate_mat[c1](l0==0?#0:l0>=c1-2?l1==l0:l1==c1-1?#00:l1==c1-2?c0:l1==l0
    ?c0*c0-c0-#00:0)
    ```

    (We let the values for remaining in the final absorbing states be 1 rather than `c0*c0`, as the effect is the same and the code is easier. `#00` is element `l0` of `u0`.)

    Finally, we finish as the previous problem, with renumbered constants. This gets the same results as the previous problem for $n = 6$. Some examples include $n = 4$, probability 15/18 = 0.535714; $n = 10$, probability 23758489/52907400 = 0.449058; $n = 20$, probability 0.41546; $n = 50$, probability 0.397067; $n = 100$, probability 0.391497. (The last two of these require more than 128 bit integers, and the last takes about three minutes on my computer.) These solutions agree with the given solutions, except differing in the last figure for $n = 20$ as the solutions here are rounded, but the given solutions are truncated, as shown by the ellipsis.

65. These are all simple problems to solve, with probability expressions:

    (a) `same(sorted5d6)`

    (b) `get(3,groups(sorted5d6))`

    (c) `list_eq(groups(sorted5d6),{0,1,1,0,0})`

    (d) `list_eq(groups(sorted5d6),{2,0,1,0,0})`

    (e) `last(runs(sorted5d6))`

    (f) `get(3,runs(sorted5d6))`

    (The last of these can have a true value other than 1.) These have probabilities 1/1296 = 0.000771605, 25/1296 = 0.0192901, 25/648 = 0.0385802, 25/162 = 0.154321, 5/162 = 0.0308642, 10/81 = 0.123457. All are as the given solutions.

66. A direct solution that uses the randomness pool, here to be initialised by `-pool 10 6`, is `v0:=sorted5d6;do2(r1:=mode_max(v0);vloop_set0(e0==r1?r1:pool_d(6)));same(v0)`. The probability of a Yahtzee is 347897/7558272 = 0.0460286, as the given solution.

(The given solution uses a Markov chain, but we do not need to do that here – although it would run more quickly, as this solution takes two seconds to run on my computer.)

67. Ignoring the scores and just considering the numbers of dice, we can construct a Markov chain where the state is the number of dead dice. The transition matrix is:

$$\frac{1}{243}\begin{pmatrix} 32 & 80 & 80 & 40 & 10 & 1 \\ 0 & 48 & 96 & 72 & 24 & 3 \\ 0 & 0 & 72 & 108 & 54 & 9 \\ 0 & 0 & 0 & 108 & 108 & 27 \\ 0 & 0 & 0 & 0 & 162 & 81 \\ 0 & 0 & 0 & 0 & 0 & 243 \end{pmatrix}$$

If we let that matrix be represented by `u0` then we look at the first row of the matrix `u1 = mmat_absorb_visits(u0)`, whose non-final elements are, when divided by `c0 = last(u1)`, the mean number of times each state is entered, including for starting in the first state. Those numbers of rolls are thus the ratio of the list `u2 = head5(u0)` to `c0`.

We now need the mean value of a roll in each state. First we need the probability of a score in each state, which is the ratio of {32,16,8,4,2} to {243,81,27,9,3} or of {32,48,72,108,162} to 243. Then the mean of the score in each state is the number of dice, which is {5,4,3,2,1}, times the mean of each non-dead die (i.e. of 1,3,4,6) which is 7/2.

Putting all this together, the mean is 837242/52117 = 16.0647, which agrees with the given solution.

If the problem were generalised to other numbers of dice, as it is in the given solution, all of the numbers in the transition matrix and in the weighting lists could be computed by the program, rather than simply being calculated by the user as here. However, we do not do this here, but rather proceed to considering the distribution of scores. Note that the further questions posed in the solutions are not amenable to the use of Monaco.

Considering the second part of the problem the probability of specific scores, in principle, this game can be simulated in exact mode using the randomness pool up to a maximum score by truncating the game at a maximum score of interest. But this takes too long even for a maximum score of zero – which can require fifteen die rolls. (It is practical, but still quite slow, for four dice and a maximum score of zero.)

Thus instead we use approximate mode. Equal to 2 or 5 is 2 modulo 3, and hence we can use `r0:=5;until(w0:=5d6;r1:=count_eq(w0%3,2);r1?r0-=r1:r2+=w0,r0==0);r2`. The required results are from `-histogram`, and as 20 is the largest value requested we can add `-range 0 20`. The probabilities for 0, 1 and 20 are about 0.205, about 0.0212 and about 0.0200, respectively, with the last digit definitely uncertain in the last two cases. The given solutions for scores of zero and one confirm that, as the latter is closer to 0.0213. The given solution does not attempt to determine the probability of a score of 20 (or even 2).

68. We can determine this result exactly by proceeding by considering the number of unkept dice n = 1, 2, … We set the largest value of n of interest as `c0`, and here consider `-c0 5`, producing results for each n up to that value. We also add the trivial case n = 0.

We start by recording the results for n = 0, 1, 2, … in v0. We could use rational numbers, but it is easier to use integers, and v0 is a scaled version of the required mean values. As we are including n = 0, we need to set `-s0 c0+1`. We will use three other lists, v1, v2 and v3, but only need to set the length of any one, we use `-s1 c0`, as the others are then deduced.

The required scaling can be deduced from that when considering n unkept dice, we have previously rolled up to $n(n\text{-}1)/2$ dice, and thus the mean value for n unkept dice must be a multiple of $6^{n(n-1)/2}$, a value to be stored as r2. We then require the value 6^{n-1}, which is r1. Using r0 for n, we can loop through the values of these three variables by using the expression `r12:=1;rloop0(c0,incr0;`*term*$_1$`;r2*=r1*=6;`*term*$_2$`);`*term*$_3$. Note that the expression rules allow us to use `incr0` as indicated to produce the required 1-based loop.

The required result is determined from the rational or real value of v0 divided by r2; r2 is updated at the end of that loop, but v0 is updated in *term*$_2$, so that is correct. Final output can then be by letting *term*$_3$ be `lwrite(v0,r2);nline;rlwrite(v0,r2)`. This includes the obvious result that rolling 0 dice has a mean of 0.

Also straightforward is *term*$_2$. In *term*$_1$, the new value of the mean, scaled by the new value of r2, is computed as r3. Before inserting that into v0, v0 must be updated from the old value of r2 to the new value of r2, which is, as seen above, by multiplying by r1. Thus *term*$_2$ is `v0*=r1;e0:=r3`.

term$_1$ now considers all possible rolls of n = r0 dice, which is done by making *term*$_1$ be `r3:=wmask_sum10(6,w20:=replace_eq(v1+1,3,0);`*term*$_4$`)`. Note that by using w20 the unwanted elements of v2, the die rolls after replacing 3s with 0s, are all zeros. The use of `wmask_sum` is because we want a mean, and the use of a sum rather than an average scales up by 6^n, as described above. Also as noted above, this gives the wanted result of r3.

term$_4$ now must be the score for the best set of these dice to retain. We loop through all masks of length n – note that there is a default 2 omitted before the comma – by letting *term*$_4$ be `wmask_min30(,w30==r0?intmax:`*term*$_5$`))`. We use masks in which 0 is that the die is kept, 1 is that the die is re-rolled, so the total number of kept dice is `sum(w30)`, or just w30. We are not allowed to re-roll all dice, so we exclude the case `w30==r0`, which in a minimum calculation can be simply to let this be an otherwise unobtainable large value.

term$_5$ now calculates the scaled mean score for the dice roll v2, with mask v3. This is `vget0(w30)+r2*dot(!v3,w20)`, the two parts of which are for re-rolled and kept dice. For re-rolled dice this is simply a lookup in v0, which is already scaled appropriately. For kept dice the score is the indicated dot product – note that `!v3` reverses the mask, and w20 ensures that there are no unwanted terms in the summation, although this should be unnecessary – but this dot product must be scaled by the number of previously rolled dice, r2.

Putting that all together, see the summary section below, the final output is:

```
{0,3,79/18,2261/432,544369/93312,504205555/80621568}
{0,3,4.38889,5.2338,5.83386,6.25398}
```

These results agree with the given solution.

The given solution poses an additional unanswered question concerning a score of zero, but these are not considered here.

69. There is a weighted term that is designed for situations like this. Rolling until reach a sum of `c0` = *M* the term that lists the die rolls (followed by zeros) is `until_sum[c0]d6`. It does not stop if we roll a 1, but we can check the list for that condition. If we assign that term to `v0`, then our score is `all_ne(v0,1)?v0:0`. Testing this with different values of *M*, the results include:

    M = 19, mean = 245632423/30233088 = 8.12462
    M = 20, mean = 492303203/60466176 = 8.14179
    M = 21, mean = 492303203/60466176 = 8.14179
    M = 22, mean = 1474037749/181398528 = 8.12596

    The mean is thus maximised for either *M* = 20 or *M* = 21. These results match the given solution.

    The given solution poses the unanswered question as to if players playing with *M* = 2 and *M* = 64 or *M* = 65 compete, who will win most often?

    Because players can score zero, the game has an indefinite length and the problem cannot be handled directly in exact mode. It could, in principle, be handled as a Markov chain, where the transition probabilities can be computed using information derived from the approach above. However, that would require a matrix with order just over 10000, which is not practical – and would almost certainly involve calculations that exceeded the maximum possible integer size.

    Instead, we use approximate mode. We set a target of `-c0 100`, and the two players' values of *M* for our first result as `-c1 2` and `-c2 64`. We can write a function `f0` that rolls until reaching a total of `p0` or greater or a 1 is rolled, returning the score, as `until(r0+=r9:=d6,r9==1|r0>=p0);r9==1?0:r0`. We then use that to play the game. We assume that a player will stop at a score of `c0` (i.e. 100) even if he has not reached a total of *M*, and thus player 1 will take a turn that is `r1+=f0(c1?<c0-r1)` and player 2 will take a turn that is `r2+=f0(c2?<c0-r2)`. Rather than checking whether player 1 has won after his turn and whether player 2 has won after his turn, it is easiest to take both turns, check if either has won using `r1>=c0|r2>=c0` but then produce a final result as whether player 1 has won, ignoring whether player 2 would have also won, as `r1>=c0`.

    Putting that all together, as in the summary section below, and playing ten million games, for that game the probability of player 1 winning is about 0.774 and of player 2 winning is about 0.226. This might have been due to first player advantage, but swapping `c1` and `c2`, the advantage of winning with *M* = 2 only drops to about 0.762. Using *M* = 65 the probabilities are about 0.785 and 0.773. Given those choices, *M* = 2 is clearly the better approach.

70. In principle, this calculation is straightforward in Monaco. However it might require 50 die rolls, with a number of results of 6^{50}, which requires 130 bits, so we have to use a version of Monaco using extended integers. We can use the weighted term `until_sum100select{...}` to continue until we reach a total of 100. But we also need to stop if we roll a 1, and we can do this with a hack, by letting `{...}` be `{`*x*`,2,3,4,5,6}` for some *x* that is sufficiently larger than 100 – 1000 will do, and then we have succeeded if our sum is less than *x*. The probability is 0.010197, requiring only 234318 games to be run, and suitably weighted. This (and the exact fraction not repeated here) agrees with the given solution.

71. Like problem 14, this problem is one that involves intelligence, but which we can solve, again working backwards from where we are most constrained to where we are least constrained.

    To do so, we produce a solution that for every possible position in the game, i.e. set of die rolls up to this point, determines whether to roll again or stop. Those positions can be characterised by a mask, a list of length 6 – or, better practice, length c0 that is initialised to 6 – that contains 0s and 1s according to whether the corresponding value – 1 to 6 in element numbers 0 to 5 – has been rolled. Because we need to start from the situation where all numbers have been rolled, the mask – which will be v2 – which starts with all zero elements, is a mask that indicates which values have not been rolled.

    We also need to map that mask to a numerical value – which will be r0 - in the range 0, inclusive, to pow(2,c0), exclusive. For convenience we let s0 be that latter value, and s1 be 2*s0, as we will use v0 and v1 of those lengths, as will be described. Rather than convert back and forth between r0 and v2, we loop over v2 and also increment r0 by using what will be the first `-eval` expression, to do all the calculations, it being r0:=-1;lists_loop2(1,incr0;*term*$_1$), with *term*$_1$ to be determined.

    Now v0 is to be a list, one element per possible position, that is 1 if we should roll, 0 if we should not roll. v1 is to be a list, one rational number, i.e. two elements, per possible position, each being the mean score in that position when playing optimally. In any position – i.e. for any v2 and r0, the latter the current index into v0 and, per rational number rather than per element, into v1 – the new values of the optimal choice and mean value can be calculated using only already calculated values, This is possible because of the guaranteed order in which v2 is looped over.

    As *term*$_1$, we first calculate, as two element lists representing rational numbers, v3 and v4 that are the mean scores if we continue and stop, respectively. Those will be using *term*$_2$ and *term*$_3$. We then update v0 by e0:=ratio_gt(v3,v4) and update v1 by [ratio_vset1(r0,e0?v3:v4)] – using [] as ratio_vset is a list function. Thus *term*$_1$ is v3:=*term*$_2$;v4:=*term*$_3$;e0:=ratio_gt(v3,v4);[ratio_vset1(r0,e0?v3:v4)] using two further terms to be determined, *term*$_2$ and *term*$_3$.

    The easier of these two required terms is *term*$_3$, this is simply the sum of the rolled dice values, converted to a rational list. For convenience we record the dice faces as u1, given by sequence[c0]+1, and then *term*$_3$ is {dot(!v2,u1),1}, the logical not operator ! being needed as v2 indicates which faces have not yet been rolled.

    For *term*$_2$ we loop over the possible next dice rolls, as an index r1 (the die roll is r1+1, but we do not need that), accumulating either *term*$_4$ if the roll is new, or zero (as a rational list) if not, then dividing by c0 to take an average. The loop and division can be combined using the function ratio_rmean. The die roll is new if element r1 of v2 is 1, thus, using that e21 can be used as a list as shown, *term*$_2$ is ratio_rmean1(c0,e21?*term*$_4$:rzero).

    We complete *term*$_2$ with that *term*$_4$ is ratio_vget1(r0-i01), using the list u0 defined as geometric[c0](1,2). This depends on that we know that the loop lists_loop2 that varies v2 does so with element zero changing most rapidly, and so on, thus the change in the index r0 according to whether element r1 is changed is 2^{r1}, i.e. element r1 of u0 or i01. Because we have limited to the case that this is a not previously rolled value, the index if that value is now rolled is r0-i01, and the mean value is extracted from v1.

This completes the evaluation of v0 and v1; the combined `-eval` expression is in the summary section below. We now need to use them to report our required results. From v1 we only need the final mean value when starting with no dice rolled, which is the rational list v5:=tail2(v1). We can output that as a rational number and a real number using write_ratio(v5) and rwrite_ratio(v5), giving the mean score playing to optimise that of 223/36 = 6.19444. This agrees with the given solution. However, we also need to know the optimising strategy.

The approach used here – a compromise between ease of producing the output and ease of reading it – produces 64 lines of output. Each line contains a list and a number. The list contains the rolled numbers, or zero if the number is unrolled – so, for example {1,2,3,0,0,0} means that 1, 2 and 3 have been rolled, but 4, 5, and 6 have not. The number is 0 if we should stop, 1 if we should continue. Some example (non-consecutive) lines are:

```
{1,2,3,4,5,6} 0
{0,0,0,0,0,6} 1
{1,2,3,0,0,0} 0
{0,0,0,0,0,0} 1
```

This output – all 64 lines – can be produced using:

```
r0:=-1;lists_loop2(1,incr0;lwrite(v2?0:u1);space;write(e0);nline)
```

The given solution has a description of a rule to use, and by chance, the solution here matches it exactly. Why by chance? Because there are situations where rolling or not has equal mean, one such being having rolled 2 and 5. The given solution formulation says do not roll again, and so does the above use of the program. But if we change the ratio_gt it includes to ratio_ge then the overall mean is unchanged, but the advice has flipped to roll again. That the two match is not chance – the given solution uses the corresponding inequality, but we did not know that when constructing our solution (although not rolling when unnecessary is probably the most obvious approach).

The given solution then asks about the probability of a score of zero. It would be possible to adapt this solution to compute that, but this has not been done here.

72. This is an instance of what can be considered as a generalisation of the previous problem. In the previous problem the dice face values were in u1, with length c0. Instead now we use c0 for the number of faces per die, and c1 as the number of dice. We first introduce a new list u2 – which was previously implicitly all 1s – as the distribution of the roll values from c1; u2 can be created as multi_dist[c1](1[c1]), and now s0 is pow(2,k2), u0 is geometric[k2](1,2) and u1 is sequence[k2]+c1.

    In calculating v3, we now cannot use ratio_rmean1 because rather than the c0 values of its second arguments being equally weighted, the k2 values of its second argument are weighted by the elements of u2. Thus, we replace the function ratio_rmean1 by the function ratio_rsum1 with three changes. First, the first argument is now k2 not c0. Second, the second argument is multiplied by i21 using the function ratio_imult, which can be limited to the case that the second argument is non-zero. Third, the result of ratio_rsum1 is divided by sum(u2), or just u2, using the function ratio_idiv.

Putting that all together gives the first `-eval` expression as in the summary section below.

Using `-c0 6` and `-c1 1` replicates the previous problem solution. Using `-c0 6` and `-c1 2` gives a mean score of 513389/34992 = 14.6716, as the given solution.

To compare the deduced maximising strategy with the given strategy, it is easiest to only output cases where the dice should be rerolled, and report the number of values rolled so far, replacing the final `-eval` expression by:

```
r0:=-1;lists_loop2(1,incr0;e0&(lwrite(v2?0:u1);space;write(count(!v2));nline))
```

This reduces the number of cases to be checked to 148. It can be further reduced by noting that it contains one case with no values yet rolled, all 11 cases with one value rolled and all 55 cases with two values rolled. These can be excluded from the output using (split on two lines for convenience):

```
r0:=-1;lists_loop2(1,incr0;
e0&((r1:=count(!v2))>2&(lwrite(v2?0:u1);space;write(r1);nline)))
```

This reduces the number of cases to be compared to 81. It can be split into 68 cases with three values rolled and 13 cases with four values rolled. Some manual work (which in the former case can be usefully combined with modifying the above to report the information differently) shows that these match the given solution, except that this solution suggests stopping after three values seen when those are {2,8,10}, {4,5,10} or {4,6,12}. Similarly to the previous problem, if we replace `ratio_gt` by `ratio_ge` then these three cases – and only those three cases – change from stopping to continuing, with no change to the mean.

73. Although apparently similar to problem 71, this is significantly different. A state consists of the last die rolled and the current total, and the latter is potentially open-ended, which is not good for exact mode. We need to cap that, either theoretically or by setting a limit where dice are no longer rolled, as described below. In either case we fix c1 as a value that if our score is equal to or greater than that value, we stop – we use c0 as the dice size, i.e. 6.

    So for a last die roll of *m*+1 and a current score of *n+1*, we use element *m*+c0**n* of v0 as whether we roll again (1) or stop (0), and the same number as the index into the list v1 of mean scores. Rather than including a score of 0, with no previous die roll, our final mean result will be to average over the elements of v1 that represent the position after one roll.

    We need to construct v0 and v1 backwards. First, for *n* = c0 to c0+c1-1 we need to set the state to always stop, with mean value equal to *n*. The former we get "for free" from the initialisation of v0. Second, we need to loop from *n* = c0-1 to *n* = 1. For each such value of *n* we need to consider each value of *m*. We only have loops of the form `rloop$(`*k*`,...)` from 0 to *k*-1, assuming *k* > 0, but we can convert this to a loop by replacing r$ – or using a different variable – before the ... We need to define the length of v0 by `-s0 c0*(c1-1+c0)`. Rather than using `-s1`, we need to initialise v1 with valid zero rational numbers, so we use `-v1 rzero[2*s0]`.

We thus have three stages, the two stages indicated above and one to produce the required output. It is thus convenient to use three `-eval` options, although as usual they could be combined. In addition it is useful to define the indexing function `f0` such that the required index into v0 or v1 for a roll of p0 giving a total of p1 is given using `-f0 (p0-1)+c0*(p1-1)`.

The first `-eval` option has the expression:

```
rloop0(c0,r0+=c1;rloop1(c0,incr1;r2:=f0(r1,r0);[ratio_vset1(r2,ratio(r0))]))
```

Breaking that down, `rloop0` together with `r0+=c1` loops `r0` from `c1` to `c1+c0-1`, as required, and `rloop1` together with `incr1` loops `r1` from 1 to `c0`. Note that these are actual scores and dice rolls, not indexed from zero. The required index is `r2`, determined using `f0`. Setting v1 uses the list function `ratio_vset1`, hence the need for the `[]`, and `ratio(r0)` converts the score to a rational number, as required.

The second `-eval` option has a similar structure:

```
rloop0(c1-1,r0:=c1-1-r0;rloop1(c0,incr1;r2:=f0(r1,r0);term))
```

After its modification, `r0` is the required current score from `c1-1` down to 1, `r1` is as previously. Note that we compute some impossible cases for `r0` < `r1`, but this is easier than avoiding them, they are not used.

term is divided into three parts. First, v2, the mean score if rolling again, is computed. Second, v3, the mean score if not rolling again is the obvious `v3:=ratio(r0)`. Third, we select the greater of v2 and v3 as the new value to put in v1, and if we choose v2 we update v0. (The default zeros in v0 cover the case when we choose v3.)

v2 is calculated by (split here on two lines to avoid a reduced font size):

```
v2:=ratio_rmean3(c0,[incr3];r4:=f0(r3,r0+r3);
[r3!=r1]?ratio_vget1(r4):rzero)
```

Breaking that down, the `ratio_rmean3` loops over the possible rolls once `r3` is incremented – we need the `[]` around `incr3` as the second argument of `ratio_rmean3` is a list (a rational number). `r4` is now the score if that roll is not a failure – we could move it into the second operand of the `?:` operator for a small efficiency gain, but we do not need that and this is a simpler direct equivalent of the previous case. We fail if `r3`, our new roll, equals `r1`, the previous roll, with a score of zero, given by `rzero`. If we succeed then we use the previously calculated rational mean value in v1.

We now select between v2 and v3 using `ratio_ge`. We could equally well used `ratio_gt` – see the discussion for problem 71 – but it turns out that using `ratio_ge` more closely corresponds to the given solution, although the mean values are the same. Denoting the rational mean value by *list*, we use *list* as `ratio_vset1(r2,`*list*`)`, and we compute *list* – and set an element of v0 if required – using `[ratio_ge(v2,v3)]?e02:=1;v2:v3)]`.

There remains only the third `-eval` expression, to output the required results. The simplest part to output – but requiring the most explanation of its meaning, see below – is v0, that records the strategy. It can be output, using `head` to exclude the results we initialised v0 with, by using `mat_write[c0](head[c0*(c1-1)](v0));blank`, the final

blank giving a blank line between the two pieces of output. The final result has to be computed as the average mean score after one roll, for which the roll and the score are equal (to r0 as used below). We can assemble those mean scores into a list v4 by using rloop0(c0,incr0;r1:=f0(r0,r0);[ratio_vset4(r0-1,ratio_vget1(r1))]), all of the pieces of which are as already used. We then can form the average as the rational number v5 using v5:=ratio_lmean(v4), and finally output the overall mean as rational and real values using write_ratio(v5);space(2);rwrite_ratio(v5).

For standard dice we use -c0 6 and here we use -c1 22. Theory – in the given solution – is that 21 is sufficient, but we here need to confirm from the results that we do not re-roll for a score of 21, regardless of the die roll that put us there, so we use 22 to confirm that, which it does as shown below.

Considering those output in reverse order, the mean score is (a not very informative) 678747596227/78364164096, which equals 8.66145. This agrees with the given solution, although that is quoted only as about 8.7. The other output – with line breaks added here for clarity – is:

```
{{1,1,1,1,1,1},{1,1,1,1,1,1},{1,1,1,1,1,1},{1,1,1,1,1,1},
{1,1,1,1,1,1},{1,1,1,1,1,1},{1,1,1,1,1,1},{1,1,1,1,1,1},
{1,1,1,1,1,1},{1,1,1,1,1,1},{1,1,1,1,1,1},{1,1,1,1,1,1},
{1,1,1,1,1,1},{1,1,1,1,1,1},{1,1,1,1,1,1},{1,1,1,1,1,0},
{1,1,1,1,0,0},{1,1,1,0,0,0},{1,1,0,0,0,0},{1,0,0,0,0,0},
{0,0,0,0,0,0}}
```

The six numbers in each sub-list are the strategy when the last roll was 1 to 6, 0 for stop, 1 for continue; the sub-lists are for totals of 1 to 21 – that the last one is all zeros means that we set a sufficiently large value of c1, because if we never re-roll for a total of *n* (regardless of the last roll) then we also never re-roll for any total *m* > *n*. The strategy is exactly as the given solution, although it lacks both a simple rule (although that could be easily created from that result) and a theoretical explanation.

Again, this could be adapted to compute the probability of zero, or with more work, any other value. We have already seen that exact values as rational numbers are fairly useless, and that the given solution is quoted to a low precision (for reasons that are unclear). Thus, we could get a distribution, and any other required statistics, sufficiently accurately using approximate mode. Using the given solution's characterisation of the optimum strategy, this is a straightforward use of -c0 6 -c1 c0*(c0+1)/2 and the expression until(r2:=r1;r0+=r1:=d[c0],r1==r2|r01>c1);r1==r2?0:r0. But we can do better than that. Switch back to exact mode, add a use of the randomness pool using -pool c1 c0 and replace d[c0] by pool_d(c0) and this runs (on my computer) about as fast as only a hundred thousand results in approximate mode.

So breaking the usual pattern here of not giving full output, the -table output in this exact case is:

```
 N      P(N)         P(<=N)      P(>=N)
 0   0.557061      0.557061   1
16   0.0257565     0.582818   0.442939
17   0.0503962     0.633214   0.417182
18   0.071095      0.704309   0.366786
19   0.0819984     0.786308   0.295691
```

```
20  0.0759857    0.862293  0.213692
21  0.0525672    0.914861  0.137707
22  0.0451863    0.960047  0.0851394
23  0.0252684    0.985315  0.0399531
24  0.0113625    0.996678  0.0146847
25  0.0030414    0.999719  0.00332228
26  0.000280884  1         0.000280884
```

The probability of a score of zero agrees with the given solution of about 56%.

74. The easiest way to solve this problem for 6-sided dice is simply to implement it using the randomness pool. We let `c0` be the number of dice faces, initially 6, and the game can be played by the expression `while(r1<c0&(r2:=pool_d(c0))>r1,r1:=r2;r0+=r1)`. Using the test `r1<c0`, which otherwise is unnecessary, we avoid needing an extra die in the pool, hence requiring at most `c0` rolls of the `c0`-sided die, and hence the pool can be set by `-pool c0 c0`. Also, this is the correct behaviour because the problem description indicates that once 6 is reached you do not roll again (and this is relevant to the next part of the problem). The mean is 6, which agrees with the given solution.

    For the number of dice rolled, we can just add `incr3` before the final `)` and the mean number of rolls is 70993/46656 = 1.52163, which agrees with the given solution.

    As usual, we cannot determine the asymptotic result using Monaco, we can only increase `c0` = *n*, and see what happens. For *n* = 10 the mean number of dice rolled is 1.59374; for *n* = 15 it is 1.63288; for *n* = 20 it is too large for 64 bit integers, but using 128 bit integers it is 1.6533, and is now using nearly ten million evaluations. That number, and hence the time taken, is the limiting factor rather than integer size, as based only on integer size 128 bits would be sufficient for *n* = 25. However, the case *n* = 25 takes four hundred million evaluations, and on my computer takes about five minutes, to give a result of 1.66584. It is clear that, say *n* = 30 is not practical this way, and it is also still not clear (especially if you know the answer) what the asymptotic result is. A different approach is needed.

    If we are only considering the number of rolls, not the scores, this problem becomes a simple Markov chain with *n*+2 states – a start state and states 1 to *n* for last roll *n*, and a final state where you fail. That still gets slower with increasing *n*, but not as rapidly. The analysis is, with one minor change noted below, as problem 1 and later. Because each state is directly linked to the score transitioning to it, and the output from the function `mmat_absorb_visits` contains the required information about mean numbers of state visits, it would be possible to modify this solution to also determine the mean score, but that is not needed here.

    We can start with *n* = 6 and the transition matrix:

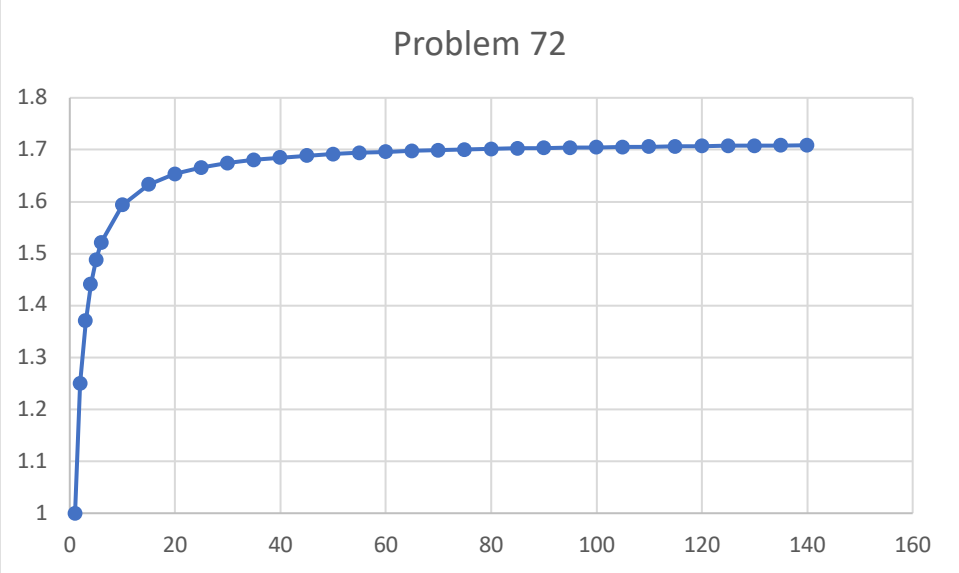
$$\frac{1}{6}\begin{pmatrix} 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 2 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 3 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 4 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 5 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 6 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 6 \end{pmatrix}$$

We initialise u0 from that matrix as usual, and then we want the mean number of transitions (die rolls) from the start state until reaching the absorbing state, except that we do not count the roll that puts us into the absorbing state (a roll which we do not actually make from the state where we have just rolled 6, as described above) so we proceed as problem 1, except subtracting 1 from the result, which we can do by swapping the initialisation of c0 and c1, and subtracting c1 when initialising c0. Note that we no longer need c0 for the number of dice faces once we have set u0. The result is as expected.

Now we can proceed for more die faces by writing an expression to create u0 given the number of faces c0., which we again now do need. A suitable expression to create u0 is `ccreate_mat[c0+2](l0>=c0?l1>c0?c0:0:l1<=l0?0:l1<=c0?1:l1)`. And then using u0 as before – though we now need c1 and c2 rather than c0 and c1.

Repeating the examples above – switching to 128 bit integers at the same point – the results are the same. But our next result for $n = 30$ is too large for 128 bit integers, so switching to the largest available integers, 1024 bits, our results are $n = 30$: 1.67432; $n = 35$: 1.68044; $n = 40$: 1.68506; $n = 45$: 1.68868, $n = 50$: 1.69159; $n = 55$: 1.69398; $n = 60$: 1.69597; $n = 65$: 1.69766; $n = 70$: 1.69912; $n = 75$: 1.70038; $n = 80$: 1.70148; $n = 85$: 1.70246; $n = 90$: 1.70333; $n = 95$: 1.70411; $n = 100$: 1.70481; $n = 105$: 1.70545; $n = 110$: 1.70603; $n = 115$: 1.70656; $n = 120$: 1.70704; $n = 125$: 1.70749; $n = 130$: 1.7079; $n = 135$: 1.70828; $n = 140$: 1.70864, and that is as far as we can get (at least considering multiples of 5).

We can plot these results using Excel – with values from $n = 1$ to $n = 5$ added to give a more complete plot – as:



It is plausible that the graph is heading for an asymptotic value, and if so probably of about 1.71, but this is not certain.

The given solution actually has a formula for this value, $\left(\frac{n+1}{n}\right)^n - 1$, and this can be added for comparison to the Excel spreadsheet to show a match for all values of n considered. (That then has an asymptotic value of e-1, or about 1.718, so in fact closer to 1.72 than the guessed 1.71 above, convergence is quite slow.)

75. This problem is similar to those from problem 71, since as we do not lose already rolled scores, we can ignore them when determining the best way to play from this point. Thus a state to determine the value of is just the mask of which values have not yet been rolled (so that, working backwards as usual, we start from a mask of all zeros indicating that all values have been rolled). However, a roll can leave the process in the same state, which is a new situation that we must handle.

    To do this, first we set s0 and v1 as problem 73, and u0 and u1 as problem 71. We set s2 as problem 71, and as that problem, we use a `-eval` expression that has the form `r0:=-1;lists_loop2(1,incr0;`*term*$_1$`)`. v2 is now a mask of the unrolled values.

    term$_1$ will have four parts. First, we note that if we roll a die, the added score has two parts: the contribution from the die roll itself, which will be v3, and the contribution from future die rolls, which will be v4. Then we combine those using `v5:=ratio_add(v3,v4)`, and then we use v5 to update the strategy.

    v3 is easily computed by `v3:=ratio(sum(v2?u1:-u1),c0)`. Skipping for the moment over the calculation of v4, we update the strategy using `e0:=ratio_gt(v5,rzero)` and if that is true – we can combine using & - use `[ratio_vset1(r0,v5)]`, the `[]` being needed as `ratio_vset` is a list function. We now have the v0 that defines the strategy, but before considering how to use it, which will be in a separate expression, we return to setting v4.

    In setting v4 we – usually, there is a special case to be described – loop over the possible die rolls using the loop `ratio_rsum1(c0,...)`. To find that, and normalise it, we note that computing the mean value across possible die rolls, where *k* of the die rolls are new, the current mean value is a sum of *k* mean values from already calculated mean values, each at the usual offset `r0-i01`, see problem 71, plus `c0-`*k* values returning to the same state. Rearranging that, we find that the current mean value is the sum of the mean values in the previously calculated states, only, divided by *k* rather than c0. *k* is simply determined from the mask v2 as `sum(v2)` or the integer v2. We thus have a v4 that is usually `ratio_idiv(ratio_rsum1(c0,[e21]?ratio_vget1(r0-i01):rzero),v2)`, which we denote as *term*$_2$. However, that term cannot be computed when v2 is zero. But that is the state in which all die values have occurred, so we must stop rolling, with a value of zero. v4 is thus set by `v4:=[v2]?`*term*$_2$`:rzero`.

    We put all of this together as shown in the summary section below.

    So now we can output the strategy using a separate option `-eval`. This can be as problem 71, except that we only need the strategy part of the output. It is convenient to augment it with output of `dot(!v2,u1)`, the total of the unrolled values, for each state, as that allows a fast comparison with the given solution, which this agrees with (i.e. that we re-roll if this value is less than 11).

    We now note that the given solution comments on that experiment shows that implementing this strategy has a mean score of about 8.7. We can verify that, and slightly improve on it. We have to do this in approximate mode, because the game is of unbounded length – for example however many 1s we start with, we keep rolling. We could use the given solution characterisation of the strategy, but here we use the computed strategy in v0.

We discard the second `-eval` option (the output) and add a main expression, which can be (split on two lines here to avoid a smaller font):

```
s9:=c0;until(r9:=dz[c0];r1:=r9+1;r8+=(e9?-r1:r1);e9:=1,
!get(binary(!v9),v0));r8
```

The repeated use of `binary` is inefficient, we could do better, but this was easy to write and fast enough. Running for a hundred million evaluations, the mean is about 8.73 (the 95% confidence interval is `[8.72922, 8.73575]`).

76. This is a problem of a similar form to the last few problems, but actually simpler. Letting `c1` be the maximum score (`c0` is 6 for dice size) we only need `s0` = `c1+1` states 0 to `c1`, the current score. Again we use `v0` for strategy, a simple 0 for don't roll, 1 for roll, and `v1`, a list of `s0` rational values, for mean scores, initialised as `rzero[s0]`, although initial values are actually not needed.

    The first `-eval` option, which calculates `v0` and `v1`, loops `r0` from 0 to `c1`, but we need to start from a score of `c1`, so let `r1:=c1-r0`. Then we average over the possible `c0` die rolls using `ratio_mean2`, needing to increase `r2` by 1, which we do using `[incr2]` to make this a list term (of unknown length, deduced as one by late list length deduction, then optimised away). This gives us the mean if we roll the die of `v2`, and the expression `rloop0(s0,r1:=c1-r0;v2:=ratio_mean2(c0,[incr2];`*term*$_1$`);`*term*$_2$`)`.

    term$_1$, the score, is straightforward. If the die is rolled then the new total is `r1+r2`, or the integer `r12`. Taking account of that if this total is above `c1` then we score zero, *term*$_1$ is `[r12>c1]?rzero:ratio_vget1(r12))`.

    term$_2$ now sets `v3`, the result of not rolling the die, and then compares `v2` and `v3` and updates `v0` and `v1`. Setting `v3` is straightforward, so *term*$_2$ is `v3:=ratio(r1);`*term*$_3$. *term*$_3$ is now composed of two parts. The first part is `v4:=[ratio_gt(v2,v3)]?e01:=1;v2:v3`, which compares the two possible mean scores, and sets `v4` to the larger, also setting the appropriate element of `v0`. (Note that there is no need to set it to zero, that is its initial value.) The second part, `[ratio_vset1(r1,v4)]`, sets the appropriate element of `v1`.

    For output, this also is an expression in two parts, in a second `-eval` option as usual. The first part loops through `v0` and reports the maximum mean strategy using `rloop0(s0,write(r0);space(2);write(e0);nline)`. The second part reports the mean value, starting from a total of zero, i.e. before any dice were rolled, as rational and real values using `v4:=head2(v1);write_ratio(v4);space(2);rwrite_ratio(v4)`. The two can be separated by a blank line using `nline`.

    The mean result with `c1` equal to 10 is 40817/5832 = 6.9988, which agrees with the given solution, as does the strategy to get it, of re-rolling with a score of 5 or less.

    The problem then asks about other values of *k*, i.e. `c1`. Results from the program are *k* = 1: 1/6 = 0.166667; *k* = 2: 1/2 = 0.5; *k* = 3: 1; *k* = 4: 7/4 = 1.75; *k* = 5: 49/18 = 2.72222; *k* = 6: 49/12 = 4.08333; *k* = 7: 3017/648 = 4.65586; *k* = 8: 20629/3888 = 5.30581; *k* = 9: 46991/7776 = 6.04308; *k* = 10: 40817/5832 = 6.9988; *k* = 20: 16.66; *k* = 50: 46.6667; *k* = 100: 96.6667. (The last two require larger integer sizes.) The first value at which the player should "stick" – *s* in the given solution – in those cases are 1, 1, 1, 2, 2, 3, 4, 5, 5, 6, 15, 45, 95, which again match the given solutions.

77. We start by parameterising the problem with `-c0 6` (number of dice faces), `-c1 10` (modulo check for failure) and with c2, a multiple of c1, that defines the maximum value we analyse up to. First we need to determine c2. While it is possible to determine a c2 that definitely works, it is easiest to just increase c2 until we always stop and results do not change.

    We will set v0 to the rational mean scores when starting from a score of 0 to c2-1. In the first of three `-eval` options we set v0 to contain a minimum value for each current score *n*, following the strategy of never re-rolling. This minimum value is *n*, unless *n* is a multiple of c1, when it is zero. After setting s0 to 2*c2, we thus set this initial v0 using the expression `rloop0(c2,[ratio_vset0(r0,r0%c1?ratio(r0):rzero)])`.

    In the second `-eval` option we calculate v0. To do so we loop improving v0 until no further improvements are made. We set r2 if an improvement is made, so the expression used is `r2:=1;while(r2,r2:=0;`*term*$_1$`)`, where r2 might be set in *term*$_1$.

    Next we will only update elements of v0 from 0 to c2-c1-1. This means that values in c0 for c2-c1 and above might not be accurate, and thus the final result must not depend on them. This will be so if our results indicate that results from c2-c1-c0 to c2-c1-1 are reliable but are never re-rolled. This will happen for any c2 for which results are stable. Only updating the indicated elements – and not those that are a multiple of c1, other than zero – means that *term*$_1$ has the form `rloop0(c2-c0,(r0==0|r0%c1)&(`*term*$_2$`))`.

    term$_2$ then starts by setting v1 to a worst case value if re-rolling, based on current incomplete knowledge. This is `v1:=ratio_rmean1(c0,ratio_vget0(r0+r1+1))`. We then check if this is better than the not re-rolling score r0, which concludes *term*$_2$ with `ratio_gt(v1,ratio_vget0(r0))&(`*term*$_3$`)`, where *term*$_3$ sets r2, as noted above, and updates the result in v0 for a score of r0, so *term*$_3$ is `r2:=1;[ratio_vset0(r0,v1)]`.

    We then proceed to results output, using the third `-eval` option, up to c2-c1-1, so the required expression has the form `rloop0(c2-c1,(r0==0|r0%c1)&(`*term*$_4$`;nline)`, the `nline` putting the output for each value of r0 on a separate line. We produce five outputs: `write(r0)`, `rwrite_ratio(v1)`, `write_ratio(v2)`, `rwrite_ratio(v2)` and `write(ratio_gt(v1,ratio(r0)))`, with v1 defined as in the second `-eval` expression and v2 defined as `ratio_vget0(r0)`. These five outputs are the score, the mean score if re-rolling, two versions of the maximum mean score, and whether to re-roll (1 = yes, 0 = no).

    Increasing c2 by c1 until the mean score starting from zero does not change, this happens between c2 = 40 and c2 = 50. There are, as there must be, six (i.e. c0) consecutive do not roll results, from 34 to 39.

    We are interested in two things: when to re-roll and the mean score from the start state. It turns out that we need greater than 128 bit integers, here we use 1024 bit integers. The former result is then to re-roll all scores up to 33, except 24 and 25. The latter result is 13.2171, or to be exact 162331123011053862884431/12281884428929630994432. This agrees with the given solution.

    We do not here reproduce all results in the given solution, just three values. For c1 = 6, results do not change between c2 = 24 and c2 = 30, and are to re-roll for all scores up to 17, only, with mean score 7.22145. For c1 = 15, results do not change between c2 = 60 and c2 = 75, and are to re-roll for all scores up to 53, except 39 to 41, with mean score

20.9709. For c1 = 20, results do not change between c2 = 100 and c2 = 120, and are to re-roll for all scores up to 93, except 54 to 56 and 74 to 78, with mean score 28.7591. These all agree with the given solution.

78. The two digit problem is straightforwardly solved. It is useful to define s0 as 6, the die size, and c0 as 10, the base used. The sum of the die face values is c1, equal to s0*(s0+1)/2. For each die face r1 – which is r0+1, with r0 looping from 0 to s0-1 – we compare the mean result putting that digit into the c1 position, and into the 1 position. However, rather than comparing mean values it is easier to compare total values, and those are s0*c0*r1+c1 and s0*r1+c0*c1. We put this together as the `-eval` expression `rloop0(s0,r1:=r0+1;e0:=s0*c0*r1+c1>s0*r1+c0*c1)`, and then we report the list v0 as {0,0,0,1,1,1}, indicating that digits 1 to 3 should be put in the ones position, digits 4 to 6 should be put in the tens position. Not requested, but provided in the given solutions, the mean value can be determined by changing the value of e0 to `max{s0*c0*r1+c1,s0*r1+c0*c1}` and reporting `rwrite_ratio{v0,s0*s0}`. Note that this uses s0*s0 because the use of v0, meaning sum(v0), introduces another factor of s0. (This can be made more efficient by using `rsum0` rather than `rloop0` and eliminating v0, but that is not worthwhile here.) That mean value is 45.25, as in the given solution. For explanatory purposes, call that value *y*.

    The three digit problem has three possible starting points, the first die roll, value r1, is put in the ones, tens and hundreds position. In the ones position the mean value is r1+10*y*. In the hundreds position the mean value is 100*r1+*y*. In the tens position we need to re-run the original problem with c0 equal to 100. The strategy is the same, putting 4 to 6 in the hundreds position, 1 to 3 in the ones position, and the mean is 427.75. For explanatory purposes, call that value *z*. The mean value if r1 is put in the tens position is 10*r1+*z*.

    Taking into account that *y* and *z* are integer multiples 181 and 1711 of ¼, we can set c8 and c9 to those integers, and find the strategy v0 that is the power of ten to assign the digit to by setting e0 to `find_max{4*r1+10*c8,40*r1+c9,400*r1+c8}`. The strategy is {0,0,1,1,2,2}, meaning that 1 or 2 is put in the ones position, 3 or 4 is put in the tens position, 5 or 6 is put in the hundreds position. The mean value can be found by replacing `find_max` by `max` and reporting `rwrite_ratio(v0,4*s0)`, and is 504. This agrees with the given solution.

Characterisation of Solutions

The problems, and their solutions in this document, can be characterised as follows. Note that this only considers the problems as posed, not supplementary problems posed by the given solutions, whether answered or not. Some categories – exact solutions are assumed unless indicated otherwise – are:

- Completely solved, or solved as well as the given solution: 1-4, 10-11, 14-16, 18-22, 25-26, 28, 31, 35-37, 40-43, 49, 52-53, 57-59, 61-63, 65-73, 75, 78 (45 problems).
- Solved for some specific parameters, but not a general solution or any asymptotic or other limiting case solution; in some cases cannot produce all of the given solutions: 5-9, 12-13, 17, 23-24, 27, 29-30, 32-33, 38-39, 46, 54-56, 60, 64, 74, 76-77 (26 problems).
- Solved, but part of the solution is approximate: 34, 45 (2 problems).
- Solved, but the solution is approximate: 44 (1 problem).
- Solution is not possible, although an illustration of at least part of the solution or an example is possible: 47-48, 50 (3 problems).
- Solution is not possible and no illustration of the solution is possible: 51 (1 problem).

Summary of Runs

This is output from the option `+parameters`, but omitting the initial `Parameters:` line.

1. `-u0 {{5,1},{0,6}} -u1 mmat_absorb_visits(u0) -c0 sum(dmat_row(0,u1)) -c1 last(u1) -eval write_ratio(c01);space(2);rwrite_ratio(c01)`

2. `-u0 {{5,1,0},{5,0,1},{0,0,6}} -u1 mmat_absorb_visits(u0) -c0 sum(dmat_row(0,u1)) -c1 last(u1) -eval write_ratio(c01);space(2);rwrite_ratio(c01)`

3. `-u0 {{5,1,0},{4,1,1},{0,0,6}} -u1 mmat_absorb_visits(u0) -c0 sum(dmat_row(0,u1)) -c1 last(u1) -eval write_ratio(c01);space(2);rwrite_ratio(c01)`

4. `-u0 {{0,1,1,1,1,1,1,0},{0,1,0,1,1,1,1,1},{0,0,1,0,1,1,1,2},{0,1,0,1,0,1,1,2},{0,1,1,0,1,0,1,2},{0,1,1,1,0,1,0,2},{0,1,1,1,1,0,1,1},{0,0,0,0,0,0,0,6}} -u1 mmat_absorb_visits(u0) -c0 sum(dmat_row(0,u1)) -c1 last(u1) -eval write_ratio(c01);space(2);rwrite_ratio(c01)`

   `-u0 {{0,1,1,1,1,1,1,0},{0,0,0,1,1,1,1,2},{0,0,0,0,1,1,1,3},{0,1,0,0,0,1,1,3},{0,1,1,0,0,0,1,3},{0,1,1,1,0,0,0,3},{0,1,1,1,1,0,0,2},{0,0,0,0,0,0,0,6}} -u1 mmat_absorb_visits(u0) -c0 sum(dmat_row(0,u1)) -c1 last(u1) -eval write_ratio(c01);space(2);rwrite_ratio(c01)`

   `-u0 {{0,2,2,2,0},{0,2,1,2,1},{0,1,2,1,2},{0,2,1,1,2},{0,0,0,0,6}} -u1 mmat_absorb_visits(u0) -c0 sum(dmat_row(0,u1)) -c1 last(u1) -eval write_ratio(c01);space(2);rwrite_ratio(c01)`

   `-u0 {{0,2,2,2,0},{0,1,1,2,2},{0,1,1,1,3},{0,2,1,0,3},{0,0,0,0,6}} -u1 mmat_absorb_visits(u0) -c0 sum(dmat_row(0,u1)) -c1 last(u1) -eval write_ratio(c01);space(2);rwrite_ratio(c01)`

5. `-exact -probability -statistics -c0 6 count_diff(sorted[c0]d6)==6`

   And similarly for other values of c0.

6. `-u0 {{5,1,0,0,0,0,0},{4,1,1,0,0,0,0},{4,1,0,1,0,0,0},{4,1,0,0,1,0,0},{4,1,0,0,0,1,0},{4,1,0,0,0,0,1},{0,0,0,0,0,0,6}} -u1 mmat_absorb_visits(u0) -c0 sum(dmat_row(0,u1)) -c1 last(u1) -eval write_ratio(c01);space(2);rwrite_ratio(c01) -v0 dmat_identity7 -noreset -nowarnings v0:=mmat_mult(u0,v0);write(number+1);space(2);rwrite([dmat_get(0,6,v0)]/[first(dmat_rsum(v0))]) 24`

   And similarly for other integer sizes and numbers of evaluations.

   `-u0 {{5,1,0,0,0,0,0},{4,1,1,0,0,0,0},{4,1,0,1,0,0,0},{4,1,0,0,1,0,0},{4,1,0,0,0,1,0},{4,1,0,0,0,0,1},{0,0,0,0,0,0,6}} -u1 mmat_absorb_visits(u0) -c0 sum(dmat_row(0,u1)) -c1 last(u1) -eval write_ratio(c01);space(2);rwrite_ratio(c01) -c2 5000 -probability -statistics do[c2](r0:=distribute(dmat_row(r0,u0)))==6 1000000`

   `-probability -statistics -c2 5000 until(r2:=d6;r1:=r2==r1+1?r1+1:r2==1,r1==6|incr0==c2);r1==6 1000000`

7.
```
-u0
{{0,6,0,0,0,0,0},{0,1,5,0,0,0,0},{0,1,1,4,0,0,0},{0,1,1,1,3,0,0},{0,
1,1,1,1,2,0},{0,1,1,1,1,1,1},{0,0,0,0,0,0,6}} -u1
mmat_absorb_visits(u0) -c0 sum(dmat_row(0,u1)) -c1 last(u1) -eval
write_ratio(c01);space(2);rwrite_ratio(c01) -v0 dmat_identity7
-noreset -nowarnings
v0:=mmat_mult(u0,v0);write(number+1);space(2);rwrite([dmat_get(0,6,
v0)]/[first(dmat_rsum(v0))]) 24
```

   And similarly for other integer sizes and numbers of evaluations.

8.
```
-exact -probability -statistics -c0 1 -c1 1
overlap(sorted[c0]d6,sorted[c1]d6)
```

   And similarly for other values of c0 and c1.

9.
```
-u0
{{0,6,0,0,0,0,0},{0,1,5,0,0,0,0},{0,0,2,4,0,0,0},{0,0,0,3,3,0,0},{0,
0,0,0,4,2,0},{0,0,0,0,0,5,1},{0,0,0,0,0,0,6}} -u1
mmat_absorb_visits(u0) -c0 sum(dmat_row(0,u1)) -c1 last(u1) -eval
write_ratio(c01);space;rwrite_ratio(c01)
```

```
-c0 6 -c1 c0+1 -f0 p1==p0?p1:p1==p0+1?c1-p1:0 -u0
ccreate_seq[c1*c1](f0(l/c1,l%c1)) -u1 mmat_absorb_visits(u0) -c2
sum(dmat_row(0,u1)) -c3 last(u1) -eval
write_ratio(c23);space(2);rwrite_ratio(c23)
```

   And similarly for other values of c0.

10.
```
-c0 6 -c1 (c0+1)*(c0+2)/2 -s0 c1*c1 -f0 p0*(2*c0+3-p0)/2+p1 -f1
c1*f0(p0,p1)+f0(p2,p3) -f2 p4&vset0(f1(p0,p1,p2,p3),p4) -eval
rloop0(c0+1,rloop1(c0+1-r0,f2(r0,r1,r0,r1,r0);f2(r0,r1,r0+1,r1-1,r1)
;f2(r0,r1,r0,r1+1,c0-r0-r1)));v1:=mmat_absorb_visits(v0);r0:=sum(
dmat_row(0,v1));r1:=last(v1);write_ratio(r01);space(2);rwrite_ratio(
r01)
```

```
-c0 6 -c1 (c0+1)*(c0+2)/2 -s0 c1*c1 -f0 p0*(2*c0+3-p0)/2+p1 -f1
c1*f0(p0,p1)+f0(p2,p3) -f2 p4&vset0(f1(p0,p1,p2,p3),p4) -eval
rloop0(c0+1,rloop1(c0+1-r0,f2(r0,r1,r0,r1,r0);f2(r0,r1,r0+1,r1-1,r1)
;f2(r0,r1,r0,r1+1,c0-r0-r1)));v1:=mmat_absorb_visits(v0);r0:=sum(
dmat_row(0,v1));r1:=last(v1);write_ratio(r01);space;rwrite_ratio(
r01) -v2 dmat_identity[c1] -noreset -nowarnings
v2:=mmat_mult(v0,v2);write(number+1);space(2);rwrite([dmat_get(0,c1-
1,v2)]/[first(dmat_rsum(v2))]) 24
```

   And similarly for other integer sizes and numbers of evaluations.

11.
```
-u0
{{0,6,0,0,0,0,0},{0,1,5,0,0,0,0},{0,0,2,4,0,0,0},{0,0,0,3,3,0,0},{0,
0,0,0,4,2,0},{0,0,0,0,0,5,1},{0,0,0,0,0,0,6}} -u1 mmat_mult(u0,u0)
-u2 mmat_absorb_visits(u1) -c0 sum(dmat_row(0,u2)) -c1 last(u2)
-eval write_ratio(c01);space(2);rwrite_ratio(c01)
```

```
-u0
{{0,6,0,0,0,0,0},{0,1,5,0,0,0,0},{0,0,2,4,0,0,0},{0,0,0,3,3,0,0},{0,
0,0,0,4,2,0},{0,0,0,0,0,5,1},{0,0,0,0,0,0,6}} -u1 mmat_mult(u0,u0)
-u2 mmat_absorb_visits(u1) -c0 sum(dmat_row(0,u2)) -c1 last(u2)
-eval write_ratio(c01);space(2);rwrite_ratio(c01) -v0
dmat_identity7 -noreset -nowarnings
v0:=mmat_mult(u1,v0);write(number+1);space(2);rwrite([dmat_get(0,6,
v0)]/[first(dmat_rsum(v0))]) 12
```

   And similarly for other integer sizes and numbers of evaluations.

12. `-exact -c0 1 -statistics count_diff(sorted[c0]d6)`

    And similarly for other values of c0.

13. `-exact -histogram -c0 3 max(sorted[c0]d6)`

14. `-c0 6 -c1 50 -f0 c0*p0+p1 -s0 f0(c1) -v1 copy[s0](rnull) -g0 ratio_lmean(mid[2*c0](2*f0(p0),v1)) -f1 [ratio_vset1(f0(p0,p1),q2)] -eval rloop1(c0,f1(c1-1,r1,ratio(r1+1))) -eval rloop0(c1-1,r0:=c1-2-r0;v2:=g0(r0+1);rloop1(c0,v3:=ratio(r1+1); ratio_gt(v2,v3)?vset0(f0(r0,r1),1);f1(r0,r1,v2):f1(r0,r1,v3))) -eval rloop0(c1,v2:=g0(r0);write(c1-r0);space(2);rwrite_ratio(v2);space(2) ;write_ratio(v2);nline);nline;rloop0(c1-1-r0,write(r0+1);space(2); lwrite(mid[c0](f0(r0),v0)?0:sequence+1);nline)`

15. `-exact -probability -statistics -c0 17 any_eq(sorted[c0]d6,6)`

    And similarly for other values of c0.

    ```
    v0:=order_dz_dist6(number+1,number);write(number+1);space;
    rwrite_ratio{last(v0),v0} 24
    ```

    And similarly for other integer sizes.

16. `-exact -probability -statistics -c0 21 overlap(sorted[c0]d6,{1,2})==2`

    `-exact -probability -statistics -c0 23 overlap(sorted[c0]d6,{1,2,3})==3`

    `-exact -probability -statistics -c0 27 count_diff(sorted[c0]d6)==6`

    And similarly for other values of c0.

17. `-exact -probability -statistics -c0 6 count_eq(sorted[c0]d6,6)==1`

    `-exact -probability -statistics -c0 12 count_eq(sorted[c0]d6,6)==2`

    `-exact -probability -statistics -c0 36 -c1 6 count_eq(sorted[c0]d6,6)==c1`

    And similarly for other values of c0, and in the third case, c1.

18. `-c0 100 -c1 10 -v1 unit[c1+1] -s0 s1*s1 -eval cloop[c1](dmat_vset0(l,0,5);dmat_vset0(l,l+1,1));dmat_vset0(c1,c1,6) -eval do[c0](v1:=mat_mult(v1,v0)) -eval r0:=last(v1);r1:=pow(6,c0);write_ratio(r01);nline;rwrite(r0/r1)`

19. `-u0 {{0,6,0,0,0,0,0,0},{0,0,5,0,0,0,0,1},{0,0,0,4,0,0,0,2},{0,0,0,0,3,0, 0,3},{0,0,0,0,0,2,0,4},{0,0,0,0,0,0,1,5},{0,0,0,0,0,0,0,6},{0,0,0,0, 0,0,0,6}} -u1 mmat_absorb_visits(u0) -c0 sum(dmat_row(0,u1)) -c1 last(u1) -eval write_ratio(c01);space(2);rwrite_ratio(c01)`

    `-exact -statistics find_eq(occurs(6d6),1)+1`

    `-exact -statistics -pool 6 6 s0:=6;xuntil6(r0:=pool_dz(6);incr1,ince0==2,incr1)`

    `-exact -statistics -pool 12 6 s0:=6;xuntil12(r0:=pool_dz(6);incr1,ince0==3,incr1)`

    `-statistics find_eq(occurs(12d6),2)+1 10000000`

    `-statistics find_eq(occurs(18d6),3)+1 10000000`

20. `-exact -probability -statistics sorted(10d6)`

    ```
    -exact -probability -statistics -c0 10 -pool c0 6
    until(r0:=r1;r1:=pool_d(6),r1<r0|incr2==c0);r1>=r0
    ```

    And similarly for other values of c0.

21. `-exact -probability -statistics list_eq(sorted2d6,sorted2d6)`

    `-exact -probability -statistics list_eq(sorted3d6,sorted3d6)`

    `-exact -probability -statistics list_eq(sorted6d6,sorted6d6)`

22. `-exact -probability -statistics count_eq(sorted6d6,6)>=1`

    `-exact -probability -statistics count_eq(sorted12d6,6)>=2`

    `-exact -probability -statistics count_eq(sorted18d6,6)>=3`

23. ```
    -c0 4 -f0 pow(5,p1)*number_combin(p0,p1) -u0
    ccreate_mat[c0+1](l0<=l1?f0(c0-l0,c0-l1):0) -u1
    mmat_absorb_visits(u0) -c1 sum(dmat_row(0,u1)) -c2 last(u1) -eval
    write_ratio(c12);space(2);rwrite_ratio(c12)
    ```

    ```
    -c0 7 -f0 pow(5,p1)*number_combin(p0,p1) -u0
    ccreate_mat[c0+1](l0<=l1?f0(c0-l0,c0-l1):0) -eval
    rwrite(mmat_absorb_time(u0,unit))
    ```

    And similarly for other values of c0.

24. `-exact -probability -statistics -c0 1 same(pattern_dist[6*c0]rand6)`

    ```
    -exact -probability -statistics -c0 1 +pscale c0*c0*sqrt(c0)
    same(pattern_dist[6*c0]rand6
    ```

    And similarly for other values of c0.

25. ```
    -exact -c0 6 -c1 c0/2 -c2 c1*(pow(2,c0)-1) -c3 c2+2 -s0 c3*c3 -f0
    p0<c1?p0:c0-1-p0
    rloop0(c1,incr0;e0:=2);rloop1(c2,rloop0(c0,r2:=mat_elem[c3](r1+1,min
    {c2,c1*bitor{r1/c1,pow(2,abs(r0-r1%c1))}+f0(r0)}+1);ince02));vset0(-
    1,c0);v1:=mmat_absorb_visits(v0);r0:=sum(dmat_row(0,v1));r1:=last(v1
    );write_ratio(r01);space(2);rwrite_ratio(r01)
    ```

26. ```
    -c0 6 -c1 c0+1 -s0 c1*c1 -s1 c0 -f0 count_eq(counts(q0),1) -eval
    vmask_loop1(c0,v2:=v1;rloop0(c1,w20:=sequence;r1:=f0(v2);r2:=c1*r0+
    r1;ince02)) -eval
    v3:=mmat_absorb_visits(v0);r0:=sum(dmat_row(0,v3));r1:=last(v3)
    -eval write_ratio(r01);space(2);rwrite_ratio(r01)
    ```

27. ```
    -exact -statistics -output
    %[write_ratio{number_permut(c0,c1),pow(c1,c1)}] -c0 6 -c1 6
    product(count_seq[c1](pattern_list[c0]rand[c1]))
    ```

    And similarly for other values of c0 and c1.

28. `-exact -probability -statistics 3d6==14`

    `-exact -probability -statistics 5d6==14`

    ```
    -c0 3 -c1 6 -c2 5 -c3 6 -u0 sum_dz_dist[c0][c1] -u1
    sum_dz_dist[c2][c3] -c4 max(c02) -c5 min(c02+k01) -eval
    c5>c4&rloop0(c5-c4,r0+=c4;r1:=r0-c0;r2:=r0-c2;i01*u1==i12*u0&(lwrite
    (c0123);space(2);write(r0);space(2);write_ratio{i01,u0};nline))
    ```

    And similarly for other values of c0 to c3.

29. `-exact -probability -statistics 2d5==9`

    `-exact -probability -statistics 2d10==9`

    ```
    -c0 2 -c1 5 -c2 2 -c3 10 -u0 sum_dz_dist[c0][c1] -u1
    sum_dz_dist[c2][c3] -c4 max(c02) -c5 min(c02+k01) -eval
    c5>c4&rloop0(c5-c4,r0+=c4;r1:=r0-c0;r2:=r0-c2;i01*u1==i12*u0&(lwrite
    (c0123);space(2);write(r0);space(2);write_ratio{i01,u0};nline))
    ```

    And similarly for other values of c0 to c3.

30. `-exact -probability -statistics -c0 4 sum(tail3(sorted[c0]d6))==18`

    And similarly for other values of c0.

31. `-eval lwrite(0[3]#max_dz_dist[3][6](4))`

32. `-exact -probability -statistics dot(sorted3d6,{1,1,-1})==0`

33. ```
    -c0 6 -c1 5 -u0 {{0,c1-1,c0-c1+1},{0,c1-1,c0-c1+1},{0,0,c0}} -u1
    mmat_absorb_visits(u0) -c2 sum(dmat_row(0,u1)) -c3 last(u1) -c45
    ratio_add(ratio_mult(c23,{c1,2}),{c0,2}) -eval
    write_ratio(c45);space(2);rwrite_ratio(c45)
    ```

    ```
    -statistics -c0 6 -c1 5 until(r0+=r1:=d[c0],r1>=c1) 10000000
    ```

34. ```
    -c0 2 -c1 6 -u0 sum_dz_dist[c0][c1] -c2 pow(2,k0) -s0 c2*c2 -eval
    rloop0(c2,rloop1(k0,r2:=bitor{r0,bitshift(1,r1)};r3:=mat_elem[c2](r0
    ,r2);e03+=i01)) -eval v1:=mmat_absorb_visits(v0) -eval
    r0:=sum(dmat_row(0,v1));r1:=last(v1);write_ratio(r01);space(2);
    rwrite_ratio(r01)
    ```

    ```
    -statistics -c0 2 -c1 6 -s0 c0*(c1-1)+1
    until(r0:=c0@dz[c1];e0:=1;incr1,all(v0)) 10000000
    ```

    And similarly for other values of c0.

35. `-exact -statistics -c0 5 find_eq(until_sum[c0]d6)`

    And similarly for other values of c0.

36. ```
    -c0 4 -u0
    ccreate_mat[c0+1](l1?l0<c0?(5-(l1-l0-1)%c0+c0)/c0:l1==c0:0) -u1
    mmat_absorb_visits(u0) -c1 sum(dmat_row(0,u1)) -c2 last(u1) -eval
    write_ratio(c12);space(2);rwrite_ratio(c12)
    ```

    And similarly for other values of c0.

    ```
    -u0 {2,3,3,4,4,5} -c0 4 -c1 c0+1 -s0 c1*c1 -eval
    rloop0(c0,rvloop1(u0,r2:=c1*r0+((r0+r1)%c0?:c0);ince02));r0:=-1;e0:=
    1 -eval
    v1:=mmat_absorb_visits(v0);r0:=sum(dmat_row(0,v1));r1:=last(v1)
    -eval write_ratio(r01);space(2);rwrite_ratio(r01)
    ```

    And similarly for other values of u0 and c0.

    ```
    -exact -c0 4 -c1 c0+1 -s0 c1*c1
    v2:=6d6;rloop0(c0,rvloop1(v2,r2:=c1*r0+((r0+r1)%c0?:c0);ince02));r0:
    =-1;e0:=1;v1:=mmat_absorb_visits(v0);r0:=sum(dmat_row(0,v1));r1:=
    last(v1);r0==c0*r1|(lwrite(v2);space(2);write_ratio(r01);space(2);
    rwrite_ratio(r01))
    ```

    And similarly for other values of c0.
    ```

37. `-exact -statistics -c0 5 find_eq(until_sum[c0]d[c0])`

    And similarly for other values of c0.

38. `-exact -probability -statistics -c0 5 sum(until_sum[c0]d6)==c0`

    `-probability -statistics -c0 50 until(r0+=d6,r0>=c0)==c0 100000000`

    ```
    -c0 50 -c1 c0+2 -s0 c1*c1 -eval
    rloop0(c0,rloop1(6,r2:=c1*r0+(r0+r1+1?<c0+1);ince02));r0:=-1;e0:=1;
    r0:=-c1-2;e0:=1;r01:=tail2(dmat_row(0,mmat_absorb_states(v0)));
    rwrite_ratio{r0,r01}
    ```

    ```
    -precision 10 -c0 50 -c1 c0+2 -s0 c1*c1 -eval
    rloop0(c0,rloop1(6,r2:=c1*r0+(r0+r1+1?<c0+1);ince02));r0:=-1;e0:=1;
    r0:=-c1-2;e0:=1;r01:=tail2(dmat_row(0,mmat_absorb_states(v0)));
    rwrite_ratio{r0,r01}
    ```

    And similarly for other values of c0.

39. ```
    -c0 5 -c1 2 -c2 c0+2 -s0 c2*c2 -eval
    rloop0(c0,rloop1(6,r2:=r0+r1+1;r2:=c2*r0+(r2<c0?r2:r2<c01?c0:c0+1);
    ince02));r0:=-1;e0:=1;r0:=-c2-2;e0:=1;r01:=tail2(dmat_row(0,
    mmat_absorb_states(v0)));rwrite_ratio{r0,r01}
    ```

    And similarly for other values of c0 and c1.

40. ```
    -u0 {0,1,1,1,1,1,1} -u1 accum_dist(u0,u0) -eval
    write_ratio{dot(u1,sequence),u1};space(2);write(find_ge(sigma(u1),u1
    /2));space(2);write(find_max(u1))
    ```

    ```
    -u0 {0,1,1,1,1,1,1} -u1 accum_dist(u0,u0) -u2 accum_dist(u1,u0)
    -eval
    write_ratio{dot(u2,sequence),u2};space(2);write(find_ge(sigma(u2),u2
    /2));space(2);write(find_max(u2))
    ```

41. `-exact +statistics -pool 6 6 (r0:=d6)@((r1:=pool_d(6))>=r0?r1:0)`

42. ```
    -c0 6 -u0 {{1,c0-1},{0,c0}} -u1 mmat_absorb_visits(u0) -c1
    dmat_diag(mmat_absorb_diag(u0),u1) -u2
    ratio_add({first(u1),c1},{c0,2}) -eval
    write_ratio(u2);space(2);rwrite_ratio(u2)
    ```

    ```
    -c0 5 -c1 6 -c2 20 -c3 (c2-c0+1)/c1 -c4 pow(c1,(c3+1)*c0) -u0
    {0}#1[c1-1]#{0} -u1 0[c1]#{1} -s01 (c3+1)*c1+1 -v0 head(u0) -v1
    head(u1) -eval
    do[c3](v0*=c1;v0+=head(sum_dist(v1,u0));v1:=head(sum_dist(v1,u1)))
    -eval v2:=multi_dist[c0](v0+v1);v3:=head[c2](v2) -eval
    lwrite(v3,c4);blank;rlwrite(v3,c4);blank;write_ratio{v2-v3,c4};space
    (2);rwrite_ratio{v2-v3,c4}
    ```

43. ```
    -u0
    {{3,1,0,0,1,0,1,0},{0,3,1,1,0,0,0,1},{0,0,4,0,0,0,0,2},{0,0,0,3,0,0,
    0,3},{0,0,0,0,4,1,0,1},{0,0,0,0,0,5,0,1},{0,0,1,0,0,1,4,0},{0,0,0,0,
    0,0,0,6}} -u1 mmat_absorb_visits(u0) -c0 sum(dmat_row(0,u1)) -c1
    last(u1) -eval write_ratio(c01);space(2);rwrite_ratio(c01)
    ```

    ```
    -c0 3 -s0 pow(4,c0) -f0
    bitor{p0,bitshift(1,p1-1,c0),bitshift(p0,p1,c0)} -eval
    cloop[pow(2,c0)](cloop6(cset(f0(l0,l1+1),dmat_vset0(l0,l2,dmat_get(
    l0,l2,v0)+1)))) -eval v1:=mmat_absorb_visits(v0) -eval
    r0:=sum(dmat_row(0,v1)) -eval r1:=last(v1) -eval
    write_ratio(r01);space(2);rwrite_ratio(r01)
    ```

    The latter similarly for other values of c0.

44. `-statistics -f0 pow(sqrt(p0),2)==r0 until(incr1,f0(r0+=d6)) 100000000`

45. `-statistics until(incr1,is_prime(r0+=d6)) 100000000`

    `-exact -statistics -pool 26 6 -f0 r9:=p0;r9>1&!is_prime(r9) until(incr1,f0(r0+=pool_d(6)))`

46. `-exact -probability -statistics -f0 r0:=p0;while(r0>=10,r0/=10);r0 f0(product(sorted2d6))==1`

    And similarly for other numbers of dice.

47. `-probability -statistics same(repeat2{floor(5.8*uniform)+1}) 10000000`

48. `-histogram -range 2 6 do_sum2(real_dist(0.38332764225046719182826783 97,0.14694008131330216014651697 41,0.11265238984384009963206170 58,0.10795693748179925338483297 81,0.11587205926654175072308109 30,0.13325088984404954428523940 95))+2 1000000000`

49. `-exact -c0 6 v0:=sorted6dz[c0];v1:=sorted6dz[c0];list_le(v0,v1)&(v2:=outer_sum(v0,v1);v3:=count_seq(v2);list_eq(v3,sequence6#rsequence5+1)&(lwrite(v0+1);space(2);lwrite(v1+1)))`

    And similarly for other values of c0.

50. `-c0 10 -histogram +chi sequence[2*c0-1]+2 (sequence[c0]+1)#(rsequence[c0-1]+1) real_dist(0.07236,0.14472,0.1,0.055279,0.127639,0.127639,0.055279,0.1,0.14472,0.07236)+real_dist(0.13847,0,0.2241,0,0.13847,0.13847,0,0.2241,0,0.13847)+2 10000000`

52. `-exact -probability -statistics d{2,2,4,4,9,9}>d{1,1,6,6,8,8}`

    `-exact -probability -statistics d{1,1,6,6,8,8}>d{3,3,5,5,7,7}`

    `-exact -probability -statistics d{3,3,5,5,7,7}>d{2,2,4,4,9,9}`

53. `-exact -output %#ny v0:=sorted6dz6;v1:=sorted6dz6;v2:=outer_sum(v0,v1);v3:=count_seq11(v2);all(v3)`

    `-exact -output %#ny v0:=sorted6dz6;v1:=sorted6dz6;list_le(v0,v1)&(v2:=outer_sum(v0,v1);v3:=count_seq11(v2);all(v3))`

    `-exact -output %#ny v0:=sorted6dz6;v1:=sorted6dz6;list_le(v0,v1)&(v2:=outer_sum(v0,v1);v3:=count_seq11(v2);all(v3)&max(v3)<=6)`

    `-exact -output %#ny v0:=sorted6dz6;v1:=sorted6dz6;list_le(v0,v1)&(v2:=outer_sum(v0,v1);v3:=count_seq11(v2);all(v3)&max(v3)<=5)`

    `-exact v0:=sorted6dz6;v1:=sorted6dz6;list_le(v0,v1)&(v2:=outer_sum(v0,v1);v3:=count_seq11(v2);all(v3)&max(v3)<=5&(lwrite(v0+1);space(2);lwrite(v1+1);space(2);lwrite(sort(v3))))`

    `-exact -output %#ny v0:=sorted6dz6;v1:=sorted6dz6;list_le(v0,v1)&(v2:=outer_sum(v0,v1);v3:=count_seq11(v2);all(v3)&max(v3)-min(v3)<=4)`

```
-exact -output %#ny
v0:=sorted6dz6;v1:=sorted6dz6;list_le(v0,v1)&(v2:=outer_sum(v0,v1);
v3:=count_seq11(v2);all(v3)&max(v3)-min(v3)<=3)
```

```
-exact
v0:=sorted6dz6;v1:=sorted6dz6;list_le(v0,v1)&(v2:=outer_sum(v0,v1);
v3:=count_seq11(v2);all(v3)&max(v3)-min(v3)<=3&(lwrite(v0+1);space(2
);lwrite(v1+1);space(2);lwrite(sort(v3))))
```

54. ```
-exact -c0 10 -c1 19
v0:=combine6from[sequence[c0]+1];v1:=outer_sum(v0,v0);(r1:=vcount10(
is_prime(e10)))>=c1&(lwrite(v0);space(2);write(r1))
```

And similarly for other values of c0 and c1.

55. ```
-exact -c0 2 -c1 2 -c2 4
v0:=combine[c0]from[2*sequence[c1]+1];v1:=combine[c0]from[2*sequence
[c1]+2];(v2:=outer_sum(v0,v1);(r0:=vcount20(is_prime(e20)))>=c2&(
lwrite(v0);space(2);lwrite(v1);space(2);write(r0)))
```

And similarly for other values of c0, c1 and c2.

56. ```
-exact -numbers -s0 2 -s1 2 -c0 30 -c1 10 -u0
{0}#recurrence9({1,2},{1,1})
v0:=combine[s0]from[sequence[c0+1]];v1:=combine[s1]from[sequence[c0+
1]];v2:=outer_sum(v0,v1);sum_eq(v2,u0)==s2&(vwrite0;space;vwrite1;
space;vwrite2)
```

And similarly for other values of s0 and s1 (and if required, c0 and c1).

57. ```
-u0 {{0,31,5,0},{30,0,0,6},{0,0,36,0},{0,0,0,36}} -c01
tail2(dmat_row(0, mmat_absorb_states(u0))) -eval
write_ratio{c0,c01};space(2);rwrite_ratio{c1,c01}
```

58. ```
-exact -c0 36 -c1 18 -c2 36
r8:=d[c1];r9:=d[c2];r8<r9&(v0:={{0,c0-r8,r8,0},{c0-r9,0,0,r9},{0,0,
c0,0},{0,0,0,c0}};r01:=tail2(dmat_row(0, mmat_absorb_states(v0)));r0
==r1&(write(r8);space(2);write(r9)))
```

And similarly for other values of c0, c1 and c2.

59. ```
-u0
{{0,0,1116,180,0},{0,0,961,335,0},{0,900,0,0,396},{0,0,0,1296,0},{0,
0,0,0,1296}} -c01 tail2(dmat_row(0, mmat_absorb_states(u0))) -eval
write_ratio{c0,c01};space(2);rwrite_ratio{c1,c01}
```

60. ```
-exact -histogram -c0 10 -c1 6 d[c0]<=>d[c1]
```

```
-exact -probability +statistics -c0 10 -c1 6
r0:=d[c0]<=>d[c1];weight(r0!=0);r0>0
```

```
-exact -probability +statistics -c0 10 -c1 6 d[c0]>=d[c1]
```

```
-exact -probability +statistics -c0 10 -c1 6 d[c0]>d[c1]
```

61. ```
-c0 12 -c1 216 -c2 27 -c3 15 -u0
ccreate_mat[2*c0+1](l0==0|l0==2*c0?l1==l0?c1:0:l1==l0-1?c2:l1==l0+1?
c3:l1==l0?c1-c23:0) -u1 dmat_row(c0,mmat_absorb_states(u0)) -c4
first(u1) -c5 last(u1) -eval
write_ratio{c4,c45};space(2);rwrite_ratio{c4,c45}
```

62. ```
-eval
v0:=multi_dist30(1[6]);v1:={sum(tail81(v0)),v0};v2:=multi_dist27(1[6
]);v3:={sum(tail63(v2)),v2};v4:=ratio_idiv(ratio_add(v1,v3),2);
write_ratio(v1);nline;rwrite_ratio(v1);nline;write_ratio(v3);nline;
rwrite_ratio(v3);nline;write_ratio(v4);nline;rwrite_ratio(v4)
```

63. ```
-u0
{{0,3,4,5,5,4,3,4,8},{0,27,0,0,0,0,0,6,3},{0,0,26,0,0,0,0,6,4},{0,0,
0,25,0,0,0,6,5},{0,0,0,0,25,0,0,6,5},{0,0,0,0,0,26,0,6,4},{0,0,0,0,0
,0,27,6,3},{0,0,0,0,0,0,0,36,0},{0,0,0,0,0,0,0,0,36}} -c01
tail2(dmat_row(0, mmat_absorb_states(u0))) -eval
write_ratio{c1,c01};space(2);rwrite_ratio{c1,c01}
```

64. ```
-c0 6 -c1 2*c0-3 -u0
ccreate_seq[c1](l==0?0:l==c1-1?c0+2:l==c1-2?4:l+2<c0?l+2:c1-l) -u1
ccreate_mat[c1](l0==0?#0:l0>=c1-2?l1==l0:l1==c1-1?#00:l1==c1-2?c0:l1
==l0?c0*c0-c0-#00:0) -c23 tail2(dmat_row(0, mmat_absorb_states(u1)))
-eval write_ratio{c3,c23};space(2);rwrite_ratio{c3,c23}
```

    And similarly for other values of c0.

65. ```
-exact -probability -statistics same(sorted5d6)

-exact -probability -statistics get(3,groups(sorted5d6))

-exact -probability -statistics
list_eq(groups(sorted5d6),{0,1,1,0,0})

-exact -probability -statistics
list_eq(groups(sorted5d6),{2,0,1,0,0})

-exact -probability -statistics last(runs(sorted5d6))

-exact -probability -statistics get(3,runs(sorted5d6))
```

66. ```
-exact -probability -statistics -pool 10 6
v0:=sorted5d6;do2(r1:=mode_max(v0);vloop_set0(e0==r1?r1:pool_d(6)));
same(v0)
```

67. ```
-u0
{{32,80,80,40,10,1},{0,48,96,72,24,3},{0,0,72,108,54,9},{0,0,0,108,
108,27},{0,0,0,0,162,81},{0,0,0,0,0,243}} -u1 mmat_absorb_visits(u0)
-c0 last(u1) -u2 head5(u1) -c1
dot(u2,{32,48,72,108,162}*{5,4,3,2,1}) -eval
write_ratio{7*c1,2*243*c0};space(2);rwrite_ratio{7*c1,2*243*c0}

-histogram -range 0 20
r0:=5;until(w0:=5d6;r1:=count_eq(w0%3,2);r1?r0-=r1:r2+=w0,r0==0);r2
10000000
```

68. ```
-c0 5 -s0 c0+1 -s1 c0 -eval
r12:=1;rloop0(c0,incr0;r3:=wmask_sum10(6,w20:=replace_eq(v1+1,3,0);
wmask_min30(,w30==r0?intmax:vget0(w30)+r2*dot(!v3,w20)));r2*=r1*=6;
v0*=r1;e0:=r3);lwrite(v0,r2);nline;rlwrite(v0,r2)
```

69. ```
-exact -statistics -c0 19 v0:=until_sum[c0]d6;all_ne(v0,1)?v0:0
```

    And similarly for other values of c0.

    ```
-probability +statistics -c0 100 -c1 2 -c2 64 -f0
until(r0+=r9:=d6,r9==1|r0>=p0);r9==1?0:r0
until(r1+=f0(c1?<c0-r1);r2+=f0(c2?<c0-r2),r1>=c0|r2>=c0);r1>=c0
10000000
```

    And similarly for other values of c1 and c2.

70. `-exact -probability -statistics sum(until_sum100select{1000,2,3,4,5,6})<1000`

71. `-c0 6 -s0 pow(2,c0) -s1 2*s0 -u0 geometric[c0](1,2) -u1 sequence[c0]+1 -eval r0:=-1;lists_loop2(1,incr0;v3:=ratio_rmean1(c0,[e21]?ratio_vget1(r0-i01):rzero);v4:={dot(!v2,u1),1};e0:=ratio_gt(v3,v4);[ratio_vset1(r0,e0?v3:v4)]) -eval r0:=-1;lists_loop2(1,incr0;lwrite(v2?0:u1);space;write(e0);nline);v5:=tail2(v1);write_ratio(v5);space;rwrite_ratio(v5)`

    And similarly for `ratio_ge` replacing `ratio_gt`.

72. `-c0 6 -c1 1 -u2 multi_dist[c1](1[c0]) -s0 pow(2,k2) -s1 2*s0 -u0 geometric[k2](1,2) -u1 sequence[k2]+c1 -eval r0:=-1;lists_loop2(1,incr0;v3:=ratio_idiv(ratio_rsum1(k2,[e21]?ratio_imult(ratio_vget1(r0-i01),i21):rzero),u2);v4:={dot(!v2,u1),1};e0:=ratio_gt(v3,v4);[ratio_vset1(r0,e0?v3:v4)]) -eval r0:=-1;lists_loop2(1,incr0;lwrite(v2?0:u1);space;write(e0);nline);v5:=tail2(v1);write_ratio(v5);nline;rwrite_ratio(v5)`

    And similarly for other values of c1 and for `ratio_ge` replacing `ratio_gt`.

    `-c0 6 -c1 2 -u2 multi_dist[c1](1[c0]) -s0 pow(2,k2) -s1 2*s0 -u0 geometric[k2](1,2) -u1 sequence[k2]+c1 -eval r0:=-1;lists_loop2(1,incr0;v3:=ratio_idiv(ratio_rsum1(k2,[e21]?ratio_imult(ratio_vget1(r0-i01),i21):rzero),u2);v4:={dot(!v2,u1),1};e0:=ratio_gt(v3,v4);[ratio_vset1(r0,e0?v3:v4)]) -eval r0:=-1;lists_loop2(1,incr0;e0&(r1:=count(!v2);lwrite(v2?0:u1);space;write(r1);nline))`

    `-c0 6 -c1 2 -u2 multi_dist[c1](1[c0]) -s0 pow(2,k2) -s1 2*s0 -u0 geometric[k2](1,2) -u1 sequence[k2]+c1 -eval r0:=-1;lists_loop2(1,incr0;v3:=ratio_idiv(ratio_rsum1(k2,[e21]?ratio_imult(ratio_vget1(r0-i01),i21):rzero),u2);v4:={dot(!v2,u1),1};e0:=ratio_gt(v3,v4);[ratio_vset1(r0,e0?v3:v4)]) -eval r0:=-1;lists_loop2(1,incr0;e0&((r1:=count(!v2))>2&(lwrite(v2?0:u1);space;write(r1);nline)))`

73. `-c0 6 -c1 22 -s0 c0*(c1-1+c0) -v1 copy[2*s0](rzero) -f0 (p0-1)+c0*(p1-1) -eval rloop0(c0,r0+=c1;rloop1(c0,incr1;r2:=f0(r1,r0);[ratio_vset1(r2,ratio(r0))])) -eval rloop0(c1-1,r0:=c1-1-r0;rloop1(c0,incr1;r2:=f0(r1,r0);v2:=ratio_rmean3(c0,[incr3];r4:=f0(r3,r0+r3);[r3!=r1]?ratio_vget1(r4):rzero);v3:=ratio(r0);[ratio_vset1(r2,[ratio_ge(v2,v3)]?e02:=1;v2:v3)])) -s4 2*c0 -eval mat_write[c0](head[c0*(c1-1)](v0));blank;rloop0(c0,incr0;r1:=f0(r0,r0);[ratio_vset4(r0-1,ratio_vget1(r1))]);v5:=ratio_lmean(v4);write_ratio(v5);space(2);rwrite_ratio(v5)`

    `-exact -table -c0 6 -c1 c0*(c0+1)/2 -pool c1 c0 until(r2:=r1;r0+=r1:=pool_d(c0),r1==r2|r01>c1);r1==r2?0:r0`

74. `-exact -statistics -c0 6 -pool c0 c0 while(r1<c0&(r2:=pool_d(c0))>r1,r1:=r2;r0+=r1;incr3)`

    `-u0 {{0,1,1,1,1,1,1,0},{0,0,1,1,1,1,1,1},{0,0,0,1,1,1,1,2},{0,0,0,0,1,1,1,3},{0,0,0,0,0,1,1,4},{0,0,0,0,0,0,1,5},{0,0,0,0,0,0,0,6},{0,0,0,0,0,0,0,6}} -u1 mmat_absorb_visits(u0) -c1 last(u1) -c0 sum(dmat_row(0,u1))-c1 -eval write_ratio(c01);space(2);rwrite_ratio(c01)`

```
-c0 6 -u0
ccreate_mat[c0+2](l0>=c0?l1>c0?c0:0:l1<=l0?0:l1<=c0?1:l0) -u1
mmat_absorb_visits(u0) -c2 last(u1) -c1 sum(dmat_row(0,u1))-c2 -eval
write_ratio(c12);space(2);rwrite_ratio(c12)
```

The first and third of these similarly for other values of c0.

75.
```
-c0 6 -s0 pow(2,c0) -v1 rzero[s0] -s2 c0 -u0 geometric[c0](1,2) -u1
sequence[c0]+1 -eval
r0:=-1;lists_loop2(1,incr0;v3:=ratio(sum(v2?u1:-u1),c0);v4:=[v2]?
ratio_idiv(ratio_rsum1(c0,[e21]?ratio_vget1(r0-i01):rzero),v2):rzero
;v5:=ratio_add(v3,v4);(e0:=ratio_gt(v5,rzero))&[ratio_vset1(r0,v5)])
-eval
r0:=-1;lists_loop2(1,incr0;lwrite(v2?0:u1);space;write(e0);space;
write(dot(!v2,u1));nline)
```

```
-c0 6 -s0 pow(2,c0) -v1 rzero[s0] -s2 c0 -u0 geometric[c0](1,2) -u1
sequence[c0]+1 -eval
r0:=-1;lists_loop2(1,incr0;v3:=ratio(sum(v2?u1:-u1),c0);v4:=[v2]?
ratio_idiv(ratio_rsum1(c0,[e21]?ratio_vget1(r0-i01):rzero),v2):rzero
;v5:=ratio_add(v3,v4);(e0:=ratio_gt(v5,rzero))&[ratio_vset1(r0,v5)])
-statistics
s9:=c0;until(r9:=dz[c0];r1:=r9+1;r8+=(e9?-r1:r1);e9:=1,!get(binary(!
v9),v0));r8 100000000
```

76.
```
-c0 6 -c1 10 -s0 c1+1 -v1 rzero[s0] -eval
rloop0(s0,r1:=c1-r0;v2:=ratio_rmean2(c0,[incr2];[r12>c1]?rzero:
ratio_vget1(r12));v3:=ratio(r1);v4:=[ratio_gt(v2,v3)]?e01:=1;v2:v3;[
ratio_vset1(r1,v4)]) -eval
rloop0(s0,write(r0);space(2);write(e0);nline);nline;v4:=head2(v1);
write_ratio(v4);space(2);rwrite_ratio(v4)
```

And similarly for other values of c1.

77.
```
-c0 6 -c1 10 -c2 50 -s0 2*c2 -eval
rloop0(c2,[ratio_vset0(r0,r0%c1?ratio(r0):rzero)]) -eval
r2:=1;while(r2,r2:=0;rloop0(c2-c0,(r0==0|r0%c1)&(v1:=ratio_rmean1(c0
,ratio_vget0(r0+r1+1));ratio_gt(v1,ratio_vget0(r0))&(r2:=1;[
ratio_vset0(r0,v1)])))) -eval
rloop0(c2-c1,(r0==0|r0%c1)&(v2:=ratio_vget0(r0);v1:=ratio_rmean1(c0,
ratio_vget0(r0+r1+1));write(r0);space;rwrite_ratio(v1);space;
write_ratio(v2);space;rwrite_ratio(v2);space;write(ratio_gt(v1,ratio
(r0)));nline))
```

And similarly for other values of c1 and c2.

78.
```
-s0 6 -c0 10 -c1 s0*(s0+1)/2 -eval

rloop0(s0,r1:=r0+1;e0:=s0*c0*r1+c1>s0*r1+c0*c1) -eval vwrite0
```

```
-s0 6 -c0 10 -c1 s0*(s0+1)/2 -eval
rloop0(s0,r1:=r0+1;e0:=max{s0*c0*r1+c1,s0*r1+c0*c1}) -eval
rwrite_ratio{v0,s0*s0}
```

And similarly for `-c0 100`.

```
-s0 6 -c8 181 -c9 1711 -eval
rloop0(s0,r1:=r0+1;e0:=find_max{4*r1+10*c8,40*r1+c9,400*r1+c8})
-eval vwrite0
```

```
-s0 6 -c8 181 -c9 1711 -eval
rloop0(s0,r1:=r0+1;e0:=max{4*r1+10*c8,40*r1+c9,400*r1+c8}) -eval
rwrite_ratio{v0,4*s0}
```