Fifty Challenging Problems in Probability

Solutions and Illustrations Using the Program *Monaco*

This document describes how the program *Monaco*[1] can be used to address the 56 problems in the book *Fifty Challenging Problems in Probability with Solutions* (Addison Wesley 1965, Dover 1987) by Frederick Mosteller, available at `https://mbapreponline.files.wordpress.com/2013/07/fifty_challenging_problems_in__2.pdf`.

Most of this document assumes significant familiarity with Monaco. Without that, this document can only be used as an indication as to some of the sorts of problems Monaco can solve – see the later section that summarises which problems Monaco can and cannot address. However, the problems in this book are in many ways atypical of the problems Monaco is most suited for, such as problems involving die rolls and drawing cards from decks.

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 some 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 book. However, the solutions in this document 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 easy, even when this is not demonstrated. In addition, most of these problems require the use of approximate mode, either because they involve continuous distributions or because they have large parameters. Default random numbers are used in these cases.

In order to save space, actual Monaco output – most often from the option `-statistics`, but also from output functions in an expression – is 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 not mentioned when used. However, both are included in the summary section described below.

Although it is a minority of cases, 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 re-created using version 2.50 of the program, but were first created well before that.

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". These are quoted to the precision displayed rather than the usually recommended approach of quoting to the precision indicated by the confidence interval. 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 – as reported by the option `+parameters`, which does not report itself – except for runs that just change parameter values; these can be created by simple modification of the given parameters, as is noted in each relevant case.

Discussions of the 56 problems are:

1. The easiest way to handle this problem in Monaco is using a brute force search. We can consider all numbers `r0` of red socks from 1 to `c0` and all numbers `r1` of black socks from 1 to c1. We can do this in exact mode using `r01:=die(c01)` – note that in exact mode we loop over all values. We then pick two socks numbered r2 and r3, counting from 0 to `r01-1`, such that r2<r3.

   This can be implemented as `rloop2(r01-1,rloop3(r01-r2-1,r3+=r2+1;`*term*`))`, where *term* checks whether r2 and r3 are both red socks, which as r2<r3 is simply the test `r3<c0`, and counts how often this occurs as r9, thus *term* is `r3<c0&incr9`.

   Defining `r8:=r01*(r01-1)/2`, the probability of two red socks is now r9 divided by r8, and so the probability of two red socks is a half if `2*r9==r8`, and in this case we can output `r0`, r1 and `r01`, the numbers of red and black socks and the total number of socks. Setting `c0` and c1 to 100, we find the only three solutions in that range are 3, 1, 4; 15, 6, 21 and 85, 35, 120. The first of these answers part (a) of the problem, 4 socks. The second of these has an even number of socks and thus answers part (b) of the problem, 21 socks. The third of these is mentioned in the given solutions.

   Note that although brute force is not elegant, and the given solution uses some algebra, it also has to use some brute force, and thus this problem can be considered a satisfactory use of the program.

2. We cannot use the program to solve this problem for all possible probabilities, only specific values. However, the wording of the question implies, but does not prove, that the solution depends only on the relative probabilities, not their absolute values. We thus will let the probability of winning the first and third matches be the rational number `c0` / c1, and the probability of winning the second match be the rational number c2 / c3.

   We can demonstrate the solution using the alternative cases `-c0123 {1,3,2,3}` and `-c0123 {2,3,1,3}`. An instance of winning a match with probability the rational number `p0` / p1 is implemented by the function `f0[p0>random(p1)]`. For two games in a row to be won the second match must be won and either of the other two matches must be won, which is the result of `f0(c2,c3)&(f0(c0,c1)|f0(c0,c1))`.

   In the first case, with the easier match in the middle, the probability of winning is 10/27; in the second case, with the harder match in the middle, the probability of winning is 8/27. We should thus, in at least this example, play the champion twice and the father once. We can try other values, for example the one in the given solution with either `-c0123 {2,5,4,5}` and `-c0123 {4,5,2,5}`, and get the same probabilities 0.384 and 0.512 as in the given solution.

3. Similarly to the previous problem, we can only illustrate this for example values of *p*, which we let be the rational number `c0 / c1`. Using the same function `f0` as the previous problem, the three man jury returns the correct result with probability `2@f0(c0,c1)+f0(1,2)>=2`. For example, using `-c01 {2,3}` the correct result has a probability of 2/3, and similarly for other examples, the result is the same as the single juror. This is the given solution, that both are always equal.

4. The straightforward way to address this problem, to implement it, does not work in exact mode owing to the indefinite number of die rolls. We can however use approximate mode as `until(incr0,d6==6)` and for ten million attempts, with default random numbers, the mean is about 6.00096, which we can guess means an exact result of 6.

   We can get an exact result by using an absorbing Markov chain, which has the very simple transition matrix `-u0 {{5,1},{0,6}}` and we can count the transitions until reaching the absorbing state of having rolled a 6 from the matrix `v0:=mmat_absorb_visits(u0)` and can output the normalised result using `write_ratio(ends(v0))`. We can put these together as a `-eval` option. The output is 6, as expected and matching the given solution.

5. Making the solution to the problem – which without the program does not live up to the description as challenging – as obvious as possible, we are using real numbers and so must use approximate mode. We let the coin centre be (`x0`,`x1`) and without loss of generality let this lie in the square with both coordinates from zero to one, thus `x0:=xuniform;x1:=xuniform`. Then with coin diameter `b0`, either coordinate, as `n0`, is suitable if the function `f0` is true, where `f0` is defined as `n0-b0/2>0&n0+b0/2<1`. The overall probability is `f0(x0)&f0(x1)`. This is, with `-b0 0.75` and with ten million results, about 0.0624273. We can see what this figure probably is by adding the option `-fraction 100`, and that is then about 1/16, which is the given solution.

6. It does not matter which number is bet on, so here we assume that is 6. The problem is then simply solved using `get(count_eq(3d6,6),{-1,1,2,3})` and the result is -17/216, i.e. a loss of 17/216, as the given solution.

7. The number of turns that Mr. Brown wins if he plays for `c0` turns on a wheel with `c1` numbers, on one of which he wins and on the rest of which he loses, is given by `r0:=sum(selection_list[c0]from[unit[c1]])`. If a win is rewarded by `c2`, plus his stake returned, then his winnings are given by `r0*(c2+1)-c0`. For the game as described, `c0`, `c1` and `c2` are 36, 38 and 35. This problem is too large to be solved exactly using 64 bit integers, but can be solved with 192 or more bit integers. Mr. Brown's net gain against the casino is -1.89474, i.e. he makes a loss. Interpreting the bet with his friend is not clear from the question, as it does not make the case of breaking even clear, but the given solution indicates that the bet is to lose all turns. If that bet is for the sum `c3`, in the problem as given equal to 20, that adds further winnings of `r0?c3:-c3`, with a combined mean gain of 2.79042. These results agree, to the precision quoted, with the given solution.

8. This can simply use the expression `same(combine13from[copy13(sequence4)])` and the probability is 1 in 158753389900 or 6.29908×10^{-12}, as the given solution.

9. This problem is best handled as an absorbing Markov chain, with states start, points of 4, 5, 6, 8, 9, 10, lose, win. The transition matrix is:

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

   This can be set as the list `u0`, see the summary in the final section. The winning probability can be found by letting `v0:=mmat_absorb_states(u0)` and the winning probability is then `last(dmat_row(,v0))` divided by `first(dmat_rsum(v0))`. The winning probability is 244/495, or 0.492929, as the given solution.

10. This problem is not suitable for the use of Monaco.

11. This problem is not suitable for the use of Monaco.

12. This problem is not suitable for the use of Monaco.

13. This problem is not suitable for the use of Monaco.

14. The direct approach here, using `until(r0:=dz[s0];e0:=1;incr1,all(v0))`, where `s0` is the number of coupons, i.e. 5, only works in approximate mode, where the result is about 11.4156. We can set this up as a Markov chain similarly to problem 9; for the matrix see the final summary section. The result is 137/12 or 11.4167, which agrees with the given solution.

15. We can create a suitable row as `v0:=permute[c01]from[step[c01](c0)]`, where `c0` and `c1` are the numbers of the two kinds of people, here 8 and 7. The total number of pairs in that row can be determined as `count(frest(v0)^lrest(v0))`. The mean result is 112/15 or 7.46667, which is the given solution.

16. Setting up this problem using the program is harder than noticing the obvious solution. However, since the work can be reused in the following problem, this has been done here.

    Let the number of rounds be set by `-c0 3`, and then a list of the players, of whom only two matter, can be set as `-u0 head[pow(2,c0)]{1,2}`, the number of players now being `k0`. We let `r0` be the current number of players, initially `r0:=k0`. We then run the first `c0-1` rounds of the tournament – we do not run the final – using `do[c0-1](r0/=2;`*term*`)`. *term* now runs the round and moves the winners to the front of the list `v0` by being `rloop1(r0,r2:=2*r1;r3:=r2+1;e01:=max{e02,e03})`. Finally, we want to know if the two relevant players meet in the final, which would be `e02&e03` if `r2` and `r3` were 0 and 1 at this point. As they are not, we first set `r23:={0,1}`.

That event, that the two players meet in the final has probability 4/7, as the given solution.

17. We start with the approach in the previous problem, extended to include the final. So everything before the outer loop is unchanged, and that loop is now `do[c0]`. We would like to make *term* `rloop1(r0,r2:=2*r1;r3:=r2+1;e01:=dz2?e02:e03)` as now jousts have a random result, but there are two problems. First, we cannot use `dz2` inside a variable loop in exact mode. So we replace it by `pool_dz(2)` as we know the number of jousts is `k0-1` and can initialise the pool using `-pool k0-1 2`. Second, we have discarded the information we want, so we can add `r9|=(e02&e03)` before changing `e01`. Our final result is `r9`. There is much unnecessary work here – we not need to randomly choose between knights numbered 0 and could stop once our result is known. But this is easier, and fast enough.

    The result is a probability of 1/4, as the given solution.

18. This is a simple calculation that does not need the program, although the program could be used as the calculator of the required binomial coefficient. It is also simple to use the expression `sum(sorted100dz2)==50`, although this hides that binomial coefficients are used in its implementation. More than 256 bit integers are required. The resulting probability is 0.0795892, which agrees with the given solution.

19. These are all cases of requiring at least `c0` sixes when `6*c0` dice are thrown. For convenience we renumber the dice {0,0,0,0,0,1} and count the true results, i.e. `count(select[6*c0]from{0,0,0,0,0,1})>=c0`. For `c0` = 1, 3, 3 the probabilities are 0.665102, 0.618667, 0.597346, which match the given solution.

20. We here perform some analysis before using the program, but the (relatively) hard part of the problem uses the program, and thus we consider this as using the program to solve the problem.

    A's initial options are to shoot at B, shoot at C, or fire to miss – the latter is not specified as an allowed option, but is a known approach in such problems. Shooting at C is pointless because a hit gives the duel to B. So the choice is between shooting at B and firing to miss. We consider each in turn, it is convenient to start with the case of firing to miss.

    If A fires to miss, B will eliminate C. That leaves the situation where it is A's shot and there is only B left. A has one shot at B, then loses. A's probability of success is 0.3, B's is 0.7, C's is zero. There is no point in using the program here.

    If A fires at B and misses, probability 0.7, then this is as the previous case. If A fires at B and hits, this is a new scenario, A and C left standing, C's shot. Assume that A's probability of success in this case is p, then A's overall probability of success is $0.7\times0.3+0.3p = 0.3\times(0.7+p)$. It remains to calculate p, and for that we can use the program. We use this approach if $p > 0.3$.

We can consider this as a Markov chain with four states: C to shoot, A to shoot, C wins, A wins. The transition matrix is:

$$\begin{pmatrix} 0.0 & 0.5 & 0.5 & 0.0 \\ 0.7 & 0.0 & 0.0 & 0.3 \\ 0.0 & 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{pmatrix} = \frac{1}{10}\begin{pmatrix} 0 & 5 & 5 & 0 \\ 7 & 0 & 0 & 3 \\ 0 & 0 & 10 & 0 \\ 0 & 0 & 0 & 10 \end{pmatrix}$$

We use the same expression as problem 9, with the new matrix, to get a probability p = 3/13 = 0.230769. As p < 0.3, the solution to the problem is that A should first fire to miss, then on all subsequent shots fire at his remaining opponent. This agrees with the given solution.

21. We start by considering drawing with replacement. We let the numbers of balls in the two urns, as `-c0123 {2,1,101,100}`, urn A before urn B, red before black. We copy this to variables at the start of the expression as `r0123:=c0123`, but we also need the constants as described below. We select an urn by `r9:=dz2`, 0 being urn A, 1 being urn B. We could draw a ball, 0 red, 1 black, by `distribute([r9]?r23:r01)` but this cannot be used in exact mode, and so we instead use the randomness pool by `pool_dist([r9]?r23:r01)` and make this the function `f0`, without arguments. Now we draw two balls by `r4:=f0;r5:=f0`. We need to initialise the pool, which can be by `-pool 1 c01*c23`, used twice. We could combine them, but this is easier when we consider the sampling without replacement case following. We could also, use a least common multiple to reduce the pool size, but there is no need to do this.

    Now we need to collect useful statistics for this case. The simplest way to do so is to use the numbers 0 to 7 to record the values of r9, r4 and r5 as 4*r4+2*r5+r9. How we use this is considered below.

    Now we consider the case without replacement. Between setting `r4` and `r5` we use `f1` to update the urn, that being by `r9?r4?decr3:decr2:r4?decr1:decr0`, we do not use its result. If we otherwise continue as above, we need to change the second setting of the pool size to `-pool 1 (c01-1)*(c23-1)`.

    Now we need to analyse the list results, and this requires processing after main expression evaluation has completed, a task for the option `-output`, using an output string `%[`*expression*`]`, where we now consider what *expression* should be. It is convenient to set `r0` to r7 – whose previous values we now do not need – to the numbers of occurrences of the results 0 to 7. First setting `-s9 8`, one way to do this is using `results_loop9(e9:=results(r9));r01234567:=v9`.

    We now can consider, for each case, what the probability of getting our guess right is. If, for example, we see RR, this can occur in `r0`+`r1`, or `r01`, ways. We would guess the more likely of these, i.e. a number of occurrences of `r0?>r1`. If we see RB this is `r2?>r3`. If we have just seen a first draw of red, then the probability of being correct is `(r0?>r1)+(r2?>r3)` divided by `r0123`, which we can report as a real number using `rwrite_ratio`. We then, in each of the cases, i.e. two runs, can similarly produce four probabilities, the probabilities of being right when first drawing red or black. With replacement these probabilities are 0.593966 and 0.598802; without replacement these probabilities are 0.570213 and 0.697605. This tells us that we are better replacing when we draw red first, but not replacing when we draw black first.

However, this has not told us which urn we should guess. We can extend *expression* by adding this – this will provide four cases in each run, but we will only use two from each. For example, having drawn RR our choice – 0 for urn A, 1 for urn B – is `r1>r2`. All four results can be conveniently output as a list using `lwrite(r1357>r0246)`. Our results for RR, RB, BR, BB with replacement are {0,1,1,1}, without replacement are {0,0,0,1}. Making the better choices as to whether to replace, this becomes {0,1,0,1}.

The conclusions on drawing RR, RB, BR or BB are thus urns A, B, A or B, respectively. This says we consider the first ball as to whether to replace, replacing red only, and then the second ball as to which urn to guess: red to guess urn A, black to guess urn B. This agrees with the given solution.

22. We cannot use the program to solve for general *a* and *b*, only for selected values, which we let be `c0` and `c1`. We can construct the, not yet randomised, list of all votes as -1 and 1 for the two candidates using `-u0 step[c01](c0)?1:-1` and then can create a random list of the votes as `permute[k0]from[u0]`. If we let that be *list*, then our required event is `any_eq(sigma(`*list*`))`.

    Two example cases – which both require 128 bit integers – are that $a = 15$, $b = 10$ has probability 4/5, and that $a = 20$, $b = 10$ has probability 2/3. These – and that the probability is 1 when $a = b$, although this case has been excluded by the stated problem, but is considered in the given solution – are consistent with the hypothesis that the probability is $1-(a-b)/(a+b) = 2b/(a+b) = 2/(1+r)$ where $r = a/b$, which is the given solution.

23. We cannot use the program to solve for general *N*, only for selected values, which we let be `c0`. For the required event no tie, we can use `xuntil[c0](r0+=ds2,r0==0,1)`. For $N = 2$ and $N = 3$ the probability is 1/2; for $N = 4$ and $N = 5$ the probability is 3/8; for $N = 6$ and $N = 7$ the probability is 5/16; for $N = 8$ and $N = 9$ the probability is 35/128; for $N = 10$ and $N = 11$ the probability is 63/256; for $N = 12$ and $N = 13$ the probability is 231/1024. These are 1/2, 3/8, 10/32, 35/128, 126/512, 462/2048 and the sequence 1, 3, 10, 35, 126, 462 can be recognised (through inspection of Pascal's triangle, or as `https://oeis.org/A001700`) as being the binomial coefficient ${}^{2m-1}C_m$, where $m = N/2$ (truncated) and the probabilities are ${}^{2m-1}C_m/2^{2m-1}$. This is equivalent to the given solution ${}^{2m}C_m/2^{2m}$, as that solution notes.

24. This problem is not suitable for the use of Monaco.

25. This problem has a well-known ambiguity in the meaning of the phrase random chord. Here we choose what is probably the most natural definition, joining two random points of the circle's circumference. As all points are equivalent, we choose one to be (1, 0), assuming a unit circle with centre at the origin. The other point is at $(\cos\theta, \sin\theta)$ where θ is `x0:=2*pi*xuniform`. The required test for the chord length is then `real_gt(sqrt(sum_sq(1-cos(x0),sin(x0))),1)`. This can only be done in approximate mode; over ten million results the probability is about 0.66654, from which we guess that it is 2/3. This is the given solution (c).

26. This problem is solved using the expression:

    ```
    x0:=60*xuniform;x1:=60*xuniform;real_lt(abs(x0-x1),5)
    ```

This can only be used in approximate mode. Over ten million results the probability is about 0.159563. The given result is 23/144, or 0.159722. The two agree to the expected degree.

27. With *n* boxes and *n* coins per box, and defining `c0` as *n*, the probability of not being detected is given by the expression `none(selection[c0]from[unit[c0]])`. This can be run in exact mode for small values of *n*, but is computationally impractical for *n* = 100, where approximate mode is needed, where we can simplify to the expression `none(select[c0](unit[c0]))`. For ten million results this probability is about 0.365993. The given result is quoted only as 0.366,which this result agrees with, but the given solution could be calculated more precisely as 0.366032, and then the results agree to the expected degree.

    Other values of *n* could be evaluated, but would not significantly add to the value of using the program in this way.

28. Similarly to the previous problem we define `-c0` *n*, `-c1` *m*, `-c2` *r*, and use the expression `count(select[c0](rstep[c0](c1)))==c2`. For example with *n* = 10, *m* = 2, *r* = 3 the probability is 0.201327. The given solution formula is the real number `[number_combin(c0,c2)]*pow(c1/c0,c2)*pow(1-c1/c0,c0-c2)`, which has the same value, so the two results agree.

   Other values of *n*, *m* and *r* could be evaluated, but would not significantly add to the value of using the program in this way.

29. The reference to many plates suggests a Poisson distribution, and then, for `c0` plates, the probability is the real value `poisson_dist(c0,c0)`. For `c0` = 3 this result is 0.224042, which is as the given solution, to the precision reported there.

    Increasing `c0`, results include `c0` = 10 result is 0.12511; `c0` = 100 result is 0.039861; `c0` = 1000 result is 0.0126146; `c0` = 10000 result is 0.00398939. This appears to be declining as the square root of `c0`, so multiplying those results by `sqrt(c0)` gives `c0` = 10 result is 0.395633; `c0` = 100 result is 0.39861; `c0` = 1000 result is 0.398909; `c0` = 10000 result is 0.398939; `c0` = 100000 result is 0.398942; `c0` = 1000000 result is 0.398942. The limiting value appears to be slightly under 0.4. Some investigation identifies that 0.398942 is $1/\sqrt{2\pi}$, so it appears that the limiting value for *n* plates might be $1/\sqrt{2\pi n}$. This is the given solution.

30. We can solve this by using `poisson_dist` multiple times. We use `-c0 20`; we would need to modify our expression if `poisson_dist(c0,0)` could equal zero, but it does not in this case. We can sum over even numbers to produce the probability `x0` by `until(x0:=x1;x2:=poisson_dist(c0,r0),r0+=2;x1:=x0+x2;real_eq(x0,x1))` and then we can output `x0`. This is reported as 0.5, and we might guess this the exact result is a half. However, the given solution shows that the exact solution is greater than this by ½e^{-40}, but that is negligibly small, being about 2×10^{-18}; real numbers cannot reliably record that difference when summing multiple terms, each with their own error. Note that in general, for mean *m*, the given result is ½(1+e^{-2m}), which can alternatively be evaluated as `exp(-c0)*cosh(c0)`.

31. We here make the simplifying assumptions that all dates are equally likely and that we can ignore leap years. Then with *n* = `c0` people, the probability that there is at least one repeated birthday is `!different(pattern_list[c0]rand365)`. This needs

large integers: 128 bits is insufficient, but 256 bits is sufficient. Note that using `sorted[c0]dz365` rather than `pattern_list[c0]rand365` is not computationally feasible. Here we just report the two values of *n* that indicate when this reaches a probability of 0.5: *n* = 22 probability is 0.475695; n = 23 probability is 0.507297. Thus the solution is that 23 people are needed. This agrees with the given solution. That solution considers the more general case, but here we have just considered the case asked – although we could change 365 to c1 and be able to handle any specific case.

32. Unfortunately, even 1024 bit integers are insufficient to solve this problem exactly – using `selection_list` the number of evaluations is small, but the number of results is too large. We thus must use approximate mode (if using the program; this is very much a case where a solution not using the program is much simpler) and the expression `any(select[c0](unit365))`.

    Using approximate mode it is hard to pin down the exact transition from a probability below 0.5 to one above it, due to the uncertainty in the results that we produce. Here we use a 99.99% confidence interval, rather than the default 95%, by using `-confidence 99.99`, and use a hundred million results for the cases quoted here. We then have for 252 people a confidence interval of [0.498882, 0.499271] and for 253 people a confidence interval of [0.500297, 0.500686]. We thus have reasonable confidence that the answer is 253 people. This matches the given solution.

33. This problem is not suitable for the use of Monaco.

34. As problem 32, this needs approximate mode, but to be practical final runs are only ten million results. At this point even 95% confidence intervals overlap, and so the answer produced here is not definite. With `c0` as the number of workers, the number of days productivity is `c0*(365-count_diff([c0]dz365))`. With 363 workers the mean total productivity is 48942.5 days; with 364 workers the mean total productivity is 48943.2 days; with 365 workers the mean total productivity is 48943.3 days; with 366 workers it is 48942.3 days, and thus we estimate the optimum factory size as 364 or 365 workers, which are too close to separate – and even the other figure quoted, or ones beyond them, are possible based on those results. The given solution is 364 workers, and we are consistent with that result.

35. Using the program an exact answer is not possible because in a finite number of steps one cannot be sure if the man will later fall off the cliff. Nevertheless, we will set a maximum number `c0` of steps. The values of `c0` that we will try do not allow exact mode, so we will use ten million results in approximate mode. The expression `xwhile[c0](r0>=0,r0+=(dz3?1:-1),1)` has result 1 if the man has not fallen off the cliff, -1 if he has. For `c0` equal to 1000 he has fallen off the cliff with a probability of about 0.499874, or has not done so with a probability of 0.500126. We can make a guess that the man has a probability of one half of falling off the cliff, and this is the given solution.

36. This problem can be solved as an absorbing Markov chain. We let the states be player *M*'s funds: 1, 2, 0, 3, so we start in the first state and want the probability of finishing in the last state. The transition matrix is:

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

    Otherwise we proceed as problem 9. The solution is that *M* wins with probability 4/7, which is the given solution.

37. Although not specifically mentioned in the problem, this problem assumes roulette as problem 7, with a probability of winning each spin of 18/38, or 9/19. If the gambler makes a single spin, his probability of reaching $40 is 9/19. The problem is thus if he plays single bets until reaching $0 or $40, what is his probability of the latter?

    This is again a Markov chain, as problem 36, except that we will number the states 0 to 40 and thus start in state 20, not in the first state, but we still consider the case of finishing in the last state. Setting up the required transition matrix by hand would be tedious, so we need to get the program to calculate the transition matrix. We start by setting up four constants: c0, the target result, i.e. 40; c1, the starting state, i.e. 20; c2, the relative probability of winning a spin, i.e. 9; c3 the relative probability of losing a spin, i.e. 10. It is convenient to also define `-c4 c0+1` and `-c5 c23`, the number of states and the probability normalising factor, respectively.

    The transition matrix u0 can then be created as the result of the expression:

    ```
    ccreate_mat[c4][c4]
    (l0==0?l1==0??c5:l0==c0?l1==c0??c5:l1==l0+1?c2:l1==l0-1??c3)
    ```

    As the previous problem, we set v0 to `mmat_absorb_states(u0)`. Then the modified r0 and r1 that the output is based on are given by `r0:=last(dmat_row(c1,v0))` and `r1:=get(c1,dmat_rsum(v0))`.

    To evaluate this needs large integers; 1024 bits is more than enough so we use that here. The probability of winning is 0.108398. The given solution is only reported to the limited precision 0.11, but its calculation can be simplified to $1/\left\{\left(\frac{20}{18}\right)^{20}+1\right\}$ and exactly agrees. The gambler should thus stake all his money at once.

38. This problem is not suitable for the use of Monaco.

39. This is simply the mean value of the minimum of three uniform random numbers `min(xuniform,xuniform,xuniform)`. This can be made a real result using the function `real_result`, and the mean then reported by using `-real_stats`. This has to be approximate, and over ten million results is about 0.249973, which we guess to have an exact value 0.25, which the given solution confirms.

40. This is `find_ne(permute52from[copy4(unit13)],0)+1`; the +1 is needed because the program counts from zero. This needs more than 128 bit integers, but 256 bits is sufficient. The result is 10.6, which is also the given solution.

41. This problem is not suitable for the use of Monaco.

42. The first part of the problem is simply `x0:=uniform` and real result `min(x0,1-x0)`, which `-real_stats` for ten million results reports a mean value of about 0.249975. We can guess that to have an exact value 0.25, which the given solution confirms. The second part of the problem most conveniently sets `x1:=min(x0,1-x0)` and then is the real result `x1/(1-x1)`, which handled similarly is about 0.386248. The given solution is 2 $\log_e 2 - 1$, which is 0.386294, although only quoted as 0.386, and the results agree to the expected degree.

43. We can create the breakpoints and arrange for them to be such that x0 < x1 by using `x0:=xuniform;x1:=xuniform;xsort01`. Next, we create the three lengths as x0, x1 and x2 by `x2:=1-x1;x1-=x0`. We can then sort those lengths in ascending order by using xsort012. We then need three separate runs, all in approximate mode, to estimate the mean values x0, x1 and x2, each as a real result. For ten million results each these are about 0.111132, 0.277803 and 0.611065. We can guess that these have exact values 0.111111..., 0.277777... and 0.611111..., which are 1/9, 5/18 and 11/18. We can make these guesses easier by adding `+fraction 50` in each case. This guesswork matches the given solution.

44. If we let the probability of winning be c0 divided by c1, and choose c2 plays, then we win if `sum(selection_list[c2]from[rstep[c1](c0)])>c2/2` is true. The mean value of that, for a winning probability of 0.45, i.e. `-c0 9 -c1 20`, is, for values of n = c2, the number of plays: n = 2 probability is 0.2025, n = 4 probability is 0.241481, n = 6 probability is 0.255264, n = 8 probability is 0.260381, n = 10 probability is 0.261563, n = 12 probability is 0.260685. We appear to have peaked, and found that the best number of plays is 10. This agrees with the given solution in this case, but we have, as usual, not found a general solution as also requested.

45. For (a) we can reduce the work by noting that there is no need to shuffle both decks, and the result is the mean value of `count(shuffle(sequence52)==sequence52)`. This needs approximate mode as the problem is much too large for exact mode. Over ten million results we get a mean of about 0.999856, which we can guess to have the exact value 1, which is the given solution. (b) is the same problem for general n, (a) having n = 52, but using the program we can only produce results for specific values of n.

46. We can get the probability of each number of matches in the previous problem by using `-histogram` rather than `-statistics`. We assume the same case (a) and for ten million results we see only 0 to 10 matches, more than 10 matches requiring longer runs. For example, the probabilities of 0 to 2 matches are about 0.367954, 0.367825 and 0.183959. The given solution does not provide these numbers, but does note that the first value tends to e^{-1} or 0.367879 as n becomes large, and this suggests that n = 52 can be considered to be fairly large.

47. We do have to use some analysis outside the program to consider this problem. Clearly we only ever stop if the current dowry is the largest seen so far, or fail if there are no more. We also must be increasingly likely to stop the longer we continue for. But no choices other than stop or continue make sense – if there were a probability of trying this dowry or continuing, one must be better than the other (except in rare cases where they are the same, but then there is no harm in assuming either is

better). So our strategy must be pick a number, here *n* = c1, and examining *n* dowries, pick the next one that is larger than any seen so far.

With 100 dowries, this is too large to implement exactly – and would be even for expressions that minimise randomness, rather than the simpler one used here. But as we are using approximate mode, simple is clearer and easier to use. We can conveniently let the dowries be the numbers 1 to c0, where c0 is the number of dowries, i.e. 100. We thus use a list of the dowries `v0:=shuffle(xsequence[c0])`. We then set our target `r0` as the largest value in the first c1 elements of v0, i.e. `r0:=max(head[c1](v0))`. We succeed if the first element in the remaining elements of v0 is equal to c0, i.e. setting `v1:=tail[c0-c1](v0)` and `r1:=find_gt(v1,r0)`, we succeed if `e1==c0`.

Finding the optimum value with only approximate results cannot guarantee to find the optimum solution. Allowing a hundred million results: for *n* = 35 the probability is about 0.370731; for *n* = 36 the probability is about 0.371041; for *n* = 37 the probability is about 0.37108; for *n* = 38 the probability is about 0.370837. The best guess is *n* = 37, but the 95% confidence interval – and we would like a higher level of confidence than that – for *n* = 36 is [0.370946, 0.371135], and for *n* = 37 it is [0.370986, 0.371175], which are almost the same and greatly overlap, so we cannot even be sure if it is *n* = 36 or *n* = 37, and could even be neither.

Noting that the given solution uses a parameter *s* such that *n* = *s*-1, the given solution is that the optimum value for 100 dowries is *s* = 38, so the optimum value of *n* is 37. So we actually exactly agree with the given solution, but this is fortunate, we could easily have only been close to it, and consider our success accordingly.

48. This problem is not suitable for the use of Monaco, but unlike previous such cases this is not because it is fundamentally unsuited, but because pursuing an approach similar to problem 47 we have two parameters to handle – the value of the largest dowry seen so far (which should either be, or be converted to, a uniform random value between 0 and 1) and investigating different numbers of dowries seen so far. This is too large a programme of runs to be realistic to pursue, particularly approximately.

49. This problem is not suitable for the use of Monaco. The solution could be illustrated for some possible figures, but there is little value in doing so.

50. The problem here does not specify how *b* and *c* are to be determined. So we break with our usual practice and consult the given solution, which is that – not using the notation there – we let *b* and *c* be uniformly distributed on the interval from -*x* to +*x*, determine the probability, and then let *x* tend to infinity. The case of having a real root is $b^2 \geq c$. We can use the program to test this for various values of *x*, but this must be approximately. We let *x* be the constant b0, and a suitable expression is `x0:=2*b0*uniform-b0;x1:=2*b0*uniform-b0;real_ge(x0*x0,x1)`. We use ten million results for each run.

For *x* = 1 the probability is about 0.666637; for *x* = 10 the probability is about 0.894495; for *x* = 100 the probability is about 0.966662; for *x* = 1000 the probability is about 0.989467; for *x* = 10000 the probability is about 0.996673; for *x* = 100000 the probability is about 0.998949; for *x* = 1000000 the probability is about 0.999667.

Clearly the probability is tending to one, and thus, in the sense defined here, most quadratic equations of the form given have real roots. This is the given solution.

51. Using the program, we can only address this problem incompletely by answering what the probability is for a large number of steps. If we let that number of steps be c0, we can use the expression:

    ```
    xuntil[c0](dz2?dz2?incr0:decr0:dz2?incr1:decr1;r2:=r0==0&r1==0,r2,0)
    ```

    However, using a million results per run, if we set c0 to 1000 then the proportion of runs that we have returned to the origin is about 0.67814, while if we set c0 to 10000 then the probability that we have returned to the origin is about 0.738996. Knowing the answer to the problem, that the probability is 1 and that we are a long way off that value, this is clearly not going to be a practical proposition to demonstrate the solution, and so we consider that this problem is not suitable for the use of Monaco.

52. As problem 51 – although knowing that the solution is less than 1, even more so – this problem is not suitable for the use of Monaco.

53. We can only illustrate this for selected values of a, which we let be b0, and ℓ, which we let be b1, and only in approximate mode.

    We can assume that the needle's centre is a distance x0 from the nearest line, so x0 is uniformly distributed between 0 and a. We can assume that the needle has a random rotation x1, where a rotation of zero is normal to the lines, that is uniformly distributed between -pi/2 and pi/2, other rotations being equivalent to these. The needle then crosses the line if x0 < b1*cos(x1). Thus we use the expression:

    ```
    x0:=b0*uniform;x1:=pi*uniform-pi/2;real_lt(x0,b1*cos(x1))
    ```

    There is no reason not to use `-b0 1` and then only vary b1. The only case we will consider here is `-b1 0.5`, where using ten million results the probability is about 0.31826. The given solution says this is $2\ell/\boldsymbol{\pi}a = 1/\boldsymbol{\pi}$, which is 0.318310, so the results agree to the expected degree.

54. Compared to the previous problem, we adopt the changed notation of just using b0 for ℓ. We let the centre of the needle be at coordinates (x0, x1) with rotation x2. To make the following problem easier, we let x0 and x1 each be uniformly distributed from 0 to 1, but we let the rotation x2 only vary between 0 and pi/2 – other rotations having the same result by symmetry.

    The needle then crosses a vertical line (using the usual graph meaning of the term, not actually vertical) if either of the following is true: x0-b0*cos(x2) < 0 or x0+b0*cos(x2) > 1 and crosses a horizontal line if either of the following is true: x1-b0*sin(x2) < 0 or x2+b0*sin(x2) > 1. We could rearrange these to simplify the expression, but again with the following problem to consider we do not do so. The resulting expression is given in the final section below.

    Again, we only consider the case `-b0 0.5`. For ten million results the mean number of crossings is about 1.27301. The given solution says this is $8\ell/\boldsymbol{\pi} = 4/\boldsymbol{\pi}$, which is 1.27324, so the results agree to the expected degree.

55. Starting from the solution to the previous problem, the number of crossings here is sum of two uses of the function f0 that takes the coordinates of the two ends in the

same axis direction and counts the lines crossed; so for example the first use of `f0` is as `f0(x0-b0*cos(x2),x0+b0*cos(x2))`. We can define `f0` taking into account the limited rotation used, and thus the constraints on the arguments `n0` and `n1` of `n0` < 1 and `n1` > 0, so that `f0` is `floor(n1)-floor(n0)`.

We illustrate this here with `-b0 3`, which over ten million results has a mean value of about 7.63948. The given solution says this is $8\ell/\boldsymbol{\pi} = 24/\boldsymbol{\pi}$, which is 7.63944, so the results agree to the expected degree.

56. This problem has too many variables – five: four numbers of balls and a number to be drawn – to be suitable for the use of Monaco. On consulting the given solution – and armed with the knowledge that since this book was published Fermat's Last Theorem has been proved for all cases, not just all computationally feasible cases – the problem is impossible, and so this problem is unsuitable for solution by any means, other than a solution that says it is impossible, as the given solution does.

Characterisation of Solutions

The problems, and their solutions in this document, can be characterised into the following categories. Exact solutions are assumed unless indicated otherwise. Any additional material provided, including any additional problems, in the given solutions is not considered in selecting a category here.

- Completely solved, or solved as well as the given solution to the given problem 1, 4, 6-9, 14-21, 31, 36-37, 40 (18 problems).
- Specific result produced, but general case not solved: 29, 44-46 (4 problems).
- Solution illustrated, but general case not solved: 2-3, 22-23, 53-55 (7 problems).
- Close approximate or otherwise limited precision result produced: 5, 25-26, 30, 32, 34-35, 39, 42-43, 47 (11 problems).
- Close approximate result produced, but general case not solved, or not fully solved: 27-28, 50 (3 problems).
- Not suitable for the use of Monaco: 10-13, 24, 33, 38, 41, 48-49, 51-52, 56 (13 problems). In the last of these cases, the problem is impossible.

The program is useful in most cases (43 out of 56), but only a complete solution in about a third of the problems.

Summary of Runs

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

1.
   ```
   -exact -c01 100
   v0:=sequence[c01];r01:=die(c01);rloop2(r01-1,rloop3(r01-r2-1,r3+=r2+
   1;r3<r0&incr9));r8:=r01*(r01-1)/2;2*r9==r8&(write(r0);space(2);write
   (r1);space(2);write(r01))
   ```

2.
   ```
   -exact -probability -statistics -c0123 {1,3,2,3}
   f0[p0>random(p1)];f0(c2,c3)&(f0(c0,c1)|f0(c0,c1))
   ```
   And similarly for other values of c0123.

3.
   ```
   -exact -probability -statistics -c01 {2,3}
   f0[p0>random(p1)];2@f0(c0,c1)+f0(1,2)>=2
   ```
   And similarly for other values of c01.

4.
   ```
   -statistics until(incr0,d6==6) 10000000
   ```
   ```
   -u0 {{5,1},{0,6}} -eval
   v0:=mmat_absorb_visits(u0);write_ratio(ends(v0))
   ```

5.
   ```
   -probability -statistics -fraction 100 -b0 0.75
   f0[real_gt(n0-b0/2,0)&real_lt(n0+b0/2,1)];x0:=xuniform;x1:=xuniform;
   f0(x0)&f0(x1) 10000000
   ```

6.
   ```
   -exact -statistics get(count_eq(3d6,6),{-1,1,2,3})
   ```

7.
   ```
   -exact +noexact -statistics -c0 36 -c1 38 -c2 35
   r0:=sum(selection_list[c0]from[unit[c1]]);r1:=r0*(c2+1)-c0
   ```
   ```
   -exact +noexact -statistics -c0 36 -c1 38 -c2 35 -c3 20
   r0:=sum(selection_list[c0]from[unit[c1]]);r1:=r0*(c2+1)-c0+(r0?c3:-
   c3)
   ```

8.
   ```
   -exact -probability -statistics
   same(combine13from[copy13(sequence4)])
   ```

9.
   ```
   -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
   ,0,25,6,3},{0,0,0,0,0,0,0,36,0},{0,0,0,0,0,0,0,0,36}} -eval
   v0:=mmat_absorb_states(u0);r0:=last(dmat_row(,v0));r1:=first(
   dmat_rsum(v0));write_ratio(r01);space(2);rwrite_ratio(r01)
   ```

14.
    ```
    -statistics -s0 5 until(r0:=dz[s0];e0:=1;incr1,all(v0)) 10000000
    ```
    ```
    -u0
    {{0,1,0,0,0,0},{0,1,4,0,0,0},{0,0,2,3,0,0},{0,0,0,3,2,0},{0,0,0,0,4,
    1},{0,0,0,0,0,5}} -u1 mmat_absorb_visits(u0) -c0
    first(dmat_rsum(u1)) -c1 last(u1) -eval
    write_ratio(c01);space(2);rwrite_ratio(c01)
    ```

15.
    ```
    -exact -statistics -c0 8 -c1 7
    v0:=permute[c01]from[step[c01](c0)];count(frest(v0)^lrest(v0))
    ```

16.
    ```
    -exact -probability -statistics -c0 3 -u0 head[pow(2,c0)]{1,2}
    v0:=permute[k0]from[u0];r0:=s0;do[c0-1](r0/=2;rloop1(r0,r2:=2*r1;r3:
    =r2+1;e01:=max{e02,e03}));r23:={0,1};e02&e03
    ```

17.
```
-exact -probability -statistics -c0 3 -u0 head[pow(2,c0)]{1,2} -pool
k0-1 2
v0:=permute[k0]from[u0];r0:=s0;do[c0](r0/=2;rloop1(r0,r2:=2*r1;r3:=
r2+1;r9|=(e02&e03);e01:=pool_dz(2)?e02:e03));r9
```

18.
```
-exact -probability -statistics sum(sorted100dz2)==50
```

19.
```
-exact -probability -statistics -c0 1
count(selection_list[c0*6]from{0,0,0,0,0,1})>=c0
```

And similarly for other values of c0.

20.
```
-u0 {{0,5,5,0},{7,0,0,3},{0,0,10,0},{0,0,0,10}} -eval
v0:=mmat_absorb_states(u0);r0:=last(dmat_row(,v0));r1:=first(
dmat_rsum(v0));write_ratio(r01);space(2);rwrite_ratio(r01)
```

21.
```
-exact -s9 8 -output
%[results_loop9(e9:=results(r9));r01234567:=v9;rwrite_ratio{(r0?>r1)
+(r2?>r3),r0123};nline;rwrite_ratio{(r4?>r5)+(r6?>r7),r4567};nline;
lwrite(r1357>r0246);nline] -c0123 {2,1,101,100} -pool 1 c01*c23
-pool 1 c01*c23 -s0 4
f0[pool_dist([r9]?r23:r01)];r0123:=c0123;r9:=dz2;r4:=f0;r5:=f0;4*r4+
2*r5+r9
```

```
-exact -s9 8 -output
%[results_loop9(e9:=results(r9));r01234567:=v9;rwrite_ratio{(r0?>r1)
+(r2?>r3),r0123};nline;rwrite_ratio{(r4?>r5)+(r6?>r7),r4567};nline;
lwrite(r1357>r0246);nline] -c0123 {2,1,101,100} -pool 1 c01*c23
-pool 1 (c01-1)*(c23-1) -s0 4
f0[pool_dist([r9]?r23:r01)];f1[r9?r4?decr3:decr2:r4?decr1:decr0];
r0123:=c0123;r9:=dz2;r4:=f0;f1;r5:=f0;4*r4+2*r5+r9
```

22.
```
-exact -probability -statistics -c0 15 -c1 10 -u0 step[c01](c0)?1:-1
any_eq(sigma(permute[k0]from[u0]),0)
```

And similarly for other values of c0 and c1.

23.
```
-exact -probability -statistics -c0 2 xuntil[c0](r0+=ds2,r0==0,1)
```

And similarly for other values of c0.

25.
```
-probability -statistics
x0:=2*pi*xuniform;real_gt(sqrt(sum_sq(1-cos(x0),sin(x0))),1)
10000000
```

26.
```
-probability -statistics
x0:=60*xuniform;x1:=60*xuniform;real_lt(abs(x0-x1),5) 10000000
```

27.
```
-exact -numbers -noeval +noexact -c0 100
none(selection[c0]from[unit[c0]])
```

```
-probability -statistics -c0 100 none(select[c0](unit[c0])) 10000000
```

28.
```
-exact -probability -statistics -c0 10 -c1 2 -c2 3
count(select[c0](rstep[c0](c1)))==c2
```

```
-c0 10 -c1 2 -c2 3 -eval
rwrite([number_combin(c0,c2)]*pow(c1/c0,c2)*pow(1-c1/c0,c0-c2))
```

29.
```
-c0 3 -eval rwrite(poisson_dist(c0,c0))
```

```
-c0 10 -eval rwrite(sqrt(c0)*poisson_dist(c0,c0))
```

And both similarly for other values of c0.

30.
```
-c0 20 -eval
until(x0:=x1;x2:=poisson_dist(c0,r0),r0+=2;x1:=x0+x2;real_eq(x0,x1))
;rwrite(x0)
```

31.
```
-exact -probability -statistics -c0 22
!different(pattern_list[c0]rand365)
```

And similarly for other values of c0.

32.
```
-probability -statistics -confidence 99.99 -c0 252
any(select[c0](unit365)) 100000000
```

And similarly for other values of c0.

34.
```
-statistics -c0 363 c0*(365-count_diff([c0]dz365)) 10000000
```

And similarly for other values of c0.

35.
```
-histogram -c0 1000 xwhile[c0](r0>=0,r0+=(dz3?1:-1),1) 10000000
```

36.
```
-u0 {{0,2,1,0},{1,0,0,2},{0,0,3,0},{0,0,0,3}} -eval
v0:=mmat_absorb_states(u0);r0:=last(dmat_row(,v0));r1:=first(
dmat_rsum(v0));write_ratio(r01);space(2);rwrite_ratio(r01)
```

37.
```
-c0 40 -c1 20 -c2 9 -c3 10 -c4 c0+1 -c5 c23 -u0
ccreate_mat[c4][c4](l0==0?l1==0??c5:l0==c0?l1==c0??c5:l1==l0+1?c2:
l1==l0-1??c3) -eval
v0:=mmat_absorb_states(u0);r0:=last(dmat_row(c1,v0));r1:=get(c1,
dmat_rsum(v0));write_ratio(r01);space(2);rwrite_ratio(r01)
```

39.
```
-real_stats real_result(min(xuniform,xuniform,xuniform)) 10000000
```

40.
```
-exact +noexact -statistics
find_ne(permute52from[copy4(unit13)],0)+1
```

42.
```
-real_stats x0:=uniform;real_result(min(x0,1-x0)) 10000000
```

```
-real_stats x0:=uniform;x1:=min(x0,1-x0);real_result(x1/(1-x1))
10000000
```

43.
```
-real_stats +fraction 50
x0:=xuniform;x1:=xuniform;xsort01;x2:=1-x1;x1-=x0;xsort012;
real_result(x0) 10000000
```

```
-real_stats +fraction 50
x0:=xuniform;x1:=xuniform;xsort01;x2:=1-x1;x1-=x0;xsort012;
real_result(x0) 10000000
```

```
-real_stats +fraction 50
x0:=xuniform;x1:=xuniform;xsort01;x2:=1-x1;x1-=x0;xsort012;
real_result(x0) 10000000
```

44.
```
-exact -probability -statistics -c0 9 -c1 20 -c2 2
sum(selection_list[c2]from[rstep[c1](c0)])>c2/2
```

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

45.
```
-statistics count(shuffle(sequence52)==sequence52) 10000000
```

46.
```
-histogram count(shuffle(sequence52)==sequence52) 10000000
```

47.
```
-probability -statistics -c0 100 -c1 35
v0:=shuffle(xsequence[c0]);r0:=max(head[c1](v0));v1:=tail[c0-c1](v0
);r1:=find_gt(v1,r0);e1==c0 100000000
```
And similarly for other values of c1.

50.
```
-probability -statistics -b0 1
x0:=2*b0*uniform-b0;x1:=2*b0*uniform-b0;real_ge(x0*x0,x1) 10000000
```
And similarly for other values of b0.

51.
```
-probability -statistics -c0 10000
xuntil[c0](dz2?dz2?incr0:decr0:dz2?incr1:decr1;r2:=r0==0&r1==0,r2,0)
1000000
```
And similarly for other values of c0.

53.
```
-probability -statistics -b0 1 -b1 0.5
x0:=b0*uniform;x1:=pi*uniform-pi/2;real_lt(x0,b1*cos(x1)) 10000000
```

54.
```
-statistics -b0 0.5
x0:=uniform;x1:=uniform;x2:=pi/2*xuniform;(real_lt(x0-b0*cos(x2),0)|
real_gt(x0+b0*cos(x2),1))+(real_lt(x1-b0*sin(x2),0)|real_gt(x1+b0*
sin(x2),1)) 10000000
```

55.
```
-statistics -b0 3
f0[floor(n1)-floor(n0)];x0:=uniform;x1:=uniform;x2:=pi/2*xuniform;f0
(x0-b0*cos(x2),x0+b0*cos(x2))+f0(x1-b0*sin(x2),x1+b0*sin(x2))
10000000
```