Twenty Problems in Probability

Solutions and illustrations using the program *Monaco*

Although principally intended for application to probability problems arising from games, the program *Monaco*[1] can be applied to a wider range of probability problems. This document describes its application to all but one of the problems in *Twenty Problems in Probability*, `https://www.math.ucdavis.edu/~gravner/MAT135A/resources/chpr.pdf`.

Using the program is not the ideal way to address those problems. The indicated reference gives analytically derived solutions to the problems, demonstrating that, at least for those problems, there is a better approach, providing not just solutions but insight into the problem. Furthermore, the program is completely unsuited to even illustrating one of the problems, and for many of the problems the program can only be used to produce solutions for selected values of a parameter *n*, not a proved result for all *n*, although in most cases results for more values of *n* than are given here can be produced. In addition, for most of these problems the program can only produce approximate solutions, although usually good ones. In fact, all twenty problems exhibit at least one of these limitations – cannot be solved, can only be solved approximately, or can only be solved for specific values of *n*.

Nevertheless, despite these limitations, there is possible value both in showing how such problems can be solved if you are not able to do so analytically – and the solutions to many of those problems are not easy, and might not be possible for many other problems – and to verify the given solutions, for at least some values of *n* (and in one case, to confirm the interpretation of the problem). In other cases the program can illustrate the problem, or its analytic solution, even if the program cannot be used to solve the problem. In one example – problem 16 – the problem has been extended beyond the version in the given reference; an additional reference that provides a solution to the extended problem is provided.

For each problem – other than the one exception – this document describes how to build up the program's main expression that can be used to solve or illustrate the problem, together with any required options. A summary of these options and expression, using the program's option `+parameters`, for each problem is presented at the end of this document.

Note that the solutions in the reference are misnumbered from question 8 onwards. The problem numbering is used here.

1. We can use the program to produce an approximate mode solution – the problem is much too large for an exact solution – by proceeding passenger by passenger following the rules.

   We start by parameterising the problem with `-s0 100`, the number of passengers and their seats. We number the passengers `0`, …, `s0-1`, and record their seat states in the corresponding elements of `v0` as either zero for occupied or containing the required passenger number plus one if unoccupied. We thus start with `v0:=xsequence`.

   We put the first passenger in a random seat with `r0:=dz[s0];e0:=0`. We then loop through the passengers `r1` = `1`, …, `s0-1` using `loop0(s0-1,r1:=r0+1,`*term*`)` where *term* seats the passenger and is `r2:=(e01?r1:bselect(v0)-1);e02:=0`. In that

[1] See `http://www.mnemosyne.uk/monaco`. The results in this document were created using version 2.34 of the program.

term, `e01` is true if the passenger's seat is unoccupied. If it is unoccupied it is selected, otherwise a random non-empty seat is selected, subtracting one from the recorded value to convert it back to a seat number. The selected seat number in either case is assigned to `r2`, and that seat is then set to occupied by `e02:=0`. We finish the expression with `r2==r1`, checking if the last passenger got the right seat.

Output for ten million results in approximate mode using the options `-probability -statistics` is:

```
Number of results          = 10000000
Number of false results    = 5000654
Number of true results     = 4999346
Probability                = 0.499935
95% confidence interval    = [0.499625, 0.500244]
```

This is consistent with the given result of ½.

2. The first part of this problem does not need the program, it is straightforward to consider that the probability that the next elevator is a down elevator is the probability that the elevator is above the 13th floor, a probability of 2/14 = 1/7. We could simulate this part of the problem using approximate mode – this problem requires uniformly distributed real numbers for the elevator position and thus cannot use exact mode – or, as is done below, simulate it as the case *n* = 1 of the second part of this problem.

   The second part of this problem can be simulated in approximate mode. We renumber the floors 0 to 14 and parameterise using `-c0 14 -c1 12` and for the first part of the problem, `-c2 1`, where `c0` is the number of floors, `c1` is Mr. Smith's floor number and `c2` is the number of elevators. For each elevator in turn we let its initial position be `x1:=c0*xuniform` and its state be `r1:=dz2`, where the state is true if going down, false if going up. We put these together as $term_1$ is `x1:=c0*xuniform;r1:=dz2`. Then the time `x2` for that elevator to reach floor `c1` – we assume a time unit of one per floor – and the state `r2` when it reaches floor `c1` are given by the term $term_2$, which is `r2:=real_gt(x1,c1);x2:=(r2?r1?x1-c1:2*c0-c1-x1:r1?x1+c1:c1-x1)`. Note that the calculation of `x2` considers the four cases defined by `r1` and `r2` separately.

   We let `x0` be the time for the first elevator to reach floor `c1`, which is a minimum of `c2` values and we can thus initialise as its maximum possible value of `2*c0`. The state that elevator is in when it reaches floor `c1` is `r0`, which does not need initialising as there will be an elevator with time less than `2*c0`. Then we want the state of the elevator for the first elevator. This can be found by looping over all `c2` elevators, using the expression `x0:=2*c0;do[c2](`$term_1$`;`$term_2$`;real_lt(x2,x0)&(x0:=x2;r0:=r2));r0`. This updates `x0` and `r0` for each elevator if earlier than the first so far – always for the first elevator due to the initial value of `x0`.

   We then use the options `-probability -statistics`, and the output for ten million evaluations in approximate mode for a single elevator is:

```
Number of results          = 10000000
Number of false results    = 8573336
Number of true results     = 1426664
Probability                = 0.142666
95% confidence interval    = [0.14245, 0.142883]
```

This is consistent with the given result of 1/7, or about 0.142857.

If we next try 2 elevators, i.e. `-c2 2`, then the last two lines above become:

```
Probability                = 0.24473
95% confidence interval    = [0.244464, 0.244997]
```

This is consistent with the given result of ½{1 − (10/14)^n} for n = 2, or 12/49, or about 0.244898.

3. We can simulate this problem in approximate mode; it is much too large for a direct solution in exact mode. The problem is not entirely clear, but based on the solution given, the interpretation of this problem is that after the first round, we are betting on a winner of each game not knowing who won the previous games. Presumably we assume our previous bets won (although this makes no difference to the mean).

   It is convenient to assume that, rather than betting at random, we always bet on the first player, and that the players are randomly ordered. We use the ordering such that if we have a list of length 2^m, player 0 plays player 2^{m-1}, player 1 plays player 2^{m-1}+1 etc. That list of 2^m will contain 0s and 1s, according to whether we did or did not successfully bet on that player up to this point. We then set the second half of the list to 0s, as we are no longer betting on those players, and then create a new list of length 2^{m-1} where the new element 0 is randomly either the old element 0 or the old element 2^{m-1}, which is 0, and so on. Thus at each stage we halve the length of the list and randomly set each element to zero with probability ½, because with randomly ordered players each game result is random with probability ½. Our score for each round is then the points for this round times the number of remaining 1s.

   Implementing this, at each stage the list has length `r1` and consists of the first `r1` elements of `v0`, i.e. `w01`, which is initialised to all 1s with `v0:=1`. We update `w01` by `w01&=repeat{dz2}`, the number of successes is `count(w01&=repeat{dz2})`. We let `c0` be the number of rounds, here 6, and set `s0:=pow(2,c0-1)`. We initialise `r1` to `s0`, and at each stage our score is `(s0/r1)*count(w01&=repeat{dz2})`, which we add to a running total `r0` before halving `r1`. The final result is `r0`.

   Assuming that we set `c0` using the option `-c0 6`, we have the final expression:

```
s0:=pow(2,c0-1);v0:=1;r1:=s0;
do[c0](r0+=(s0/r1)*count(w01&=repeat{dz2});r1/=2);r0.
```

   With `-statistics` the output for ten million evaluations in approximate mode is:

```
Number of results          = 10000000
Mean                       = 31.4974
Standard deviation         = 13.4405
Standard deviation of mean = 0.00425027
95% confidence interval    = [31.489, 31.5057]
Minimum result             = 2
Maximum result             = 160
```

   This is consistent with the given result of 31.5.

4. This problem can be solved using the program, although this solution is based on the given solution rather than having been independently created. We thus assume here that the

given solution is understood. However, to provide some added value, this solution then demonstrate its use.

The solution (before use) is in two stages. First we determine the value after each game, and before the first game. Second we determine the bet before each game. We record each of those two values after *n* games in the list `v$` where `$` is *n*, i.e. `v0` to `v7`, although we do not need `v7` for the betting. There are 2^n possible states after *n* games (we include cases that do not occur in practice as one team having won 4 games play stops). We thus use those `v$` with length $2^{\$}$, and can set those lengths with the option `-s01234567 geometric8(1,2)`. Each state is a 7 bit number, 1 for a win by the backed team, with the most recent result as the least significant bit.

The first stage, setting the values, proceeds from `v7` to `v0`. `v7` is initialised by `vloop_set7(2*c0*(bitcount(r7)>3))`, where `c0` is the initial stake, 100000, equal to $1000 in cents. (We use the observation in the given solution that no fractional cents occur.) This includes setting cases that do not occur, as noted above, but that has no effect.

The remaining `v$` are then initialised one from the last. We use the function `g0` to do this, so, we initialise `v6` from `v7` using `v6:=g0(v7)`, and we continue similarly up to `v0:=g0(v1)`. Internally, `g0` uses the list `v9`, which is a fixed length, the maximum needed is set by `-s9 s6`. `g0` only sets the required elements of `v9`, so we could finish `g0` with `set_size(head(v9),size(q0)/2)`, but as we will reuse that, we make that the definition of another function `g9`, and we finish `g0` with `g9(q0)`. `v9` is set in `g0` by `r9:=size(q0)/2;[wloop_set9(sum(sget2(r9,q0))/2)]`, where the ordering of the games from most recent ensures that the values to be summed are adjacent and `sget` can be used.

Determining each set of bets from the values, and overwriting `v0` to `v6` (we do not need `v7`) proceeds in that order. We use a similar function `g1` to determine what to subtract from `v0` to `v6`, using `v0-=g1(v1)` up to `v6-=g1(v7)`. `g1` is identical to `g0`, except that it uses `min` rather than `sum`, and that is not divided by two.

We put the initialisation of `v0` and the seven uses each of `g0` and `g1` together using the option `-eval`. After this we have `v0` to `v6` with the required betting patterns. We can then use a second `-eval` option to report `v0` to `v1`, using `vwrite0` to `vwrite6` with `blank` to separate outputs. For example `v0` is `{31250}`, meaning that we always bet $312.50 on the first game, while v2 is {25000,37500,37500,25000}, which says that if we have lost two games we bet $250, if we have won one and lost one (in either order, which must be the case) we bet $375, if we have won two we bet $250.

A summary of the results, based on the current score is:

0-0 bet $312.50

1-0 or 0-1 bet $312.50

2-0 or 0-2 bet $250, 1-1 bet $375

3-0 or 0-3 bet $125, 2-1 or 1-2 bet $375

3-1 or 1-3 bet $250, 2-2 bet $500

3-2 or 2-3 bet $500

3-3 bet $1000

So now we can demonstrate this in action. (We could simplify, knowing that the order of wins and losses is not relevant, but do not do that here. A series is run as:

```
r5:=c0;until(r4:=f0;r1*=2;r1+=r2:=dz2;r5+=r2?r4:-r4,
incr0==7|(r3+=r2)==4)
```

where `f0`, to be described, gives the required bet `r4` based on `r0` games with state `r1`, `r2` is the game result and `r3` is the match result so far, and `r5` is our stake plus winnings minus losses. `f0` is then `get(r0,{e01,e11,e21,e31,e41,e51,e61})`.

We could run this – slightly modified using `xuntil` instead of `until` to limit the loop – in exact mode, but is more satisfactory a demonstration in approximate mode. A histogram over ten million series produced a final balance of 0 in 4999908 series and 200000 (i.e. $2000) in 5000092, which are as expected.

5. This problem is not suitable for the program, but we can implement the given solutions exactly, in the second part of the problem for any reasonable number of players.

   In the first part, we set the hat colours as `v0:=3dz2`. Each player `r0`, from 0 to 2, can set `r1:=sum(v0)-e0` and sees the same colour if `r1!=1`, the colour seen being `r1/2`. The player then makes a successful guess if `r1/2!=e0`. We thus have a single success if `v0:=3dz2;rcount0(3,r1:=sum(v0)-e0;r1!=1&r1/2!=e0)==1`.

   The exact mode `-probability -statistics` output is:

```
Number of results          = 8
Number of false results    = 2
Number of true results     = 6
Probability                = 0.75 = 3/4
```

   This is the given result.

   In the second part, we let `s0` be *n*, the number of players. We let the hat colours be `v0`, given by `v0:=repeat{dz2}`. Then player `r0`, counting from zero, looks at lower numbered player hat numbers, from 0, inclusive, to `r0`, exclusive, any whether any are true. This is given by `any(w0)`, as although this checks all `s0` elements of `w0`, not just `r0` elements, the last `s0-r0` elements of `w0` are all zero and thus cannot contribute to the result of `any(w0)`. If the bet is made, whether it is a success is determined by `e0`, so the bet, if any, made and its outcome are thus given by `any(w0)?0:e0?r1:-r1`, where `r1` is the bet level at player `r0`. We could use `r1:=pow(2,r0)`, but instead initialise `r1:=1` and update `r1` at the end of the loop for `r0` by `r1*=2`.

   We want to accumulate bets, so the obvious loop to use is `rsum0`. However, this means that the result of the calculation of the bet must be the result of the loop term, but we need `r1*=2` to happen after that. We can make those both happen by making the loop term `return0(any(w0)?0:e0?r1:-r1,r1*=2)`, which we call *term*. Putting all of that together, except keeping *term* as that notation, the expression is given by `v0:=repeat{dz2};r1:=1;rsum(s0,`*term*`)>0`, the final `>0` being because we are counting successful bets, not their size. We assume `s0` is set by an option, and here we use the example `-s0 5`.

The output from `-probability -statistics` for that example in exact mode is:

```
Number of results          = 32
Number of false results    = 1
Number of true results     = 31
Probability                = 0.96875 = 31/32
```

This is the given result $1 - 2^{-n}$, which in this case, for $n = 5$, is 31/32.

6. This problem is not suitable for the program, but we can implement an example of the given solution, in approximate mode due to the distributions used.

   We use `r0` and `r1` for *X* and *Y*. We need example distributions for *X* and *Y*, i.e. for `r0` and `r1`, which could be different, but here we let them both be `random(c0)` for `c0` defined by an option, here we use `-c0 100`. However, we cannot have *X* = *Y*, and so we set `r0` and `r1` using `until(r0:=random(c0),r1:=random(c0),r0!=r1)`. This is not possible in exact mode; we could produce an alternative that is, but as the rest of the problem is not possible in exact mode we do not bother.

   We then reveal *X*, and can choose whether to accept *X* as larger according to a test using *X* and a random variable *G*, and for the given solution we let *G* be `neg_exp`, which as indicated cannot be used in exact mode, with the test whether to accept *X* being *X* > *G*, i.e. `real_gt(r0,neg_exp)`. We are successful if that term equals `r0>r1`.

   Ten million results in approximate mode have `-probability -statistics` output:

```
Number of results          = 10000000
Number of false results    = 4846155
Number of true results     = 5153845
Probability                = 0.515385
95% confidence interval    = [0.515075, 0.515694]
```

   As the given result tells us to expect, this is, with a very high confidence, greater than one half. Experimentation shows that it gets closer to one half as `c0` increases.

7. This problem is not suitable for the program, but we can use it to provide an illustrative example of the behaviour the problem is about. The example here can use exact mode, but many other possible illustrations would require approximate mode.

   We let *n* be `c0` and *k* be `c1`. We will use distributions, and will let the initial distribution of the p_i, with equal probabilities of all days, be `u0:=1[c0]`. We can create a sorted list – because we do not care about order – of the *k* birthdays using the weighted term `selection_list[c1]by[u0]`, and can check whether all of the birthdays are different by using `different(selection_list[c1]by[u0])`. As an example we let *n* = 10 and *k* = 5, i.e. we use the options `-c0 10 -c1 5`.

   The exact mode output with `-probability -statistics` is:

```
Number of evaluations      = 2002
Number of results          = 100000
Number of false results    = 69760
Number of true results     = 30240
Probability                = 0.3024 = 189/625
```

Now we try slightly changing that distribution to `u0:={3,5}#4[c0-2]`, after noting that the previously used `u0:=1[c0]` is the same distribution as `u0:=4[c0]`, and that this is a small change to the latter.

The output now becomes:

```
Number of evaluations      = 2002
Number of results          = 102400000
Number of false results    = 71864320
Number of true results     = 30535680
Probability                = 0.2982 = 1491/5000
```

As the given result tells us to expect, there is now a small reduction in the probability of all different birthdays.

8. We can produce a solution to the problem using the program, but we must use approximate mode because of the use of the uniform real distribution. This means that expressing the result in the form $r+s\pi$ is not possible, the best that can be done is to determine closeness of an approximate result to a given result of that form.

   A suitable expression is `round(uniform/uniform)%2==0`. Ten million results in approximate mode with the options `-probability -statistics` gives the output:

   ```
   Number of results          = 10000000
   Number of false results    = 5351143
   Number of true results     = 4648857
   Probability                = 0.464886
   95% confidence interval    = [0.464577, 0.465195]
   ```

   This is consistent with the given result of $(5-\pi)/4$, which is about 0.464602.

   We could instead start from the knowledge that the result is $(5-\pi)/4$, and use that simulation to estimate π. To do this we replace `-probability -statistics` with the option `-output %~r[5-4*mean]`, and get the estimate `3.14046`. This is not a very good estimate of π, but it is good to three significant figures. To improve it by one more significant figure would take a run one hundred times longer. It would be possible to extend this example to also output a confidence interval for that value of π, to show that about three significant figures is expected, but this has not been done here.

9. We can implement this problem using the program by proceeding as follows. We loop through the $2n$ ends of the strings, which we can number from zero, proceeding along joined strings, restarting when we have made a loop. Each odd numbered end will be the other end of the string whose first end is the preceding even numbered end. This can then be tied to either the one preceding free end, or to a following random end. If this is the k-th odd end, again counting from zero, i.e. end $2k+1$, then there will be $2n-2k-2$ following ends, so the probability of connecting to the preceding free end is $1/(2n-2k-1)$. If we do so connect, we add one to the count of loops. Note that at an even numbered end we either take the other end of a known string, which is always possible, or a new random free end, which is also always possible. So we only need to do any work for odd ends. This then directly leads to the given solution, without needing the program, but here we consider it using the program. Note that we do not need to use the end numbers to implement this solution.

Assuming that we set `c0` to *n* using the option `-c0` *n*, then we can implement this solution using the expression `rloop1(c0,random(2*(c0-r1)-1)|incr0);r0`. Here `r1` is *k* and `r0` is the count of loops. The term `random(…)` is zero with the required probability of completing a loop, and `r0` is increased by one when we do that, i.e. when that term is zero.

We can run that in approximate mode – see below for why, and how to improve on this – for ten million results with `-c0 3` with output from `-statistics` to get:

```
Number of results          = 10000000
Mean                       = 1.53354
Standard deviation         = 0.618469
Standard deviation of mean = 0.000195577
95% confidence interval    = [1.53315, 1.53392]
Minimum result             = 1
Maximum result             = 3
```

The given result is 1+⅓+⅕ = 23/15, or about 1.53333, and the above result is consistent with that. However, we can do better. We cannot run that expression in exact mode due to the use of the function `random` with a non-constant argument, but we can use exact mode if we use the randomness pool and replace `random` by `pool_dz`. A suitable size pool can be set by using `pool_set[semifactorial(2*c0-1)]`.

This then gives us the output:

```
Number of evaluations      = 15
Number of results          = 15
Mean                       = 1.53333 = 23/15
Standard deviation         = 0.618241
Minimum result             = 1
Maximum result             = 3
```

We can get exact answers for larger values of *n*, but only up to a point. For example for *n* = 10 the solution:

```
Number of evaluations      = 654729075
Number of results          = 654729075
Mean                       = 2.13326 = 31037876/14549535
Standard deviation         = 0.961527
Minimum result             = 1
Maximum result             = 10
```

took my computer nearly 3 minutes. Above about that point we need to revert to approximate mode. Of course using the given solution, exact values for any plausible *n* are easily calculated directly.

10. This problem is not suitable for the program, which cannot ever determine limiting values, but sometimes, as in this problem, we can show behaviour consistent with a limiting value. Here, we have the additional restriction that we can only determine real values, we have no way to estimate the required integer values.

    We can determine the value of $\sqrt{n}\,p_n$, even for very large values of *n*, in exact mode by using the weighted term `total2d[n]`, as that requires only 2*n*-1 evaluations, instead of, for example, requiring ½*n*(*n*+1) evaluations when using the otherwise equivalent term

`sum(sorted2d[`*n*`])`. We can test whether `r0:=total2d[`*n*`]` is a perfect square by using `r1:=sqrt(r0);r1*r1==r0`. However, we do not want the probability of that being so, but $\sqrt{n}$ times that probability. We can report that scaled probability by using the option `-output %~r[sqrt(c0)*mean]`. It would be possible to also output a confidence interval for that value, but this has not been done here.

As usual we use `-c0` *n* to set the required parameter. The output for *n* = 1,000,000 is `0.552284`, and for *n* = 10,000,000 it is `0.552285`, which suggests – although it does not prove – that the limit is very close to the latter value. The given solution tells us that the limit is $4(\sqrt{2}-1)/3$, which is about 0.552285, and our results are indistinguishably (at the reported precision) close to that.

11. This problem is not suitable for the program, but we can use it to implement the given solution. This must be in approximate mode both due to the indefinite number of coin tosses required and the particular use of real numbers.

    We use that we can set a variable to a random value in an option, and that it has that single value for all evaluations of the main expression. We will use `x0` for the intended probability α that Alice wins, and `x1` for the probability *p* that the coin comes up heads. We initialise these with `-x0 uniform -x1 uniform`. Here we use what the default random number generator gives us for `x0` and `x1`, which are respectively about 0.670627 and about 0.10076. We could instead pick any numbers – and then we could instead use the constants `b0` and `b1` – or we could use the option `-new` to get new numbers each run. But using `x0` and `x1` as described above provides a suitable example here.

    Each unfair die roll is the result of `f0[rbernoulli(x1)]`. The fair dice roll is then the result of `f1[until(r8:=f0;r9:=f0,r8!=r9);r8<r9]`. Rolling until we get a head and counting rolls is the result of `f2[until(incr7,f1);r7]`. The `p0`-th fractional bit of `x0` is the result of `f3[floor(pow(2,p0)*x0)%2]`. With those definitions the rest of the main expression is simply `f3(f2)`.

    For ten million results in approximate mode, using `-probability -statistics`, the output is:

```
Number of results          = 10000000
Number of false results    = 3294584
Number of true results     = 6705416
Probability                = 0.670542
95% confidence interval    = [0.67025, 0.670833]
```

    This is consistent with the intended result of 0.670627.

12. This problem is not suitable for the program, nor is implementing the given solution.

13. We can implement the problem using the program. This must be in approximate mode, because of the indefinite number of visits to the numbers.

    Using the option `-s0` *n*, we use the list `v0` to record whether a number has been visited or not. Using `r0` for the walker's current number, which by default starts at zero, we initialise `v0` as its default zeros plus `e0:=1`. We then continue until all numbers are visited, using `until(`*term*`,all(v0))` for a *term* that needs to change `r0`, and record

the visit, which can be simply `r0+=ds2;e0:=1`. It is not necessary to add `r0%=s0`, to set `r0` to the actual number, as that modular calculation is performed by `e0:=1`.

The result of interest is whether each number was the final number visited, which is `r0`, modulo *n*, after the loop. To collect multiple statistics we can use list results, and the required list result can be added using `list_result(unit[s0](r0))`. Again, we do not need to apply a modulo *n* adjustment to `r0`, as that is also done by the function `unit[s0]`.

The required output uses the options `+probability -list_stats` and, for ten million results in the necessary approximate mode, due to the indefinite number of random values needed, with *n* = 5 is:

```
Number of list results      = 10000000
List false result numbers   = {10000000,7502047,7500010,7498414,7499529}
List true result numbers    = {0,2497953,2499990,2501586,2500471}
List probabilities          = {0,0.249795,0.249999,0.250159,0.250047}
List 95% conf intervals     = {[0,4.63705e-07],[0.249527,0.250064],
                               [0.249731,0.250267],[0.24989,0.250427],
                               [0.249779,0.250316]}
```

This is consistent with the given result that the probabilities are 1/(*n*-1) = ¼ for all numbers other than zero.

14. This problem is not suitable for the program. We can use it to provide approximate answers to some probabilities p_n, and thus determine some approximate values of $p_n^{1/n}$, but – even apart from the usual problems in assessing limiting behaviour – here as *n* increases p_n gets smaller, there are few positive examples, and the estimates of p_n and $p_n^{1/n}$ have increasingly large uncertainty until, for any reasonable size run, there are no true results to use to estimate p_n as anything other than "about zero", and $p_n^{1/n}$ becomes completely uncertain. To demonstrate this, and see if anything can be deduced about the solution, we produce some results, necessarily in approximate mode, for *n* = 10, *n* = 20 and *n* = 30, parameterised as usual with `-c0` *n*.

    A suitable expression to simulate the problem is:

    ```
    x1:=uniform;do_all[c0-1](x0:=x1;x1:=uniform;real_le(x0+x1,1))
    ```

    Note that we do not need to create all the random variables at once, but only as needed – the loop terminates as soon as a loop term is false. For each evaluation of the loop term, `x0` is X_i and `x1` is X_{i+1}. For the output, we can use the option:

    ```
    -output %~r[mean]%_%~r[pow(mean,1/c0)]%_
    [%~r[pow(mean_bound(0),1/c0)],%_%~r[pow(mean_bound(1),1/c0)]]
    ```

    This option outputs four values: estimates of p_n, $p_n^{1/n}$, and a confidence interval for the latter. This output for ten million results with *n* = 10, *n* = 20 and *n* = 30 is:

    ```
    0.013886 0.652015 [0.651675, 0.652356]

    0.0001528 0.644475 [0.642861, 0.646093]

    1.1e-06 0.632965 [0.619871, 0.645669]
    ```

    The last of these shows only 11 positive results, and the confidence interval is significantly wider. No definite conclusion can be drawn from these results, although we might have guessed that the limit is about 0.64, but it could be lower. The given result is that the limit

is $2/\pi$, which is about 0.636620, which means that the guess would have been right, but of low precision.

15. We cannot solve this problem in general using the program. We can implement it exactly, and thus confirm the claim made, but only for small values of *n*.

    We start with the usual `-c0` *n* and then determine *A*, *B* and *C* as `r0`, `r1` and `r2` by using the term `do[c0](v0+=shuffle(xsequence3));r012:=sort(v0)`. To then determine the two required probabilities, a_n and b_n, we produce the two corresponding events using the terms `r1==r0&r2==r1` and `r1==r0+1,r2==r1+1`, which we refer to as $term_a$ and $term_b$. We can determine the probabilities of these two events in the same run by using `list_results(`$term_a$`,`$term_b$`)` and their list statistics.

    To assess if the required tests are always true we need to use exact results, which we can do for small values of *n*. We can report whether the test $4a_n \le b_n$, is true using the option `-output %[real_le(4*list_mean(0),list_mean(1))]`. [There is a possible issue here with the use of real numbers and rounding if the equality case occurs, which we could correct for, but have not done so here, as it does not appear to be needed.]

    Rather than attempting both tests $4a_n \le b_n$ and $4a_{n+1} \le b_{n+1}$ in the same run, we just run a sequence of runs for all possible values of *n*. The use of exact mode means that the time for each run increases significantly with increasing *n*, so we here consider only $n = 1$ to $n = 10$. These all report $4a_n \le b_n$ except for when $n = 2$, which means that we have tested the indicated result up to $n = 10$. (As $4a_n \le b_n$ for $n = 10$, we do not need to test $n = 11$ to confirm the result for $n = 10$.) However, testing any further is increasingly impractical, and soon impossible, in this manner. We could possibly use approximate mode to investigate higher values of *n*, and even use confidence intervals to give better substance to any tests, but this has not been attempted here, and its success would depend on whether we ever have $4a_n = b_n$, which we could never be certain about. We thus do not know if there are any other cases like $n = 2$ where it is necessary to allow for the case $4a_{n+1} \le b_{n+1}$.

16. This problem can be approximately, but not exactly, solved by the program. It stretches the program, owing to that it has only ten real variables. (An alternative is used in the extension to the problem described below.)

    To solve the problem using just nine of those variables for coordinate values, we note that, without loss of generality, one of the four points on the sphere, *A*, can be assumed to be (1,0,0), the centre of the sphere being (0,0,0). The other three points can then be *B* (`x0`,`x1`,`x2`), *C* (`x3`,`x4`,`x5`) and *D* (`x6`,`x7`,`x8`), leaving `x9` as a working variable, which will be sufficient.

    To set *B*, *C* and *D*, we will have a function `f0` with no arguments that sets `x0`, `x1` and `x2` to a random point on the sphere; we will then copy `x0`, `x1` and `x2` to `x3`, `x4` and `x5`, thus setting *C*, repeat `f0`, copy `x0`, `x1` and `x2` to `x6`, `x7` and `x8`, thus setting *D*, and repeat `f0`, thus setting *B*. `f0` can be defined by picking a random point in the axis-aligned cube with volume 8 that the sphere, with volume $4\pi/3$, just fits into, repeating until the point is inside the cube (taking on average $6/\pi$, or slightly under two, attempts each), and scaling the point onto the sphere.

The function `f0` can be implemented by:

```
f0[until(x0:=2*uniform-1;x1:=2*uniform-1;x2:=2*uniform-1,
real_le(x9:=sum_sq(x0,x1,x2),1));x9:=sqrt(x9);x0/=x9;x1/=x9;x2/=x9]
```

Next, for each triangle, e.g. *ABC*, we determine on which side of it the origin *O* is. We can start with the normal vector to the triangle with magnitude its area – which we do not care about – as the cross-product of the vectors $\overrightarrow{AB}$ and $\overrightarrow{AC}$, or using **a**, **b**, and **c** for the position vectors of *A*, *B* and *C* relative to the origin *O*, (**b** - **a**) × (**c** - **a**). We then see how this is aligned with the vector from O to any of the points, for convenience *A*, where $\overrightarrow{OA}$ is simply **a**. We determine that from the scalar triple product **a**.(**b** - **a**) × (**c** - **a**) of the three indicated vectors. This is equal to the determinant of the 3 × 3 matrix whose rows are the coordinates of the three vectors **a**, **b** - **a** and **c** - **a**. We can create a function `m1` of nine real variables `n0` to `n8` to calculate the determinant of any 3 × 3 matrix, which we can conveniently do by using a function `m0` of four variables `n0` to `n3` that calculates the determinant of any 2 × 2 matrix, as `m0[n0*n3-n1*n2]`.

With that function `m0`, the function `m1` can be defined by:

```
m1[n0*m0(n4,n5,n7,n8)-n1*m0(n3,n5,n6,n8)+n2*m0(n3,n4,n6,n7)]
```

We will use `m1` four times, and can discard the magnitude of the result. We can also convert the sign to simply true or false. This can use the function `f1` whose arguments are the coordinates of the points *A*, *B* and *C* in the example above. `f1` can be defined by:

```
f1[real_gt(m1(n0,n1,n2,n3-n0,n4-n1,n5-n2,n6-n0,n7-n1,n8-n2),0)]
```

We do not care – we cannot easily know, it depends on the orientation of *A*, *B*, *C* and *D* – what the convention is for false and true, but we do need our four triangles traversed in the same direction so that each uses the same convention. We achieve that if when we pass along an edge of a triangle in one direction, we must pass along it in the opposite direction when traversing the other triangle that shares that edge. This can be done by using the triangles *ABC*, *DCB*, *CDA* and *BAD* , each in that order.

We thus use `f1` four times to get four results `r0` to `r3`:

```
r0:=f1(1,0,0,x0,x1,x2,x3,x4,x5);r1:=f1(x6,x7,x9,x3,x4,x5,x0,x1,x2);
r2:=f1(x3,x4,x5,x6,x7,x8,1,0,0);r3:=f1(x0,x1,x2,1,0,0,x6,x7,x8)
```

and the final result, for *O* to be inside the tetrahedron *ABCD*, is then `same(r0123)`.

The usual `-probability -statistics` output for ten million results in approximate mode, needed due to the real distributions used, is:

```
Number of results          = 10000000
Number of false results    = 8751132
Number of true results     = 1248868
Probability                = 0.124887
95% confidence interval    = [0.124682, 0.125092]
```

This is consistent with the given result of ⅛ = 0.125.

This problem can be extended to consider more points, and a general solution for *N* points is provided in a paper at `https://www.mscand.dk/article/view/10655`. That paper also generalises the problem to *n* dimensions but here we consider only the case *n* = 3. (It also generalises the point distribution, but with that of interest here included.)

The paper actually considers the probability that the *N* points lie within a single hemisphere, but that is equivalent to the case that the centre of the sphere is not within the polygon (or polytope in higher dimensions) defined by the *N* points (their convex hull) and thus the probabilities in that paper and the probabilities required here are complementary (sum to one).

As usual we use the option `-c0` *N*. Note that we must have *N* > *n*, i.e. here *N* ≥ 4.

We now have insufficient real variables, so we use the real array. To contain *N* points we initialise the real array size using the option `-rq 3*c0`, and to initialise the real array for each evaluation of the main expression we can use `f0` to set one point (`x0`, `x1`, `x2`) by:

```
f0[until(x0:=2*uniform-1;x1:=2*uniform-1;x2:=2*uniform-1;
x9:=sum_sq(x0,x1,x2),real_le(x9,1));x9:=sqrt(x9)]
```

Next we use `m0` to set the `p0`-th point in the real array by:

```
m0[f0;rqset(3*p0,x0/x9);rqset(3*p0+1,x1/x9);rqset(3*p0+2,x2/x9)]
```

We can set all the points in the real array using `rloop0(c0,m0(r0))`. Note that we do not need to use (1,0,0) as one of the points.

Now we will consider each possible set of three points by defining `v9:=sequence[c0]` and looping through all those combinations setting `r9:=3` and then the loop:

`wcombin_all9(r012:=head3(v9);v8:=tail(v9);`*term*`)`

which has put the point numbers into `r0`, `r1` and `r2` and the other point numbers into `v8`, where the length of `v8` and the to be used `v7` are set by `s8:=s7:=c0-3`. We then evaluate *term* for each of these, and we need all sets of points to produce a true *term*.

term implements an algorithm that considers those sets of three points as possible triangles on the outside of the polygon defined by the *N* points. (We can ignore the case where the polygon has any non-triangular faces as having probability zero in theory and negligible probability in practice.) We thus consider the orientation of the triangle relative to each other point – we will ignore this triangle if those are not all the same, as it is not an outside face. Each other point is then a loop through `v8`. We put those `s8` = `s7` orientations into `v7`. Our test thus passes if `!same(v7)`, otherwise a further test is needed. *term* thus takes the form `rloop8(s8,e78:=`*term*$_1$`);!same(v7)|`*term*$_2$.

We now use a function `f1` that has six arguments: three point indices `r0`, `r1` and `r2` that become `p0`, `p1` and `p2`, and the real coordinates of a point that become `n3`, `n4` and `n5`. This has the advantage that if used with only three arguments, the other arguments are zero, the centre of the sphere, which must have the same orientation as the other points. Thus *term*$_1$ is `f1(r0,r1,r2,rqget(3*e8),rqget(3*e8+1),rqget(3*e8+2)` and *term*$_2$ is `f1(r0,r1,r2)==first(v7)`.

There remains only to define `f1`. This uses `m2`, which in turn uses `m1`; these are renumbered `m1` and `m0` used above. `f1` is then:

```
f1[x0:=rqget(3*p0);x1:=rqget(3*p0+1);x2:=rqget(3*p0+2);
real_gt(m2(n3-x0,n4-x1,n5-x2,rqget(3*p1)-x0,
rqget(3*p1+1)-x1,rqget(3*p1+2)-x2,rqget(3*p2)-x0,
rqget(3*p2+1)-x1,rqget(3*p2+2)-x2),0)]
```

This uses the same algorithm as the original problem, where because the vector **a** is subtracted from each of the vectors **b**, **c** and the external point – previously just the origin – we now call **d**, we use (`x0`,`x1`,`x2`) for **a**. **a**, **b** and **c** are determined from the indices `p0`, `p1` and `p2` and **d** is (`n3`,`n4`,`n5`).

The output for ten million results with *N* = 4 is then:

```
Number of results          = 10000000
Number of false results    = 8750028
Number of true results     = 1249972
Probability                = 0.124997
95% confidence interval    = [0.124792, 0.125202]
```

This is again consistent with the previously given result ⅛.

We now consider *N* = 5 to *N* = 8, where the probability result lines become, in turn:

```
Probability                = 0.312347
Probability                = 0.499993
Probability                = 0.656225
Probability                = 0.773546
```

The first three can be recognised (adding the option `+fraction` can help) as close to $^{5}/_{16}$, ½ and $^{21}/_{32}$. The fourth is harder (and `+fraction` does not help) but if it is assumed that the result has a denominator of a power of 2, it is close to $^{99}/_{128}$. If making that assumption – which is true – adding the option `+pscale pow(2,c0-1)` can help here.

These results – or the suggested values – can be verified using the referenced paper. As noted above, its formulation has the complementary probability to that required here. Using *q* for this complementary probability, and *p* = 1–*q* for the probability required here, the paper shows that the complementary probability is given by:

$$q = 2^{-(N-1)} \sum_{k=0}^{n-1} \binom{N-1}{k}$$

For the case we are concerned with, for *n* = 3, this can be simplified to:

$$q = \frac{½N(N-1)+1}{2^{N-1}}$$

with the results:

N = 4	$q = {}^{7}/_{8}$	$p = {}^{1}/_{8} = 0.125$
N = 5	$q = {}^{11}/_{16}$	$p = {}^{5}/_{16} = 0.3125$
N = 6	$q = {}^{16}/_{32} = {}^{1}/_{2}$	$p = {}^{1}/_{2} = 0.5$
N = 7	$q = {}^{22}/_{64} = {}^{11}/_{32}$	$p = {}^{21}/_{32} = 0.65625$
N = 8	$q = {}^{29}/_{128}$	$p = {}^{99}/_{128} = 0.7734375$

The simulated results are consistent with these results.

17. This problem cannot be solved using the program for general *m* and *n*, but can be solved for specific *m* and *n*, even exactly as long as *mn* is small enough, for larger *m* and *n* we need to use approximate mode. As usual we use `-c0` *m* and `-c1` *n*.

    The solution is to create the grid as `v0` using `v0:=[c0*c1]dz2`, then create the grid components using `v1:=grid_comp(v0)` and then the number of components is `max(v1)+1`. The mean can be reported as usual using `-statistics`.

    Some example results, together with the given bounds are:

    m = 4, *n* = 4, mean = 5.1431, minimum = 2, maximum = 6.

    m = 4, *n* = 5, mean = 6.02819, minimum = 2.5, maximum = 7.

    m = 4, *n* = 6, mean = 6.91334, minimum = 3, maximum = 8.

    The last of these takes over 10 seconds on my computer. For larger grids here are some approximate results using ten million grids. In all cases not just the mean but all of the confidence intervals are within the given bounds.

    m = 5, *n* = 6, mean = 8.06154, minimum = 3.75, maximum = 28/3 = 9.33333.

    m = 6, *n* = 6, mean = 9.20708, minimum = 4.5, maximum = 32/3 = 10.6667.

    m = 7, *n* = 7, mean = 11.6347, minimum = 6.125, maximum = 13.5.

    m = 8, *n* = 8, mean = 14.325, minimum = 8, maximum = 50/3 = 16.6667.

    The last of these took over 20 seconds, so larger grids become impractical for the ten million results used here. However, in some of those cases fewer results could be used, while still having a confidence interval that is well within the given bounds.

18. This problem can be simulated by the program, requiring approximate mode due to the real distributions used.

    We define a point on the circle by its angle from (1,0). It is convenient to do this in rotations, hence in the range -½ to +½, and scale the result by π only at the end. Thus the three points have angles `x0:=uniform-0.5` and similarly for `x1` and `x2`. We can then sort these three values into ascending order using `xsort012`. There are then four cases. If `x0` > 0 then `x2` should be wrapped around to `x2-1` and zero is in the interval from `x2` to `x0`. If `x2` < 0 then `x0` should be wrapped around to `x0+1` and zero is again in the interval from `x2` to `x0`. Otherwise, if x1 > 0 then zero is in the interval from `x0` to `x1`, or if `x1` < 0 then zero is in the interval from `x1` to `x2`.

    Putting those together, the interval length, before scaling by `2*pi`, is:

    ```
    [real_gt(x0,0)|real_lt(x2,0)]?x0-x2+1:[real_gt(x1,0)]?x1-x0:x2-x1
    ```

    This, with the required scaling by `2*pi`, is then the argument of `real_result`.

    Output in approximate mode – required because we are using uniformly distributed real numbers – over ten million results using the option `-real_stats` is:

    ```
    Number of real results     = 10000000
    Mean                       = 3.14165
    Standard deviation         = 1.40523
    Standard deviation of mean = 0.000444373
    95% confidence interval    = [3.14078, 3.14252]
    ```

```
Skewness                    = 7.58094e-06
Minimum result              = 0.00162788
Maximum result              = 6.2824
```

This is consistent with the given result of π, or about 3.14159.

19. This problem can be solved exactly by the program for small values of *n*, the number of sock pairs. Larger – but still limited – values of *n* are considered below. For convenience, we modify the problem to number the socks from 0 to *n*-1 rather than 1 to *n*, which has no effect on the results.

    With the usual option `-c0` *n*, we can first set up a list whose consecutive pairs of elements are the sock pairs as `v0:=shuffle(copy2(sequence[c0]))`. Then we can set up a list of differences as `v1:=mat_col2(1,delta(v0))`. The latter works because it first finds a list of all the differences in `v0` (starting with the first element of `v0`) as `delta(v0)`. We only want the sock pair differences, elements 1, 3, 5, … of that list (counting from zero). We can get this by treating that list as a matrix with row length 2, and taking its last (i.e. second) column. Finally, our result is `all_le(abs(v1),1)`.

    Exact mode output from `-probability -statistics` for *n* = 5 is:

```
Number of results           = 3628800
Number of false results     = 3548160
Number of true results      = 80640
Probability                 = 0.0222222 = 1/45
```

    The given formula for the result is:

$$p_n = \frac{(2^{n+1} + (-1)^n)2^n n!}{3(2n)!}$$

    which evaluates to 1/45 when *n* = 5, so our result is as given.

    Taking the given expression beyond *n* = 5 is not practical because `shuffle` is an expensive (in terms of the number of evaluations required) function in exact mode, more so than necessary, because it is only necessary to select socks 1, 3, …, 2*n*-1 randomly, a total of (2*n*-1)!! possibilities (the semifactorial of 2*n*-1) rather than (2*n*)! possibilities. This can be accomplished in exact mode, but not by a single function replacement for `shuffle`. One possible approach would be to use a randomness pool of size `semifactorial(2*c0-1)`, which would increase the maximum practical value of *n* in exact mode. However, how much *n* could be increased by using this approach has not been determined, because this has not been done here – and neither has the alternative of switching to approximate mode been done.

20. We can solve this problem for individual values of *n* using the program, but only approximately due to the uniform distribution used, and not for all *n*.

    As usual we use `-c0` *n*, and in order to be able to handle examples with n > 5 we use the real array, which we initialise with the option `-rq 2*c0`. We then fill it with 2*n* uniform random values, except that for convenience each of the *n* interval bounds, in consecutive elements of the real array, is sorted with lower bound before upper bound. We can do this with:

```
rloop9(c0,x0:=uniform;x1:=uniform;xsort01;r0:=2*r9;^0:=x0;incr0;^0:=x1)
```

Note that `^0` is element `r0` of the real array.

We now assume a function `f1` with parameter `p0` that returns true if interval number `p0` overlaps all intervals – testing this can include itself, as an interval always overlaps itself. Then the remainder of the expression is `rany8(c0,f1(r8))` as we want to know if there is any interval that does this. We use `r8` to loop over the intervals because `r9` will be used in `f1`. (We could avoid this by using `rsave9` in `f1`, but we do not do that here.)

Now assuming a function `f0` with real parameters `n0`, `n1`, `n2` and `n3` that returns true if the intervals [`n0`, `n1`] and [`n2`, `n3`] overlap, `f1` can be defined by:

```
f1[r0:=2*p0;r1:=r0+1;rall9(c0,r2:=2*r9;r3:=r2+1;f0(^0,^1,^2,^3))]
```

Here the first interval is [`^0`, `^1`], determined from `p0`, and for the second interval we loop over all intervals – we want to know if we have an overlap for all second intervals – to determine the second interval [`^2`, `^3`] from the interval number `r9`.

There just remains `f0`, which can be defined by:

```
f0[real_lt(n0,n2)?real_ge(n1,n2):real_le(n0,n3)]
```

Putting that all together, for ten million results in approximate mode – we cannot use exact mode when using uniform random real numbers – the output using `-probability -statistics` is:

```
Number of results          = 10000000
Number of false results    = 3333849
Number of true results     = 6666151
Probability                = 0.666615
95% confidence interval    = [0.666323, 0.666907]
```

This is consistent with the given result of ⅔ for all *n*. We could use other values of *n* to demonstrate this independence, but that has not been done here.

Summary of the Program Parameters Used

The summary is output from the option `+parameters`, omitting its initial `Parameters:` line.

1. `-s0 100 -probability -statistics v0:=xsequence;r0:=dz[s0];e0:=0;rloop0(s0-1,r1:=r0+1;r2:=(e01?r1:bselect(v0)-1);e02:=0);r2==r1 10000000`

2. `-c0 14 -c1 12 -c2 1 -probability -statistics x0:=2*c0;do[c2](x1:=c0*xuniform;r1:=dz2;r2:=real_gt(x1,c1);x2:=r2?r1?x1-c1:2*c0-c1-x1:r1?x1+c1:c1-x1;real_lt(x2,x0)&(x0:=x2;r0:=r2));r0 10000000`

   and similarly with `-c2 2`.

3. `-statistics -c0 6 s0:=pow(2,c0-1);v0:=1;r1:=s0;do[c0](r0+=(s0/r1)*count(w01&=repeat{dz2});r1/=2);r0 10000000`

4. `-c0 100000 -s01234567 geometric8(1,2) -s9 s6 -g9 set_size(head(v9),size(q0)/2) -g0 r9:=size(q0)/2;[wloop_set9(sum(sget2(r9,q0))/2)];g9(q0) -g1 r9:=size(q0)/2;[wloop_set9(min(sget2(r9,q0)))];g9(q0) -eval vloop_set7(2*c0*(bitcount(r7)>3));v6:=g0(v7);v5:=g0(v6);v4:=g0(v5);v3:=g0(v4);v2:=g0(v3);v1:=g0(v2);v0:=g0(v1);v0-=g1(v1);v1-=g1(v2);v2-=g1(v3);v3-=g1(v4);v4-=g1(v5);v5-=g1(v6);v6-=g1(v7) -eval vwrite0;blank;vwrite1;blank;vwrite2;blank;vwrite3;blank;vwrite4;blank;vwrite5;blank;vwrite6`

   `-histogram -c0 100000 -s01234567 geometric8(1,2) -s9 s6 -g9 set_size(head(v9),size(q0)/2) -g0 r9:=size(q0)/2;[wloop_set9(sum(sget2(r9,q0))/2)];g9(q0) -g1 r9:=size(q0)/2;[wloop_set9(min(sget2(r9,q0)))];g9(q0) -f0 get(r0,{e01,e11,e21,e31,e41,e51,e61}) -eval vloop_set7(2*c0*(bitcount(r7)>3));v6:=g0(v7);v5:=g0(v6);v4:=g0(v5);v3:=g0(v4);v2:=g0(v3);v1:=g0(v2);v0:=g0(v1);v0-=g1(v1);v1-=g1(v2);v2-=g1(v3);v3-=g1(v4);v4-=g1(v5);v5-=g1(v6);v6-=g1(v7) r5:=c0;until(r4:=f0;r1*=2;r1+=r2:=dz2;r5+=r2?r4:-r4,incr0==7|(r3+=r2)==4) 10000000`

5. `-exact -probability -statistics v0:=3dz2;rcount0(3,r1:=sum(v0)-e0;r1!=1&r1/2!=e0)==1`

   `-exact -probability -statistics -s0 5 v0:=repeat{dz2};r1:=1;rsum0(s0,return0(any(w0)?0:e0?r1:-r1,r1*=2))>0`

6. `-probability -statistics -c0 100 until(r0:=random(c0);r1:=random(c0),r0!=r1);real_gt(r0,neg_exp)==(r0>r1) 10000000`

7. `-exact -probability -statistics -c0 10 -c1 5 u0:=1[c0];different(selection_list[c1]by[u0])`

   `-exact -probability -statistics -c0 10 -c1 5 u0:={3,5}#4[c0-2];different(selection_list[c1]by[u0])`

8. `-probability -statistics round(uniform/uniform)%2==0 10000000`

   `-output %~r[5-4*mean] round(uniform/uniform)%2==0 10000000`

9. `-statistics -c0 3 rloop1(c0,random(2*(c0-r1)-1)|incr0);r0 10000000`

   `-exact -statistics -c0 3 pool_set[semifactorial(2*c0-1)];rloop1(c0,pool_dz(2*(c0-r1)-1)|incr0);r0`

   and the latter similarly with `-c0 10`.

10. `-exact -probability -output %~r[sqrt(c0)*mean] -c0 1000000 r0:=total2d[c0];r1:=sqrt(r0);r1*r1==r0`

    and similarly with `-c0 10000000`.

11. `-probability -statistics -x0 uniform -x1 uniform f0[rbernoulli(x1)];f1[until(r8:=f0;r9:=f0,r8!=r9);r8<r9];f2[until(incr7,f1);r7];f3[floor(pow(2,p0)*x0)%2];f3(f2) 10000000`

12. The program is not used.

13. `+probability -list_stats -s0 5 e0:=1;until(r0+=ds2;e0:=1,all(v0));list_result(unit[s0](r0)) 10000000`

14. `-probability -c0 10 -output %~r[mean]%_%~r[pow(mean,1/c0)]%_[%~r[pow(mean_bound(0),1/c0)],%_%~r[pow(mean_bound(1),1/c0)]] x1:=uniform;do_all[c0-1](x0:=x1;x1:=uniform;real_le(x0+x1,1)) 10000000`

    and similarly with `-c0 20` and `-c0 30`.

15. `-exact -output %[real_le(4*list_mean(0),list_mean(1))] -c0 1 do[c0](v0+=shuffle(xsequence3));r012:=sort(v0);list_result{r1==r0&r2==r1,r1==r0+1,r2==r1+1}`

    and similarly with `-c0 2` to `-c0 10`.

16. `-probability -statistics m0[n0*n3-n1*n2];m1[n0*m0(n4,n5,n7,n8)-n1*m0(n3,n5,n6,n8)+n2*m0(n3,n4,n6,n7)];f0[until(x0:=2*uniform-1;x1:=2*uniform-1;x2:=2*uniform-1,real_le(x9:=sum_sq(x0,x1,x2),1));x9:=sqrt(x9);x0/=x9;x1/=x9;x2/=x9];f1[real_gt(m1(n0,n1,n2,n3-n0,n4-n1,n5-n2,n6-n0,n7-n1,n8-n2),0)];f0;x3:=x0;x4:=x1;x5:=x2;f0;x6:=x0;x7:=x1;x8:=x2;f0;r0:=f1(1,0,0,x0,x1,x2,x3,x4,x5);r1:=f1(x6,x7,x9,x3,x4,x5,x0,x1,x2);r2:=f1(x3,x4,x5,x6,x7,x8,1,0,0);r3:=f1(x0,x1,x2,1,0,0,x6,x7,x8);same(r0123) 10000000`

    `-probability -statistics -c0 4 -rq 3*c0 f0[until(x0:=2*uniform-1;x1:=2*uniform-1;x2:=2*uniform-1;x9:=sum_sq(x0,x1,x2),real_le(x9,1));x9:=sqrt(x9)];m0[f0;rqset(3*p0,x0/x9);rqset(3*p0+1,x1/x9);rqset(3*p0+2,x2/x9)];m1[n0*n3-n1*n2];m2[n0*m1(n4,n5,n7,n8)-n1*m1(n3,n5,n6,n8)+n2*m1(n3,n4,n6,n7)];f1[x0:=rqget(3*p0);x1:=rqget(3*p0+1);x2:=rqget(3*p0+2);real_gt(m2(n3-x0,n4-x1,n5-x2,rqget(3*p1)-x0,rqget(3*p1+1)-x1,rqget(3*p1+2)-x2,rqget(3*p2)-x0,rqget(3*p2+1)-x1,rqget(3*p2+2)-x2),0)];s8:=c0-3;rloop0(c0,m0(r0));v9:=sequence[c0];r9:=3;wcombin_all9(r012:=head3(v9);rloop8(s8,r7:=vget9(r8+3);e8:=f1(r0,r1,r2,rqget(3*r7),rqget(3*r7+1),rqget(3*r7+2)));!same(v8)|f1(r0,r1,r2)==first(v8)) 10000000`

    and the latter similarly with `-c0 5` to `-c0 8`.

17. `-exact -statistics -c0 4 -c1 4`
    `v0:=[c0*c1]dz2;v1:=grid_comp[c0][c1](v0);max(v1)+1`

    and similarly for other values of `c0` and `c1`.

18. `-real_stats`
    `x0:=uniform-0.5;x1:=uniform-0.5;x2:=uniform-0.5;xsort012;real_result`
    `(2*pi*([real_gt(x0,0)|real_lt(x2,0)]?x0-x2+1:[real_gt(x1,0)]?x1-x0:`
    `x2-x1)) 10000000`

19. `-exact -probability -statistics -c0 5`
    `v0:=shuffle(copy2(sequence[c0]));v1:=mat_col2(1,delta(v0));all_le(`
    `abs(v1),1)`

20. `-probability -statistics -c0 10 -rq 2*c0`
    `f0[real_lt(n0,n2)?real_ge(n1,n2):real_le(n0,n3)];f1[r0:=2*p0;r1:=r0+`
    `1;rall9(c0,r2:=2*r9;r3:=r2+1;f0(^0,^1,^2,^3))];rloop9(c0,x0:=uniform`
    `;x1:=uniform;xsort01;r0:=2*r9;^0:=x0;incr0;^0:=x1);rany8(c0,f1(r8))`
    `10000000`