102 Combinatorial Problems from the Training of the USA IMO Team

Solutions and Illustrations Using the Program *Monaco*

Introduction

This document describes how the program *Monaco*[1] can be used to address the problems in the book *102 Combinatorial Problems from the Training of the USA IMO Team* (T. Andreescu and Z. Feng), see
`https://www.tandfonline.com/doi/pdf/10.1017/S0025557200175692`.

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.

The purpose of this document is to investigate the applicability of Monaco to these problems. In some problems Monaco is an ideal tool for the task, in others it is less so, and in many it cannot be used at all. The best that can be done – or at least just the best this author can do – is presented, whether a full solution, a partial solution, an illustration of the problem, or simply no solution. However, if a solution is fast enough it is included, even if it could be improved to be faster. Most of the introductory problems have a solution, but most of the advanced problems do not. In many cases – even in 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" refer to one of the solutions to the problems that make up most of the book. However, the solutions in this document were created independently from those solutions. (In a few cases, the given solutions were consulted to confirm that their size made the problem computationally impossible using Monaco, at least a far as this author can see.)

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. Most problems are handled in a way that makes changing the problem parameters easy, even when this is not demonstrated.

In order to save space, actual Monaco output – most often from the option `-statistics`, but also from some other options or from output functions in an expression – is not reproduced here, but is instead extracted from Monaco's exact mode, specified by the option `-exact`, is always used, except that in some cases exact answers are produced without a main expression using the option `-eval`; in this case the option `-exact` is not required.

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.

These are combinatorial problems, where often the required solution is a number of combinations or permutations. This can sometimes be solved by the program just from the output from the option `-numbers`, without evaluating an expression. In some other cases candidates combinations or permutations are looped through and marked as true or false according to whether they match a required pattern or not. The number of matching combinations or permutations is then given by the number of true results in the output using the options `-exact -probability -statistics`, often referred to here as the usual options.

---

[1] See `http://www.mnemosyne.uk/monaco/`. The results in this document were re-created using version 2.55 of the program, but were first created using an earlier version of the program.

Solutions: Introductory Problems

1. This problem is too easy to seriously use the program, or it can be used simply as a calculator of `number_combin(25,2)`. But to actually simulate the problem – using 1 to 26 as A to Z – can use `sorted{d26,d26,26}>0` and count the true results. The solution is 300, as the given solution.

2. This can be solved by looping over locker numbers `r0` counting the cost as `r1` using `ilength` to count digits until the indicated sum is reached, reporting `r0`. This can use the option `-eval` with expression `until(r1+=2*ilength(incr0),r1>=13794);write(r0)`. The solution is 2001, as the given solution.

3. The program cannot be used to solve for a general n; the best it can do is to count the number of odd values for a specific n, or for a loop over values of n, here the first `c0` suitable values. With `-exact` making a random value a loop, this can use the expression `r0:=2*d[c0]+1;xrloop1(1,(r0-1)/2,r2+=number_combin(r0,r1)%2);write(r2)`. With `c0` equal to 20, the output is 20 odd numbers. Changing the output to `r2%2|write(r2)` there is no output, even for `c0` equal to 500, which requires the largest available integers.

4. Using the expression `r0:=d2001;(r0%3==0|r0%4==0)&r0%5!=0`, the solution, the number of true results, is 801, as the given solution.

5. This is essentially a generalisation of problem 2. We can start with the loop `until(r1+=ilength(incr0),r1>=1983)`, then use `r0` and `r1` to find the required solution as `r0/pow(10,r1-1983)%10`, which is 7, as the given solution. (`r0` is 697, `r1` is 1983.)

6. The program cannot handle 25 boys and 25 girls, the number of permutations is too large, even handled efficiently. Instead we here consider 15 boys and 15 girls, which still needs 128 bit integers for its 2.65×10^{32} results, but only 1.55×10^{8} evaluations. Cases where one person has two girl neighbours – using 0 for boys and 1 for girls – are tested for using `v0:=shuffle_by{c0,c0};any(v0&rotate(v0,2))`, where `c0` is 15. All results are true.

7. This problem is also too simple to need the program, but the tournament can be simulated using the expression `do4(incr1;dz2&veswap0(r0,r1);incr0);vresult0`, initialised with `-v0 sequence5`. Using `-exact -output %#lx` 16 different lists are reported, as the given solution.

8. Letting `c0` be 8, we can number the socks 0 to `c0-1` and similarly the shoes, and then create a shuffled list of the two using `shuffle_by[2[c0]]`. For all lists of interest the equal numbered sock is before the shoe. Creating all those lists would be too time consuming, but we only need the number of lists, which is the number of evaluations 81729648000. The given solution is $16!/2^8$, which is equal to that value.

9. This problem is too large for solution using Monaco.

10. This problem can be solved without the program, and to use the program most of that solution is used. The program's solution could be extended to present the rational numbers, but this is not done here. The program solution also hides various details.

    Defining `c0` as 20, we can define the constant lists, `u0:=primes[number_primes(c0)]` and `u1:=factors(factorial(c0),u0)`. `u0` is a list of all the prime numbers less than or equal to `c0`, and `u1` is a list of their multiplicities when factoring 20!. We do not need to know how many prime numbers are required, it is calculated for us, but is available as `k0` (or `k1`). We then use part of the mathematical solution to note that as rational numbers are in lowest form, for each prime all copies of it must be in either the numerator or denominator. We can create an allocation of each to the numerator or denominator as `v0:=[k0]dz2` or `!v0`. using the option `-exact`. We must have the numerator less than the denominator, which we can test as `product(pow(u0,v0??u1))<product(pow(u0,!v0??u1))`. Counting those produces the result 128. We could skip these last steps, we have `pow(2,k0)` rational numbers that might be above or below 1, but in matching pairs, so the solution must be

pow(2,k0-1). We would not however have the rational numbers available, and would have barely used the program.

11. We can create all sequences, regardless of separation, using combin5(xsequence18)), and denoting that *list*, the required sequences are those for which all_ge(xdelta(*list*,2) is true, which we count as usual, with solution 2002, as the given solution.

12. This is a straightforward problem without the program and, in addition to only being able to illustrate cases for specific value of N, either involves effectively using the solution or producing a very inefficient solution. Adopting the latter, we use s0 for N, and consider the case $N = 6$. (Anything more than about $N = 8$ is not practical.) We can loop through all pairs of values r0 and r1 such that 0 ≤ r0 < r1 < s0 that represent the persons numbered r0 and r1 shaking hands or not, adding to their counts of handshakes in the list v0 using the term do[triangle(s0-1)](incr0==s0&(r0:=incr1+1);dz2&(ince00;ince01)). We then count the number of people who have shaken hands with everyone else as r2:=count_eq(v0,s0-1), and then are only interested in the case that this is not everyone, but when it is want the result r2, using conditional(r2<s0,r2). We are then interested in the maximum value of this, which is available using -statistics, with -exact as usual. The solution is 4, as the given solution.

13. The term pattern_dist...rand4 can be used to find lists of four numbers that add up to ... However here we want odd numbers, and can use the term 1+2*pattern_dist£rand4 to find lists of four numbers that add up to 2*£+4. We want this to equal 98, so £ is 47. This requires 128 bit integers. However, it creates non-increasing lists and the problem wants all possible lists. We can create those using the function shuffle, but then will include duplicate lists. The easiest – though probably not the most efficient – solution is to use shuffle, record the lists using the function list_result, and count the different lists using the option -output %#lx. The solution is 19600, as the given solution.

14. This problem is unsuitable for solution using Monaco.

15. This can be checked by looping r0 from 1000 to 1999, which can be done in exact mode using r0:=d1000+1000, then converting r0 and r0+1 to lists using the function n_ary4, and finally adding those lists, which is without carrying, and checking whether all elements of that sum are less than or equal to 9. Putting that together gives the expression r0:=d1000+1000;all_le(n_ary4(10,r0)+n_ary4(10,r0+1),9). The solution, using the usual options, is 156, as the given solution.

16. The three professors can take any three chairs as permute3from{0,1,2,3,4,5,6,7,8}), then if that is *list* then that suitability can be checked as all_ge(xdelta(sort(*list*)),3). With the usual options the solution is 60, as the given solution.

17. This problem is too large for solution using Monaco.

18. Using numbering such that the comparison block is all zeros for convenience, this is simply count{dz2,dz3,dz4,dz4}==2, and using the usual options the result is 29, as the given solution.

19. This is straightforwardly done using v0:=7dz10 for all numbers, using v1:=mid3(0,v0) for first three digits, then is memorable if list_eq(v1,mid(3,v0))|list_eq(v1,mid(4,v0)). The solution, reported using the usual options, is 19990, as the given solution.

20. Letting c0 be 7 and s0 be c0*c0, we can create a board v0 with two squares painted using r01:=combine2from[sequence[s0]];e00:=e01:=1. Then the board and its three rotations are v0, grid_rot(v0,1), grid_rot(v0,2) and grid_rot(v0,3), and we can pick a representative from these that does not depend on their order using the variadic list function min. We can then use list_result, and the result wanted is the exact number of different list results, reported by -output %#lx. The solution is 300, as the given solution.

21. The possible arrangements are `shuffle{21,31,41,51,61,71,81}`, and one way to use that is to assign it to `r0123456`. Then we can test `r0123` etc. for divisibility by three using `!(r0123%3|r1234%3|r2345%3|r3456%3)`. With the usual options, the solution is 144, as the given solution.

22. v0 and v1 can be used as masks of the two sets as `v0:=6dz2;v1:=6dz2`. These cover all elements if `all(v0|v1)`. To only represent a pair of sets in one way also only consider if `list_le(v0,v1)`. This can still include a pair twice if `v0` and `v1` are equal, so if both tests pass use `list_result(v0#v1)` and the option `-output %#lx`. The solution is 365, as the given solution.

23. This would be computationally challenging even if the solution for n were small. Given its actual result, 253, solving this problem is not practical for Monaco.

24. This problem is unsuitable for solution using Monaco.

25. This can be simply solved by writing 100 as a binary number, then interpreting it as a ternary number. Letting `c0` be 100, this can use `n_ary(3,binary[log(2,c0)+1](c0))`, then use `write` in an option `-eval`. The solution is 981, as the given solution.

26. We can number the 27 cards with an obvious pattern and select three by `combin3(sequence27)`, which we denote *list*$_1$. We can convert each to 3 numbers, producing a list with length 9 using `n_ary9(3,n_ary(27,`*list*$_1$`))`, which we denote *list*$_2$. We convert that list to three lists of the attributes, not the cards, using `vslice012(`*list*$_2$`)`. Each attribute passes or fails using `f0` defined as `same(q0)|different(q0)`. We thus pass or fail by `f0(v0)&f0(v1)&f0(v2)`. With the usual options the solution is 117, as the given solution.

27. This problem is unsuitable for solution using Monaco.

28. The line-up can be implemented and the number of boy/girl pairs counted using `count(xdelta(shuffle_by{7,13}))`. The mean is then found using the options `-exact -statistics` and is 91/10, as the given solution.

29. Let `s0` be the number of lockers, equal to 1024. `v0` is the state of the lockers, each 1 for open, 0 for closed, We can implement the process controlled by `r0` as the current locker position, or -1 or `s0` if we have travelled to the end of the row, and by `r1` as the direction of travel, +1 or -1. We start with `r0:=-1` and `r1:=1`, except that to start the process, opening the first locker, we let `e0:=1` before setting `r0`. The main loop, in a `-eval` option, is `while(any_eq(v0),f0;r0+=r1;f0;e0:=1)`, where `f0` advances to the next locker, never beyond the range 0 to `s0-1`. `f0` is defined by `while(r1<0?f2:f1,r1:=-r1)`, where `f1` and `f2` advance in the two directions; each might go off the end of the row but is true if it does – in which case the direction is reversed by `r1:=-r1` to advance in the new direction. Then `f1` is `r0:=find_eq(v0|rstep(r0));r0==s0` and `f2` is `r0:=rfind_eq(v0|step(r0));r0<0`. Note that the search functions `find_eq` and `rfind_eq` are from the end of the list, so `rstep` and `step` are used to only search from position `r0`. `r0` is then a position in `v0`, counting from zero, but lockers are numbered from one, so the final output is `write(r0+1)`. The solution is 342, as the given solution.

30. To avoid overflows, 128 bit integers are used, but having verified that, simple multiplication is used. We let `c0` and `c1` be 31 and 19, and thus n is `pow(2,c0)*pow(3,c1)` and is `c2`. We loop `r0` through all the factors of n^2 as `r0:=pow(2,dz[2*c0+1])*pow(3,dz[2*c1+1])`. Then that satisfies the required condition as `r0<c2&c2%r0`. With the usual options the solution is 589, as the given solution.

31. This problem is unsuitable for solution using Monaco.

32. The program cannot handle numbers of the size required, but it does not need to, we are only interested in the first digit. We will need more than that, but the first `c1` digits will suffice and the first attempted value of this, 10, works. The mean we will reduce values

that are greater than or equal to `c2`, equal to `pow(10,c1)`. We let `c0` be 4000. We count powers of 9 using `r0`, and number with a leading 9 as `r1`. We check, not very efficiently, that a number has a leading 9 using `f0` defined as `r9:=p0;while(r9>10,r9/=10);r9==9`.

We use a `-eval` option where, to verify we have not over-truncated, we use `r2` and `r3` for lower and upper bounds of the first `c1` digits of the powers of 9. We start with `r2` and `r3` equal to 1, and then, in an `until` loop, multiply `r2` and `r3` by 9, check their first digits for being 9 using `f0`, producing an error if different, increment `r1` if the first digit was 9, and if r3 has reached `c2`, divide `r2` and `r3` by 10, rounding down and up, respectively. We finish if incremented `r0` has reached `c0`. The loop is:

```
until(r23*=9;r4:=f0(r2);r5:=f0(r3);error_if(r4!=r5);r1+=r5;
r3>=c2&(r2/=10;r3+=9;r3/=10),incr0==c0)
```

The solution is a final output of `r1`, assuming no error, and is 184, as the given solution.

33. This problem is unsuitable for solution using Monaco.

34. We let `s0` be the number of numbers, 15. We consider these numbers divided into three groups, two subsets of a minimal S for those subsets, and numbers not in that minimal S. These are encoded in the list `v0`, as ±1 for the two subsets, and 0 for other numbers. We consider all possibilities of this in a `-eval` option by looping `r0` through the values 0 to `c0-1`, where `c0` is `pow(3,s0)`, then letting `v0` be `n_ary(3,r0)-1`. We must exclude the case of two empty subsets by only taking any action when `any(v0)` is true. We then exclude cases where the two subsets have the same sum by only proceeding to exclude if `dot(v0,xsequence)` is zero, i.e. the two subsets have the same sum. We exclude using the function `f0`, where what matters is the numbers in either subset, not which one, the minimal S, which we can represent as `binary(boolean(v0))`.

    The purpose of `f0` is to use that minimal S, to exclude all possible sets S, all supersets of the minimal S. This can be done recursively, adding each possible missing number to the next excluded S. In order for this to be acceptably efficient we must stop the recursion if we reach an already excluded S, where we record excluded S in the list `v1`, with length `s1` given by `pow(2,s0)`. It is convenient to split `f0` into `f0` and `f1`, where `f0` sets `r9` to `p0`, then only continues if `r9` is false, using `f1(r9)`. However, it is necessary to protect `r9` against returning to this function, and hence `f0` is `r9:=p0;e19|rsave9(f1(r9))`. `f1` also sets `r9:=p0` and mark this set as processed using `e19:=1`. It then loops through each number, setting it in the new S. To set a number in the set representation `r9` means bitwise or-ing with each power of 2. Including the needed protection of `r9` and of `r8`, the power of 2, the remaining part of `f1` is `r8:=1;do[s0](rsave89(f0(bitor{r9,r8}));r8*=2)`.

    Returning to the `-eval` expression, there remains to use the zeros in `v1` to find the largest sum of any permitted S, which can use `rmax1(s1,e1?:dot(binary[s0](r1),xsequence)`, whose value can be output as the solution. This is 61, as the given solution.

35. This problem is unsuitable for solution using Monaco.

36. This problem is unsuitable for solution using Monaco.

37. The triangles can be numbered (coloured) using `12dz4`, then adjacent differences determined using `cdelta`, and then `all` used to check no adjacent triangles are the same. The solution is 531444, as the given solution.

38. This problem is unsuitable for solution using Monaco.

39. Candidate matrices can be created as `v0:=16ds2`. The row sums and column sums are available as `dmat_rsum(v0)` and `dmat_csum(v0)`. The easiest way to test for the required condition is applying `none` to their concatenation. With the usual options the solution is 90, as the given solution.

40. We cannot solve this for general n using Monaco, only for small specific values of n. Letting `c0` be n, and for convenience letting `c1` be `c0-1`, the vertex colours can be created as `v0:=[c0*c0]dz2`. This is valid and practical only for $2 \le n \le 5$. In those cases we can loop through the squares, stopping if one does not have two vertices of each colour using `rall0(c1*c1,sum(get(r0+r0/c1+{0,1,c0,c0+1},v0))==2)`. The solutions for $n = 2,3,4,5$ are 6, 14, 30, 62, which are as the given solution formula $2^n - 2$. However, that general formula cannot be confirmed.

41. This problem is unsuitable for solution using Monaco.

42. This problem is unsuitable for solution using Monaco.

43. This problem is unsuitable for solution using Monaco.

44. This problem is unsuitable for solution using Monaco.

45. Letting `s0` be $n = 7$, `v0:=dz2` with `-exact` creates mask for all subsets – including empty subset, which has zero sum and thus can be included. The summation can loop `r0` from 0 to `s0-1`, which as numbers are 1 to `s0` in descending order means numbers to be added or subtracted are `s0-r0`. We add or subtract according to whether `r1` is +1 or -1; we start with addition, but because it is convenient to invert `r1` before addition, we initialise `r1:=-1`. The summation can use `vsum0`; if `s0` is zero we add zero and take no action, otherwise we invert `r1` and add the relevant term multiplied by `r1`, thus the argument of `vsum0` can be the term `e0&(r1:=-r1)*(s0-r0)`. The result of `vsum0` is then the value to be summed over all values of `v0` and can be reported using the option `-output %#mt`. The solution is 448, as the given solution.

46. This problem can be solved recursively, where we use a list mask where 0 indicates the square has gone, 1 where it is still present. The mask has length `c0*c1` where `c0` and `c1` are the dimensions of the grid, i.e. 5 and 7. The solution will use the list `v0`, so we set `s0` to `c0*c1`. A simple recursive approach takes far too long, so we terminate whenever we have seen a grid before. For that we convert the grid *list* to the integer `binary(`*list*`)` and record new grids in the array. For ease of creation and explanation we use four recursive functions `f0` to `f3`, declared as it will turn out, by `+f0 q +f1 q +f2 qp +f3 qp`. We use a single -eval option that is just `f0(1[s0]);write(qsize)`, where `1[s0]` is the initial grid, and since each new grid is added to the array, the total number of different grids is `qsize`.

    `f0` then checks whether the grid is new, and if not adds it to the array and hands the grid to `f1`, defining `f0` as `r9:=binary(q0);qany_eq(r9)|(qbpush(r9);f1(q0))`. There are then a number of possible bites equal to the number of squares left in the grid. We can loop `r0` through these, simply numbered from 0 – they will be converted to an actual square number in `f2`. Because this function is used recursively, we protect `r0` from modification elsewhere (by returning to `f1`) using `rsave0`. This gives us the definition of `f1` as `rloop0(count(q0),rsave0(f2(q0,r0)))`.

    `f2` converts the square number counting from 0, skipping absent squares, to the square number, by defining `f2` as `f3(q0,find_gt(sigma(q0),p1))`. `f3` then removes squares with `v0:=q0;r8:=p1;r9:=r8%c1;r8/=c1;xrloop6(,r8,xrloop7(,r9,veset0(c1*r6+r7)))`, and then continue the recursion using `f0(v0)`. We need a variable to allow us to change the grid, but it is only temporary. Note that the `xrloop` functions loop from 0 to r8 or r9, inclusive.

    The solution is 792, as the given solution.

47. This problem is unsuitable for solution using Monaco.

48. This problem is unsuitable for solution using Monaco.

49. This problem can be solved by a combination of the program and a final step outside the program. We let `c0` be 15 and `s0` be `2*c0`. Then we want to create the circle of `c0` each boys and girls. We could consider boys and girls as, separately, interchangeable, and

select `c0` places for girls. But instead, both for efficiency and to handle the rotation limitation, we put a specific girl, let's say Alice, in seat 0, then pick `c0-1` seats for girls, who will be 1s, leaving `c0` seats for boys, who will be 0s. We do this with the term `v0:={1}#mset[s0-1](combin[c0-1](sequence[s0-1]),1)`. We then use f0 to determine if we can pair off seats 0 and 1, 2 and 3 etc. We succeed if `f0(v0)|f0(rotate(1,v0)` where the rotation allows for possibility of pairing seats `s0-1` and 0, 1 and 2 etc. If we succeed we use `vresult0` and a partial solution is found using `-output %#lx`, and it is 32767.

This is only a partial solution, among the girls we have just fixed Alice, and among the boys we have no fixed seating. So we have to multiply that partial solution by `factorial(c0-1)` and `factorial(c0)`. However, as the given solution does not multiply these out, we do not do so here either. The given solution is $14!\,15!\,(2^{15}-1)$, which our solution matches as $2^{15}-1$ is 32767.

50. This problem is unsuitable for solution using Monaco.
51. This problem is unsuitable for solution using Monaco.

Solutions: Advanced Problems

1. This problem is unsuitable for solution using Monaco.
2. The program cannot answer this for a general n, only small specific values of n. For n given by `c0`, the indicated sum, for a random permutation, is given by the term `sum(abs(shuffle(u0)-u0))`, which can be compared with `(c0*c0-1)/2` and counted using the usual options. This can be compared with the given solution implemented as the option `-output %[c0*square(factorial((c0-1)/2))]`. The solution and given solution agree for $n = 3, 5, 7, 9, 11$ with values 3, 20, 252, 5184, 158400.
3. With `c0` equal to 15, the sequence of heads (0) and tails (1) is `v0:=[c0]dz2`. Then we can create a list reporting consecutive pairs as HH, HT, TH, TT equal to 0 to 3 by using `count_seq(2*lrest(v0)+frest(v0))` and, using the function `list_eq`, compare that to `{2,3,4,5}`. The solution, using the usual options, is 560, as the given solution.
4. This problem is unsuitable for solution using Monaco.
5. This problem is unsuitable for solution using Monaco.
6. This problem is unsuitable for solution using Monaco.
7. Letting `s0` be 19, the possible patterns of deliveries (1) and no deliveries (0) are given by `v0:=dz2`. The first condition is `all_le(frest(bursts(v0,1)))` and the second condition is that `all_lt(frest(bursts(!v0,1)),2)`. Combined with `&` and using the usual options, the solution is 351, as the given solution.
8. This problem is unsuitable for solution using Monaco.
9. This problem is unsuitable for solution using Monaco.
10. This problem is too large for solution using Monaco.
11. This problem is unsuitable for solution using Monaco.
12. We let `c0` be 2000, the number of cards. We are interested in the penultimate card, so let `s0` be `c0-1`. (This also avoids a special case for the last card.) We put the cards, numbered by position, in the array as `qvreset(sequence[c0])`. We then run the process up to the penultimate card as `until(e0:=qfpop;incr0;qbpush(qfpop),qsize==1)`. We then use `write(last(v0))`, which as we count positions in the deck from zero, is the solution 927, as the given solution.
13. This problem is unsuitable for solution using Monaco.
14. We put details of the two types of events – adding a letter, typing a letter, `c0` of each – into `v0`. First we use the encoding 1 for the former, -1 for the latter, setting the order – regardless of before or after lunch – as `v0:=shuffle_from[copy[c0]{1,-1}]`. We only proceed if all letters are only typed after being received, or `all_ge(sigma(v0))`. We then recode the events in `v0` as letter number 1 ... `c0` or typing the last letter received as 0, using `v0:=v0>0??occurs(v0)+1`. We now find the action after the penultimate letter, which was typed before lunch, by `r2:=find_eq(v0,c0-1)+1`; this is the first action that might be typing the penultimate letter, and thus is the earliest possible last action before lunch. We then loop `r0` over all possible last actions before lunch, `xrloop0(r2,s0-1,`*term*`)`, where *term* is used to process the action sequence `v0` for each `r0`. Within *term* we use `r1`, which starts, as usual, as 0 and is modified by `e0?decr1:incr1`. `r1` then satisfies `r1<=0` if the penultimate letter has not yet been typed, so it is before lunch, or `r1>0` if it has been typed so it might be after lunch, after action `r0` in each case. We process the action sequence in this case using `f0(r0)` meaning actions 0 to `r0` are before lunch, and actions `r0+1` to `s0-1`, if any, are after lunch.

    `f0` loops over the whole of the action sequence, using the array to represent the current letter queue, using `rloop1(s0,`*action*`)`, where *action* depends on whether `e01` is a letter or

a typing, and on whether r1>p0, i.e. r1>r0. As well as processing the actions by building the letter queue at the back of the array, the after lunch typed letter list is created at the front of the array. Note that the former is empty after all actions, so we are left with the after lunch typed letters – in reverse order, but that does not matter. *action* is thus e01>0?qbpush(e01):r1>p0?qfpush(qbpop):qbpop. We can ensure that each possible queue is only counted once by using qresult. The final solution is obtained from the option -output %#lx, and is 704, as the given solution.

15. This problem is unsuitable for solution using Monaco.
16. This problem is unsuitable for solution using Monaco.
17. This problem is unsuitable for solution using Monaco.
18. This problem is unsuitable for solution using Monaco.
19. This problem is unsuitable for solution using Monaco.
20. This problem is unsuitable for solution using Monaco.
21. This problem is unsuitable for solution using Monaco.
22. We let s0 be 15 and after letting u0 be xsequence[0], the values in the problem, we consider possible subsets indicated by the mask v0:=dz2. We are going to check if that has the required property using conditional(*condition*,v0), for a condition *condition*, where the sum v0 is the size of the subset, and we can get the maximum subset size that passes the test using the maximum value from the output from -statistics. The first part of the condition – not necessary the way the rest of the condition is coded here, but could be if a better coding were produced – is v0>=3. We now want to consider three element subsets of that, but it is easier – and while not fast, good enough – to loop over all possible masks by rall0(pow(2,s0),*term*), for a term *term*, then because we are only interested in three element subsets of the subset indicated by v0, but require all of them to have the indicated property, term can include v1:=binary(r0)*v0;v1!=3. Note that we want to ignore other than three element subsets, so consider those a success, so we use v1:=binary(r0)*v0;v1!=3, where the final v1 is the subset size. The test in the three element case is !is_square(product(v1?u0:1))). Putting this together with the correct logical operators, the solution, the maximum value as indicated, is 10, as the given solution.
23. This problem can be solved for specific n but not general n. We let c0 be n, and possible solution will be v0. The sum of the elements of v0 must be c0+1, the number of elements $0, \ldots, n$ so we can set v0 to each possible lists that might, in some order, be a solution using v0:=ipartition[c0+1]. We then loop over all permutations of that using vpermut_loop0, where the now permuted v0 is a solution if v0 equals count_seq(v0), checked using list_eq. If equal then we use vwrite0 to report v0.

    There are no solutions for $n = 1,2,5$. For $n = 3$ there are two solutions: {2,0,2,0} and {1,2,1,0}. For $n = 4$ there is one solution {2,1,2,0,0}. For $n = 6, \ldots, 15$, the values that can be handled with 64 bit integers, there is one solution, from {3,2,1,1,0,0,0} to {12,2,1,0,0,0,0,0,0,0,0,0,1,0,0,0}, all of the pattern $\{n-3,2,1,\ldots,1,0,0,0\}$ where ... is all zeros. This is as the given solution.
24. This problem is unsuitable for solution using Monaco.
25. This problem is too large for solution using Monaco.
26. The program cannot handle a general n, but can handle specific values. We let c0 be n, u0 be the allowed coefficients {0,1,2,3} and c1 be the value for which the polynomial is to be evaluated, $m = 2$. The largest power of m that can be used in an evaluation to n is $\log_m n$, so, allowing for counting from 0, we let c2 be log(c1,c0)+1. Then evaluating a polynomial uses geometric[c2](c1), which as it is constant we assign to u1. The final

test, that the polynomial evaluates to n, for coefficients that in exact mode loop over all possibilities, is `dot(select(u0),u1)==c0`. With the usual options, some example solutions are 6 for $n = 10$, 13 for $n = 25$, 51 for $n = 100$, and 501 for $n = 1000$, all as the given solution.

27. This problem is unsuitable for solution using Monaco.

28. This problem is too large for solution using Monaco.

29. This problem can only be solved for specific values of n. Letting `s0` be n, the triangle can be implemented by `v0:=dz2;r0:=s0;while(r1+=w0;decr0,w0^=frest(v0)#{0});r1`, provided $n > 1$ (which would require modification of that term to allow). Note that the value of the `{0}` is never used, it is there to match lengths. Some solutions, the maximum result are 2 for $n = 2$, 4 for $n = 3$, 10 for $n = 5$, 37 for $n = 10$, and 140 for $n = 20$. It is convenient to add the option `-output %[(s0*s0+s0+1)/3]` to report the given solution for (successful) comparison.

30. This problem is unsuitable for solution using Monaco.

31. As usual, we can only handle specific values of n, and only small values. We let `c0` be n, and let `s0` be the limiting value $\frac{n}{2}(n^2 - 2n + 3)$, or `c0*(c0*c0-2*c0+3)/2`. We then colour numbers using `v0:=dz2`. We are now going to consider combinations of n numbers, and by setting `r0:=c0` we can do this in `w10`, initialised with `v1:=sequence[s0]`. The loop `wcombin_any` puts each combination in `w10`, and within that loop `v2:=head[c0](v1)` puts it in `v2`, then `v3:=mset[s0](v2,1)` converts that to a mask `v3` of the numbers from 1 to `s0`. We are only interested in the case that this is monocoloured, which we can check using `subset(v3,v0)|subset(v3,!v0)`. We require the indicated ordering, which we can test using `sorted(xdelta(v2))`. Note that as differences are used it does not matter that we counted from 0 not 1. The required solution is that the probability using `-probability -statistics` is 1, which it is for $n = 2,3,4$, as the given solution.

32. As usual, we can only handle specific values of n, and only small values. We let `c0` be n. We first use `v0` to allocate the values $0, \ldots, 3n - 1$ (the correction to starting at 1 is later) to subsets 0 to 2 using `v0:=shuffle_from[copy[c0](sequence3)]`. The required subsets can now be put into `v1` to `v3`, with the indicated correction) using `v123:=sort_index(v0)+1`, provided that `s1` to `s3` are set to `c0`. We can then use the inclusive-or of the three terms of the form `overlap(outer_sum(v1,v2),v3)` to test the required sum, which we check for always being possible using the usual options looking for probability 1, which we find for $n = 1,2,3,4,5,6$, as the given solution.

33. This problem is unsuitable for solution using Monaco.

34. This problem is unsuitable for solution using Monaco.

35. This problem is unsuitable for solution using Monaco.

36. This problem is unsuitable for solution using Monaco.

37. This problem is too large for solution using Monaco.

38. This is another problem that can be implemented for small value of m and n, but not in general. It is easier to implement using the program if the roles of rows and columns are switched, which makes no difference to the validity of a solution, so that has been done here. This description is as switched.

    We thus have m rows, set as `c0`, and n columns set as `c1`, also interpreted as the row length. Values in the array are in the range 0 to `c0-1` – it makes no difference using 0-based values rather than 1-based values. The matrix (described as an array in the problem) is thus initialised, covering all possibilities, as `v0:=shuffle_by[c1[c0]]`. We will manipulate this matrix, but also need the unmanipulated matrix, so we manipulate `v1`, set to `v0`. We do not need to shuffle all rows, we can leave the first unshuffled, so we consider shuffling

all rows numbered r0 by xrloop0(1,c0-1,*term*), where *term* is as if performing the shuffle in place in v1, although implemented as vset1(c1*r0,shuffle(mat_row[c1](r0,v1))).

We then look to see if all columns have the required property, most easily checked using the function different, as rall1(c0,different(mat_col[c1](r1,v1))). If this I so then v0 has passed the test and we can record it using vresult0. We require all possible v0 to pass the test, which we can do by comparing the number of recorded v0, given by %#lx in the option -output, with the total number of v0. One way to make that comparison is to finish the expression with n_ary(c0,v0), which is different for each v0, and also use %#nd in -output, so the two numbers must be the same for the required solution. This has been verified, matching the given solution, for (m,n) equal to (2,2), (2,3), (2,4), (3,2), (3,3), (3,4), (4,2) and (4,3). The case (4,4) is too large for the program.

39. This is another problem that can be implemented for small value of n, but not in general. It uses the randomness pool, which is first initialised, but how depends on its use, so we postpone considering that. It I more convenient to use numbering from zero rather than one, and this makes no difference to the solution. We let c0 be n and s0 be the number of subsets, i.e. $n-2$ or c0-2. Each subset has a length from 2 to $n-1$, which can be created by v0:=dz[c0-2]. We then can create a mask for each subset by looping r0 using rloop0(s0,*term*) where the subset's length is used in *term* as e0. As that is variable, we use the randomness pool to create each mask as pool_combin_dist(e0,1[c0]), and letting that be *mask*, we join the masks together to form v1 using vset1(r0*c0,*mask*), having let s1 be s0*c0. Each of the s0 masks is a combination from c0 elements, so we set the randomness pool size using do[s0](pool_bset[c0]).

    Now we need to consider all possible permutations of U, and we are looking for one – we do not need more – with the required property, by initialising v2 as sequence[c0] and using vpermut_any2(*test*), where *test* checks v2 against all of the masks using rall0(s0,f0(mid[c0](c0*r0,v1)), f0 using v2 directly and the indicated selected mask as its parameter q0. We are looking to split all subsets, hence the use of rall. f0 first sets v9 to q0, and also uses v8 initialised as v8:=0[c0]. The loop rloop9(c0,e9|*setting*) loops through the subset mask, where *setting* is only concerned with values not in the subset. *setting* then sets the element of v2 corresponding to the matching position in v2 using vset8(find_eq(v2,r9),1). Finally we check for splitting, using that if not split, all elements must be at the start and/or end of the list, which can be tested by v8:=xbursts(v8,1);first(v8)+last(v8)!=v8.

    We want to be able to do this in all cases, so with the usual option we require a probability of 1, which we get for $n = 3,4,5,6$, as the given solution.

40. We can directly implement this for fixed values of n, but as usual not for a general n. There is however one complication. We must use the randomness pool as, in general, there is a variable number of columns we can take a pebble from, and an unknown number of movements. We do not however know the required size of the pool. We can however use the option +pool_info to see if a chosen size is adequate (or unnecessarily large, but we should not need this). We may have to use it more than once in order to reach a final suitable value. A suitable starting value is that for the previous value of n. It is experimentally observed that for all n used, the required pool size is the product of a power of 2 and a power of 3, and can be set in that way, for example for $n = 18$, the largest value of n used, use -pool 26 2 -pool 13 3 rather than -pool 1 106993205379072. The required pool size is 1 for $n = 2, \dots, 5$; using (a,b) to represent $2^a 3^b$, the experimentally determined exact pool sizes for $n = 6, \dots, 18$ are $(1,0), (1,0), (3,0), (4,0), (7,0), (8,0), (11,1), (14,1), (14,3), (17,4), (17,7), (24,9), (26,13)$. The number of actual evaluations in the case $n = 18$ is 80603756.

    To represent a position we need a maximum number of columns, s0. This is clearly never larger than the number of pebbles, so we use the same number – although it could be

reduced. We start with the pebbles arranged as `v0:=s0*unit`. For any `v0`, a second list `v1` indicates which of the first $n-1$ columns has a moveable pebble, set by the term *term* or `v1:=lrest(v0)&xdelta(v0)<=-2`. Pebble movement, until that is not possible, can then be implemented as `while(any(`*term*`),r0:=pool_dist(v1);dece0;incr0;ince0)`.

Finally, as all we are asked to confirm is the uniqueness of the final position we can finish with `vresult0` and use the option `-output %#lx`. If we wanted to know what that position is we could use the option `-lists`. In all cases $n = 2, \ldots, 18$ a unique solution is reported, as the given solution.

41. This problem is too large for solution using Monaco.

42. This problem is too large for solution using Monaco.

43. This is another problem that can only be illustrated by specific examples. However, here there are three parameters, k, m and n, so creating multiple results is not straightforward. Instead here we create a solution that will work for small enough parameters, but only illustrate a single use. We let `s0`, `c0` and `c1` be k, m and n. Not all combinations are valid, and testing that is not obvious, so we add `error_if_not(1<c0&c0<=c1-1&c1-1<=s0)` at the start of the single use of `-eval` that is the solution. Also, once we have finished, whether we have the right solution is also not obvious, so we report (but do not use) the given solution `c1<triangle(c0)?s0:s0-floor_div(2*c1-c1*(c0-1),2*c0)` in the option `-output`. The one example here will be of the case for larger `c1`, i.e. n. The rest of the `-eval` expression will calculate the required maximum value, and output it using `write`.

    That maximum value will represent the set S by the mask `v0`, and as we require the maximum value of something to be calculated which we will call *value*, over all possible sets S, the term required is `lists_max0(1,`*value*`)`. For *value*, we now look for subsets of S, represented by the mask `v1`, checking whether the summation condition is satisfied. If it is we are not interested, but if it is not this is a possible value, equal to the size of S, i.e. the integer `v0`. The summation condition causes non-interest if true for any subset of S, thus, letting the condition be *condition*, *value* is `lists_any1(v0,`*condition*`)?0:v0`. We are only interested in cases where the subset has n elements, so *condition* is 0 if `v1!=c1`. (This is thus not an efficient way to loop through only such lists, but it suffices here.) The sum of the subset represented by `v1` is then `dot(v1,xsequence)`, and the remainder of *condition* is if this is equal to `c1`.

    The example case used here is $k = 10$, $m = 3$, $n = 8$. The output, including the solution and the given solution, is 10.

44. This problem is unsuitable for solution using Monaco.

45. This problem can only be solved for specific values of n, and the only values of n that are not too large are $n = 2$ and $n = 3$. We let `c0` be n, then let `s0` be `c0*c0`. We are interested in the maxima of the minima of some rational numbers, including across evaluations where the program does not directly provide that functionality. However, all ratios have one of the numbers $1, \ldots n^2$ as a denominator, so if we let `c1` be the lowest common multiple of those values, `lcm_seq(,s0)`, then we can multiply all results by `c1` to give an integer, and report results using the option `-scale 1 c0`. The solution is then the maximum result reported using the option `-statistics`.

    We thus start the main expression with `v0:=shuffle(xsequence)`. After this we will loop the variables `r0`, `r1` and `r2` so that we compare element (`r0`,`r1`) with other elements in its row or column indexed using `r2`. We want the minimum such ratio, so we loop with `rmin0`, `rmin1` and `rmin2`. This could use `rmin0(c0,rmin1(c0,rmin2(c0,`*term*`)))`, but *term* uses the value `r3:=dmat_get(r0,r1,v0)` which is best calculated outside the `rmin2` loop. *term* is then the minimum of a row and column ratio, calculated using `f0(r3,r0,r2)` and `f0(r3,r2,r1)`, where `f0` is given by `f1(p0,dmat_get(p1,p2,v0))`. `f1` now selects only the cases where the ratio is greater than one; as we are looking for a minimum, when a case

is not selected we use a large value so it will be ignored, and `intmax` is suitable. Thus `f1`, the ratio times `c1` or `intmax`, is `r9:=p1;p0>r9?c1*p0/r9:intmax` – here we take advantage of that we know `p0` is `r3` so do not need to copy it to a variable to avoid recalculation.

The solution – the maximum value as noted above – is then $\frac{3}{2}$ for $n = 2$ and $\frac{4}{3}$ for $n = 3$, as the given solution.

46. This problem is unsuitable for solution using Monaco.
47. This problem is unsuitable for solution using Monaco.
48. This problem is unsuitable for solution using Monaco.
49. This problem is unsuitable for solution using Monaco.
50. This problem can be solved by continuing to output values of n until the problem becomes too large, as set by the equal valued parameters `s0` and `s1`, or too slow, requiring the user to abort the run. Here we use `s01` equal to 10000. Note that as this is the length of two lists, memory use as well as time increases with this parameter. `s01` defines the length of `v0` and `v1` that are masks representing the sets A_k and B_k, initialised by that `v0` is empty so does not need initialising, and `v1` represents {0} so can be initialised by `v1:=unit`. We also use r0 for n, initialised by `r0:=1`. In the first argument of the function until we update `r0` using `incr0`, we set the new `v0` using a temporary `v2:=shift(-1,v1)`, we set the new `v1` using `v1:=(v0|v1)^(v0&v1)` – note that as the union contains the intersection we can use the symmetric difference – and then set `v0:=v2`. The cases of interest, when $B_n = \{0\}$ can be determined as when `none(frest(v1))` and then n is reported as `write(r0)` with a convenient `nline`. This can continue until `last(v1)`, as after that `v0` cannot be correctly determined.

    Using `s01` equal to 10000, as noted, the solution, the output values of n are the powers of 2 from 2 to 16384. Also, not reported but could be by replacing `r0:=1` by `write(r0:=1)`, $n = 1$ is also a solution. This is as the given solution.
51. This problem is unsuitable for solution using Monaco.

Characterisation of Solutions

Introductory Problems

Problems that can be solved: 1, 2, 4, 5, 7, 8, 10, 11, 13, 15, 16, 18-22, 25, 26, 28-30, 32, 34, 37, 39, 45, 46, 49 (28 problems).

Problems that can be solved for some specific examples, but not in general: 3, 6, 12, 40 (4 problems).

Problems that cannot be solved: 9, 14, 17, 23, 24, 27, 31, 33, 35, 36, 38, 41-44, 47, 48, 50, 51 (19 problems).

Advanced Problems

Problems that can be solved: 3, 7, 12, 14, 22, (5 problems).

Problems that can be solved for some specific examples, but not in general: 2, 23, 26, 29, 31, 32, 38-40, 43, 45, 50 (12 problems).

Problems that cannot be solved: 1, 4-6, 8-11, 13, 15-21, 24, 25, 27, 28, 30, 33-37, 41, 42, 44, 46-49, 51 (34 problems).

Summary of Runs

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

Introductory Problems

1. `-eval write(number_combin(25,2))`

   `-exact -probability -statistics sorted{d26,d26,26}>0`

2. `-eval until(r1+=2*ilength(incr0),r1>=13794);write(r0)`

3. `-c0 20 -eval r0:=2*d[c0]+1;xrloop1(1,(r0-1)/2,r2+=number_combin(r0,r1)%2);write(r2)`

   `-c0 500 -eval r0:=2*d[c0]+1;xrloop1(1,(r0-1)/2,r2+=number_combin(r0,r1)%2);r2%2|write(r2)`

4. `-exact -probability -statistics r0:=d2001;(r0%3==0|r0%4==0)&r0%5!=0`

5. `-eval until(r1+=ilength(incr0),r1>=1983);write(r0);space;write(r1);space;write(r0/pow(10,r1-1983)%10)`

6. `-exact -probability -statistics -numbers +noexact -c0 15 v0:=shuffle_by{c0,c0};any(v0&rotate(v0,2))`

7. `-exact -output %#lx -v0 sequence5 do4(incr1;dz2&veswap0(r0,r1);incr0);vresult0`

8. `-exact -numbers -c0 8 v0:=shuffle_by[2[c0]]`

10. `-exact -probability -statistics -c0 20 u0:=primes[number_primes(c0)];u1:=factors(factorial(c0),u0);v0:=[k0]dz2;product(pow(u0,v0??u1))<product(pow(u0,!v0??u1))`

11. `-exact -probability -statistics all_ge(xdelta(combin5(xsequence18)),2)`

12. `-exact -statistics -s0 6 do[triangle(s0-1)](incr0==s0&(r0:=incr1+1);dz2&(ince00;ince01));r2:=count_eq(v0,s0-1);conditional(r2<s0,r2)`

13. `-exact -output %#lx list_result(shuffle(1+2*pattern_dist47rand4))`

15. `-exact -probability -statistics r0:=d1000+1000;all_le(n_ary4(10,r0)+n_ary4(10,r0+1),9)`

16. `-exact -probability -statistics all_ge(xdelta(sort(permute3from[sequence9])),3)`

18. `-exact -probability -statistics count{dz2,dz3,dz4,dz4}==2`

19. `-exact -probability -statistics v0:=7dz10;v1:=mid3(0,v0);list_eq(v1,mid(3,v0))|list_eq(v1,mid(4,v0))`

20. `-exact -output %#lx -c0 7 s0:=c0*c0;r01:=combine2from[sequence[s0]];e00:=e01:=1;list_result(min(v0,grid_rot(v0,1),grid_rot(v0,2),grid_rot(v0,3)))`

21. `-exact -probability -statistics r0123456:=shuffle{21,31,41,51,61,71,81};!(r0123%3|r1234%3|r2345%3|r3456%3)`

22. `-exact -output %#lx`
    `v0:=6dz2;v1:=6dz2;all(v0|v1)&list_le(v0,v1)&list_result(v0#v1)`

25. `-c0 100 -eval write(n_ary(3,binary[log(2,c0)+1](c0)))`

26. `-exact -probability -statistics`
    `f0[same(q0)|different(q0)];vslice012(n_ary9(3,n_ary(27,combin3(`
    `sequence27))));f0(v0)&f0(v1)&f0(v2)`

28. `-exact -statistics count(xdelta(shuffle_by{7,13}))`

29. `-s0 1024 -f1 r0:=find_eq(v0|rstep(r0));r0==s0 -f2`
    `r0:=rfind_eq(v0|step(r0));r0<0 -f0 while(r1<0?f2:f1,r1:=-r1) -eval`
    `e0:=1;r0:=-1;r1:=1;while(any_eq(v0),f0;r0+=r1;f0;e0:=1);write(r0+1)`

30. `-exact -probability -statistics -c0 31 -c1 19 -c2`
    `pow(2,c0)*pow(3,c1)`
    `r0:=pow(2,dz[2*c0+1])*pow(3,dz[2*c1+1]);r0<c2&c2%r0`

32. `-c0 4000 -c1 10 -c2 pow(10,c1) -f0 r9:=p0;while(r9>10,r9/=10);r9==9`
    `-eval`
    `r23:=1;until(r23*=9;r4:=f0(r2);r5:=f0(r3);error_if(r4!=r5);r1+=r5;r3`
    `>=c2&(r2/=10;r3+=9;r3/=10),incr0==c0);write(r1)`

34. `-s0 15 -c0 pow(3,s0) -s1 pow(2,s0) +f0 p +f1 p -f1`
    `r9:=p0;e19:=1;r8:=1;do[s0](rsave89(f0(bitor{r9,r8}));r8*=2) -f0`
    `r9:=p0;e19|rsave9(f1(r9)) -eval`
    `rloop0(c0,v0:=n_ary(3,r0)-1;any(v0)&(dot(v0,xsequence)|f0(binary(`
    `boolean(v0)))));write(rmax1(s1,e1?:dot(binary[s0](r1),xsequence)))`

37. `-exact -probability -statistics all(cdelta(12dz4))`

39. `-exact -probability -statistics`
    `v0:=16ds2;none(dmat_rsum(v0)#dmat_csum(v0))`

40. `-exact -probability -statistics -c0 2 -c1 c0-1`
    `v0:=[c0*c0]dz2;rall0(c1*c1,sum(get(r0+r0/c1+{0,1,c0,c0+1},v0))==2)`

    And similarly for other values of `c0`.

45. `-exact -output %#mt -s0 7`
    `v0:=dz2;r1:=-1;vsum0(e0&(r1:=-r1)*(s0-r0))`

46. `-c0 5 -c1 7 -s0 c0*c1 +f0 q +f1 q +f2 qp +f3 qp -f3`
    `v0:=q0;r8:=p1;r9:=r8%c1;r8/=c1;xrloop6(,r8,xrloop7(,r9,veset0(c1*r6+`
    `r7)));f0(v0) -f2 f3(q0,find_gt(sigma(q0),p1)) -f1`
    `rloop0(count(q0),rsave0(f2(q0,r0))) -f0`
    `r9:=binary(q0);qany_eq(r9)|(qbpush(r9);f1(q0)) -eval`
    `f0(1[s0]);write(qsize)`

49. `-exact -output %#lx -c0 15 -s0 2*c0 -f0 all(fold[c0](weave[c0](q0)))`
    `v0:={1}#mset[s0-1](combin[c0-1](sequence[s0-1]),1);(f0(v0)|f0(rotate`
    `(1,v0)))&vresult0`

Advanced Problems

2. `-exact -probability -statistics -output`
   `%[c0*square(factorial((c0-1)/2))] -c0 3 -u0 xsequence[c0]`
   `sum(abs(shuffle(u0)-u0))==(c0*c0-1)/2`

   And similarly for other values of `c0`.

3. `-exact -probability -statistics -c0 15`
   `v0:=[c0]dz2;list_eq(count_seq(2*lrest(v0)+frest(v0)),{2,3,4,5})`

7. `-exact -probability -statistics -s0 19 v0:=dz2;all_le(frest(bursts(v0,1)))&all_lt(frest(bursts(!v0,1)),2)`

12. `-c0 2000 -s0 c0-1 -eval qvreset(sequence[c0]);until(e0:=qfpop;incr0;qbpush(qfpop),qsize==1); write(last(v0))`

14. `-exact -output %#lx -c0 9 -s0 2*c0 -f0 qclear;rloop1(s0,e01>0?qbpush(e01):r1>p0?qfpush(qbpop):qbpop); qresult v0:=shuffle_from[copy[c0]{1,-1}];all_ge(sigma(v0))&(v0:=v0>0??occurs (v0)+1;r2:=find_eq(v0,c0-1)+1;xrloop0(r2,s0-1,e0?decr1:incr1;r1>0&f0 (r0)))`

22. `-exact -statistics -s0 15 u0:=xsequence[s0];v0:=dz2;conditional(v0>=3&rall0(pow(2,s0),v1:= binary(r0)*v0;v1!=3|!is_square(product(v1?u0:1))),v0)`

23. `-exact -c0 3 v0:=ipartition[c0+1];vpermut_loop0(list_eq(v0,count_seq(v0))&vwrite0 )`

    And similarly for other values of `c0`.

26. `-exact -probability -statistics -c0 10 -u0 {0,1,2,3} -c1 2 -c2 log(c1,c0)+1 -u1 geometric[c2](,c1) dot(select(u0),u1)==c0`

    And similarly for other values of `c0`.

29. `-exact -statistics -output %[(s0*s0+s0+1)/3] -s0 2 v0:=dz2;r0:=s0;while(r1+=w0;decr0,w0^=frest(v0)#{0});r1`

    And similarly for other values of `s0`.

31. `-exact -probability -statistics -c0 2 -s0 c0*(c0*c0-2*c0+3)/2 v0:=dz2;r0:=c0;v1:=sequence[s0];wcombin_any10(v2:=head[c0](v1);v3:= mset[s0](v2,1);(subset(v3,v0)|subset(v3,!v0))&sorted(xdelta(v2)))`

    And similarly for other values of `c0`.

32. `-exact -probability -statistics -c0 1 -s123 c0 v0:=shuffle_from[copy[c0](sequence3)];v123:=sort_index(v0)+1;overlap (outer_sum(v1,v2),v3)|overlap(outer_sum(v1,v3),v2)|overlap(outer_sum (v2,v3),v1)`

    And similarly for other values of `c0`.

38. `-exact -output %#lx %#nd -c0 2 -c1 2 v1:=v0:=shuffle_by[c1[c0]];xrloop0(1,c0-1,[vset1(c1*r0,shuffle( mat_row[c1](r0,v1)))]);rall1(c0,different(mat_col[c1](r1,v1)))& vresult0;n_ary(c0,v0)`

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

39. `-exact -probability -statistics -c0 2 -s0 c0-2 -s1 s0*c0 -f0 v9:=q0;v8:=0[c0];rloop9(c0,e9|(veset8(find_eq(v2,r9),1)));v8:= xbursts(v8,1);first(v8)+last(v8)!=v8 do[s0](pool_bset[c0]);v0:=dz[c0-2]+2;rloop0(s0,[vset1(r0*c0, pool_combin_dist(e0,1[c0]))]);v2:=sequence[c0];vpermut_any2(rall0(s0 ,f0(mid[c0](c0*r0,v1))))`

    And similarly for other values of `c0`.

40. `-exact -output %#lx -pool 1 1 -s0 2 v0:=s0*unit;while(any(v1:=lrest(v0)&xdelta(v0)<=-2),r0:=pool_dist(v1);dece0;incr0;ince0);vresult0`

    And similarly, but with a suitable pool size, for other values of `s0`.

43. `-output %[c1<triangle(c0)?s0:s0-floor_div(2*c1-c1*(c0-1),2*c0)] -s0 10 -c0 3 -c1 8 -eval error_if_not(1<c0&c0<=c1-1&c1-1<=s0);write(lists_max0(1,lists_any1(v0,v1!=c1?0:dot(v1,xsequence)==c1)?0:v0))`

    And similarly for other values of `s0`, `c0` and `c1`.

45. `-exact -statistics -c0 2 -s0 c0*c0 -c1 lcm_seq(,s0) -scale 1 c1 -f1 r9:=p1;p0>r9?c1*p0/r9:intmax -f0 f1(p0,dmat_get(p1,p2,v0)) v0:=shuffle(xsequence);rmin0(c0,rmin1(c0,r3:=dmat_get(r0,r1,v0);rmin2(c0,min{f0(r3,r0,r2),f0(r3,r2,r1)})))`

    And similarly for other values of `c0`.

50. `-s01 10000 -eval v1:=unit;r0:=1;until(incr0;v2:=shift(-1,v1);v1:=(v0|v1)^(v0&v1);v0:=v2;none(frest(v1))&(write(r0);nline),last(v1))`

    And similarly for other values of `s01`.