VAUVAU

One solution is to simulate minute by minute, keeping track of what mood each of the dogs is in, and how long he's been that way. We can write in an array how many dogs were aggressive during each minute and easily look up the answer for each of the heroes.

The other solution is using modular arithmetic. For example, the first dog is aggressive in minute M if $0 \leq (M-1) \bmod (A+B) \leq A-1$, where mod represents the remainder operator.

VECI

We increase X by 1 until we've found a solution. If we don't find one in 1 000 000 iterations, then there is surely no such number and we output zero. What remains is to find a simple procedure to check if two integers X and Y consist of the same digits.

Separating an integer into digits can be done by repeatedly observing the last digit (remainder when dividing it by 10) and dividing the number by 10, while the number is not zero.

To check if X and Y consist of the same digits, we count how many times each of the 10 digits appears in each number. If the counts match for all digits, the numbers consist of the same digits.

LEKTIRA

The problem here is to implement the procedure in the problem statement. Using two for loops we pick two places where to split the given word. After reversing the order of letters in each sub word we check if the resulting word is better than the currently best result.

MUZICARI

The solution to this problem is in reducing it to the classical knapsack problem. Suppose the capacity of the knapsack is the duration of the concert, the objects are musicians, and the weights are the lengths of breaks. We want to distribute the objects into two sacks of equal capacity.

One way to do this is to fill one sack as much as possible and put all other objects in the other knapsack.

Once we've established which objects (musicians) go into which knapsack, the actual times can be obtained by having all musicians in the same knapsack take breaks one after another in any order.

We use dynamic programming to solve the knapsack problem. Let dp(P, W) be 1 if it is possible to find a subset of the first P objects such that their weights sum up to W, and 0 if it is not possible.

The recursive relation for calculating dp(P, W) relies on observing the two options we have for using object P, of weight w(P):

- If we decide that object P is not part of the subset (does not go in the sack), then we need to use some of the previous P−1 objects to obtain the target weight W. The expression dp(P−1, T) tells us exactly that.
- If we decide that object P is part of the subset (goes in the sack), then we need to use some of the previous P−1 objects to obtain the weight W−w(P). The expression dp(P−1, T−w(P)) tells us exactly that.

POKLON

First sort the intervals in descending order by their lower bound, breaking ties in increasing order of their upper bounds. Let dp(I) be the length of the longest sequence starting with interval I.

We can see that dp(I) = max{ 1 + dp(J), for each interval J completely contained in interval I }. If I does not contain any other intervals, then dp(I) = 1.

Because of how we sorted the intervals, interval I cannot contain interval J if I < J. A direct implementation of the above idea has time complexity $O(N^2)$ and does not get all points.

To efficiently calculate dp(I), we introduce a new array of integers A containing 1 000 000 elements and initialized to zero. The value A(hi) tells us the length of the longest sequence starting with an interval whose upper bound is exactly hi.

Now the relation dp(I = [lo,hi]) can be expressed as dp(I) = max{ 1 + A(x), for lo ≤ x ≤ hi }. In other words, we find the length of the longest sequence whose upper bound is inside I. Again, the lower bound of that interval will also be within I, because the intervals are processed in descending order of lower bounds.

Finally, to efficiently find the largest element in the subsequence of A with indices [lo, hi], we can use an interval tree or Fenwick tree.

KOCKE

This task can be solved in many ways and with different strategies, but various cases can seriously complicate solutions. The solution described here is considered simplest by the author, in terms of coding complexity and how many special cases need handling.

The gist of the solution is a subroutine which moves a specific cube (which hasn't yet joined another cube) to some pair of coordinates, making sure the cube does not join other cubes en route (except perhaps when arriving at its destination), and that the robot does not move any other cube.

The subroutine uses breadth-first search. The state is two pairs of coordinates, the coordinates of the robot and the cube it is moving. See the sample source code for implementation details.

After the subroutine is done, we need to analyze the starting situation:

1. The robot can reach every square without pushing cubes.
2. The robot is trapped and cannot move without moving cubes.
3. The robot can't reach every square without pushing – some square is surrounded by four cubes.

Case 3 is solved by pushing the upper of those four cubes one square down (using the subroutine). Now we already have four cubes joined forming the upper part of the letter T, so we simply move the remaining cube (again using the subroutine) to complete the letter.

Case 2 is converted into case 1 with one push. We need to make sure the push doesn't join two cubes.

Finally, case 1 is solved by choosing the final coordinates of the letter T (for example, (8,8), (7,8), (7,7), (7,6) and (6,8)) and then bringing the five cubes to the five coordinates with five calls of the subroutine.

If we always choose the cube with the largest x-coordinate as the cube to move next, then there will always exist a sequence of moves to bring it to its final coordinates, since the robot can reach every square without pushing cubes (including the square immediately to the left of the cube), and after that we no longer need to worry about the cube joining other on the way to its final destination.