Analysis

The problem Ruler was a bit more unorthodox in comparison to most problems, given on high-school competitions. Still, the main idea – to pre-calculate locally the answers for all possible inputs – was somehow intuitive and I doubt many people will miss it. The 1 second time limit was given so the competitors are able to score a reasonable amount of points even if they didn't hardcode the answers in their code (without pre-calculation, a good solution can find the answers for $N \leq 11$ or $N \leq 12$, depending on the optimizations applied).

The problem's setup is actually known (https://en.wikipedia.org/wiki/Golomb_ruler), however the solution (heuristics) are ad-hoc and usually not known by the vast majority of competitors; thus making it an okay problem. There is currently no known effective algorithm for finding neither the ruler's length, nor the ruler's marks. There are some approximation ones, which yield close-to-optimal results, but in this case, we're interested in the *optimal* ones, thus making them infeasible.

Of course, with access to the Internet one can very easily (and quickly) find all the answers for this problem. The constraints of this competition (no internet and no smartphones) fortunately makes this approach impossible.

Let's now look at the solution which we had in mind.

In the given solution we will call the markers of the ruler just "numbers" (the distance from the beginning of the ruler) – indeed, we are only interested in non-negative integer positions (where the markers are) which are the natural numbers. The distance between two markers is the absolute difference between the respective numbers.

Dynamic Programming?

Some of the competitors can intuitively be drawn to a dynamic programming approach, in which the state is last used number and which distances between markers are already used. This turns out to be wrong, as we need the numbers we've used already (not the distances between them) – it is possible to have two different states (sets of ruler markers) which yield the same set of distances between pairs of markers! This way the dynamic programming gets confused that it has already seen this state before and stops backtracking, when in fact it needs to continue. Once thinking about it, it is fairly obvious that if we have the same set of used distances, but different sets of used numbers (markers), adding a new marker at a certain position would yield different new forbidden distances, thus different new state.

If we need in the state which numbers we have already used, then it makes no sense to use dynamic programming at all (we have to brute-force all possibilities anyway, and each of them is considered only once). Thus, a backtrack would be much more efficient as it will have the same speed but would require much less memory.

Backtrack

The first problem we face is that we don't know what the length L of the ruler should be. Obviously it needs to be at least N. The simple observation that if we have N markers we also have $N * (N - 1) / 2$ distances between them which should be distinct, we get to a much larger lower bound. Another, even better lower bound which we can use is $ans[N - 1] + 1$ (the answer for a ruler with one less mark plus one). Later we'll see that we can easily know this value without sacrificing much time to calculate it (see Observation 2).

Observation 1

The main observation in order to be able to get 100 points on the task is that we can calculate all answers and put them in our code, achieving O(1) answering for a given N. This way we increase the TL of our solution to 5 hours, instead of the given 1 second. We already mentioned this at the beginning of the analysis, but is important enough to be repeated here. We can always do this whenever we have relatively small number of inputs (in this task – only 10) and the answer to each input is relatively short. Such tasks are somewhat common on university contests (ACM ICPC, for example).

Observation 2

To calculate the solution for N we can calculate and use as we like all answers for N – 1, N – 2, … 1 without this slowing down our solution too much. Since the amount of work is exponentially increasing with the increase of N (at least twice as much work in comparison to N – 1, but in fact much more). We can also apply observation 1 (calculate N = 1 and put it in the code, then calculate N = 2 and put it in the code, then calculate N = 3 and put it in the code and so on, thus decreasing the calculation time to O(1) for smaller answers.

Optimization 1: Using bitwise operations

Since the allowed numbers, as well as distances between them, are relatively small (as it turns out, up to 127), we can use bitwise operations (either implementing a bitset ourselves, or using the one in the STL) to keep and update the state. This way operations, which would otherwise be linear, can become constant.

It is important to use the bitwise tricks not only to keep the state (which distances are already used and where are the marks), but also update it. We should figure out how, adding a new number, we can constantly update the state.

Imagine we have already put 10 numbers (it makes sense to put the numbers in increasing order) and the largest of them is X. The next number can be at positions X + 1, X + 2, …, etc. Imagine you already have the distances from all other numbers to the last one (X). Trying X + 1 as a position for the new one, we can have this mask, shift it left (<<=) by 1, and add a bit at position 1 (distance 1 to X). Then, trying X + 2 we can shift this mask one more time (<<= 1) achieving a constant update of the new distances. Checking if the new distances are okay can be done with a single AND (&) with the used distances mask. If the result is non-zero, then we have already used some of the distances and cannot put the new number there. All these operations are O(1) and are implemented in the STL bitset. They can easily be implemented by ourselves which makes the execution twice faster, but requires some more code. The only linear thing remaining is trying X + 1, X + 2, … L. This is also what we can apply heuristics to.

Optimization 2: Using the answers for smaller rulers

Since in the beginning we fix the length of the ruler (L), at any point we know how much space we have left until its end. Let this number be some integer R. We also know how many marks we must put (which is some integer M). M will obviously be less than N, since we're putting a mark at the start of the ruler, position 0, thus we'll have the ans[M] calculated. If at any time we're at a position, for which ans[M] > R, then there is no way we can finish our ruler and can stop backtracking this branch of the search tree.

Optimization 3: Heuristic for increasing distances

As we said before, in order to put N numbers we will need at least N * (N – 1) / 2 positions. If we end up in a state, in which the remaining positions until the end of the ruler is R and M marks to put, and R < M * (M – 1) ? 2, then we can cut this branch right away.

Optimization 4: Heuristic for increasing *unused* distances

We can build up on the previous optimization, since we already know that some of the distances are "used" and cannot be used once again. Imagine we have already used distances 1, 3, 5, 6, 7, 10, 11, and 13 (and possibly larger, which we don't care about at the moment), the last used number is 42, and there are 5 more marks which we should put. The least realistic configuration for the last 5 numbers would be:

1. 44 (using distance 2 to 42)
2. 48 (using distance 4 to 44)
3. 56 (using distance 8 to 48)
4. 65 (using distance 9 to 56)
5. 77 (using distance 12 to 65)

Of course, this configuration can still be invalid (and in this case IS invalid – for example the distance from 48 to 42 is 6, which is already used), but still can be used as a fast heuristic to eliminate some branches of the search tree. In this case, instead of 5 * (5 – 1) / 2 = 10 we can see that we will need *at least* 77 – 42 = 35 positions.

In short, instead of using the formula M * (M – 1) / 2 we will use the sum of the first M unused distances.

Optimization 5: The better half

If we are to put N numbers in L positions, then at least half of the numbers will be either in the first or in the second half of the positions. Since the ruler is valid no matter which end we choose to be its start (we have a mark at the left *and* at the right ends), then we can fix that at least N / 2 of the marks will be in its first half. The check here is if by the (L / 2 + 1)-th position we have put at least N / 2 of the marks – if we haven't we can trim this branch of the tree. Please note that it *is* possible that we have an answer in this branch – however, we're guaranteed to find its symmetric one in the rest of the search.

Optimization 6: Search direction

The function of the answer is monotonically increasing, since if the answer for N is X, then the answer for N + 1 will need at least one more position for the new mark. If we allow the last mark to be not at the end of the ruler, but some point before it (for example at position L – 2 for a ruler with length L), then the function becomes also continuous.

So, instead of trying out ans[N – 1] + 1, ans[N – 1] + 2, … etc., until we find the answer for N, we can use binary search. As an upper bound we can use ans[N – 1] * 2 + 1, since we can get the ruler for N – 1, extend it by ans[N – 1] + 1 and put the last mark there. The distance from the new marker to all older ones will be at least ans[N – 1] + 1, and the distances between all markers from the old ruler are at most ans[N – 1], then all new distances will be unique and not seen before, thus making it a valid (but not necessarily optimal) ruler with N markers.

Although the binary search significantly helps for finding the answer more quickly, it is not the best we can do. "WAT!?" you may be thinking – how come BS is not optimal? Well, in this sort of problems (exhaustive backtracks) it often happens that examining L which doesn't yield an answer is much slower than

examining L which has (especially if there is enough additional space, i.e. L is much larger than the minimal length for this N). Thus, the "negative" answers – cases, in which the chosen L is not enough to build a valid ruler with N marks – are evaluated much slower, than "positive" ones – in which the chosen L is enough. Cases, in which the chosen L is enough and much larger than needed are evaluated very quickly, as answer is being found almost immediately.

So, what we can do here is start from the upper bound and go downwards, until we reach L for which we don't have an answer. Because of the monotonicity of the function we can stop right away.

An even further optimization is skipping Ls for which we are sure to have an answer. Imagine, we tried a length L, which is much larger than the optimal. We found a solution, whose largest mark is at position L – X. There is no need to examine L – 1, L – 2, …, L - X as we are sure they have an answer – we can continue the search from L – X - 1.

Author: Alexander Georgiev