Info(1)cup 2020 solutions

Alexa Tudose, Tamio-Vesa Nakajima

February 28, 2020

1 Problem Table tennis

The author of this problem is Tamio-Vesa Nakajima. In this problem, you are given a sorted sequence of $M = N + K$ integers, and are asked to eliminate K of them so that the resulting sequence is *balanced*. A sequence $a_1 \dots a_N$ is balanced if and only if $S = a_1 + a_N = a_2 + a_{N-1} = \dots$. It is guaranteed that this is possible. The solutions for the various subtasks follow.

1.1 Subtask 1

Here $K = 1$ and an $O(M^2)$ complexity solution is accepted. Thus, for each integer in the sequence, try to eliminate it, and see if the resulting sequence is well balanced in $O(M)$. Output a correct solution upon finding it.

1.2 Subtask 2

Now, $K = 1$, but an $O(M)$ complexity solution is needed. To do this, consider the first and last element in the solution. These are either the first and last element in the input; or the first and second to last; or the second and the last. If we could somehow find a solution for a given sum S, if possible, in linear time, then we can solve this subtask.

Now, consider an S, and try to find a subsequence of a that is balanced, where the sum of corresponding elements is S. First, initialise the solution r. Now, consider the sum of the first and last elements of a. If that sum is greater than S, then clearly the last element of a cannot be part of any solution (since no integer in a could possible sum to S with it). So we can eliminate it, and continue on from there. Otherwise, if the sum is less than S, we can eliminate the first element of a, for the same reasons. If, on the other hand, the sum is S precisely, then add the first and last elements of a to the solution r, remove them from a, and continue from there. If r ever reaches N elements, then we have found a solution. If a ever becomes empty before r reaches N elements, then no solution exists. Since at each step of the algorithm we eliminate at least an element of a, this algorithm takes at most M steps to terminate.

Thus, to solve this subtask, try setting S to $a_1 + a_M, a_1 + a_{M-1}, a_2 + a_M$, and checking for which a solution exists.

1.3 Subtask 3

Now, $K = 2$, and an $O(M)$ complexity is needed. Now try setting S to all sums $a_i + a_{M+1-j}$ where $i + j \leq 4$, and do a similar algorithm to subtask 2. This works since clearly the first and last element of the result sequence must be a pair from this set.

1.4 Subtask 4

Now, we want an $O(M^3)$ complexity solution. To accomplish this, set S to all possible sums $a_i + a_j$, checking each one in linear time.

1.5 Subtask 5

This subtask rewards a brute-force solution, as $N + K \leq 1$. Here, try all $N + K$ take K ways of selection the K numbers which are eliminated, and see which one leads to a correct solution.

1.6 Subtask 6 & 7

Here an $O(MK^2)$ solution is wanted. To do this, we must find $O(K^2)$ candidate values for S, and check each one as before. Clearly the first element in the solution sequence is among the first $K + 1$ elements of a, and likewise the last element in the solution sequence is among the last $K + 1$ elements of a (since at most K elements are not in the solution). So, trying the sum of all of these pairs gives us our $O(K^2)$ candidates.

1.7 Subtask 8

Now, we must solve the problem in $O(MK)$. We need to find $O(K)$ candidates for the value of S, and check each as before. Suppose that $M > 4K$ (since otherwise the solution from subtask 4 works). Consider the first K and the last K elements of a. Note that the first K and the last K elements of the solution are among these. Thus, if we consider the sum of all pairs (a_i, a_j) where $i \leq K$ and $j \geq N$, the correct sum must appear K times. Thus, enumerate these $4K^2$ sums, and select only those that appear K times. S must appear among these, and there are at most $4K$ such sums. So these are our $O(K)$ candidates for S.

2 Problem Trampoline

The author of this problem is Alexa Tudose.

2.1 Subtask 1

To solve this subtask, we can start a Breath-First Search (BFS) for each query from the given starting position. At each step, depending on the colour of the trampoline we are currently on, we can either expand to one or two other trampolines. In the end,

we should check whether the given ending position was visited in the BFS or not. s we may need to visit up to $O(R \times C)$ trampolines for each query, the overall complexity is $O(T \times R \times C)$.

2.2 Subtask 2

To solve this subtask, we need to use the following observation:

Observation 1. *Whenever Little Square is on a green trampoline that is not on the same line as the required ending position, it is optimal for him to go one step down. Otherwise, he should go one step to the right.*

Now, for each query, we can simulate the process from the given starting position and then check whether the required ending position was visited or not. Note that for each query we need to make at most $R + C$ steps. Thus, we get the following complexity: $O(T \times (R + C))$.

2.3 Subtask 3

In this subtask, the starting and ending trampolines are on adjacent lines, which means that Little Square needs to make exactly one step down. As we can deduce from observation 1, he should go one step down when he first encounters a green trampoline, and he should go one step to the right in all other cases. Let y_t be the smallest column such that $y_t \geq y_{start}$ and there is a green trampoline on (x_{start}, y_t) – we take $y_t = \infty$ if there is no such column. Now, Little Square can go from (x_{start}, y_{start}) to (x_{finish}, y_{finish}) if and only if $y_t \leq y_{finish}$. To find y_t, we need to do a binary search on the positions of the green trampolines, which leads to the complexity $O(\log N)$ for each query. Thus, the total complexity is $O(N + T \times \log N)$.

2.4 Subtask 4

By using observation 1, we can deduce, for each green trampoline, which will be the next green trampoline that we will encounter on our path. More precisely, for a green trampoline i at position (x_i, y_i), we say that $Next[i] = j$, if j is another green trampoline with $x_j = x_i + 1, y_j \geq y_i$ and there are no green trampolines on line x_j, on columns between y_i and $y_j - 1$. We can first compute $Next[i]$ for each green trampoline i by using binary search on the positions of all green trampolines. Note that it is possible for a green trampoline i not to have a "next" (there is no such j as described above). In this case, we can signal this by setting $Next[i] = \infty$. After computing $Next[]$, we can solve each query as follows:

1. Find the first green trampoline that Little Square will encounter on his path. That is, find y_t, where y_t is the smallest column such that $y_t \geq y_{start}$ and there is a green trampoline on (x_{start}, y_t).
2. Until we reach a green trampoline on line $x_{finish} - 1$, we "jump" from the green trampoline i we are currently on to $Next[i]$.

3. Now, suppose we are on trampoline i, on position (x_i, y_i) and $x_i = x_{finish} - 1$. If $y_i \leq y_{finish}$, then the answer is `Yes`, otherwise it is `No`.

Also, pay attention to the fact that if at some step of our algorithm there is not a "next green trampoline" then the answer is `No`. For each query, the complexity of our algorithm is $O(N)$, as we may visit all the green trampolines during step 2. Overall, the complexity is $O(T \times N)$.

2.5 Subtask 5

The solution for this subtask is based on the solution for the previous subtask. We again need to find $Next[i]$ for each green trampoline i, but we also compute$father[i][j] = Next[Next[Next[\ldots Next[i]]]$, where $Next$ is applied 2^j times. Now, when we need to make a certain number of "jumps" during step 2, say K, we write K as sum of distinct powers of 2. Suppose $K = 2^{p1} + 2^p + \ldots + 2^{ps}$ and suppose that after step 1 we are at the green trampoline (x_t, y_t). We first perform 2^{p1} jumps, which leads to the green trampoline $father[t][p1]$. Afterwards, we perform 2^{p2} jumps, which leads to $father[father[t][p1][p2]$. We continue this process until we perform all the K steps. Now, the complexity for step 2 of our algorithm is $O(\log N)$, so the overall complexity becomes $O(N \times \log N + T \times \log N)$.

An alternative solution is to compute, for each green trampoline i, $SqFather[i] = Next[Next[\ldots Next[i]]$, where $Next$ is applied $\lfloor\sqrt{N}\rfloor$ times. When we need to make a certain number of "jumps" during step 2, we can use the in information found in $SqFather[]$ and $Next[]$ instead of using information found in $father[][]$. This solution leads to an overall complexity $O(N \times \sqrt{N} + T \times \sqrt{N})$, which is fast enough to get 100 points.

3 Problem Football

The author of this problem is Tamio-Vesa Nakajima. In this problem the contestant is asked to solve a variant of the Nim game (whose description can be found online). Rather than piles and objects, the game here is played with classes and students. Moreover, the players are "polite" – they never take more students than in the previous turn, and never take more than K students in a turn. The solutions to the various subtasks follow.

3.1 Subtask 1

When $K = 1$, each player takes one student each turn. The classes are irrelevant, all that matters is the total number of students. If it is even, the second player wins, otherwise the first player wins. It is worth noting that the sum overflows the `int` data type.

3.2 Subtask 2

In this subtask, a brute-force solution is awarded points. Create a function $wins(v)$ that returns 1 if and only if a player wins if starting from a sequence of classes of sizes

described in vector v. Make the function memoize its results. When calculating a result, simply try all possible moves, and use *wins* recursively – if there exists a move that leads to a losing state, then the initial state is winning; otherwise, it is not. It can be proved that, if the answers are memoized, at most 2^N of them must be calculated.

3.3 Subtask 3

A better brute-force solution is needed here. Create an array $d[i_1][i_2][i_3][i_4][i_5]$. This will be 1 if and only if, starting from a state with i_j classes of size j (for $j = 1, 2, 3, 4, 5$), the player who moves first wins. Initialise this array so that each index is between 0 and 10 (it thus has 10^5 total entries). Calculate this array recursively similarly to the previous subtask, by simulating all possible moves, and checking if a losing state can be reached. When answering a query, look up the solution in the array.

3.4 Subtask 4

Here, $N = 1$. Suppose that $K = 1$. Then the solution is determined by the parity of the number of students (like in subtask 1). Otherwise, suppose there are an odd number of students. Then, if the first player takes one student, the second is forced to take one also, and so on – and the first player wins. So we know how to solve the problem if $K = 1$ or N is odd. Now, what if $K > 1$ and N is even?

Suppose this is the case, and note that the player who selects an odd number will certainly lose (since the opponent is given a state where there are an odd number of students, and thus can select one student, and win). So we can consider that both players only make even moves, and "give up" instead of playing an odd move. But, then we can just divide K and the size of the lone class by 2, and reach an equivalent game state!

This observation completes the solution. Simply continue to divide the class size and K by two until either the class size becomes odd, or K becomes 1, at which point the problem can be easily solved.

3.5 Subtask 5 + 6 + 7

As before, if K is one then the game is solved, and if the total number of students is odd, the game is also solved. So suppose K is not one and there are an even number of students. The key observation is that we can divide all class sizes by two (ignoring the remainder) without changing the winner of the game (since the only case in which that remainder will be relevant is if an odd move is made, in which case the game is decided). So adapt the solution from the previous subtask, dividing K and all class sizes by two, until either their sum is odd, or K is one.

4 Problem Cheerleaders

The authors of this problem are Alexa Tudose and Tamio-Vesa Nakajima. One possible solution is to use the Backtracking technique to generate all possible sequences of moves of reasonable length. There are multiple possible optimizations which can decrease the running time. For instance, note that it is redundant to make two big swaps consecutively. Additionally, it can be proved that it is enough to generate only sequence of moves whose length does not exceed $2N$. Depending on implementation, a solution based on Backtracking can pass up to the first 3 subtasks. Another possible solution is to start a DFS/BFS from the given state (i.e. from the initial arrangement of the cheerleaders). At each step, we can expand from the current state to two other states (i.e. the arrangements obtained by applying dance move 1 or dance move 2). Of course, we must not visit any state more than once. In order to efficiently check whether a state has already been visited, we need to use an efficient hashing method to encode each state. It can be proved (as you will see later) that the number of states reachable from the initial state is at most $2^N \times N$. Also, for each state it takes $O(2^N)$ to find the two states that we can expand to. This gives the overall complexity $O(N \times 2^{2^N})$ if implemented properly, which passes the first 3 subtasks. If the hashing method is less efficient, it may lead to other complexities, such as $O(N^2 \times 2^{3^N})$, which passes only the first 2 subtasks.

4.1 Developing a better solution

To get a better solution, it is necessary to devise a simpler way to define the given dance moves. Let $p[i]$ represent the position where the i-th cheerleader (in increasing order of height) is placed. We use 0-indexing for both cheerleaders and positions. So $p[0]$ represents the position where the shortest cheerleader is placed, whereas $p[2^N - 1]$ represents the position where the tallest cheerleader is placed. Now, the two types of dance moves can be described as follows:

1. Big swap: $p[i] \coloneqq p[i]\texttt{xor}1 \ll (N-1)$, for all i. In other words, this operation flips the most significant bit (for all elements of p).

2. Big split: $p[i] := (p[i] \gg 1)|((p[i]\&1) \ll (N-1))$ In other words, this operation moves the least significant bit to the position of the most significant bit, which is equivalent to a cyclic shift to the right (for all elements of p).

Moreover, any possible sequence of dance moves is equivalent to the following:

1. Apply a certain number (say *cnt_shifts*) of cyclic shifts to p. This can be achieved by doing big swap repeatedly *cnt_shift* times.

2. Apply `xor` operator to all elements of p by the same value (say *xor_val*). This can be achieved by doing big swap and big split in a certain order.

4.2 Subtask 4

In this subtask, it is guaranteed that we can make $p = \langle 0, 1, \ldots, N-1 \rangle$, but we want to find out how to do this. We can try each possible value (from 0 to $N-1$) for *cnt_shifts*. Let p_{cnt_shifts} denote the array that would be obtained if we applied *cnt_shifts* cyclic shifts at step 1. Then, in order to obtain $p[i] = i$ in the end, we must take $xor\_val = p_{cnt\_shifts}[0]$. To check whether we indeed have a solution, we must further check that $p_{cnt\_shifts}[i]\texttt{xor}xor\_val = i$, for all i. Once we obtain a solution in terms of *cnt_shifts* and *xor_val*, we should find out a solution as a sequence of dance moves.

4.3 Subtask 5

Our target in this subtask is to minimize the number of inversions of p. We can again try each possible value for *cnt_shifts*. Now, once *cnt_shifts* is fixed, consider two elements of the array obtained after step 1: $p_{cnt\_shifts}[i]$ and $p_{cnt\_shifts}[j]$, where $i < j$. Also, let k be the most significant bit where these two elements differ (i.e. maximum k such that $p_{cnt\_shifts}[i]\&(1 \ll k) \neq p_{cnt\_shifts}[j]\&(1 \ll k)$). These two elements will form an inversion in the end if and only if one of the following conditions is true:

(a) $p_{cnt\_shifts}[i] < p_{cnt\_shifts}[j]$ and $xor\_val\&(1 \ll k) \neq 0$

(b) $p_{cnt\_shifts}[i] > p_{cnt\_shifts}[j]$ and $xor\_val\&(1 \ll k) = 0$

This is true because the bit which decides which element is greater will always be k. Thus, whether the two elements will form an inversion or not in the end will depend only on whether bit k is 1 in *xor_val*. This observation immediately leads to an $O(N \times 2^{2^N})$ solution, which cannot pass subtask 5, but can pass the first 3 subtasks. For all bits k, we want to find $potential[k][0]$ and $potential[k][1]$, which denote the number of inversions that would be created if k were set to 0 or 1, respectively. For all pairs of elements of p_{cnt_shifts}, we can find the most significant bit k where they differ and increase either $potential[k][0]$ or $potential[k][1]$, considering cases a and b listed above. In the end, we fix each bit k to the value that generates less inversions. Note that each bit is fixed independently of the others. To solve the last subtask, we again aim to find $potential[k][0, 1]$. We also need to maintain a frequency table *cnt*, where $cnt[i][j]$ represents the number of elements that have the prefix of length i (the number formed by the most significant i bits) equal to j. Initially, *cnt* is set to 0. We then iterate through all elements in p_{cnt_shifts}. When we are at element $p_{cnt\_shifts}[i]$, we can use *cnt*to find out, for each bit k, how many elements $p_{cnt\_shifts}[j]$, where $j < i$, first differ from $p_{cnt\_shifts}[i]$ at bit k. Having this information, we can update potential. Afterwards, we update *cnt*, by inserting all the prefixes of $p_{cnt\_shifts}[i]$. This solution gives the overall complexity $O(N^2 \times 2^N)$, which is fast enough to get 100 points.