Problem Analysis (Unofficial)
The 2026 ICPC Asia Japan Online First-Round Contest

Problem A: Find the Strongest Card

Proposer: Kazuhiro Inaba Author: Kazuhiro Inaba

There are various ways to solve this problem. For example, if the input contains a 2, output 2; otherwise, if it contains a 1, output 1; otherwise, output the maximum value. Another approach is to compare the input values not directly, but by their remainders after adding 10 and dividing by 13, and output the element with the largest remainder.

Problem B: Vending Machines

Proposer: Mitsuru Kusumoto Author: Mitsuru Kusumoto

The problem can be solved in $O(n)$ time by placing vending machines greedily from left to right. More specifically, for the leftmost remaining door at position x, place a vending machine at $x+d$ and remove all doors covered by it. Repeat this until no door remains. Since the length of the room can be as large as 10^8, if $x+d$ exceeds 10^8, the vending machine must be placed at 10^8 instead of $x+d$; this does not affect the answer.

Since the upper bound on n is small, an $O(n^2)$ implementation will also be accepted.

Problem C: Water Remaining

Proposer: Kiyoshi Ishihata Author: Fumihiko Yamaguchi

This problem asks for the amount of water left in a tank with a stepped bottom after the panels at both ends come off and some of the water flows out.

Water remains in section k if there is a shallower section (one with a smaller depth) on each side of section k. In other words, the condition is $\min_{1\le i<k}(d_i) < d_k$ and $\min_{k<i\le n}(d_i) < d_k$. When water remains, the height of the water surface in section k after the outflow is

$$\max\left(\min_{1\le i<k}(d_i), \min_{k<i\le n}(d_i)\right).$$

If this value is smaller than d_k, the difference is the amount of water remaining in that section. Sum this value over all k.

If, for every k, we independently find the minimum bottom depth among the sections on each side of section k, the asymptotic time complexity will be $O(n^2)$. Since $n \le 100$, this is fast enough to be accepted.

For all sections, the minimum depth among the sections closer to one end than section i—that is, $\min_{1\le j\le i}(d_j)$ for every $1 \le i \le n$—can be computed in $O(n)$ time by maintaining the minimum seen so far. Similarly, the minimum depth toward the other end can be computed for every section in $O(n)$ time. For each section k, take the larger of these two minima and subtract it from d_k. This gives an overall time complexity of $O(n)$.

Problem D: Frequency Sequence

Proposer: Tomohiro Oka Author: Tomohiro Oka

The key to this problem is recognizing the pattern in the sequence.

If we track how the array describing "which value has appeared how many times" changes, we can regard the pattern as changing every time s appears.

For example, when $s = 5$, the sequence is as follows.

$$\begin{aligned}
&\underline{5},\\
1,1,2,1,3,1,4,1,&\underline{5},\\
2,2,3,2,4,2,&\underline{5},\\
3,3,4,3,&\underline{5},\\
4,4,&\underline{5},\\
&\underline{5},\\
6,1,6,2,6,3,6,4,6,&\underline{5},\\
7,1,7,2,7,3,7,4,7,&\underline{5},\\
&\vdots
\end{aligned}$$

For $0 \le i \le s$, the $(i+1)$-st occurrence of s is at position $s^2 + 1 - (s-i)^2$. After that, the value at every position $s^2 + 1 + 2sj$ $(j \ge 1)$ is s. Split the sequence immediately after every occurrence of s, and number the resulting segments starting from 0.

First determine which segment contains the k-th term. If $k \le s^2 + 1$, find the smallest i for which $s^2 + 1 - (s-i)^2 \ge k$, either by binary search or by looping over i. If $k > s^2 + 1$, the segment can be determined from the integer quotient obtained by dividing $k - s^2 - 2$ by the period $2s$.

Segment 0 consists only of a_1, whose value is s. For a segment i with $1 \le i \le s$, its length is $2s - 2i + 1$. Writing the segment as $b_1, \ldots, b_{2s-2i+1}$, the values at odd positions of b start at i and increase by 1, while all values at even positions are i. Thus,

$$b = (i, i, i+1, i, i+2, \ldots, s-1, i, s).$$

For a segment i with $i \ge s+1$, its length is $2s$. Writing the segment as $c_1, \ldots, c_{2s}$, all values at odd positions of c are i, while those at even positions increase from 1. Thus,

$$c = (i, 1, i, 2, \ldots, i, s).$$

Problem E: Shopping Master

Proposer: Masataka Yoneda Author: Masataka Yoneda

First consider the case where the total number of bonus gems, $\sum b_i$, is at least $n-1$. If we first use coins to buy any bottle of sake with $b_i \ge 1$, we can always buy every remaining bottle without any additional coins. Specifically, keep using the bonus gems obtained to preferentially buy bottles with $b_i \ge 1$.

Next consider the case where $\sum b_i \le n-2$. Let the sum be S. We must initially buy $n - S$ bottles with coins; for example, if the sum is $n-2$, we must buy two bottles with coins. At least one of the bottles initially bought with coins should satisfy $b_i \ge 1$, but the others may have $b_i = 0$. Conversely, as long as this condition is met, there is always a way to buy all remaining bottles without additional coins (the proof is analogous to the case where the sum is $n-1$).

Therefore, the following algorithm suffices. In general, buy the cheapest $n - S$ bottles with coins. If all of those bottles have $b_i = 0$, replace one of them with the cheapest bottle satisfying $b_i \geq 1$. However, if every b_i is 0, no such bottle exists, so this case must be handled separately. The answer can be as large as 10^{14}, so a 64-bit integer type such as `long long` is required.

Problem F: Optimizing a Map Application

Proposer: Masataka Yoneda Author: Masataka Yoneda

At first glance, this appears to be a standard shortest-path problem. However, the side length n can be as large as 10^9, so a straightforward breadth-first search taking $O(n^2)$ time is impractical. The key observation is to partition the plane at every x- and y-coordinate that forms a boundary of a building. Since there are m buildings, this partitions the plane into roughly $2m \times 2m$ regions. This technique is known as coordinate compression.

It is then enough to compute the shortest distances from Central Station to the four corners of every region. Indeed, if a destination lies inside a region A, there is always a shortest path from Central Station to the destination that passes through one of the four corners of A. To compute the shortest distances to these corners, construct a graph whose vertices are the corners and whose edges connect horizontally or vertically adjacent vertices, and run Dijkstra's algorithm on it. The time complexity is $O(m^2 \log m)$.

The greatest challenge in this problem is implementation. In addition to requiring rather more code than is usual for the domestic preliminary contest, including special handling for the 1×1 Central Station, a correct program must handle edge cases such as buildings touching along an edge or at a corner and destinations lying on region boundaries. Thus, many teams may have found the solution but still failed to obtain an accepted submission within the contest time.

Problem G: Avoid Collision

Proposer: Taishi Amagai Author: Taishi Amagai

Fix the paths of the two pieces. If the two pieces can occupy the same cell at the same time, that cell must lie in row $h := \lfloor n/2 \rfloor + 1$. Suppose the white piece visits columns $a, a+1, \ldots, b$ in row h, while the black piece visits columns $c, c+1, \ldots, d$ in that row. Since the two pieces must not occupy the same cell, these intervals must be disjoint. Therefore, every valid pair of paths must satisfy either $a \leq b < c \leq d$ or $c \leq d < a \leq b$.

Precompute the number U_j of paths from $(1,1)$ to $(h-1, j)$, the number V_j of paths from $(h+1, j)$ to (n, m), the number X_j of paths from $(n, 1)$ to $(h+1, j)$, and the number Y_j of paths from $(h-1, j)$ to $(1, m)$. For fixed a, b, c, d, the contribution to the answer is $U_a V_b X_c Y_d$ if every cell in columns $a, \ldots, b$ and $c, \ldots, d$ of row h is free of obstacles, and 0 otherwise.

The sum of these contributions over all valid a, b, c, d can be computed with a state-based dynamic program. For the case $a \leq b < c \leq d$, scan from left to right, using states such as "a has not been chosen," "up to a has been chosen," "up to b has been chosen," "up to c has been chosen," and "up to d has been chosen" to record how many endpoints have already been fixed.

The time complexity is $O(nm)$. Implementing the computations of U, V, X, Y as a function makes the code more concise and reduces mistakes involving indices or directions. Note, however, that depending on the implementation, $n = 2$ may require special care.

Problem H: Sorting Swim Rings

Proposer: Ryotaro Sato Author: Ryotaro Sato

First, consider the problem with the target color for each pole fixed in advance. For each pole, the consecutive swim rings of its target color starting from the bottom never need to be moved. Every other swim ring must be moved at least once. There is a simple strategy in which all of these rings are first moved to the temporary storage area and are then returned one by one. This strategy uses two operations per ring. It is also clear that there is no benefit in spending three or more operations on a single ring. Therefore, it is enough to maximize the number of rings that can be handled in just one operation.

A ring needs only one operation if, when it is moved, every unwanted ring has already been removed from its destination pole. Thus, we need to optimize the order in which unwanted rings are removed from the poles. This can be computed by a DP keyed by the set of poles from which all unwanted rings have already been removed. Choose one pole to process next. For each ring removed from it, add one operation if its destination pole has already been processed, and two operations otherwise. If the number of rings of each color on each pole is precomputed, this DP runs in $O(2^n n)$ time.

For the original problem, in which the target colors are not fixed, we can use a similar DP while deciding which color to assign to each pole. More specifically, let $dp[S][T]$ be the minimum number of operations so far when S is the set of poles already processed and T is the set of colors already used. Suppose we assign an unused color $c \notin T$ to an unprocessed pole $i \notin S$. If we know both the number of consecutive rings of color c starting from the bottom of pole i and the number of rings on pole i whose colors belong to T, then $dp[S \cup \{i\}][T \cup \{c\}]$ can be updated in $O(1)$ time. The final answer is $dp[\{1, \ldots, n\}][\{1, \ldots, n\}]$.

Only states with $|S| = |T|$ need to be considered, so the number of states is

$$\sum_{i=0}^{n} \binom{n}{i}^2 = \binom{2n}{n} \in O(4^n/\sqrt{n}).$$

Therefore, the overall time complexity is $O(nm + \binom{2n}{n} n^2)$, which is sufficiently fast in practice. The transitions can be computed more carefully to improve this further to $O(nm + \binom{2n}{n} n)$, but that is not necessary for this problem.

Problem I: Speed Limit

Proposer: Masataka Yoneda Author: Masataka Yoneda

Consider increasing the available cost by 2 at a time, so that $L = 2, 4, 6, 8, \ldots$. Whenever the available cost increases by 2, we can increase the speed on one interval by 1 km/h. This suggests a greedy algorithm that repeatedly chooses, among the changes that obey the speed limits, the one that minimizes the total travel time. For example, when $a = (4, 4, 1, 4, 4, 5, 4, 4)$ and $L = 10$, the speeds change as shown in Table 1. In particular, when increasing the cost from 8 to 10, speeding up the left interval traveled at 2 km/h saves about 0.33 hours, while speeding up the right interval traveled at 3 km/h saves about 0.42 hours; therefore, the right interval is sped up.

This greedy algorithm can in fact be proved to always produce an optimal solution, as shown below. However, since $L \leq 10^{10}$, a direct implementation takes $O(nL)$ time and cannot meet the time limit. Instead, binary-search the reduction in travel time obtained by increasing the cost by 2; call this reduction the **gain**. More precisely, construct a function that quickly computes

Table 1: Speeds on each interval as the cost increases. Newly increased speeds are highlighted. Units are km/h.

Cost	0–1 km	1–2 km	2–3 km	3–4 km	4–5 km	5–6 km	6–7 km	7–8 km	Time
2	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	8.00
4	1	1	1	**2**	**2**	**2**	**2**	**2**	5.50
6	**2**	**2**	1	2	2	2	2	2	4.50
8	2	2	1	**3**	**3**	**3**	**3**	**3**	3.67
10	2	2	1	**4**	**4**	**4**	**4**	**4**	3.25

the total cost required after applying every improvement whose gain is at least a real value x, and binary-search for a value of x that makes this cost equal to L.

We explain how to compute the cost using $a = (40, 40, 10, 40, 40, 50, 40, 40)$ and $x = 0.01$ as an example; this a is ten times the previous example. At first, the greedy algorithm increases the speed over the entire highway. Even after raising it to the minimum speed limit, 10 km/h, the gain still exceeds x, so the speed is increased to 10 km/h. The intervals whose speed can still be increased are then split into a 2 km interval on the left and a 5 km interval on the right.

- On the left, the minimum speed limit is 40 km/h. The gain falls below x when increasing the speed from 14 km/h to 15 km/h, so the speed is increased to 14 km/h.
- On the right, the minimum speed limit is 40 km/h. The gain falls below x when increasing the speed from 22 km/h to 23 km/h, so the speed is increased to 22 km/h.

Thresholds such as 14 km/h can be computed easily by solving a quadratic equation. The final speeds are $(14, 14, 10, 22, 22, 22, 22, 22)$, and the cost is 52. In this way, increase the speed of the current interval as far as possible; if it reaches the minimum speed limit before reaching the gain threshold, split the interval and solve the resulting parts recursively. The minimum speed limit can be queried in $O(\log n)$ time with a segment tree, so one cost computation takes $O(n \log n)$ time. The total running time, including the binary search, is fast enough.

Implementation notes. Special care is needed when multiple improvements have the same gain. For example, when $a = (2, 1, 2)$ and $L = 4$, the gain is 0.5 both when the cost increases from 2 to 4 and when it increases from 4 to 6. Consequently, there is no value x for which the cost is exactly L. In such a case, compute the threshold x at which the cost first reaches at least L, and then adjust the travel time to account for the excess cost. The range and number of iterations of the binary search also require care.

Proof of the greedy algorithm. Represent the speed limits as a histogram and divide it into horizontal pieces. Similarly, represent the actual driving speeds by filling the histogram from the bottom. A solution that fills only part of one horizontal piece is clearly suboptimal: filling the entire lowest such piece can reduce the travel time without changing the cost. Filling one whole horizontal piece costs 2. The reduction in travel time obtained by filling a piece is larger for lower pieces and smaller for higher pieces, so the gains are monotone. Therefore, always choosing the available piece with the largest gain is optimal, which proves the greedy algorithm.

Problem J: Maximum Scaling

Proposer: Mitsuru Kusumoto Author: Mitsuru Kusumoto

We binary-search the scale factor, fixing its value for the rest of the discussion. The problem is unchanged if, instead of translating Q', we translate P so that P contains Q', so we use this equivalent formulation.

The region on and inside the boundary of P can be represented as the intersection of n half-planes. Specifically, extend each edge of P into a line and take the half-plane on the side containing P.

Consider one of these half-planes, written as $ax + by \leq c$. If it is translated by r in the x-direction and by s in the y-direction, its inequality becomes

$$a(x - r) + b(y - s) \leq c,$$

or equivalently,

$$ax + by - c \leq ar + bs.$$

Polygon P contains Q' if and only if all n half-planes contain every vertex of Q'. Substituting the m pairs $(x, y) = (x'_1, y'_1), \ldots, (x'_m, y'_m)$ into the inequality above shows that the condition for one half-plane is

$$\max_{j=1,\ldots,m} (ax'_j + by'_j - c) \leq ar + bs.$$

Once the coefficients a, b, c are known, the left-hand side can be computed in $O(m)$ time. This gives the feasible region for the translation (r, s) with respect to one half-plane; this region is itself a half-plane. Find these feasible regions for all half-planes and check whether their intersection is empty. There are various ways to test whether the intersection is nonempty. Although efficient $O(n \log n)$ algorithms exist, an $O(n^2)$ method such as repeatedly clipping a convex polygon is sufficient for this problem.

Overall, the problem can be solved as follows:

- Compute the half-plane coefficients a, b, c for every edge of P: $O(n)$ time.
- Perform $O(\log(1/\varepsilon))$ iterations of binary search for the required precision ε.
- Compute the n half-planes describing feasible translations (r, s): $O(nm)$ time.
- Determine whether the feasible region for (r, s) is empty: $O(n^2)$ time.

Thus, the overall time complexity is $O(\log(1/\varepsilon)\, n(m + n))$.