2024 ICPC Asia Taichung Regional Contest

Problem Analysis and Hints

Problem Title		Task Author(s)*
A	The Bento Box Adventure	Meng-Tsung Tsai
B	Bowling Frame	Ting-Feng Lin
C	Cube	Peter Rossmanith
D	Drunken Maze	Peter Rossmanith
E	Beautiful Array	Kai-Chun Ho
F	Segmentation Folds	Ting-Wei Chen, Shang-En Huang
G	Grid Game	Yuhang Sheng
H	Sheet Music	Ronnie Cheng
I	Auto Complete	Chung-Sheng Wu
J	Bottle Arrangement	Ta-Wei Tu, Brian Tsai
K	Trophic Balance Species	Meng-Tsung Tsai
L	Building Castle	Ronnie Cheng
M	Selection Sort	Bangye Wu
N	Railway Construction	Ta-Wei Tu, Brian Tsai, Ronnie Cheng

*: Task authors contributed the main idea to each problem. The tasks (including the task descriptions, test data, and solution programs) are parepared and polished by the entire contest judge team and ChatGPT.

The judges would also like to thank Po-Yu Chou, Yu-Hsun Chiang, Li-Yuan Chiang, Nikil Roashan Selvam, Conor Kennedy, and Misha Ivkov for their valuable feedback to the tasks.

Problem A: The Bento Box Adventure

This is a *missing number problem*. The solution is simply $15 - (a + b + c + d)$.

Problem B: Bowling Frame

It is immediate that if one wants to build a frame of size k, then at least $k(k+1)/2$ pins are needed in total. It turns out that this is also sufficient, that if $w + b \geq k(k+1)/2$ then the construction is always possible. To see this, note that $\max\{w, b\} \geq \lceil k(k+1)/4 \rceil \geq k$, and thus if we have enough pins then we can always choose one of the two colors for the k-th row. Then, the remaining $k - 1$ rows can be constructed inductively.

Problem C: Cube

- Dynamic programming
- Use a DP table that is indexed by already used planes

This problem can be solved using dynamic programming, where we use binary numbers to represent planes that were already used.

Imagine that we are choosing the cubes from each yz-slices in the ascending order of their x-coordinates. Here, a *yz-slice* is referred to a collection of cubes (a plane) with the same x-coordinates. When we fix an x-coordinate $x = k$, we know that exactly $k - 1$ values of y-coordinates and z-coordinates must have been used. Therefore, we can recover the x-coordinate from the given subsets Y and Z of used y- and z-coordinates.

For each subsets Y and Z of the same size k, let $dp(Y, Z)$ be the minimum sum of all cubes' values chosen from the first k yz-slices, such that the chosen cubes altogether occupy all y- and z-coordinates in subset Y and Z. Now, we can write the recurrence relation:

$$dp(Y, Z) = \min_{y \in Y, z \in Z} \Big\{ dp(Y - \{y\}, Z - \{z\}) + value(k, y, z) \Big\},$$

where $value(k, y, z)$ is the number written at the coordinate (k, y, z). The base case is of course $dp(\emptyset, \emptyset) = 0$. To implement this dynamic program, we use binary number ranging from 0 to $2^n - 1$ to encode the subset of coordinates $\{1, 2, \ldots, n\}$. We can simply run two for-loops: one for enumerating all Y subsets, another for enumerating all Z subsets. Notice that the state (Y, Z) is valid only when $|Y| = |Z|$, where we can check this using `__builtin_popcount()` efficiently. For each valid state (Y, Z), we run another two for-loops to enumerate all possible $y \in Y$ and $z \in Z$ indices.

Thus, the estimated running time of this dynamic programming algorithm is then

$$2^{2n} + \sum_{k=1}^{n} \binom{n}{k}^2 k^2 = 2^{2n} + n^2 \cdot \binom{2n-2}{n-1} \approx 1.18 \times 10^8 \text{ when } n = 12.$$

This algorithm requires allocating a huge array of 2^{2n} integers, which is roughly 64MB.

Problem D: Drunken Maze

Short summary:

- BFS
- Find the shortest path in graph whose vertices are the positions in the maze augmented with the a number k and a direction r indicating that the position was reached by k consecutive steps in direction r.

Problem E: Beautiful Array

The simplest solution is to make the array $[b, b, 3a - 2b]$. The mean is a because the sum is $3a$. The median is b because the middle element after sorting the array is always b.

Task Author's Remark. There is no "correct" thinking process, and not all observations are useful. Try to get the most out of the given rules. For example, rephrasing the rules (if mean $= a$, then sum $= a \times$ array length), discovering hidden properties (if more than half elements are b, then median $= b$), making new rules as combination of known rules (if more then half but not all elements are b, you can set the remaining elements to get desired sum), and so on. Through practicing, more and more approaches will be available and eventually become your "intuition".

Problem F: Segmentation Folds

Short summary:

- Preprocessing prime numbers.
- Brute force all folding sequences.

The algorithm is going to recurse on subintervals, and so let us use L and R to denote the input values ℓ and r to avoid confusion with the subproblems that we will consider. We first observe that no matter how a segment is being folded, the interval it occupies will always be within the range $[L, R]$. Hence, the prime numbers we need to check fall within the range $[2L, 2R]$.

In preprocessing, we use a sieve method to find all primes within the range $[2L, 2R]$ and store them in an array. Recall from Euler' result that $\sum_{p<n:\ p \text{ is prime}} 1/p = \ln \ln n + O(1)$. The sieve

method includes spending $O(\sqrt{2R} \cdot \log\log R)$ time building all prime numbers up to $\lceil\sqrt{2R}\rceil$ and then apply the sieve again to the range $[2L, 2R]$ in $O((R-L)\log\log R)$ time.

From now on, checking if a number is prime can then be done in $O(1)$ time. The second part of the algorithm is to generate all possible folding sequences using a (divide and conquer style) brute force algorithm: on an interval $[\ell, r]$, the algorithm first finds the largest x such that $\ell + x$ is a prime and the smallest y such that $y + r$ is a prime. This can be done in $O(1)$ time by preprocessing all binary search results of $[2L, 2R]$ in $O(R-L)$ time. Then the algorithm simply simulates folding the segments and recurses on $[(\ell + x)/2, r]$ and $[\ell, (r + y)/2]$. This suffices to get accepted.

To complete the analysis, we now analyze the number of subproblems $[\ell, r]$ encountered during the simulation. Consider the recursion tree of the simulation. For each subproblem $[\ell, r]$ we define the potential $\Phi(\ell, r)$ to be the amount of prime numbers within the interval $(2\ell, 2r)$, *excluding* both 2ℓ and $2r$. Observe that (1) any subproblem has non-negative potential, (2) any child subproblem has potential strictly smaller than its parent, and (3) if a subproblem $[\ell, r]$ has two childrens $[(\ell + x)/2, r]$ and $[\ell, (r + y)/2]$, then the sum of their potentials is at most $\Phi(\ell, r)$. Note that in fact, $\Phi((\ell + x)/2, r) + \Phi(\ell, (r + y)/2) = \Phi(\ell, r)$ if and only if $\ell + r$ is not a prime number.

Therefore, the entire recursion tree has at most $\Phi(L, R)$ subproblems with non-zero potential, and hence a total of at most $3\Phi(L, R) \leq 6(R - L)$ subproblems.

Hence, the running time of the entire algorithm is $O((R - L)\log\log R)$.

Problem G: Grid Game

This is an implementation-heavy problem. We describe a possible walkthrough of the algorithm:

1. Clean up the lines so that each line does not overlap, and any two lines in the same column are seperated by at least one empty cell.

2. If the first/last row/column is completely filled, then removing the row/column does not affact the answer. So we clean up the boundaries and make sure each of the first/last row/column has at least one white cell.

3. Now we say that two lines are *adjacent* if there is a cell $c = (c_x, c_y)$ on the first line and a cell $d = (d_x, d_y)$ on the second line such that c and d are adjacent. That is, $\max(|c_x - d_x|, |c_y - d_y|) = 1$. Note that after Steps 1 and 2, adjacent lines will only appear on different columns.

4. Add an artificial marked box surrounding the grid and use a union-find data structure to compute the connected components of lines with respect to the above adjacency rela-

tionship.

5. Now, there are only a few cases where a cell can be a cut point, and this will involve "connecting" two lines which were not adjacent but connected in the union-find data structure. The following picture should be the best explanation.

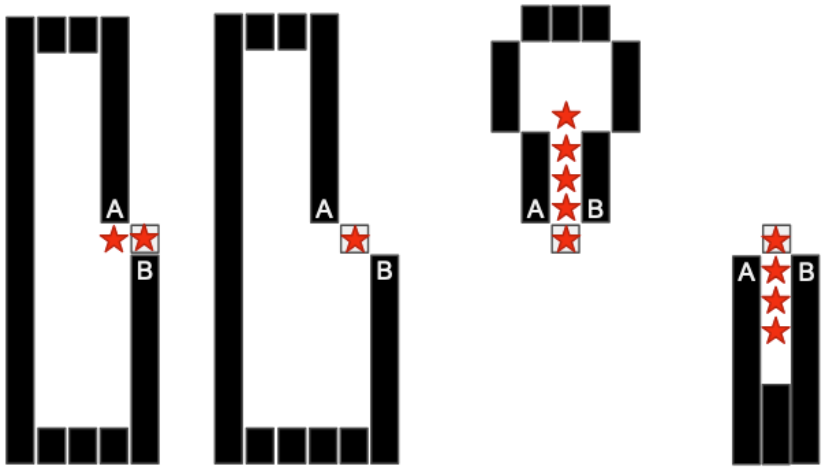


Figure 1: The possible cases of a cell being a cut point. Note that the two lines A and B can be connected through the artificial box we added around the grid.

   Note that this can only happen for lines that are on columns x and $x+1$ or x and $x+2$ (for some x).

6. Alternatively, one can carefully compress the graph (again utilizing observations given in the figure above) and run a standard cut-vertex algorithm on the compressed graph.

Problem H: Sheet Music

- For any song, we can merge the continuous subarray of the same pitch into one single element. After merging, if the length of the song becomes m, it is obvious that $\binom{n-1}{m-1}$ different songs will merge into the same shape.
- Hence, for all lengths $1 \le i \le n$, we want to calculate $D_k(i)$ — the number of songs of length i with the restriction that no two adjacent notes have the same pitch.
- Observe that $D_k(i)$ is exactly the number of length $i-1$ sequences consists of symbols "<" and ">" such that a symbol does not appear k times consecutively. A possible recurrence

relation of $D_k(i)$ can be the following:

$$D_k(i) = \begin{cases} 2^i & \text{if } 0 \le i < k, \\ 2^k - 2 & \text{if } i = k, \\ 2 \times D_k(i-1) - D_k(i-k) & \text{otherwise } i > k. \end{cases}$$

Hence, computing $D_k(i)$ for all $1 \le i \le n$ can be done in $O(n)$ time.

- The overall solution is then $\sum_{i=1}^{n} \binom{n-1}{i-1} D_k(i)$ modulo 998244353.

Problem I: Auto Complete

A naive solution is to maintain the current T after each action, and then try to match the prefix of each P_i. The time complexity of this might be $O(N^2 \times (\text{string comparison}))$. To optimize it, we can use a *trie* data structure to store the patterns. For each trie node, we need to maintain:

1. Pointers to child nodes. It would be a map of characters to another trie node.
2. The best pattern ID after this node.

Each time we add or remove a pattern, we must maintain the best pattern by choosing the longest, smallest lexicographic path from each child node. To maintain this, we might need another data structure to keep the child nodes because it is a RMQ question. A linear time update might also work if the number of child nodes is not large. The maximum number of child nodes would be the number of printable characters, which is about 94. Whether another data structure is needed may vary by the implementation.

For each `append` or `backspace` operation, we need to maintain a cursor on the trie. When appending, we walk down the trie to the corresponding trie node. When backspacing, we walk back to the parent nodes. The node cursor stop on will provide the answer. Note that, the cursor may move down to a null node or up to the root node. The current node may also be removed by the add / remove operation. These edge cases must be handled properly.

Problem J: Bottle Arrangement

There are two parts to solve in this problem. The first part is to determine if there is a valid arrangement, while the second part is to determine the minimum number of caps to be removed.

Observe that any valid arrangement splits the sequence into a non-decreasing prefix and a non-increasing suffix. To solve the first part, we first assume that all caps of the bottles have been removed, and let $b'_i = b_i - 1$.

The algorithm enumerates all possible lengths of the prefix/suffix. Once the length is fixed, the following algorithm checks if there is a valid arrangement.

Suppose that the first ℓ bottles are arranged non-decreasingly (so the last $n-\ell$ bottles will be arranged non-increasingly). In this scenario, we define the adjusted height: $a'_i = \min\{a_i, a_{i+1}, \ldots, a_\ell\}$ for all i ($1 \le i \le \ell$) and $a'_j = \min\{a_{\ell+1}, \ldots, a_j\}$ for all $j > \ell$. Without loss of generality, assume that the heights $\{b'_i\}$ are sorted, i.e., $b'_1 < b'_2 < \cdots < b'_n$. Now, for each a'_i, let c_i to be the largest *index* such that $b'_{c_i} < a'_i$, and we collect all these indices $\{c_i\}$.

Observation J.1. *There is a valid arrangement if and only if for all values $v \in \{1, 2, \ldots, n\}$, there are at most v indices whose values are no more than v. I.e., $|\{c_i \ : \ c_i \le v\}| \le v$.*

Proof. "⇒" by the pigeonhole principle; and "⇐" can be proved by a greedy assignment. □

Implementing the above algorithm naively takes $O(n^2)$ time, since we have to try all possible prefix lengths ℓ. Now, we use a data structure to speed up the checking process. Imagine that we gradually increase ℓ from 1 to n. During the increment, the values $a'_1, \ldots, a'_\ell$ can be represented using a monotonic stack, with $O(n)$ total changes. Similar analysis can be applied to the suffix when we decrease ℓ from n to 1. To simulate the actual enumeration process of $\ell = 1, 2, \ldots, n$, we precompute and memorize all changes to the monotonic stack for the suffixes, and revert the stack operations along the actual simulation.

For each stack operation, we either "insert a bulk of values v" or "remove a bulk of values v". Then, we will apply Observation J.1 to check if there is a solution. This suggests the following implementation: we keep a relative counter $C[v] := v - |\{c_i \ : \ c_i \le v\}|$ for all $v \in \{1, 2, \ldots, n\}$. Initially we have $C[v] = v$. Upon inserting (resp. deleting) the value v with multiplicity k, we decrease (resp. increase) *all* counters $C[v], C[v+1], \ldots, C[n]$ by k. By definition of $C[v]$, if some $C[v]$ value become negative after an update, then there will be no valid arrangement.

We keep two set of counters: $C[v]$ and $\hat{C}[v]$. Here $C[v]$ is for $\{b'_1, \ldots, b'_n\}$ and $\hat{C}[v]$ is for the capped bottles $\{b_1, b_2, \ldots, b_n\}$. The final algorithm maintains both $C[v]$ and $\hat{C}[v]$. If there exists valid arrangements for $C[v]$, we check the data structure for $\hat{C}[v]$ and count how many values are -1. The number of v's such that $\hat{C}[v] = -1$ will be the minimum number of caps to be removed. In conclusion, the entire algorithm can be implemented in $O(n \log n)$ time.

Problem K: Trophic Balance Species

Short summary:

- Decompose D into $O(k \log n)$ chains by a greedy algorithm
- Use DP with respect to these chains to compute the answer.

- Let us begin with the assumption that D is a DAG. Dilworth's Theorem states that the size of a largest antichain equals the size of a minimum chain decomposition. Thus, V can be partitioned into k subsets such that the nodes in each subset are passed by a single path in D. Note that the path may contain more nodes than the subset.

- Given the partition of V into k subsets, the set of nodes that a node x can reach via directed paths can be presented by a k-tuple of integers, each corresponding the length of a suffix of one of the k paths. In this way, the transitive closure of D can be computed in $O(mk)$ time using standard dynamic programming.
- We do not partition V into the minimum number of subsets. Instead, we parition V into $O(k \log n)$ subsets. This can be computed by greedily removing a path from D that covers the most number of uncovered nodes, taking $O(mk \log n)$ time. In total, the running time is $O(mk \log n)$. The approximation factor can be analyzed using an approach similar to the analysis of the $O(\log n)$-approximation algorithm for dominating sets, which is based on submodularity.
- Lastly, for an arbitrary directed graph D, one can apply SCC decomposition to reduce it to the above DAG case.
- An incorrect solution might involve partitioning V into the fewest number ℓ of node-disjoint paths. Although this can be computed in $O(\sqrt{n}m)$ time using bipartite matchings, ℓ can be much larger than k. This gives the main reason why such a solution exceeds the time limit, rather than the matching algorithm being too slow.
- To our knowledge, the algorithm described above are first presented in Topi Paavilainen's 2017 thesis and later mentioned in several papers on minimum chain decomposition for DAGs. We also independently develop it for a hardness reduction, so we believe this problem is likely solvable for teams familiar with fundamental concepts in computational graph theory.

Problem L: Building Castle

- Assuming that we have the symmetry center p of B. Let A' be the point symmetry convex polygon of A with respect to p, it can be easily derived that the minimum can happen when $B = A \cap A'$.
- Hence, all we have to do is to find the best place to put the symmetry center. According to the Brunn–Minkowski theorem, the function mapping center to the square root of the area is a convex function within the range of A. Thus, we can use any convex-like extreme point searching algorithm to find the best place.
- An implementation is to use a two-layer ternary search to find the minimum, with a Half-plane intersection algorithm to calculate $A \cap A'$. The total running time can be $O(n \log n \log^2(\epsilon^{-1}C))$.
  - However, such a solution will get TLE. One should optimize the time complexity of the Half-plane intersection to $O(n)$ per query by preprocessing the order of the

half-plane lines.

- There might still be some precision issues that need to be carefully estimated.

Problem M: Selection Sort

- Let the range where the first (resp. second) operation is applied be $[1, x]$ (resp. $[y, n]$).
- Guess y.
- If $[1, x]$ and $[y, n]$ are disjoint, find the x that is furthest from y such that the elements within the range $[x, y]$ are in the correct positions.
- Otherwse $[1, x]$ and $[y, n]$ intersect. Find the smallest x such that if you sort the first x elements then the first $y - 1$ elements are in the correct positions.
- The above can be implemented in $O(n)$ time. An $O(n \log n)$-time algorithm also can pass all the test cases within the time limit.

Problem N: Railway Construction

Let us assume for without loss of generality that all a_u's are distinct and consider first how to compute a single MST of the input graph. There are several ways of doing so, but we choose one here that will be easy to generalize to solve the whole problem.

For each vertex u, let us pick the edge (u, n_u) in the graph to a vertex n_u with the smallest value of a_{n_u}. This can be done in $O(m)$ time for all u, assuming that the vertices are already sorted in increasing order with respect to a_i's.

It is easy to see that the edges $\{(u, n_u)\}_{u \in V}$ contain the parts of the MST inside the connected components induced by them. That is, we can compute an MST on $\{(u, n_u)\}_{u \in V}$, contract these edges and compute the rest of the MST on the contracted graph.

Observation N.1. *There are at most $O(\sqrt{m})$ vertices having degree more than one in edges $\{(u, n_u)\}_{u \in V}$.*

Proof. A vertex u that has degree more than one if and only if $u = n_v$ for some v. However, note that most v's will have their n_v be among the first $O(\sqrt{m})$ vertices u with the smallest a_u's. This is because if n_v is not among the first $O(\sqrt{m})$ vertices, then v is incident to $\Omega(\sqrt{m})$ anti-edges, and thus there can be at most $O(\sqrt{m})$ such v's. □

Note that contracting all (u, n_u) corresponds to the first round of Borůvka's algorithm, and what Observation N.1 implies is that there can be at most $O(\sqrt{m})$ connected components in the graph after this. We can thus finish the MST computation by running Prim's algorithm on

the contracted graph in $O(\sqrt{m}^2) = O(m)$ time. This step requires some careful implementation to compute for each pair of connected components the minimum weight edge between them. A naive one may require querying a hash table $O(m)$ time to check whether a certain edge exists, but this is likely to have a very large constant overhead and may result in a TLE. Fortunately, there is an elegant implementation that optimizes these checks to run in just $O(1)$ time (with a small constant overhead) per call by arranging them properly. Exploring this optimization is left as an exercise for the reader.

To solve the full problem, we note that Observation N.1 actually implies something stronger, that all but $O(\sqrt{m})$ vertices are leaves in $\{(u, n_u)\}_{u \in V}$. Moreover, since the Prim's algorithm that we run after Borůvka's only adds $O(\sqrt{m})$ edges to the MST, overall the number of non-leaves in the tree is very small.

Observation N.2. *The MST has only $O(\sqrt{m})$ non-leaf nodes.*

With Observation N.2, we know that we only need to recompute the MST for the $O(\sqrt{m})$ non-leaf nodes. If we spend $O(m)$ time for each of them (using the algorithm described above), this gives an $O(m\sqrt{m})$ time algorithm for the full problem.

Additional Speedup. It turns out that spending $O(\sqrt{m}^2)$ time in Prim's algorithm for each vertex removal may still be slightly too slow, depending on the implementation. One potential speed up is to only apply the algorithm on components that are broken by the vertex removal, spending $O(\min(d_i, \sqrt{m})^2)$ time instead, where d_i is the degree of the removed vertex i in the MST. This may lead to a faster implementation. To justify this, with a careful analysis it turns out this will push some of the $O(m\sqrt{m})$ terms in the total running time to an $O(n\sqrt{m})$ term instead (though the overall time complexity is still $O(m\sqrt{m})$).