培正喇沙 編程挑戰賽
LA SALLE – PUI CHING PROGRAMMING CHALLENGE

10TH · 2026

JUL 25, 2026 (SATURDAY)

LA SALLE COLLEGE

SOLUTIONS

ID		NAME	AUTHOR / PREPARER	DIFFICULTY
A		Anniversary	Cheng Yu San Kwok Ching Him	★★☆☆☆
B		Burnside's Lemma	Wong Chun Wong Cheuk Kiu	★☆☆☆☆
C		Circle Quiz Relay	Cheng Yu San Cheng Yu San	★★★★☆
D		Doomsday of the Disguisers	Lu Yi Fung Mak Chun Hin Ivan	★☆☆☆☆
E		Emoji Riddle	Yeung Man Tsung Yuen Lok Kan Ethen	★★★★★
F		Forest Guardian Traversal	Wong Chun Lu Yi Fung	★★★★★★
G		Go Shoot!	Wong Man Hang Wong Man Hang	★★★★★
H		Hollow Knight: Silksong	Lung Sui Hei Lung Sui Hei	★★☆☆☆
I		Ice Cream Sort	Yeung Shing Hei Angus Yeung Shing Hei Angus	★★★☆☆
J		Jettisoned Joins	Wong Ho Yan Wong Ho Yan	★★★☆☆
K		KMP Evasion	Lai Hong Tung Honrich Choy Tsz Chai Berton	★★★☆☆
L		Last Order	Cheng Hei Chit Wong Tsz Him	★★★★☆

The authors expect that the problems were to be solved by the following percentage of teams

From 1 to 6 stars: >75%, 50-75%, 25-50%, 5-25%, 1-5%, <1%

A – Anniversary

Problem Summary

Given an $N \times (M+2)$ grid forming the number `10` (a line of N balloons in column 1 and an $N \times M$ perimeter loop of balloons in columns 3 to $M+2$), assign K colors to all $3N+2M-4$ balloon cells matching the exact counts $A_1, A_2, \ldots, A_K$ so that no adjacent balloons share a color, or determine if it is impossible.

Solution

The problem asks for a proper vertex coloring on a graph consisting of two disconnected components: (digit `1`) and (digit `0`).

Crucially, the length of the digit `0` loop is $L = 2N + 2M - 4 = 2(N + M - 2)$, which is **always an even integer** for any integers $N, M \geq 3$. Because L is even, the graph is bipartite.

The problem can be solved using the following approach:

1. **Sorting Colors:** Sort the available balloon colors in non-increasing order of their frequencies, and flatten them into a single sequence S of length $3N + 2M - 4$.
2. **Order Traversals:** Convert the target digit shapes into ordered 1D coordinate arrays:
   - Array v (Digit 0): Trace the outer boundary (perimeter loop) of digit 0 in a clockwise direction.
   - Array u (Digit 1): Trace the cells of digit 1 from top to bottom.
3. **Interleaved Assignment:** Iterate through the sequence S from Step 1 sequentially, popping the next available color from S and assigning it to cells in four distinct passes:
   (a) Fill odd indices of v $(v_1, v_3, v_5, \ldots)$
   (b) Fill odd indices of u $(u_1, u_3, u_5, \ldots)$
   (c) Fill even indices of v $(v_2, v_4, v_6, \ldots)$
   (d) Fill even indices of u $(u_2, u_4, u_6, \ldots)$
4. **Validation:** Run an adjacency check over the resulting grid. If no two adjacent balloon cells share the same color, output `POSSIBLE` along with the grid; otherwise, output `IMPOSSIBLE`.

Comments

The author expected around 33 to 50 teams to solve this problem, but only 21 teams solved it during the contest, which is lower than expected. We encourage contestants to explore different greedy approaches more thoroughly and be bolder when testing potential solutions.

B - Burnside's Lemma

Problem Summary

In this task, you are given N instructions on how to flip (along the horizontal or vertical axes) and rotate (clockwise or anti-clockwise) a steak. The goal is to find the final state of the steak - whether it is flipped and how many degrees it is rotated with respect to its original state.

Solution

Whether or not the steak is flipped at the end is easy to handle, just maintain a boolean variable and flip it whenever there is an ***H*** or ***V*** instruction.
To find the rotation, we have to handle it differently for each of the 4 instructions.

- ***H***: newRotation = 360 - oldRotation
- ***V***: newRotation = 180 - oldRotation
- ***C X***: newRotation = oldRotation + X
- ***A X***: newRotation = oldRotation - X

Remember to make sure the answer is a non-negative integer less than 360, you might want to do newRotation = (oldRotation % 360 + 360) % 360 after every operation.

Comments

About 50 teams are expected to solve this task during contest time.

C - Circle Quiz Relay

Problem Summary

There are N nodes and a permutation P of $\{1, 2, ..., N\}$. Starting from any node u, repeat the following traversal process until all N nodes have been visited:

1. If node u has not been visited before, mark u as visited and set u to P_u.
2. Otherwise, set u to $(u \bmod N) + 1$.

The cost of a traversal is defined to be the number of times case 2 is triggered throughout the process. The task is to find the minimum cost achievable by choosing the starting node optimally.

Solution

Since P is a permutation of 1 to N, the graph formed by directed edges $i \to P_i$ consists of a set of disjoint directed cycles. Following P from any unvisited node u will visit every node in u's cycle exactly once without encountering any previously visited nodes, until returning to u.

Suppose we choose s to be our starting node:

- Initially, the cycle containing s becomes visited.
- A pointer p starts at s and advances in order $s \to s+1 \to s+2 \ldots \pmod N$.
- Whenever p hits a node whose cycle has not been visited yet, that cycle is immediately marked as visited.
- The process terminates as soon as all cycles are marked as visited.

Notice that the cost starting from s is simply the difference between p and s when the process terminates.

Thus, the problem can be reduced to finding the minimum length of a contiguous circular window $[L, R]$ that contains at least one node from every cycle.

This problem can be solved using a sliding window algorithm:

- Find all connected cycles of P and assign a cycle ID to each cycle. Label each node with its corresponding cycle ID.
- Duplicate the array of cycle IDs to length $2N$ to handle the circular array.
- Maintain a frequency map of cycle IDs within the current window $[L, R]$.
- Expand R until all cycles are present in the window.
- Shrink L to find the minimal valid window length.

The overall time complexity of the above algorithm is $\mathcal{O}(N)$.

Comments

The author expected this task to be solved by 10–15 teams, but eventually more teams than expected were able to solve it.

The core challenge of this task is to reformulate it into a standard sliding window problem. Once you see that connection, solving it becomes straightforward.

In competitive programming, the ability to establish a bijection between an unseen, complex problem to a simple, familiar technique is very important. Most of the difficulty comes from looking past the custom rules to find the standard pattern hiding underneath, instead of implementing the solution.

D - Doomsday of the Disguisers

Problem summary

Given an N x M grid with each cell containing exactly 1 chameleon, Q distinct operations which are either killing a whole row, or a whole column of chameleons. Count the number of killed chameleons after all operations.

Solution

Initialize a 2D boolean array of size N x M with all of the entries being false. For every operation, loop through the corresponding row or column, and mark all of the entries looped through as true.

Count the number of true entries in the boolean array and output it.
Time Complexity: O(NM+Q)

There exists a better solution that can work in O(Q). To be specific, let the number of row operations be *R*, and that of column operations be *C*, the answer is simply R x M + C x N - R x C. This expression comes from adding the number of chameleons killed by the row operations, the number of chameleons killed by the column operations, then subtracting the chameleons killed by both operations.

Comments

The author expected this task to be solved by approximately 60 teams.
Solving this problem requires simple array processing or game observation. However, the second approach is significantly simpler to implement. Therefore, participants are encouraged to consider if a simpler solution exists before beginning their implementation.

E – Emoji Riddle

Problem Summary

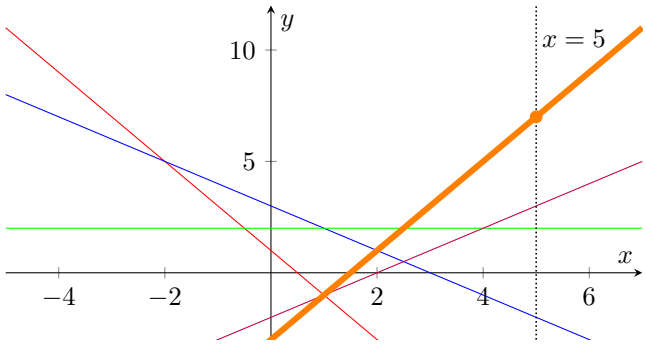
Given N emojis with feature vectors $(t_i, 1)$ and M words with feature vectors (x_i, y_i), where the relevance between an emoji i and a word j is defined as $t_i \cdot x_j + 1 \cdot y_j$. The *relevance* between an emoji riddle (a pair of emojis) and a word is the sum of the two individual relevance values.

Find the number of emoji riddles that satisfy these constraints:

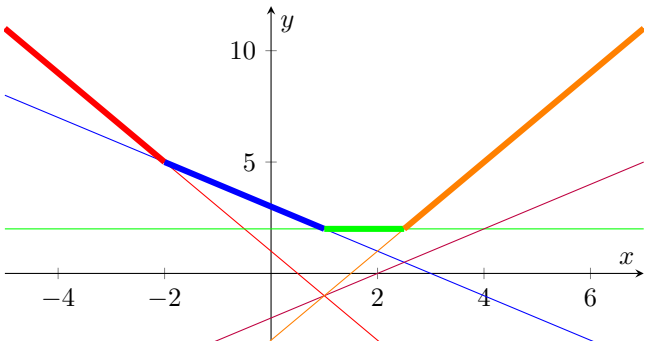
1. The first emoji points towards a word A with the highest relevance.
2. The second emoji points towards a different word B.
3. However, the entire riddle combined points towards another different word C.

Solution

We can treat each word as a line $mx + c$ on the plane, with $m = x_j$ as the slope and $c = y_j$ as the y-intercept. The t_i value of each emoji then becomes an x-coordinate, and the most relevant word would simply be the line that gives the highest y value given the x-coordinate.



If we plot all these lines out, we will notice that only lines that fall within the **convex hull** are relevant. To find the convex hull, we could sort the lines in ascending order of slope, and discard the lines that is always superseded by another line in a linear scan. After the scan, we can also compute a set of ranges, representing the range of x-coordinates each segment of the convex hull covers. Two emojis point to different words if and only if they belong to different segments in the range.



Range	Convex hull segment
$(-\infty, -2]$	
$[-2, 1]$	
$[1, 2.5]$	
$[2.5, \infty)$	

However, this doesn't give us a way to model the most relevant word for a riddle, which is given by $(t_{i_1} \cdot x_j + 1 \cdot y_j) + (t_{i_2} \cdot x_j + 1 \cdot y_j)$. The key observation is that maximizing the *sum* of the two relevance values is equivalent to maximizing the *average*, i.e. $\frac{t_{i_1} + t_{i_2}}{2} \cdot x_j + y_j$. This means we are substituting the x-coordinate of $\frac{t_{i_1} + t_{i_2}}{2}$, the midpoint!

Therefore, we reduce the problem to the following: Given a list of ranges that covers the entire range $(-\infty, \infty)$ and an array t, find the number of pairs of indices (i_1, i_2) such that t_{i_1}, $\frac{t_{i_1} + t_{i_2}}{2}$ and t_{i_2} belong to 3 different ranges. And it is guaranteed that none of these values fall on the boundary.

This becomes a fairly classic problem which can be solved in $O(N \log N)$ time: We subtract the number of invalid pairs from $\binom{N}{2}$ to obtain the final answer. The invalid pairs can be categorized into the following:

- Case #1: Pairs where t_{i_1} and t_{i_2} belong to the same range.
- Case #2: Pairs where the midpoint $\frac{t_{i_1} + t_{i_2}}{2}$ falls within the same range as t_{i_1}.
  Let $[l, r]$ be the range that t_{i_1} belongs to. This happens if $r < t_{i_2} < t_{i_1} + 2(r - t_{i_1})$.
- Case #3: Pairs where the midpoint falls within the same range as t_{i_2}. Symmetric to Case #2.

Each of the cases can be easily handled using binary search or two pointers.

Comments

The author expected this task to be solved by around 3 teams. Unfortunately, the overall performance is not as great as expected.

This task has three main components – identifying the geometric interpretation and the tool (convex hull), then making the midpoint observation, and finally doing the counting in $O(N \log N)$ time. Each of this components require a skillset where a different team member could offer. Therefore, it's a good idea to talk to your teammates about partial ideas, and you can possibly piece together a complete solution with your teammates!

F – Forest Guardian Traversal

Problem Summary

Given a forest of size N, Br Br Patapim performs the following operation until all nodes are visited:

1. Randomly pick an unvisited node.
2. Run a randomized Depth-First Search (DFS) on the tree containing this node.

Find the expected 1-indexed position of each node in the final order of visit.

Solution

By linearity of expectation, the expected position of a node can be found by summing the contributions of all other nodes:

$$\text{Expected position of } u = 1 + \sum_{v \neq u} P(\text{Br Br Patapim visits } v \text{ before } u)$$

Since all nodes within the same tree are visited continuously before moving to another tree, we can split the calculation into two parts: **Inter-tree Contributions** and **Intra-tree Contributions**.

Inter-tree Contributions

Consider the contribution between two disjoint trees, A and B, with sizes sz_A and sz_B respectively.

Only the very first node visited across both trees decides their relative order. The probability that the first visited node belongs to tree B is $\frac{sz_B}{sz_A+sz_B}$. In this case, all sz_B nodes are visited before the tree A. Thus, the total contribution of all nodes in tree B to any node in tree A is:

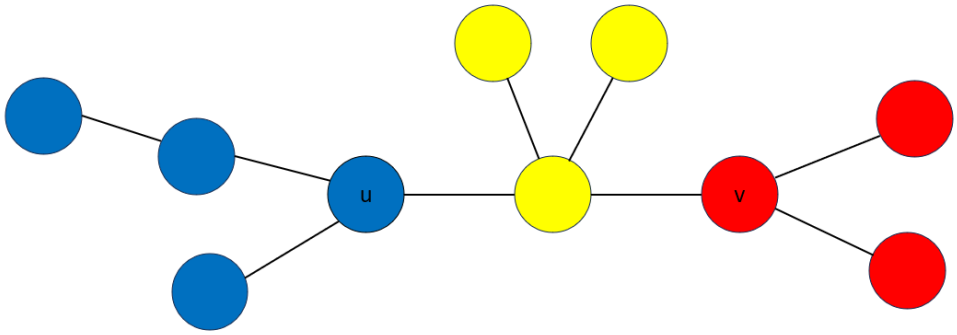
$$\text{Contribution from tree } B \text{ to tree } A = \frac{sz_B}{sz_A + sz_B} \times sz_B$$

Summing over all trees:

$$\text{Total external contribution to tree } A = \sum_{B \neq A} \frac{sz_B^2}{sz_A + sz_B}$$

A naive evaluation takes $O(N^2)$ time. However, since the contribution depends only on the tree size, we can group trees of identical sizes. This optimizes the time complexity to $O(N)$ (the proof is left as an exercise).

Intra-tree Contributions



Consider the internal contribution of a node v to a node u within the same tree of size T. Divide the tree into components by the path between u and v:

- **Red nodes:** Nodes attached to the v-side. If the DFS starts here, v must be visited before u.
- **Blue nodes:** Nodes attached to the u-side. If the DFS starts here, u must be visited before v.
- **Yellow nodes:** Nodes outside the components of u and v (attached to the path between them). If the DFS starts here, Br Br Patapim will hit the path and choose either direction with equal probability, meaning that v is visited before u with a probability of $\frac{1}{2}$.

Thus, the contribution of v to u is $\frac{\text{number of red nodes}}{T} + \frac{\text{number of yellow nodes}}{2T}$.

We can use a tree DP to compute the contribution to u from all possible v. Root the tree at u, let $sz[x]$ be the size of the subtree rooted at node x, and $dp[x]$ be the total contribution to u considering the nodes of descendants of node x only, scaled up by a factor of $2T$ to clear denominators. The DP equation for a node x is:

$$dp[x] = 2 \times sz[x] + \sum_{i \in \text{children}(x)} (dp[i] + sz[i] \times (sz[x] - sz[i]))$$

- The base term $2 \times sz[x]$ accounts for the contribution of node x itself. For every node in the subtree rooted at node x, they will all be red nodes, contributing 2 units each.
- The term $dp[i]$ pulls up the already computed sub-tree relationships within the child's branch.
- The cross-term $sz[i] \times (sz[x] - sz[i])$ accounts for the structural interaction between nodes within the subtree of i and all other nodes currently discovered in the sibling subtrees of x (plus x itself). For all nodes in the subtree rooted at node i ($sz[i]$ nodes in total), all other nodes in the subtree rooted at node x that are not in the subtree rooted at node i ($sz[u] - sz[i]$ nodes in total) will be yellow nodes, contributing 1 unit each.

Special handle the DP for the root, as we do not need to calculate the contribution of node u to node u, and all nodes in other subtrees should be blue and not yellow.

To find the answer for all other nodes, we can apply a standard rerooting DP technique, achieving a time complexity of $O(N)$.

There exist other tree DP methods without using linearity of expectation for calculating intra-tree contributions, but the equation for the tree DP might be more complex without linearity of expectation.

Comments

The author expected this task to be solved by around 1 team.

This task requires calculating expected values, efficient grouping for calculation, and standard tree DP techniques. Similar to problem E, each of this components require a skillset where a different team member could offer. Therefore, it's a good idea to talk to your teammates about partial ideas, and you can possibly piece together a complete solution with your teammates!

G – Go Shoot!

Problem Summary

We have three categories of parts: Blades, Ratchets and Bits. A valid deck contains exactly three Beyblades, so we must select exactly three distinct parts from each category. The total cost of the nine selected parts cannot exceed the budget X.

For one Beyblade, let A, D and S be the sums of Attack, Defense and Stamina of its Blade, Ratchet and Bit. Its power is

$$\max(A, D) + S.$$

We need to maximize the sum of the powers of the three Beyblades.

Solution

1. Treat each Beyblade as Attack-oriented or Defense-oriented

For one Beyblade,

$$\max(A, D) + S = \max(A + S, D + S).$$

Therefore, we may assign it one of two roles:

- an *Attack role*, whose value is $A + S$; or
- a *Defense role*, whose value is $D + S$.

Once the role is fixed, every part contributes independently. A part with attributes (c, a, d, s) contributes

$$a + s \quad \text{in an Attack-role Beyblade,}$$

and

$$d + s \quad \text{in a Defense-role Beyblade.}$$

Suppose exactly k of the three Beyblades have the Attack role. Each category must then provide exactly k Attack-role parts and $3 - k$ Defense-role parts. For example, if $k = 2$, we need two Attack-role Blades, two Attack-role Ratchets and two Attack-role Bits; the remaining part of every category has the Defense role.

After these roles are fixed, the exact pairing of parts with the same role does not affect their total contribution. Hence we can optimize the three categories independently and only require them to use the same value of k.

2. Dynamic programming for one category

Process each of the three categories separately. For one category, define

$$\mathrm{dp}[i][j][x]$$

as the maximum contribution obtained by selecting exactly i Attack-role parts and exactly j Defense-role parts, with total cost exactly x. Only states with $i + j \leq 3$ are required.

Initially,

$$\mathrm{dp}[0][0][0] = 0,$$

and every other state is impossible, represented by $-\infty$.

For every part (c, a, d, s), there are three choices:

1. skip it;
2. select it for an Attack role, increasing the value by $a + s$; or
3. select it for a Defense role, increasing the value by $d + s$.

The two selecting transitions are

$$\mathrm{dp}[i+1][j][x+c] \leftarrow \max\big(\mathrm{dp}[i+1][j][x+c], \mathrm{dp}[i][j][x] + a + s\big),$$
$$\mathrm{dp}[i][j+1][x+c] \leftarrow \max\big(\mathrm{dp}[i][j+1][x+c], \mathrm{dp}[i][j][x] + d + s\big).$$

Iterate $i + j$ and x in descending order. Thus a part cannot be used again in the same iteration, which enforces that every listed part is selected at most once.

After all parts in the category have been processed, define

$$\mathrm{best}[k][x] = \mathrm{dp}[k][3-k][x] \qquad (0 \le k \le 3).$$

We compute one such table for Blades, one for Ratchets and one for Bits.

3. Combine the three categories

Fix k. Let

$$B[p] = \text{best Blade contribution at exact cost } p,$$
$$R[q] = \text{best Ratchet contribution at exact cost } q,$$
$$I[r] = \text{best Bit contribution at exact cost } r,$$

where all three arrays use the role counts $(k, 3-k)$.

We need

$$\max_{p+q+r \le X} \big(B[p] + R[q] + I[r]\big).$$

A direct three-level loop would take $\mathcal{O}(X^3)$. Instead, first combine Blades and Ratchets by a max-plus convolution:

$$BR[x] = \max_{p+q=x} \big(B[p] + R[q]\big).$$

This takes $\mathcal{O}(X^2)$ time.

The budget is an upper bound, so the Bit part may use any cost not exceeding the remaining budget. Compute prefix maxima

$$PI[x] = \max_{0 \le r \le x} I[r].$$

The best answer for this fixed k is then

$$\max_{0 \le x \le X} \big(BR[x] + PI[X - x]\big).$$

Repeat this for $k = 0, 1, 2, 3$ and take the maximum. If every resulting state is impossible, output -1.

Why the Algorithm Is Correct

Consider any valid assembled deck. Label each Beyblade Attack or Defense according to whichever of $A + S$ and $D + S$ gives its power. If there are k Attack labels, this deck corresponds to selections of exactly k Attack-role parts and $3 - k$ Defense-role parts from every category. The category DPs

and the combination step consider those selections and their costs. Therefore, the algorithm can obtain a value at least as large as the power of every valid deck.

Conversely, consider any three category selections counted by the algorithm for some fixed k. Pair their k Attack-role parts to form k Beyblades, and pair the remaining parts to form the other $3-k$ Beyblades. For each resulting Beyblade, its actual power $\max(A+S, D+S)$ is at least its assigned role value. The resulting deck is valid and stays within the same budget. Therefore, the value produced by the algorithm cannot exceed the optimum deck power.

The algorithm's answer is thus exactly the maximum possible deck power.

Complexity Analysis

For one test case, the three category DPs take

$$\mathcal{O}((N+M+K)X)$$

time, since the two role-count dimensions contain only a constant number of states. Combining the categories for all four values of k takes $\mathcal{O}(X^2)$ time. Hence the total time complexity is

$$\boxed{\mathcal{O}\big((N+M+K+X)X\big)}.$$

The memory complexity is

$$\boxed{\mathcal{O}(X)},$$

up to constant factors for the role counts and the three categories.

Use a 64-bit integer type for DP values, and use a sufficiently small negative value to represent impossible states.

Other Approaches

There are also correct solutions that avoid the explicit $\mathcal{O}(X^2)$ convolution.

For a fixed k, one may process the categories sequentially. Before processing a category, initialize its state dp[0][0][x] using the best values obtained after the previous categories. After selecting exactly k Attack-role parts and $3-k$ Defense-role parts from the current category, carry that exact-cost array to the next category. Trying all four values of k gives an $\mathcal{O}((N+M+K)X)$ solution.

Another correct approach sorts each category by $a-d$. Among any three selected parts, an exchange argument shows that the k parts with the largest $a-d$ should receive the Attack role. An ordered choose-three knapsack can then process the categories in $\mathcal{O}((N+M+K)X)$ time after sorting. The sorting key must be exactly $a-d$; Stamina contributes equally to both roles and must not be included in the key.

Several simpler-looking approaches are incorrect:

- Replacing every part's contribution by $\max(a,d)+s$ lets different parts of the same Beyblade choose different roles, which does not match $\max(A,D)+S$.
- Computing $\max(\text{total Attack}, \text{total Defense}) + \text{total Stamina}$ chooses one role for the whole deck, whereas the three Beyblades choose their roles independently.
- Selecting parts greedily by power, power-to-cost ratio, or lowest cost does not account for the global budget and the requirement to select exactly three distinct parts from every category.
- Using ascending 0/1-DP loops silently turns the problem into an unbounded knapsack and may select the same listed part multiple times.

Common Pitfalls

- The role count k must be the same for Blades, Ratchets and Bits. Otherwise, the selected parts cannot be paired into three role-labelled Beyblades.
- The DP loops must run in descending order. Ascending loops may select the same listed part multiple times.
- DP arrays store values for an *exact* cost. Prefix maxima are needed when handling the condition that the total cost is at most X.
- Choosing Attack or Defense independently for every part is incorrect; all three parts in one Beyblade must use the same role when evaluating $A + S$ or $D + S$.

H - Hollow Knight: Silksong

Problem Summary
Given n games, each with a popularity pi. Game j cannot be released within x days before and after the release of game i if pj < pi. Find the minimum number of days needed to release all n games satisfying the popularity rule.

Solution
Observe that both game i and game j blocks the release date of each other if their popularity is not the same. Therefore, the optimal scheduling will always be releasing the games with the same popularity together (one day each, without any gaps in between), wait for x days, then release the batch of games with another popularity.
Let k be the number of unique popularity in the input. The release schedule will look like:

Batch 0	x days	Batch 1	…	Batch k-1	x days	Batch k

Overall, the number of days used to release games is n. The number of days not releasing any games (i.e. waiting) is (k - 1) * x.

The solution is n + (k - 1) * x.

Comment
The author expects around 30 - 50 teams to solve this problem.
On site performance is better than expected.

I - Ice Cream Sort

Problem Summary

You are given an array of N distinct numbers representing ticket numbers of customers in a queue. You can select a fixed integer distance D ($1 \leq D \leq N$) and swap any two elements whose indices differ by exactly D.

The goal is to find the maximum possible integer D such that the array can be fully sorted in strictly increasing order using only swaps of distance D.

Solution

Swapping elements at indices i and j with distance D means two elements can be swapped if and only if:

$$i \equiv j \ (mod\ D)$$

Thus, choosing a distance D splits the array into D independent sub-chains based on their indices modulo D. Elements can only move within their own sub-chain and can never move between different sub-chains.

To sort the entire array, every element currently at index i (1-indexed) must end up at its target sorted index p_i. For this movement to be possible:

- Every element must belong to the same sub-chain as its sorted destination, which requires $i \equiv p_i \ (mod\ D)$.
- Equivalent to: D must divide $\left|i - p_i\right|$ for all $1 \leq i \leq N$.

Hence, for the largest D, $D = gcd\left(\left\{\left|i - p_i\right|\right\}\right)$.

Time complexity: $O(NlogN)$

Comments

- About 20–30 teams are expected to solve this task during contest time.

J - Jettisoned Joins

Problem Summary

You are given two database tables and a set of key columns. Delete at most D rows in total from the two tables so that the number of rows in the resulting inner join, taken over the key columns, is minimized. Report that minimum number of rows.

Solution

Notice that for each unique tuple of values on the key columns, we only care about its number of occurrences in each table.

For each tuple T, let a_T and b_T denote the number of occurrences of T in the first and second table respectively. Every one of the a_T rows pairs with every one of the b_T rows, so T contributes exactly $a_T b_T$ rows to the inner join, and the total size of the join is $\sum_T a_T b_T$.

A tuple occurring in only one table has $a_T b_T = 0$; it contributes nothing to the join and deleting its rows is never useful, so from now on we consider only tuples that occur in both tables.

The following lemma tells us exactly what a single deletion is worth.

Lemma 1. *Suppose without loss of generality, $a_T \le b_T$, and consider deleting d rows of T from the first table (the table with fewer occurrences). If $d \le a_T$, the contribution of T becomes $(a_T - d)b_T$, so each deletion removes exactly $b_T = \max(a_T, b_T)$ rows from the join. Once $d = a_T = \min(a_T, b_T)$ the contribution is 0, and any further deletions on T remove nothing.*

Proof. The first statement is obvious. For the second statement, we can consider the case where $d = a_T + x$ where $x > 0$. Since we can only delete up to a_T rows in the first table, the remaining x deletions must be done on the second table. This gives us a total contribution of $(a_T - a_T)(b_T - x) = 0$, which is equal to when $d = a_T$ as required. □

So each tuple behaves like a bundle of identical items: it offers up to $\min(a_T, b_T)$ deletions, each worth exactly $\max(a_T, b_T)$ removed rows. This gives us the following algorithm:

1. Group the rows of each table by their key tuple, and compute a_T and b_T for every tuple T occurring in both tables. This can be done by either sorting or hashing the tuples.
2. Compute the initial join size $\sum_T a_T b_T$.
3. Sort the tuples in descending order of $\max(a_T, b_T)$.
4. Iterate through the sorted list, maintaining a remaining budget initialised to D. For each tuple, spend $t = \min(\text{budget},\ \min(a_T, b_T))$ deletions on it, subtract $t \cdot \max(a_T, b_T)$ from the answer, and reduce the budget by t. Stop when the budget reaches 0.

The algorithm runs on $O(NK \log N)$ time for the sorting version, or $O(NK + N \log N)$ expected for the hashing version.

Proof of Optimality

Claim 1. *There exists an optimal solution in which, for every tuple T, all deletions of rows of T are made from the table with fewer occurrences of T.*

Proof. Without loss of generality assume $a_T \le b_T$. Consider any deletion plan that performs $x \ge 0$ deletions of T on the first table and $y > 0$ deletions of T on the second table, so that T contributes $(a_T - x)(b_T - y)$ rows. We show that moving all of these deletions to the first table is no worse.

Case 1: $x + y \le a_T$**.** Moving all deletions to the first table gives a contribution of

$$\begin{aligned}(a_T - x - y)b_T &= a_T b_T - x b_T - y b_T \\ &\le a_T b_T - x b_T - y a_T \\ &\le a_T b_T - x b_T - y a_T + xy \\ &= (a_T - x)(b_T - y),\end{aligned}$$

where the first inequality uses $a_T \le b_T$ and $y > 0$, and the second uses $xy \ge 0$. Both plans use the same number of deletions, so the new plan is no worse.

Case 2: $x + y > a_T$**.** Consider instead deleting a_T rows from the first table and none from the second. This makes the contribution of T equal to 0, which is at most the original contribution, and it uses $a_T < x + y$ deletions, freeing up $x + y - a_T$ deletions to be spent elsewhere. Since no deletion can increase the size of the join, the new plan is no worse.

In both cases the modified plan is no worse than the original, and it performs all deletions of T on the first table. Applying this modification to each tuple yields an optimal solution of the required form. □

Claim 2. *There exists an optimal solution in which, whenever some rows of a tuple T are deleted, every tuple T' with* $\max(a_{T'}, b_{T'}) > \max(a_T, b_T)$ *contributes* 0 *rows to the inner join.*

Proof. By Claim 1 we may assume all deletions on any tuple are made from the table with fewer occurrences of that tuple.

Suppose some deletion plan deletes a row of T while some tuple T' with $\max(a_{T'}, b_{T'}) > \max(a_T, b_T)$ still contributes a non-zero number of rows. Since T' contributes more than 0 rows, fewer than $\min(a_{T'}, b_{T'})$ deletions have been spent on it, so by Lemma 1 at least one further deletion on T' is available and would remove exactly $\max(a_{T'}, b_{T'})$ rows.

Move one deletion from T to T'. By Lemma 1, undoing a deletion on T adds back exactly $\max(a_T, b_T)$ rows, and the new deletion on T' removes exactly $\max(a_{T'}, b_{T'})$ rows. The total number of deletions is unchanged, and the join shrinks by

$$\max(a_{T'}, b_{T'}) - \max(a_T, b_T) > 0,$$

so the new plan is strictly better. Each such exchange strictly decreases the size of the join, and there are finitely many deletion plans, so the process terminates at a plan satisfying the claim. □

Together these results justify the algorithm: by Claim 1 every tuple's deletions go to one side, by Lemma 1 each such deletion is worth exactly $\max(a_T, b_T)$ for up to $\min(a_T, b_T)$ deletions, and by Claim 2 an optimal solution exhausts tuples in descending order of $\max(a_T, b_T)$.

Comments

The author expected around 20 to 30 teams to solve this problem. The onsite performance was worse than expected, with only 14 teams being able to solve the problem.

A lot of teams that attempted the problem had RE/TLE submissions. Teams should be aware that unnecessarily slow methods of grouping the tuples might result in slow runtime (and hence a TLE verdict) with a large amount of unique tuples.

K - KMP Evasion

Problem summary

Given strings s (length n) and t (length m), shuffle s to minimize the number of occurrences of t in s as a substring. Output the shuffled string.

Solution

Case 1: t contains at least two distinct characters.

Find an index i such that $t_i \neq t_{i+1}$ (such i must exist as there are at least two distinct characters). Denote t_i and t_{i+1} as a and b respectively. Simply output (all occurrences of b) + (all occurrences of a) + all the other characters. (+ is concatenate)

This guarantees 0 occurrences of t as all a's are on the right of all b's.

Case 2: t is one character repeated m times. We denote that repeated character as c, and all other characters in s as non-c.

We start with an empty string ans, then repeatedly add $m - 1$ copies of c and one copy non-c to ans. If we run out of c's, just add the remaining non-c's arbitrarily. If we run out of non-c's, just add the remaining c's.

(This is just one of many methods to minimise the number of occurrences).

L - Last Order

Problem Summary

Mr. Noodle cooks N food types with distinct cooking times C_i on a single stove. At most one order can be placed per second for the first T seconds. Orders are processed in FIFO order, but duplicate food types waiting in the queue cook simultaneously as a single batch in time C_i.

Determine the maximum possible completion time to finish all orders under the worst-case order placement sequence, and output the sequence of orders $O_1, O_2, \ldots, O_T$ (where $O_t \in [1,N]$, or $O_t = 0$ if no order is placed at second t) that achieves this maximum time.

Solution

For the following solution, assume without loss of generality that $C_1 < C_2 < \cdots < C_N$.

The maximum achievable completion time is:

$$T + C_N - 2 + \sum_{i=1}^{N} C_i$$

Consider the state after second T.

All unfinished orders of the same type will eventually be cooked together. Therefore, regardless of how many unfinished orders exist, the total remaining cooking time represented by the queue is at most $\sum_{i=1}^{N} C_i$

Suppose food type x is currently being cooked.

- If the current dish being cooked started before second T, it has been cooked for at least two seconds. Its remaining cooking time is therefore at most $C_x - 2$ seconds. Hence, the completion time is at most
$$\sum_{i=1}^{N} C_i + C_x - 2$$
- If it started at second T, it has $C_x - 1$ seconds left, but its type cannot be waiting in the queue. The queue then contributes at most $\sum_{i=1}^{N} C_i - C_x$. Hence, the completion time is at most
$$C_x - 1 + \sum_{i=1}^{N} C_i - C_x = \sum_{i=1}^{N} C_i - 1$$

Therefore, the completion time cannot exceed

$$T + C_N - 2 + \sum_{i=1}^{N} C_i$$

To achieve the maximum completion time, we can construct the order sequence as follows:

1. At second $T - 1 - C_{N-1}$, order type $N - 1$.
2. At second $T - N$, order type N.
3. During seconds $T - N + 1, T - N + 2, \ldots, T - 1$, order every type of food other than N.
4. At second T, order type N.
5. For all remaining seconds, do not order anything.

Special case: For $N = 1$, handle it separately by ordering the only food type at seconds $T - 1$ and T.

Comments

The author expected this task to be solved by around 10 teams. The onsite performance is a lot worse than expected, which none of the teams has solved it during contest time.

This task may seem difficult at first, but the solution is actually quite simple. The contestants may think that dp is needed to construct the sequence that pushes all types of food into the queue, but if they actually read the constraints and think about them carefully, they will find that it is always possible to achieve the maximum possible completion time with a fixed construction.