ICPC Latin American Regional Contests – 2024

Solution Sketches

Problem A – Append and Panic!

Author: Humberto Díaz Suárez, Puerto Rico

Let's call Gabriel's original (uncorrupted) string t and let y be the sorted, duplicate-free version of t. Our input is $s = t + y$, the concatenation of these two strings.

We can determine y by sorting s and removing duplicates – exactly the operation Gabriel performed. Once we have y, the answer to the problem is simply $|t| - |y|$. The complexity of this approach is $O(|s| \log |s|)$.

Alternatively, we can solve the problem without explicitly recovering the string y. Since only the length of y matters, counting the number of distinct characters in the input is sufficient. Below is a sample amortized $O(|s|)$ implementation of this approach using Python's set data structure:

```python
s = input()
print(len(s) - len(set(s)))
```

Another approach leverages the fact that the end of the string (representing y) is sorted and contains no duplicates. Starting with an index at the last character of the string, we can move backwards as long as each previous character is lexicographically smaller than the current one. This allows us to identify the boundary between t and y and determine $|t|$ in $O(|y|)$ time.

Problem B – Biketopia's Cyclic Track

Author: Alexandr Grebennikov, Brasil

In this problem, we are given a connected, undirected, and unweighted graph G with n vertices and m edges, where each vertex has a degree of at least 3. The goal is to find a cycle in G such that, after deleting all edges of the cycle, the graph remains connected – or to determine that no such cycle exists.

It turns out that it is always possible to find such a cycle.

Pick an arbitrary vertex v_0 and perform a DFS from it. Let T be the resulting DFS tree. Call the edges of T *tree edges*, and the remaining edges of G *backward edges*. For a vertex v, denote by $d(v)$ the distance from v to the root v_0 in the tree T. Now, for any backward edge $e = uv$ of G define its *cost* as $\min(d(u), d(v))$.

Let $e' = xy$ be the backward edge with maximum cost. Assume that $d(x) < d(y)$, so x is an ancestor of y in the tree T (a property of DFS trees). Consider the cycle C consisting of e' and the path from x to y in T. We claim that this cycle meets the requirements. It is straightforward to construct this cycle in $O(n + m)$.

Proof of the claim: We need to verify that any vertex v can still connect to v_0 without using the edges of cycle C. It suffices to check this for vertices v that lie on the cycle C. Since each vertex v has a degree of at least 3, there exists an edge $e = vu$ that is not part of C.

(1) If $v = x$ then we can reach v_0 by moving up the tree T.

(2) If $v \neq x$ and e is a backward edge, then the cost of e is at most the cost of e' (by choice of e'):

$$\min(d(v), d(u)) \leq d(x).$$

Since $d(v) > d(x)$, it follows that $d(u) \leq d(x)$. Thus, we can traverse the edge vu, and then move up the tree from u to v_0.

(3) If $v \neq x$ and e is a tree edge, then this edge goes downward in the tree, so $d(u) > d(v)$. We can continue moving down the tree T until reaching a leaf w_0. Since w_0 has a degree of at least 3, it is adjacent to a backward edge w_0w_1. By a similar argument as in case (2), $d(w_1) \leq d(x)$, allowing us to move up the tree from w_1 to v_0.

Problem C – Cindy's Christmas Challenge

Author: Roberto Sales, Brasil

The problem asks us to quickly compute the edit distance between Cindy's (R, B)-sequence and each of W contiguous subsequences of Grinch's sequence.

Edit distance for single-character strings

If we take a string of repeated characters a^x, then the edit distance between an arbitrary string S and a^x is given by:

$$d(a^x, S) = \max(|S| - x, 0) + \max(x - cnt_a(S), 0) \tag{1}$$

where $cnt_a(S)$ is the number of occurrences of character a in S.

The intuition behind this formula is as follows: in the first max-term, we account for characters in S that need to be removed so that $|S| \leq x$. Next, we either replace characters in S with a, or add new a characters to it, which the second max-term accounts for.

Generalizing to two-character sequences

Let C be Cindy's (R, B)-sequence, i.e. $C = r^R b^B$, and let S represent Grinch's altered sequence, composed of characters "`r`", "`b`" and "`g`".

To calculate the edit distance between C and S, we can use a brute-force approach by splitting S into two parts. The first part will be transformed into the R "`r`" characters of C, and the second part will be transformed into the B "`b`" characters.

$$d(C, S) = \min_{0 \leq m \leq |S|} \{d(r^R, S_{[1..m]}) + d(b^B, S_{[m+1..|S|]})\} \tag{2}$$

By expanding 1 into 2, we get a big formula:

$$\begin{aligned} d(C, S) = \min_{0 \leq m \leq |S|} \{ & \max(m - R, 0) + \\ & \max(m - cnt_r(S_{[1..m]}), 0) + \\ & \max(|S| - m - B, 0) + \\ & \max(B - cnt_b(S_{[m+1..|S|]}), 0)\} \end{aligned}$$

While this can be easily computed in $O(|S|)$ in the worst case, we want to do this for $O(W)$ queries and thus we want a faster solution.

Optimizing the solution

A transformation we can make to the formula is to introduce our query parameters to the function. Let's define $f(L,U) = d(C, S_{[L..U]})$. After rearranging a few of the terms into functions t_1, t_2, t_3 and t_4, we finally get:

$$\begin{aligned} f(L,U) &= \min_{L-1 \le m \le U} \sum_{1 \le i \le 4} \max(t_i(m,L,U),0) \qquad (3)\\ t_1(m,L,U) &= m - L + 1 - R \\ t_2(m,L,U) &= R - cnt_r(S_{[L..m]}) \\ t_3(m,L,U) &= U - m - B \\ t_4(m,L,U) &= B - cnt_b(S_{[m+1..U]}) \end{aligned}$$

Now, we can see we have 4 max-terms in our formula. Each one has two components: the first only depends on m and the query terms L, U – our t_i –, and the second is zero.

Let's say a max-term *triggers* when it's strictly greater than zero. When one such max-term i triggers, it's clear that $\max(t_i(m,L,U),0) = t_i(m,L,U)$ and otherwise it is zero. Notice that if we fix a query (and thus fix the values of L, U), the t_i functions above have an interesting property: they're either monotonically increasing or decreasing w.r.t m. This means that, as m increases from $L-1$ to U, a max-term will either start triggering and never stop triggering again, or vice-versa.

With that observation, when answering each query, we can binary search, for each of the 4 *max* terms, at which point they will flip between triggering and not triggering (or vice-versa). Let's define these points as $p_i(L,U)$, where $1 \le i \le 4$. Notice that p depends on the query parameters, since the flip point might change when the query parameters change.

When solving a query L, U, if we sort $L-1, p_1(L,U), p_2(L,U), p_3(L,U), p_4(L,U), U+1$, we can notice that, for every two adjacent points in this list, the interval $[x..y)$ defined by them can be solved more easily now, since we know exactly which of the 4 maxes triggers or not. Then, in this interval between two adjacent points, the formula becomes a summation of 4 functions. Something like:

$$g(x,y,L,U) = \min_{x \le m < y} \sum_{1 \le i \le 4} \max(t_i(m,L,U),0) \qquad (4)$$

Let T be a subset of the integer interval $[1..4]$ containing only the indices of the max-terms that triggered for the interval $[x..y)$. Let's keep only these terms, and split t_i between a function that only depends on m, and a function that depends on the query parameters L, U, i.e., $t_i(m,L,U) = t'_i(m) + t''_i(L,U)$:

$$\begin{aligned} g(x,y,L,U) &= \min_{x \le m < y} \sum_{i \in T} t'_i(m) + t''_i(L,U) \qquad (5)\\ &= \sum_{i \in T} t''_i(L,U) + \min_{x \le m < y} \sum_{i \in T} t'_i(m) \qquad (6) \end{aligned}$$

Notice the sum of $t''_i(L,U)$ can be moved outside the minimum expression, and thus computed in constant time for every query. If we know how to answer $t'_i(m)$ fast for every query, we're done.

To answer that fast, we can keep $2^4 = 16$ RMQ structures (such as a Segment Tree or a Sparse Table) that stores the summation of $\sum_{i \in T} t'_i(m)$ for every possible subset T. When answering

$g(x, y, L, U)$, we can simply figure out which functions trigger, query the corresponding RMQ structure, add the query-dependent terms t''_i, and we're done. We should do that for every x, y pair based on the split points p_i we found.

By using segment trees, we should take $O(16 \times |S| \log |S|)$ to build them, and $O(W \log |S|)$ to answer the queries, which should be enough for the given constraints.

Notice cnt_r and cnt_b can be computed quickly with prefix sums.

One can also implement everything in linear time by using an RMQ structure that is $O(n)$ to build and $O(1)$ to query, but this was not required.

Alternative Formula

One might also arrive at the formula below for the edit-distance between S and a^x, through a different intuition.

$d(a^x, S) = \max(|S|, x) - \min(cnt_a(S), x)$.

One can follow the same steps described by the solution above to arrive at a solution, but such a solution gets slightly harder to implement since the resulting formulas are not so pretty.

One can prove through min/max manipulation that both formulas are equivalent.

Problem D – Diverse T-Shirts

Author: Agustín Santiago Gutiérrez, Argentina

Once abstracted, the problem reduces to finding the maximum independent set in a given graph, which is an NP-hard problem. This suggests that the given graph must have a specific structure. Indeed, based on the problem context, we can deduce that it is the **line graph of a bipartite simple graph** (`https://en.wikipedia.org/wiki/Line_graph`).

In this bipartite graph, each text color is a node, and each background color is a node. Each T-shirt model is an edge joining exactly one text-color node with one background-color node. Critically, the maximum matching in this bipartite graph gives the answer we're looking for.

Therefore, if we can reconstruct this bipartite graph from the given line graph, we can solve the problem using a basic $O(V \cdot E)$ bipartite matching algorithm. Note that $V = O(N)$ and $E = O(N)$, for the N given in input, since the "input nodes" are edges in the reconstructed bipartite graph.

To reconstruct the bipartite graph, we can use a greedy approach to identify maximal cliques incrementally from the input. These cliques represent nodes in the bipartite graph. A **maximal clique** is a complete subgraph that is not a subset of any other complete subgraph, although it is not necessarily the largest.

This approach works because the **line graph of a bipartite graph is diamond-free** (`https://en.wikipedia.org/wiki/Diamond_graph`). It is actually an even more specific graph than diamond free, that is: not all diamond-free graphs are line graphs of a simple bipartite graph.

In a diamond-free graph, after discarding isolated nodes, the maximal cliques partition the edges; each edge belongs to exactly one maximal clique. Here's why:

If an edge e connects nodes u and v and belongs to a complete subgraph induced by the set S of nodes, as well as a complete subgraph induced by another set T, then $S \cup T$ must itself be a complete subgraph. If it were not, we could find nodes $s \in S$ and $t \in T$ that are not adjacent, forming a diamond with nodes u and v. This leads to a contradiction, so an edge can belong to at most one maximal clique.

Using this property, we can enumerate all maximal cliques in $O(N^2)$ time. We start by selecting any unused edge (x, y) as the current clique, then incrementally add nodes to this clique until no node is adjacent to all nodes in the current clique. Specifically:

1. Choose an unused edge (x, y).
2. Add nodes z such that both (x, z) and (y, z) exist in the input graph.

This process reliably finds maximal cliques in the input, as each chosen node must be adjacent to every other in the set to avoid a diamond.

In our specific problem, each node x in the bipartite graph corresponds to a clique in the line graph, containing all input nodes that represent models as edges incident to x in the original bipartite graph. Any node x with degree at least 2 will form a maximal clique, which we identify through our enumeration. Remaining single-node cliques can be inferred to fully reconstruct the bipartite graph, except for isolated nodes, which correspond to unused colors and are irrelevant.

As an extra exercise / note, observe that in this case, there are $O(N)$ maximal cliques, since each input node can belong to at most two maximal cliques. Therefore, even an easy, naive implementation of the greedy algorithm for clique enumeration that takes $O(N)$ time per added node (that is, the obvious "triple for" code), will achieve $O(N^2)$ complexity. This is something which would have required being quite more careful for a completely general diamond-free graph. For example, a general complete bipartite graph is an invalid input for this problem (almost none of those are the line graph of a bipartite graph), yet it is diamond free (because it is triangle free, as it has no odd cycles). Thus each edge is a maximal clique in that case and our code for maximal clique enumeration would find that, but take cubic time to do so.

Problem E – Evereth Expedition

Author: Paulo Cezar Pereira Costa, Brasil

In this problem, we are given a permutation with missing elements, and our task is to complete the permutation so that it first strictly increases, then strictly decreases (i.e., forms a bitonic or mountain permutation), or determine if that's not possible.

The key idea is to greedily place the missing numbers at the sequence's extremes, while ensuring that both the added and existing numbers maintain the bitonic (mountain) property.

Lemma 1: In any mountain permutation of numbers from 1 to N, the number 1 must be at one of the two ends of the sequence.

Suppose 1 is not at an end but somewhere in the middle. Then, since 1 is the smallest number, the sequence cannot strictly decrease before it (as there are no smaller numbers), nor can it strictly increase after it. This contradicts the definition of a mountain permutation. Therefore, 1 must be at one of the ends.

Lemma 2: Mountain permutations have a natural recursive nature: given that the number 1 is at an end, removing it and decrementing all other numbers by 1 yields a mountain permutation of size $N - 1$.

Removing 1 leaves numbers from 2 to N. Decrementing each by 1 maps them to numbers from 1 to $N - 1$. The order of the sequence remains, and the mountain property is preserved.

Solution steps

Based on these lemmas, we can complete the sequence greedily as follows:

- Start by considering the elements between indices $l = 0$ and $r = N - 1$ of the given sequence A.

- By *Lemma 1* number 1 should be in one of the ends (A_l or A_r), hence there's no solution if:
  - number 1 is not missing and is not at one of the ends; or
  - number 1 is missing but both ends are already filled with values different from 1.
- If one of the ends is the number 1 we can adjust that end (i.e. $l = l + 1$ or $r = r - 1$)
- Similarly, if only one of the ends is currently missing, we have no choice but to put number 1 there and adjust that end.
- In both cases according to *Lemma 2* we can now reduce non-zero numbers by one and start again with a problem of size $N' = N - 1$ (until we reach the trivial case with $N' = 1$).
- The remaining question is: what to do when both ends are missing?
  - If the sequence has only zeroes we can choose any end, as regardless of our choice we can continue adding elements until the sequence is complete.
  - Otherwise, let $A_{dx} = x$ be the first non-zero value to the right of A_l, and $A_{dy} = y$ the first non-zero value to the left of A_r.
  - If $x < y$ we choose to set $A_l = 1$, and in case $y < x$ we choose to set $A_r = 1$
    * Let's dive into the case where $x < y$ as the same arguments can be applied by symmetry to $y < x$. The idea here is that there are fewer elements that can be used to make the sequence $A_l \cdots A_{dx}$ increasing, and given that $x < y$ that must be the case. If there is a solution, either $A_{dx} \cdots A_{dy}$ is increasing, or the peak of the mountain appears within it (and the sequence starts to decrease), in any case $A_l \cdots A_{dx}$ must be increasing.
  - Now, if $x = y$, there's a single non-zero element in the sequence and we'll set $A_l = 1$ when $(dx - l) < (r - dy)$ and $A_r = 1$ otherwise (i.e. we place 1 on the end with fewer zeroes to be filled). The idea is that:
    * either that non-zero element is the peak of the sequence ($x = N$), in which case regardless of our choice we'd get a valid mountain at the end; or
    * $x < N$ and we need to prioritize placing the values smaller than x on the shorter end, otherwise we might end up placing N after x and a value greater than x before it, creating a second peak which is not valid.
- We can complete the proof that the above algorithm works with an exchange argument. Lets focus on the case where $x = y$ and $(dx - l) < (r - dy)$ as the same construction below will be quite similar for the other cases.
- Suppose we have a sequence $A_l = 0, \cdots, A_{dx=dy} = X, \cdots, A_r$ where instead of choosing to place number 1 on the shorter end we place it on the longer end, and that results in a valid mountain $P = A_l = w, \cdots, X, \cdots, A_{r-1} = z, A_r = 1$.
- By *Lemma 1* and *Lemma 2* we know that either $w = 2$ or $z = 2$. If $z = 2$ we can swap A_{r-1} and A_l and still obtain a valid mountain $P' = A_l = 2, \cdots, X, \cdots, A_{r-1} = w, A_r = 1$, as all elements greater than w must be between A_{l+1} and A_{dx} (think of what happens both when $w > x$ and $w < x$).
- Now that we have both P and $P' = A_l = 2, \cdots, X, \cdots, A_r = 1$, we can trivially swap numbers 1 and 2 and still get a valid mountain permutation, which could have been obtained by making the choice earlier to place 1 on the shorted end. Hence, whenever there's a solution where number 1 is placed on the longer end we can turn that into a solution where 1 is put on the shorter end.

Note that we don't need to actually to "remove 1 and decrement remaining numbers", instead we can simply increase the value that should be set in one of the current ends. Arrays $next_i =$ "index to the first non-zero element to the right of i" and $prev_i =$ "index to the first non-zero element to the left of i", well as an array $missing_i =$ "1 is i is missing and 0 in case it appears in the sequence" can be pre-computed in $O(N)$ and with that information we can implement the algorithm above also in $O(N)$.

Alternative Approach

There's also an alternative approach based on max-flow:

Let's assume that N is present in the list.

Consider consecutive zeros ("blocks of 0s") in the array. Each block can be classified as increasing or decreasing by checking its surrounding values. By examining the values immediately before and after each block, we can determine whether each block should be increasing or decreasing.

If N is not in the input, there will be only two possible blocks of 0s where N can be placed: either the last increasing block or the first decreasing block (considering the blocks from left to right). We test both options by inserting N in an arbitrary position within each candidate block.

For a given candidate, if the arrangement of blocks does not follow the "mountain" pattern – i.e., an initial sequence of increasing blocks followed by a sequence of decreasing block – we discard it. Otherwise, we observe that each missing number can be assigned to at most two possible blocks (one increasing and one decreasing), and each block has a fixed size. This forms a straightforward maximum flow problem, and reconstructing the solution after running the flow algorithm is straightforward.

Problem F – Finding Privacy

Author: Alejandro Strejilevich de Loma, Argentina

Substrings algorithm

The statement asks for a maximal independent set of size K in the path graph P_N. W.l.o.g. we can assume that the vertices/toilets are chosen/occupied from left to right. By symmetry, just three situations may occur:

- If at least three vertices remain available and the second available vertex is chosen ($S_3 =$ "`-X-`"), then three vertices are consumed.
- If at least two vertices remain available and the first available vertex is chosen ($S_2 =$ "`X-`"), then two vertices are consumed.
- If a single vertex remains available, this vertex is chosen and consumed ($S_1 =$ "`X`").

With this in mind, it is not hard to realize that the maximal independent set exists if and only if $\lceil N/3 \rceil \leq K \leq \lceil N/2 \rceil$.

This also suggests a possible algorithm, which is to check whether K has a good value and in that case build the output string by repeating S_1, S_2 and S_3 as substrings. Notice that S_1 is used at most once at the very end of the process, while the occurrences of S_2 and S_3 can be sorted arbitrarily. We need a few observations to get a working algorithm.

If S_3 can be used, then S_1 can be avoided, because $S_3 S_1$ can be replaced by $S_2 S_2$. What happens when S_3 cannot be used? This is equivalent to having a maximum K, because S_3

"wastes" a vertex. When S_3 cannot be used (that is, K is maximum), sometimes S_1 can still be avoided. More precisely, if K is maximum and N is even ($N = 2K$), then S_1 can be avoided with the solution S_2^K; finally, if K is maximum and N is odd ($N = 2K - 1$), then S_1 is unavoidable because the only solution is $S_2^{K-1}S_1$.

To summarize the above paragraph, if $N = 2K-1$ then the only solution is $S_2^{K-1}S_1$; otherwise, the output string can be build by repeating S_2 and S_3 as substrings. Since the occurrences of S_2 and S_3 can be sorted arbitrarily, it is enough to calculate how many copies of each of them are needed. This can be done by solving the system

$$\begin{aligned} N &= 2n_2 + 3n_3 \\ K &= n_2 + n_3 \ , \end{aligned}$$

where n_2 and n_3 are respectively the numbers of copies of S_2 and S_3. The solution of the system is $n_2 = 3K - N$, $n_3 = N - 2K$. Notice that $\lceil N/3 \rceil \leq K$ implies $n_2 \geq 0$, while $K \leq \lceil N/2 \rceil$ and $N \neq 2K - 1$ imply $n_3 \geq 0$.

The complexity of the algorithm we just described is $O(K)$. The Python code below implements this approach.

```python
#!/usr/bin/env python3

K, N = map(int, input().split())

K2=K+K
K3=K+K2

if N<K2-1 or K3<N:
   print('*')
elif N==K2-1:
   print('X-'*(K-1), 'X', sep='')
else:
   print('X-'*(K3-N), '-X-'*(N-K2), sep='')
```

Naive algorithm

A possible algorithm is just trying to build a valid occupation of the toilets. If this is achieved, output the result; if the process fails, output "`*`" to inform that no valid occupation exists.

For doing so, start with a string having the chosen toilets arranged in the most possible compact form, that is, with the string $S = S_2^{K-1}S_1$. Then, while $|S| < N$ repeatedly insert "`-`" in S, at most once before each occurrence of "`X`", and at most once after the last occurrence of "`X`". Finally, output S if $|S| = N$ and one of the last two positions of S is "`X`". Otherwise output "`*`".

It is possible to implement the above algorithm with complexity $O(K)$.

Alternative approach

Even without the graph formulation it's possible to notice that the minimum number of people needed to fill the N toilets while respecting the privacy condition is $\lceil N/3 \rceil$, and the maximum number is $\lceil N/2 \rceil$.

Thus, the feasible range for K is $\lceil N/3 \rceil \leq K \leq \lceil N/2 \rceil$, meaning that deciding whether there's a solution for given values of N and K is easy.

This actually also suggests a simple approach for constructing an answer: if we attempt to occupy as many toilets as possible while maintaining the privacy condition by using a pattern like "-X-", we create a reduced problem with $N' = N - 3$ and $K' = K - 1$. We can check if this is feasible. If not, we need to occupy toilets more compactly by using "X-" (or just "X" if there's only one toilet left), reducing the problem to $N' = \max(N - 2, 0)$ and $K' = K - 1$.

Problem G – Grand Glory Race

Author: André Amaral de Sousa, Brasil

If we had a single target village as the endpoint, calculating the results for all leaf villages would be straightforward. We could perform a DFS starting from this target village, traversing each subtree to identify which leaf claims each village along the way. This DFS would also count how many villages each leaf claims, as each village would be claimed by the first runner to reach it.

The challenge here is that the endpoint ("Crown Village") can vary with each query. Recalculating a DFS for each possible endpoint would be inefficient. Instead, we leverage a key observation:

If the race finishes at village A instead of village B, where A and B are neighboring villages, only the runners who initially claimed A or B will see a change in their claimed villages. Specifically, the runner who claimed B might now be able to "steal" village A if they arrive at A first.

With this observation, we can apply a tree rerooting technique to solve the problem offline for all queries efficiently.

First, using DFS with village 1 as the root (effectively setting village 1 as the race's endpoint), we calculate the following for each vertex v:

- best_leaf[v]: the best leaf for vertex v, i.e., the leaf that will claim vertex v.
- best_dist[v]: the distance from the best leaf of v to v.
- second_best[v]: the second-best leaf for vertex v – specifically, the best leaf that originates from a different subtree than the best leaf.
- second_dist[v]: the distance from the second-best leaf of v to v.

To efficiently answer queries for all villages as possible endpoints, we perform another DFS to calculate results for each vertex as a root (the tree rerooting technique). The core idea is:

- When rerooting from a vertex v to one of its children w, only the results for v and w change, as all other parts of the tree remain unaffected.
- If best_leaf[w] = best_leaf[v], then w contributes the best leaf for v. When rerooting to w, this best leaf stops at w, so we update v's best leaf to be its second-best leaf.
- If best_leaf[w] $\neq$ best_leaf[v], then v's best leaf does not come from w, so we check whether v's best leaf should now be the best for w.

Throughout both DFS passes, we also maintain an array claimed_by[$leaf$] that counts the number of villages claimed by each leaf. In the second DFS, when we enter a vertex v, all the information we have is based on v as the root (end of the race), allowing us to answer all queries that have v as the end of the race.

Alternatively, those with good templates templates on their library can also overkill the problem with a more complex online solution that combines (LCA + Path-jumping) with Tree

Rerooting, giving us a time complexity of $O(N \log N + Q \log^2 N)$. This lets you answer queries on-the-fly by first using tree rerooting to precompute the "winning leaf in the subtree for every node with every root". Then, for each query, you can do a binary search along the path between E and F to count how many villages are claimed.

Problem H – Heraclosures

Author: Agustín Santiago Gutiérrez, Argentina

In asymptotic complexities, we will assume $N = O(M)$ for simplicity.

It is a classical exercise to compute the total execution times $T(i)$ of each function in $O(M)$ time using Dynamic Programming. One just has to apply the statement formula for each i, making sure times $T(j)$ for functions called by function i have already been computed. This can be done for example by computing a Topological Sort of the call-graph using DFS or any other technique, then that particular order can be used to update $T(i)$ values. It is common also to compute the $T(i)$ values themselves in the same DFS that computes the Topological Sort.

The main problem is how to efficiently update this computed value, as changing a function's B_i value forces recomputation of $\Theta(N)$ values of $T(i)$ in the worst case.

The intended solution has complexity $O(NM + E\sqrt{M})$.

The main idea is to first of all, compute all initial values $T(i)$ in $O(M)$ as already explained.

Then, we divide the update events into chunks of k events (where k a is a parameter of the solution algorithm). We process events in input order. After each chunk, we do a full recomputation in $O(M)$ time of all the $T(i)$ values. This recomputation is based on the up to date B_i values, having accumulated updates up to that point in time.

Thus critically, after recomputing those $T(i)$, we can completely forget from that point onwards about previous updates, and continue execution as if those B_i values were the initial input, and those updated $T(i)$ values were the initially computed ones. All of this recomputation will have total cost $O(M\frac{E}{k})$

Now, for any given query, we will only need to "correct" the currently computed $T(i)$ value at that point in time (which was based on all previous updates already, except for the current chunk), to account for the at most k updates that have happened in the current chunk.

For $T(x)$, an update changing the value B_y from v_1 to v_2 will result in a difference in the query result of $(v2 - v1) \cdot C(x, y)$, for a certain value $C(x, y)$. Critically, as we hint by the chosen notation, this value $C(x, y)$ depends only on x and y, but not on the values themselves. So if we can precompute C, an $N \times N$ matrix of coefficients, each query can be corrected in $O(k)$ time, since each correction becomes an $O(1)$ adjustment to the precomputed $T(i)$ result.

It is possible to notice that the $C(x, y)$ coefficient is simply the total number of paths from x to y in the call graph, so computing $C(x, y)$ can be done at the beginning of the program in $O(NM)$ time using dynamic programming.

Thus total runtime is $O(NM + Ek + M\frac{E}{k})$. Choosing $k = \Theta(\sqrt{M})$ gives the expected complexity.

Problem I – Inversion Insight

Author: Rafael Grandsire, Brasil

This problem can be split in two phases. First, determine the exact number of inversions, denoted by x, that the target permutation must have. Then, find the specific permutation with x inversions that corresponds to the desired position K.

The main strategy for making the algorithm efficient is to avoid listing all permutations explicitly. Instead, we'll count the number of permutations with certain properties.

To achieve this, we use a recurrence relation that counts the number of permutations with n elements and x inversions. This will be the fundamental building block to the final algorithm:

$$f(n,x) = \begin{cases} \mathrm{I}[x=0] & n=0 \\ \sum_{0\le j<n} f(n-1,x-j) & \text{otherwise} \end{cases}$$

The recurrence works by considering all possible choices for the first element of the permutation. Among the n elements, if we choose the largest element as the first, it contributes $n-1$ inversions, meaning the remaining elements must have $x-n+1$ inversions to reach x inversions in total. The same logic applies when choosing the i-th largest element as the first, resulting in $x-i$ inversions among the remaining elements.

To compute $f(n,x)$ efficiently, memoization can reduce the time complexity to $\mathcal{O}(n^2x)$ and memory complexity to $\mathcal{O}(nx)$. One can also notice that, if we define $g(n,x)$ to be the prefix sum, with respect to the argument x, of the function $f(n,x)$,

$$g(n,x) = g(n,x-1) + f(n,x)$$

then the computation of $f(n,x)$ simplifies to

$$f(n,x) = g(n-1,x) - g(n-1,x-n),$$

which can be done in $\mathcal{O}(nx)$ time and memory complexity. With this tool, we can solve both phases of the problem.

To determine the number of inversions x in the K-th permutation, we find the largest x such that

$$\sum_{0\le i<x} f(n,i) < K$$

. We then update $K = K - \sum_{0\le i<x} f(n,i)$. Now, the target permutation is exactly the K-th lexicographic permutation with x inversions.

To build this permutation, store the elements $1,\ldots,n$ in a data structure that supports **erase_kth** and **retrieve_kth** operations (an ordered statistics set is suitable here). Using the values of f, we'll decide which elements to place in each position and retrieve them efficiently from the data structure. If we place the i-th element in the first position, then the remaining elements need precisely $x-i$ inversions. The first element is therefore the j-th one in the data structure for the maximum j such that

$$\sum_{0\le i<j} f(n-1,x-i) < K$$

. We continue this process by updating $K = K - \sum_{0\le i<j} f(n-1,x-i)$, $n=n-1$, $x=x-j$, and repeating until the data structure is empty.

This algorithm runs in $\mathcal{O}(nx+n\log(n))$, which is sufficient given the problem constraints. When n is a large value, $f(n,x)$ grows faster than the Fibonnaci sequence since it sums multiple terms rather than just two. Therefore, even for very large values of k, the desired x will be very small. One can experimentally check that for $n>100$, $x<15$.

This growth rate also raises concerns about overflow, as the values of $f(n,x)$ might not fit in 64 bits integers when K is that large. Contestants may need to use 128-bit integers or any other trick to prevent overflows – such as capping $f(n,x)$ at $K+1$, since only the point where the function exceeds K is relevant, exact counts aren't necessary.

Another useful observation – though not necessary to achieve an AC here – is that because x remains small for large values of n, the operation to extract the j-th value from our data structure will consistently remove the smaller elements. Thus, rather than using a complex data structure, we can keep the elements in a simple array sorted in decreasing order. Each removal will then always occur at the end of the array, reducing the complexity to $\mathcal{O}(nx)$ and significantly improving runtime in practice. For instance, in C++, removing the j-th element from the end (`vector.erase(j-th)`) would actually require a single *memmove* to shift the j elements after the removed one forward.

Problem J – Jigsaw of Shadows

Author: Paulo Cezar Pereira Costa, Brasil

A flatlander standing at position X_i with height H_i will cast a shadow of length L_i in the eastward direction.

Given the angle of the flashlight θ, the length L_i of the shadow for a flatlander of height H_i can be calculated using trigonometry:

$$L_i = H_i \cdot \cot(\theta)$$

Here, $\cot(\theta) = \frac{\cos(\theta)}{\sin(\theta)}$. The idea is that in a right triangle formed by the flatlander, their shadow, and the light beam, $\cot(\theta) = \frac{\text{Adjacent (shadow length)}}{\text{Opposite (height)}}$. Note that since θ is given in degrees, we need to convert it to radians when computing trigonometric functions.

That means that each flatlander's shadow can be seen as an interval $[X_i, X_i + L_i]$ and we are interested in the classic problem of calculating the length of the union of intervals.

We can start by sorting the flatlanders by X_i and then iterate over them merging overlapping shadows and accumulating the total length. Notice that here all comparison can be made on integers only, and as X_i values are distinct there's no need to worry about tie-breaking. Then, zero-initialize two variables: `total_shadow_length` to accumulate the total covered length, and `current_end` to keep track of where shadows currently end.

For each flatlander and their corresponding $[X_i, X_i + L_i]$ interval:

- If $X_i + L_i >$ `current_end`, we need to update `current_end` (and set it to $X_i + L_i$).
- This will add $X_i + L_i - \max($`current_end`$, X_i)$ to `total_shadow_length`, as either the `current_end` intersects with this flatlander's shadow or we are starting a new non-overlapping one at X_i.

By the end of this process, `total_shadow_length` will contain the total length of the road covered by shadows, accounting for all overlaps.

Sorting the flatlanders takes $O(N \log N)$ time and merging intervals and calculating the total covered length takes $O(N)$ time, as each interval is processed once. Thus, the overall time complexity is $O(N \log N)$.

Another possibility would be using a line sweep technique, we can think of each interval as two "events", a "shadow start" event at X_i and a "shadow end" event at $X_i + L_i$. If we add all these events to an array and sort them we can move an imaginary line of the x-axis (iterate on the sorted events), keep track of whether there's an active shadow or not and accumulate the covered length accordingly.

Problem K – Kool Strings

Author: Luis Santiago Re, Argentina

This problem seems very easy, and it is indeed easy, but it has some interesting corner cases and details that you need to be careful with to solve it.

Almost correct solution: probably the first idea that one could think of is to separate S into blocks of contiguous elements with the same value: $B_1, B_2, ..., B_z$, and then think that the answer is $\sum \lfloor \frac{size(B_i)}{K} \rfloor$ and that it could be achieved by flipping the bits in positions of the form $B_{i_{j \cdot K}}$. For example in the "input 3" above $S = 1111100$, then the blocks would be $B_1 = 11111$ and $B_2 = 00$, and the answer would be $\lfloor \frac{5}{3} \rfloor + \lfloor \frac{2}{3} \rfloor = 1 + 0 = 1$ by making $B_1 = 11011$, which in this case is correct. But this is not always correct, because if the first or last element of a block is flipped, then it might become part of the adyacent block next to it, so this needs to be handled carefully. For example, if $S = 11111100$ and $K = 3$, the idea mentioned above will answer $2 \rightarrow 11011000$, which is wrong. A correct solution would be for example $2 \rightarrow 11010100$.

Correct solution: it is easy to see that $\sum \lfloor \frac{size(B_i)}{K} \rfloor$ is a lower bound of the solution. And below there is a way of solving the cases with $K > 2$ using that number of operations:

- For blocks of size not divisble by K, do what's mentioned in the almost correct solution above.
- For blocks of size divisble by K, do almost the same, but make a single change to it: instead of flipping the bit in the last position of the block, flip the bit in the position before the last one.

With the construction mentioned above, given that $K > 2$, it is guaranteed that the first and last element of each block are not modified, and this makes the blocks independent, as they will not be affected by the changes made in other blocks (because the first and last element of each block act as a "barriers" or separators between adyacent blocks).

All that remains is to solve $K = 2$, which can be done by simply trying the only two possible alternating strings of length N: 101010... and 010101...

Problem L – Latin Squares

Author: Arthur Nascimento, Brasil

The sequence of operations induces two permutations of size N, one for the rows and one for the columns of the matrix. Let's denote these by $r[i]$ for rows and $c[i]$ for columns. Our goal is to find a Latin square M such that $M[i][j] = M[r[i]][c[j]]$ holds for every pair (i, j).

We can analyze the "row permutation" and "column permutation" independently, as performing row swaps and column swaps are independent operations that commute with each other. When we decompose both permutations into cycles, we observe that all cycles in both permutations must have the same length. Notice that even cycles in the row permutation must match the lengths of cycles in the column permutation; there cannot be cycles of different lengths between them.

To understand why, consider that if $r[i]$ has a cycle of length A and $c[j]$ has a cycle of length B, and without loss of generality, assume $A < B$. Iterating the condition $M[i][j] = M[r[i]][c[j]]$ A times would yield $M[i][j] = M[r^A[i]][c^A[j]] = M[i][c^A[j]]$, implying that row i has repeated entries, contradicting the requirements of a Latin square.

If all cycles have the same length S (which must divide N), we can construct the Latin square by assuming that each cycle follows the structure $(1, 2, \ldots, S)$, $(S+1, S+2, \ldots, 2S)$, $(2S+1, 2S+2, \ldots, 3S)$, and so on, up to $(N-S+1, N-S+2, \ldots, N)$. If this is not the case, we can relabel the row and column indices as if it were true and apply a corrective permutation to our construction afterward.

Now, divide the matrix into blocks of size $S \times S$. For each (i, j)-th block, fill it with a Latin square that contains only numbers between $k \cdot S + 1$ and $k \cdot S + S$, where $k = (i + j) \mod (N/S)$. Essentially, this approach uses each small Latin square as a unit to create a "Latin square of Latin squares" of size $(N/S) \times (N/S)$, with each mini Latin square having the same pattern but containing numbers in ranges $[1..S], [S+1..2S], \ldots$, as defined above.