# The 2024 ICPC Caribbean Finals Qualifier

## Editorial

## Problem set developers

Humberto Díaz Suárez – Universidad de Puerto Rico, Mayagüez
Jorge Alejandro Pichardo Cabrera – Universidad de La Habana
José Carlos Gutiérrez Pérez – Universidad de La Habana
José García Negrón – Universidad de Puerto Rico, Mayagüez
Marcelo J. Fornet Fornés – ICPC Caribbean
Vladimir Otero Mariño – Universidad de las Ciencias Informáticas
Carlos Joa Fong – Altice Dominicana
Rubén Alcolea Núñez – Leil Storage

## Matcom Online Grader

Leandro Castillo Valdés – ICPC Caribbean
Frank Rodríguez Siret – Harbour Space University

October 12th, 2024

# Problem A. Accidental Command

**Author: Humberto Díaz Suárez – UPR, Mayagüez**
**Category: Strings 3**
**Solved by 2 teams**

The *accidental command* (AC) property can be formalized as follows: a command $w$ has the property if and only if there exists a pair of commands $p, q$ such that $w$ is a prefix or suffix of $p + q$. In the prefix case, we require that $w \neq p$. In the suffix case, we require that $w \neq q$. Note that $p$ and $q$ can be the same command — the statement does not require the two to be different.

The key is to analyze each command and try to represent it as a prefix or suffix. This is more efficient than examining all pairs of commands.

There are two ways that $w$ could be a prefix:

- Condition 1: $w$ is a prefix of a command $p$ (where $w \neq p$). Example: `rm` is a prefix of `rmdir`.
- Condition 2: There exist two commands $p$ and $q$ (where $w \neq p$) such that $w$ is a prefix of $(p + q)$. That implies that $p$ is a prefix of $w$. Let $w'$ be the remaining portion of the $w$. $w'$ must be a prefix of $q$. Example: `rmdir` is a prefix of `rm dir` (after concatenating).

Both of these checks can be performed efficiently (in linear time per command) using rolling hashes. Let's define some functions for convenience. $\texttt{hash}(w)$ is the hash of a string $w$. $\texttt{hash}(w, start, length)$ is the hash of a substring of $w$ with the specified starting index and length. We'll assume strings are 0-indexed. The hashes are available to us in constant time after precomputing prefix hashes.

The algorithm is as follows:

1. Initialize $H$, a list of hash maps to associate (prefix length, prefix hash) with a count — the number of times any prefix of that length appears with that hash.
2. Initialize $F$, a list of hash sets to contain (string length, string hash) entries. It provides a way to efficiently check whether some string with a certain length and hash is a valid command.
3. Precompute prefix hashes for all strings in the input. Also load those hashes into $F$ and $H$.
4. Iterate over each command $w$:
   (a) Check whether $H[|w|, \texttt{hash}(w)] > 1$. That's equivalent to whether there is more than one prefix of length $|w|$ with the same hash as $w$. If so, then there must be another string for which $w$ is a prefix. Therefore, $w$ meets condition 1 and it's an AC.
   (b) Iterate over each prefix length $x$ from 1 to $|w| - 1$:
      i. Check whether $F$ contains $(x, \texttt{hash}(w, 0, x))$. If so, then we found a prefix of $w$ with a hash matching another command.
      ii. Check whether $H$ contains the key $(|w| - x, \texttt{hash}(w, x, |w| - x))$. If so, then we found a suffix of $w$ with a hash matching a command's prefix.
      iii. If both checks succeed, then $w$ meets condition 2 and it's an AC.

The suffix checks can be implemented similarly with reversed strings. Since prefix and suffix checks can be done in $O(|w|)$ time, the time complexity of all checks for a particular command is $O(|w|)$ time. Precomputing the hashes, setting up the data structures, and performing these checks for $n$ commands takes $O(s)$ time, where $s$ is the sum of all command lengths. This is optimal.

The output step requires printing the indices of accidental commands in ascending order. This can be done in $O(n \log n)$ or $O(n)$ time.

Note that solutions relying solely on 32-bit hashes will tend to give false positives. That is, hash collisions are common enough that a solution may incorrectly identify a command as having the AC property and give incorrect results. A few teams had their submissions fail because of this. There are workarounds, such as checking matches against the actual strings, using two different 32-bit hashes, or using 64-bit hashes.

There is an alternative approach that checks prefixes and suffixes with tries instead of hashes. Its time complexity is $O(nk^2)$, where $k$ is the length of the longest command. Although its running time is poor for specially-crafted inputs, it tends to outperform hashing in most cases. Combining tries and rolling hashes gives another optimal solution which has a lower probability of false positives, allowing for 32-bit hashes to pass.

## Problem B. Bespoke Shuffle

**Author: Humberto Díaz Suárez – UPR, Mayagüez**
**Category: Combinatorics, Number Theory 3**
**Solved by 0 teams**

Let's index the deck with integers from 1 to $N$ from top to bottom. Also, let's number the cards with integers from 1 to $N$. Initially, card 1 is in position 1, card 2 is in position 2, ..., and card $N$ is in position $N$. So, we have a **permutation** of numbers from 1 to $N$.

Each basic operation can be interpreted as a permutation $P$: it takes an object (a card) in position $i$ and maps it to position $P[i]$. A sequence of basic operations corresponds to a **composition** of permutations (see the article in `https://en.wikipedia.org/wiki/Permutation_group`), which is also a permutation.

If we view the permutation $P$ of size $N$ as a graph of $N$ vertices, where there is a directed edge from vertex $v$ to vertex $P[v]$ and traverse this graph, we obtain a list of **cycles**. Therefore, we can say that a permutation consists of a list of cycles (see `https://en.wikipedia.org/wiki/Permutation#Cycle_notation`).

The **period** (aka **order**) of a permutation $P$ is the number of times we need to apply this permutation $P$ to an initial sequence $S$ of $N$ objects to get back to the original sequence $S$. The period of a permutation with $M$ cycles, whose lengths are $C_1, C_2, ..., C_M$, equals the least common multiple of $C_1, C_2, \ldots, C_M$ (see `https://en.wikipedia.org/wiki/Permutation#Order_of_a_permutation`).

Since $lcm(x, y) = \frac{x \cdot y}{gcd(x,y)}$, we can get rid of the gcd part if we work with relatively prime pairs $x$ and $y$ (i.e. $gcd(x, y) = 1$).

Back to the problem: To start, let's factorize the required period $Z$ and write $Z$ as the product of prime powers: $Z = p_1^{e_1} \cdot p_2^{e_2} \cdot p_3^{e_3}, \ldots$. Since each prime power factor does not share a factor with another prime power factor, they are pairwise relatively prime. So, if we can get cycles whose lengths are precisely these prime power factors, we are *golden* since the size $N$ of such permutation is $p_1^{e_1} + p_2^{e_2} + p_3^{e_3} + \ldots$. This size is optimal: there is no permutation of smaller size that yields period $Z$. This is important because, although the problem statement assures us that all test cases will be solvable, there are choices of $Z$ for which every suboptimal $N$ would exceed the $10^4$ upper limit (e.g. $Z = 975002$).

How do we get a permutation with specific cycle lengths $p_1^{e_1}, p_2^{e_2}, p_3^{e_3}, \ldots$?

- Let's start with a basic operation of shifting the deck by 1. This creates a giant cycle of size $N$.
- For each prime power $p_i^{e_i}$ (except the last), let's "remove" $p_i^{e_i}$ cards from the deck, going from bottom to top. We will create a cycle with these "removed" cards. We accomplish this by swapping the cards at positions 1 and *last*, where *last* is the index of the last removed card.

In this solution, the required number of basic operations equals the number of distinct prime factors of period $Z$. For $Z \leq 10^6$, the maximum number of basic operations is 7, when $Z = 2 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \cdot 13 \cdot 17$.

## Problem C. Connecting modules

**Author: Vladimir Otero Mariño – UCI**
**Category: Graph Theory 3**
**Solved by 22 teams**

**Solution 1:**

Let's begin by interpreting this problem as a directed graph with cycles. To solve this problem, it is possible to consider performing a search through the graph's nodes using DFS or BFS. The main issue lies in the cycles. All the nodes within a cycle can send messages, either directly or indirectly, to the other nodes in the cycle, including the node itself. One way to solve this problem is to divide it into two stages. The first stage involves extracting the strongly connected components and counting, within these components, the number of nodes that belong to a component containing cycles and are configured for streaming. Then, it is possible to create a condensed graph with the strongly connected components to run a search algorithm on this condensed graph to count the nodes that do not belong to cycles, are configured for streaming, and can send messages to another node that streams.

**Solution 2:**

The problem boils down to checking, for each streaming node, whether there exists a *non-empty* path starting from that node and ending at a streaming node. If we run a DFS or BFS from each streaming node, our solution will time out. Instead, we run a multi-source BFS on the *reverse* graph, where the sources are all streaming nodes. This produces the set of nodes that are reachable from any streaming node in the reverse graph. In the original graph, this set corresponds to the nodes that can reach a streaming node. In the last step, we just need to count all the nodes in this set that are also streaming nodes.

## Problem D. Dance Dance Daisy

**Author: José García Negrón – UPR, Mayagüez**
**Category: Brute force 2**
**Solved by 82 teams**

This problem can be solved with a brute-force solution. The key point is always to prioritize the arrows that score the most points for every input. Because of this, a good brute-force solution would be multiple 'for' loops ordered in decreasing priority and checking if the output equals the input. Something like this would work:

```
for a in range(100,-1,-1):
    for b in range(100,-1,-1):
        for c in range(100,-1,-1):
            for d in range(100,-1,-1):
                if a+b+c+d == 100 and a*1000 + b*100 + c*10 + (100-c-b-a)*1 == score:
                    print("Perfect:", a)
                    ... etc
```

We can make this even more efficient by recognizing that the innermost loop does not need to exist and we can calculate it mathematically:

```
for a in range(100,-1,-1):
    for b in range(100,-1,-1):
        for c in range(100,-1,-1):
            if a+b+c <= 100 and a*1000 + b*100 + c*10 + (100-c-b-a)*1 == score:
                print("Perfect:", a)
                ... etc
```

**Bonus Challenge**: Can you come up with a solution with a better time complexity than the above $O(N^3)$ solution? $N$ is the number of arrows in a configuration.

## Problem E. Einstein's Riddle

**Author: Marcelo J. Fornet Fornés – ICPC Caribbean**
**Category: Brute force, Parsing 3**
**Solved by 0 teams**

This problem can be solved by iterating over all possible solutions and checking if all constraints match the current candidate. The number of candidates is $5!^4$ after we fix the house number and find all permutations for the remaining four features. Checking the constraints can be done in constant time if the candidate is adequately represented.

An alternative solution is modeling the problem as a Boolean Satisfiability Problem. We create $5^3$ variables of the form $v_{i,j,k}$ that is true if the person that lives in house $i$ has the $k$-th value in the $j$-th feature. We should express the natural constraint of the problems: for every $i, j$, exactly one index $k$ should be true, and for every $j, k$, exactly one $i$ should be true. Then we need to deal with individual constraints on the input.

With this problem, it turns out that almost all clauses in the SAT have exactly two variables (almost a 2-SAT), except for the clauses that make sure there is at least one element True in the set $v_{i,j,*}$. We can backtrack over those clauses and set every other variable that derives from these ones following the 2-SAT graph.

## Problem F. Family ÷ Spy

**Author: Jorge A. Pichardo Cabrera – UH**
**Category: Geometry 2**
**Solved by 98 teams**

Let's focus on the $x$ coordinate. Every time you apply a translation by a vector with value $v_x$ (on the $x$ coordinate), each $P_{i_x}$ turns into $P_{i_x} + v_x$. Now if your actual sum of points is $\sum_{i=1}^{n} P_{i_x}$, then the new one after translation will be $\sum_{i=1}^{n}(P_{i_x} + v_x)$, which is the same as $n \cdot v_x + \sum_{i=1}^{n} P_{i_x}$.

Basically, let $S_x$ be your actual sum and $v_x$ the translation vector you are applying. After the operation you only need to assign $S_x := S_x + v_x \cdot n$. Now extend this to the $y$ coordinate and you are done.

Time complexity: $O(n)$

## Problem G. Ghost in the SecuriTree

**Author: Jorge A. Pichardo Cabrera – UH**
**Category: Data Structures, DFS, Trees 3**
**Solved by 24 teams**

Answer the queries offline. As you perform a DFS, put a node in some data structure (DS) when you enter it and remove it when you leave. This way you maintain the entire path from the root in the DS at all times, since it is equivalent to the DFS calls on the system stack. You need to be able to query the DS for which is the node with the k-th lowest label. This can be done with an ordered set or Fenwick tree + implicit binary search. The expected time complexity is $O(n \log n)$.

## Problem H. Hurry Chinchilla!

**Author: Humberto Díaz Suárez – UPR, Mayagüez**
**Category: Arithmetic, Probability, Graph Theory 4**
**Solved by 0 teams**

The statement describes how a packet travels around a graph representing a network. There are $n$ nodes, numbered 1 through $n$. The packet begins at node 1 and its destination is the goal node ($g$). At each node (except the goal), we choose a neighboring node at random with equal probability and forward the packet to it. If the packet arrives at the goal, then the walk ends. Otherwise, we continue passing the packet around.

The resulting walk from the start to the goal may involve edges that are **antioptimal**. An edge is antioptimal if moving the packet across the edge would not decrease the packet's shortest-path distance to the goal. The objective is to calculate the expected number of antioptimal edges in a packet's random walk.

Let $X_u$ be a random variable representing the number of antioptimal edges in a walk from node $u$ to node $g$. We can define its expected value in terms of the expected values for neighboring nodes:

$$E[X_u] = \frac{1}{deg(u)} \cdot \sum_{v \in adj(u)} (E[X_v] + I(u,v))$$

- $E[X_u]$ is the expected value of $X_u$.
- $deg(u)$ is the degree of node $u$.
- $adj(u)$ is the adjacency list of $u$.
- $I(u,v) = 1$ if edge $(u,v)$ is antioptimal. Otherwise, $I(u,v) = 0$.

$X_g$ is a special case. Walks end at the goal, so its expected value is always zero: $E[X_g] = 0$

This is a classic recursive probability problem. The objective is to calculate $E[X_1]$. But how do we solve for these expected values when they are interdependent?

We are effectively working with a system of $n$ linear equations. To simplify, let's say $y_u = E[X_u]$, and that we assign edge weights such that $w(u,v) = I(u,v)$. We substitute throughout our equations:

$$y_u = \frac{1}{deg(u)} \cdot \sum_{v \in adj(u)} (y_v + w(u,v))$$

$$y_g = 0$$

Also, we rewrite the equations to separate a constant term from our variables. It's a useful representation for later:

$$deg(u) \cdot y_u - \sum_{v \in adj(u)} y_v = \sum_{v \in adj(u)} w(u, v)$$

It's critical to note that the resulting system of equations is **indeterminate** if we don't account for the variables of nodes that have no path to the goal. Indeterminate means that some variables have multiple valid values that satisfy the equations. This condition generally causes equation-solving algorithms to fail, even if $y_1$ is well-defined, so we have to correct for it.

With the theory laid out, we can now discuss the solution. Start with a BFS originating from the goal to assign single-source distances to all nodes. This also allows us to detect unreachable nodes and to assign weights to antioptimal edges. If node 1 is unreachable, then we output `NO ROUTE`. This step runs in $O(n + k)$ time, where $n$ is the number of nodes and $k$ is the number of edges.

Next, we express our linear system as a matrix multiplication of the form $A\mathbf{y} = \mathbf{b}$, where...

- $\mathbf{y} = (y_1, \ldots, y_n)^T$ is the column vector of variables.
- $\mathbf{b}$ is the column vector of constant terms. Here, $\mathbf{b}_u = \sum_{v \in adj(u)} w(u, v)$.
- $A$ is the $n \times n$ coefficient matrix.

For convenience, initialize matrix $A$ to zeroes. We define certain elements of each row $u$ as follows:

- If node $u$ is reachable from the goal, then:
  - $A_{u,u} = deg(u)$
  - For $v \in adj(u)$: $A_{u,v} = -1$
- Otherwise, node $u$ is unreachable. Set $A_{u,u} = 1$. This prevents the issue of indeterminate equations.

There are a few ways to solve for our linear system. The simplest is some variant of Gaussian elimination. It runs in $O(n^3)$ time and has reliable accuracy. An implementation of Gaussian elimination is one of those should-have competition algorithms.

An involved alternative is to use Cramer's rule to compute $y_1$ from two matrix determinants. The determinants can be calculated using Leibniz's formula for determinants and Heap's algorithm for enumerating permutations with alternating signs. The advantage of this approach is that all calculations can be done with integer arithmetic (since all coefficients are integers). Division is only performed once at the end, allowing us to get a rational result without loss of precision. The downside is the time complexity, which is a shocking $O(n! \cdot n)$ time, assuming permutations are enumerated efficiently. The running time can be improved by pruning unreachable nodes from the graph and pruning the variable for the goal ($y_g = 0$). That gets us to $O((n-1)! \cdot n)$ time – a substantial reduction due to the factorial term.

Finally, we should mention the issue of rounding. Modern computers typically implement floating-point arithmetic according the guidelines set out by the IEEE Standard for Floating-Point Arithmetic (IEEE 754). It specifies that processors should "round half to even"[1]. This differs from how humans usually round numbers and affects a few test cases. We can add a tiny correction factor (e.g. $10^{-10}$) to ensure answers are rounded as we would expect.

[1] See `https://en.wikipedia.org/wiki/Rounding#Rounding_half_to_even`.

***Bonus:*** *Is there another way?*

We can use an iterative approximation. For every reachable node $u$, there is a sequence of approximations $(r_{u,1}, r_{u,2}, r_{u,3}, \dots)$ that approaches the true value of $E[X_u]$. Let $r_{u,1} = 0$. We define the following recurrence for $i > 1$ and $u \neq g$:

$$r_{u,i} = \frac{1}{deg(u)} \cdot \sum_{v \in adj(u)} (r_{v,i-1} + w(u,v))$$

Of course, the goal is an exception: $r_{g,i} = 0$ for $i \geq 1$.

This can be computed efficiently using dynamic programming. After $t$ iterations (for a large enough $t$), $r_{1,t} \approx E[X_1]$ . This is feasible because the graphs in this problem are relatively small. Otherwise, it could take too many iterations to achieve an accurate result.

The advantage of this approach is its simpler implementation. It isn't affected by leaving unreachable nodes in the graph (though watch out for $deg(u) = 0$) and can be done without matrix operations. The downside is that converging on a correct answer depends on performing enough iterations. This solution runs in $O(t(n + k))$ time, where $n$ is the number of nodes, $k$ is the number of edges, and $t$ is the number of iterations used. In practice, $t = 1000$ suffices for this problem.

## Problem I. Inconsecutive Digits

**Author: Marcelo J. Fornet Fornés – ICPC Caribbean**
**Category: Ad-Hoc 1**
**Solved by 147 teams**

**Solution 1:**
One possible solution is to generate all 10! permutations and check each one to see if it satisfies the given constraints. Since the problem asks for any valid solution, the process can stop as soon as the first valid permutation is found, which can then be selected as the correct answer.

Some programming languages, such as C++, offer useful functions like `std::next_permutation` to efficiently generate permutations. In languages that lack such built-in functions, a recursive approach can be implemented to generate all permutations manually.

**Solution 1b:**
Instead of generating all possible permutations of 10 digits, let's generate a random one and check if it satisfies the conditions given in the statement. If it invalid, generate another one. Do this until you find a valid permutation. The probability that a random permutation is valid and starts with $a$ and $b$ is around $10^-3$. But if you do this 10000 times, the probability of finding a valid one is near 1 (Look up the Birthday Paradox). You may increase the probability of finding an answer by fixing the first two digits of the permutation (to $a$ and $b$ respectively) and shuffling the other digits, but this was not required for this problem.

**Solution 2:**
An alternative approach to solving the problem takes advantage of the fact that any valid sequence can be printed. With this in mind, we can define a valid sequence of ten digits where the first and last digit are not consecutive. The sequence below illustrates such a case:
`0, 2, 4, 6, 8, 1, 3, 5, 7, 9`

Initially, it may seem that we could simply remove the first two digits given in the input and then construct a valid sequence from the remaining digits. However, this approach doesn't always work. Depending on the specific values of the two digits provided, extra adjustments may be required to "fix" the sequence and ensure it is valid.

A simple trick can be used to guarantee that a valid sequence is always generated. By concatenating the initial valid sequence with itself, we create a sequence of 20 digits:
`0, 2, 4, 6, 8, 1, 3, 5, 7, 9, 0, 2, 4, 6, 8, 1, 3, 5, 7, 9`

Starting from this new sequence, we can always form a valid solution by finding the position of the first occurrence of digit `b`, and from the next position onward, adding each digit to the solution, except for `a` and `b`. Once we've scanned through the sequence, our solution will be complete.

**Example Walkthrough**

Consider the first example from the problem, where $a = 4$ and $b = 9$.

We begin with the sequence: `0, 2, 4, 6, 8, 1, 3, 5, 7, 9, 0, 2, 4, 6, 8, 1, 3, 5, 7, 9`

- **Step 1:** Add $a$ and $b$ as the first two digits of the solution:
  - Solution: $\{4, 9\}$
- **Step 2:** Locate the position of $b$ in the sequence. In this case, $b = 9$ is at position 9. We'll start scanning from position 10.
- **Step 3:** Add every digit from position 10 onward, as long as it's different from $a$ and $b$:
  - Solution: $\{4, 9, 0\}$ (Added 0)
  - Solution: $\{4, 9, 0, 2\}$ (Added 2)
  - Digit 4 is skipped, as it is equal to $a$
  - Solution: $\{4, 9, 0, 2, 6\}$ (Added 6)
  - Solution: $\{4, 9, 0, 2, 6, 8\}$ (Added 8)
  - Solution: $\{4, 9, 0, 2, 6, 8, 1\}$ (Added 1)
  - Solution: $\{4, 9, 0, 2, 6, 8, 1, 3\}$ (Added 3)
  - Solution: $\{4, 9, 0, 2, 6, 8, 1, 3, 5\}$ (Added 5)
  - Solution: $\{4, 9, 0, 2, 6, 8, 1, 3, 5, 7\}$ (Added 7)
  - Digit 9 is skipped, as it is equal to $b$

Thus, the final solution for this example is {`4, 9, 0, 2, 6, 8, 1, 3, 5, 7`}.

## Problem J. Just the Coffee We Need

**Author: Humberto Díaz Suárez – UPR, Mayagüez**
**Category: Ad-Hoc 1**
**Solved by 201 teams**

Let $T$ be the temperature read from input.

- Loop over all "cold" temperatures (i.e., integers from 0 to 48). If one of them is equal to $T$, print the phrase `TOO COLD` and terminate your program.
- Loop over all "hot" temperatures (i.e., integers from 61 to 99). If one of them is equal to $T$, print the phrase `TOO HOT` and terminate your program.
- Loop over all "good" temperatures (i.e., integers from 49 to 60). If one of them is equal to $T$, print the phrase `ACCEPTED` and terminate your program.

- Otherwise, print `Are you serious?` and open a clarification request asking for a refund of your registration fee.

**Bonus Challenge**: Can you come up with a solution that has a better *time complexity*?

## Problem K. Keen Tree Design

**Author: Marcelo J. Fornet Fornés – ICPC Caribbean**
**Category: Dynamic Programming 4**
**Solved by 0 teams**

In this problem, we are asked to build an optimally balanced segment tree to minimize the cost of answering a given distribution of queries provided as part of the input.

To construct the optimal tree from the root, we should choose the best pivot and build the optimal subtrees to the left and right by splitting the original segment using this pivot. This problem is reminiscent of the "famous"Matrix Chain Multiplication problem, where the goal is to find the optimal way of adding parentheses to minimize the number of operations needed for a sequence of matrix multiplications. Adding parentheses is analogous to building a binary tree, similar to what we are trying to solve here.

Following this approach, we can model the problem using dynamic programming as follows:

$dp[i, j]$ := The cost of the optimal tree for the segment from i to j inclusive

(*) Although this model won't solve the problem, it is a natural starting point.

To compute the value of $dp[i][j]$, we will try every possible pivot $p$ within the segment and choose the one that yields the minimum cost:

$$dp[i, j] = min_{i \le p < j}(cost(i, j, p) + dp[i, p] + dp[p + 1, j])$$

If we can compute cost$(i, j, p)$ in $\mathcal{O}(1)$ time, then the entire solution could be calculated in $\mathcal{O}(n^3)$, by iterating over all possible states of the dynamic programming table, starting from smaller to larger segments. The final solution will be $dp[1, n]$.

We must consider how different queries affect cost$(i, j, p)$. Here, we encounter the following challenge: if a query fully covers our segment, we cannot immediately determine if it will reach the segment from $i$ to $j$ or stop at an ancestor node above it. However, given a fixed segment and pivot, we can determine if the query needs to traverse to one or both children. Consequently, we redefine our dynamic programming state as follows:

$dp[i, j]$ := The cost of the optimal tree for the segment from i to j inclusive, excluding the cost of calling the root query.

This way, we move the cost of calling *query* from the *dp* table to the *cost* function. In this case, the solution will be $dp[1, n] + a \cdot S$, where $S$ is the total number of queries, the sum of the frequency of all queries.

We initialize the table with $dp[i, i] = 0$, since the cost of solving a segment of size 1 is 0.

To compute the *cost* function, we need to analyze all possible ways in which we will call *query* on the children, and in which cases we will call *combine*.

- We will call *combine* if the query intersects both children but doesn't contain the current segment.
- We will call *query* on the left node if the query intersects the left child segment but doesn't contain the current segment.
- Similarly, we will call *query* on the right node if the query intersects the right child segment but doesn't contain the current segment.

With the previous characterization, we can compute the *cost* function in $\mathcal{O}(n^2)$ by checking the condition on every possible query from $x$ to $y$ on the transition $i$, $j$ with pivot $p$.

To speed up this solution, we should notice that the interesting queries always lie in the union of a constant number of rectangles, which we can compute in $\mathcal{O}(1)$ by using the Inclusion-Exclusion Principle over a cumulative sum table in 2D. See the reference solution for more details.

## Problem L. Lisan al Gaib

**Author: Jorge A. Pichardo Cabrera – UH**
**Category: Game Theory 4**
**Solved by 0 teams**

Since the number of stones does not change, we can see the problem as having all the stones in a line, separated by $n-1$ vertical bars. The number of stones between bar $i$ and $i+1$ represent the pile $i+1$, the number of stones before bar 1 is the pile 1, and the number of stones after bar $n-1$ is pile $n$. Now the game ends when all the bars are at the end (i.e. pile 1 contains all the stones). If we define the position of a bar as the number of stones it has to its right, then we can say that what an operation does is decrease the position of the bar $i$, that is, move it to the right. The game ends when all the bars are in position 0. Since the player who wins is the one who runs out of moves (and not the other way around as is common), then we are playing *Misere Nim*. All that remains is to determine the position of the $n-1$ bars: this is the suffix sum from position $n$ to 2.

In Misere Nim, the losing state is given by: $n_1 \oplus \cdots \oplus n_k = 0$ if some $n_i > 1$ and $n_1 \oplus \cdots \oplus n_k = 1$ if all $n_i \leq 1$. It's easy to check that from this state you can only move to a winning state and from any winning state it's always possible to move to a losing one.

## Problem M. Multiprocessor Scheduler

**Author: José Carlos Gutiérrez Pérez – UH**
**Category: Graph theory, Maximum flow 4**
**Solved by 0 teams**

The proposed approach to solve this problem uses maximum flow in a graph. If we take the set of all values of $s_i$, $e_i$, i.e., $S = \{s_1, s_2, \ldots, e_1, e_2, \ldots\}$, remove duplicates, and order them ascending, we get a set of values $t_1 < t_2 < \ldots < t_k$.

Let's consider a graph with a set of vertices $V = S \cup T \cup \{u_1, u_2, \ldots, u_k\} \cup \{w_1, w_2, \ldots, w_n\}$ ($S$ will be the source, $T$ the sink, each $t_i$ has an associated vertex $u_i$, and each task has an associated vertex $w_i$). Add the following edges with the capacities stated below:

1. Edges $S \to u_i$ with capacity $m(t_{i+1} - t_i)$ for each $i \in [1, k-1]$.
2. Edges $u_i \to w_j$ with capacity $t_{i+1} - t_i$ for each $i \in [1, k-1]$ and each $j$ such that $[t_i, t_{i+1}] \in [s_j, e_j]$.
3. Edges $w_i \to T$ with capacity $d_i$ for each $i \in [1, n]$.

The task can be performed if and only if the network has a flow of $value = \sum d_i$. From a flow of this value, we can construct a way to perform all the tasks.