The 2023 ICPC Caribbean Finals Qualifier

Editorial

Problem set developers

Anier Velasco Sotomayor – Harbour Space University
Humberto Yusta Gómez – Harbour Space University
Carlos Joa Fong – Altice Dominicana
Marcelo J. Fornet Fornés – ICPC Caribbean
Alben Luis Urquiza Rojas – Universidad de Las Tunas
Ernesto Teruel Velazco – Universidad de las Ciencias Informáticas
Rubén Alcolea Núñez – Leil Storage

Matcom Online Grader

Leandro Castillo Valdés – ICPC Caribbean

September 27th, 2023

2023 ICPC Qualifier Real Contest Problem Set

A Array Distribution

Author: Carlos Joa Fong – Altice Dominicana
Category: Dynamic Programming, Number Theory 3
Solved by 2 teams

First, we observe that:

1. For any positive integer x, there is a unique representation of x as a product of its prime factors (disregarding the order of the prime factors).
2. Since the products of the numbers may be too large to compare efficiently, let's compare their prime factors.
3. Instead of comparing individual prime factors, we will compare their counts: two integers are equal if they have the same count of 2s, 3s, 5s, 7s, 11s, etc., in their prime factorization.
4. The integers in the input array are up to 10, so we only need to consider primes 2, 3, 5, and 7.

Let's do case analysis of each possible integer value that may appear in input array:

- $4 = 2 \times 2$, so a 4 contributes two 2s.
- $8 = 2 \times 2 \times 2$, so an 8 contributes three 2s.
- $9 = 3 \times 3$, so a 9 contributes two 3s.
- 2, 3, 5, and 7 contributes a single instance of themselves
- $6 = 2 \times 3$ and $10 = 2 \times 5$ have two distinct prime factors and they contribute one instance of each of their two prime factors
- 1s do not change the product, so we may distribute them in any way we wish (read below for an exception to this rule).
- If the input consists of only 1s, you must assign at least one 1 to both groups (since the two groups must not be empty).
- If we tally up the counts of each of the primes and there is a prime with and odd count, there is no solution,
- Since prime 7 is never mixed with other primes, we split their instances in two halves: one half goes into group G_0 and the other one into group G_1.

The official solution brute forces the number of 6s and 10s to distribute into each group. Once we decide the number of 6s and 10s of each group, we compute the number of 2s, 3s and 5s (of the remaining numbers) that must be in group G_0 and check whether it is possible to obtain these counts:

- The case of 5s is easy, since they only appear *alone* (after deciding how to split the 10s).
- For the 2s, let's create an instance of the subset sum problem: is there a way to achieve a sum of x by selecting a certain number of 1s, 2s, and 3s (which correspond to the available counts of 2s, 4s, and 8s respectively, in the input array).
- Likewise for the case of 3s: create an instance of the subset sum problem to determine if it is possible to get a sum of x by choosing certain number of 1 and 2 (corresponding to number of 3s and 9s in the input array).

Notice that the input sequence for the subset sum problem is *fixed* per input array A, so we only need to compute it once and answer a *query* to the subset sum problem for a particular value of x in $O(1)$. Therefore, the total complexity is the cost of computing subset sum + the cost to brute force all possible ways to split 6s and 10s: $O(MAXSUM \times N + N^2)$ where $MAXSUM$ is the maximum sum of the subset sum problem. For this problem, $MAXSUM = 3 \times N$.

B Broken Watch

Author: Ernesto Teruel Velazco – UCI
Category: Ad-Hoc 2
Solved by 69 teams

Let's find the time T until the first match occurs. For all other matches, we can consider the current position as second 0, and the process will repeat itself. Since the constraints are small, we can simulate the movement of the clocks. This can be proven to take no more than 60 steps to find a match. **Time Complexity:** $O(T)$.

Another observation to find T is to consider the clock that works well as stationary and the clock that works badly as moving K-1 units. We must calculate how long the broken clock takes to pass through the "second" 0, which would be $60/GCD(60, K-1)$. Be careful with the case $K = 0$.

C Cutting Rebar

Author: Alben Luis Urquiza Rojas - Universidad de Las Tunas
Category: Combinations, Dynamic Programming 4
Solved by 5 teams

Observation: since the number of pieces that need to be cut is ≤ 18, it is possible to have an exponential time solution like bitmask.

For the solution we will use Dynamic Programming with Bitmask. For each mask, the active bits represent the pieces that have already been cut. So for each different bitmask, we keep track of the minimum number of rods used to cut those pieces and how much we have cut from the last used rod. The solution will be the bitmask with all bits active.

Time Complexity: $O(2^C \cdot C)$, such that $C = \Sigma_{i=1}^{N} C_i$, and $(1 \leq C_i \leq N)$.

D Dragonslayers

Author: Carlos Joa Fong – Altice Dominicana
Category: Greedy, Brute Force 1
Solved by 90 teams

Because there are only three knights and three dragons, we may solve the problem by brute force: generate all possible matchups of dragons to knights and for each one, check whether at least two knights beat two of their assigned dragons. This solution is accepted because there are only 6 possible matchups.

Another solution is to greedily match the weakest dragon to your second strongest knight and the second weakest dragon to your strongest knight. In this matchup, your weakest knight fights the strongest dragon (in a possibly hopeless fight), but the other two knights are matched optimally to the weakest two dragons: if they both beat their respective dragons, the answer is "POSSIBLE" regardless of the outcome in the battle of your weakest knight. Otherwise, the answer is "IMPOSSIBLE" because, in this case, your weakest knight will definitely lose to the strongest dragon.

E Stolen Table

Author: Anier Velasco Sotomayor – Harbour Space University
Category: Combinations, Inclusion-Exclusion Principle 5
Solved by 0 teams

To solve the problem, let's use the inclusion-exclusion principle. First, let's simplify the problem. Suppose we need not the condition of equality of the maximum value in a row or column, but the inequality: $max \leq a_i$ or $max \leq b_i$. In other words, the maximum in a row or column must not exceed some constant. In this case, for each cell (i, j), we can find the maximum possible value it can take, which is equal to $min(a_i, b_j)$. So, for cell (i, j), there are $min(a_i, b_j)$ possible values. Now, the number of tables that satisfy the simplified condition is: $\prod_{i=1}^{n} \prod_{j=1}^{m} min(a_i, b_j)$, let's call this the initial answer for convenience.

Now, we need to use the inclusion-exclusion principle: let's consider in which rows the initial condition is NOT met and in which columns it is NOT met. Let's say we choose x rows and y columns for which the condition is guaranteed NOT to be met. It's not difficult to **almost correctly** calculate all such tables: first, subtract one from $a_i (a'_i = a_i - 1)$ if this row is in the selected set (otherwise, leave it unchanged), do the same for columns. As a result, using the same formula as above, we can calculate all such tables: $\prod_{i=1}^{n} \prod_{j=1}^{m} min(a'_i, b'_j)$. Unfortunately, it's possible that the condition is NOT met for more than x rows and y columns. Fortunately, this can be used for the "inclusion-exclusion" technique: consider all possible subsets of rows and columns, calculate the answer for them, and add it to or subtract it from the answer depending on the parity of the size of the subset $(x + y)$. This is enough to write a slow solution.

To optimize for time, let's get rid of inclusion-exclusion for columns. Let's introduce a function: $f(b_j)$. It will calculate the answer for column j if the maximum in it is not greater than b_j. $f(b_j) = \prod_{i=1}^{n} min(a'_i, b_j)$, then we can expand $min(a'_i, b_j)$ and write it separately for cases when $a'_i \le b_j$ and $a'_i > b_j$. For convenience, we can sort the array a'. Then $f(b_j) = b_j^{n-k} \cdot \prod_{i=1}^{k} a'_i$ (k is the length of the maximum prefix of the a' array where values are less than or equal to b_j). In this case, the answer for the column is: $f(b_j) - f(b_j - 1)$. Thus, we have eliminated inclusion-exclusion for columns, and now we can calculate the answer using inclusion-exclusion ONLY for rows. But this solution is still slow.

For further optimization, let's notice how changing one element in the array a affects the function f. To do this, let's consider two cases. If $a_i > b_j$, then the value of the function $f(b_j)$ does not change after decreasing a_i by one, otherwise the value of the function is multiplied by $\frac{a_i - 1}{a_i}$. But, since the answer for the column is $f(b_j) - f(b_j - 1)$, in the end, there will be three options for how the answer for the column will change.

- $a_i > b_j$: nothing changes.
- $a_i = b_j$: $f(b_j) - f(b_j - 1) \to \frac{a_i - 1}{a_i} \cdot f(b_j) - f(b_j - 1)$.
- $a_i < b_j$: $f(b_j) - f(b_j - 1) \to \frac{a_i - 1}{a_i} \cdot [f(b_j) - f(b_j - 1)]$.

Let's introduce another function $P(x, k)$. This function calculates the answer in the case where we selected k rows with a_i equal to x and decreased them by one. To calculate it, we need to know the number of values b_i that are strictly greater than a_i (let it be bg) and equal to a_i (let it be eq). It can be noticed that the initial answer should be multiplied by:

$$\left(\frac{x-1}{x}\right)^{k \cdot bg} \cdot \left(\frac{\left(\frac{x-1}{x}\right)^{k} \cdot f(x) - f(x-1)}{f(x) - f(x-1)}\right)^{eq}$$

Now we can understand that these changes are independent for rows with different values of a_i, and to account for changes in several different values of a_i, it is sufficient to multiply the functions P. Then we can use inclusion-exclusion for one set of rows, more formally, the change in the answer for a specific value x looks as follows:

$$\sum_{k=0}^{cnt}(-1)^k \cdot P(x,k) \cdot \binom{cnt}{k}$$

In the end, it's enough to find the product of these sums for all values in the array a and multiply it by the initial answer.

P.S. In all formulas, calculations are performed modulo $10^9 + 7$. Fermat's little theorem is used for divisions and fractions. Also, by changing the answer, we mean the ratio, not the difference.

F Fibonacci Squared

Author: Rubén Alcolea Núñez - Leil Storage
Category: Arithmetic-Algebra 3
Solved by 18 teams

In this problem, you need to compute the nth term of the following modified Fibonacci sequence: $F_i = F_{i-1} + F_{i-2} + 2 \cdot i^2 + 5$.

The problem of computing the nth term of the original Fibonacci sequence can be solved using linear recurrences and matrix exponentiation in $O(\log n)$. We can express the recurrence in matrix notation. In this way, it's possible to transform the original recurrence in (1) with the form $B = M \times A$, where,

- B is a column vector that contains the term F_n,
- M is a constant matrix with the coefficients that satisfy the equation $B = M \times A$, and
- A is a column vector with the base cases of the recurrence.

$$\begin{bmatrix} f_{n+1} \\ f_n \\ (n+1)^2 \\ n+1 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 2 & 0 & 5 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 2 & 1 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} f_n \\ f_{n-1} \\ n^2 \\ n \\ 1 \end{bmatrix} \quad (1)$$

The term f_{n+k} of the sequence can be computed multiplying k times both sides of equation (1) by the matrix M, and then multiplying by the column vector A (2).

$$\begin{bmatrix} f_{n+k} \\ f_{n+k-1} \\ (n+k)^2 \\ n+k \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 2 & 0 & 5 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 2 & 1 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}^k \times \begin{bmatrix} f_n \\ f_{n-1} \\ n^2 \\ n \\ 1 \end{bmatrix} \tag{2}$$

Consequently, the final value f_n can be obtained by raising the matrix M to the power of $n-1$ $(n-2+1)$ and multiplying the result by the column vector A with the base cases $f_1 = 1$, $f_0 = 0$, $n^2 = 4$, $n = 2$ and 1.

After replacing the matrices and column vectors by their values, we get the final equation (3):

$$\begin{bmatrix} f_n \\ f_{n-1} \\ n^2 \\ n \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 2 & 0 & 5 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 2 & 1 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}^{n-1} \times \begin{bmatrix} 1 \\ 0 \\ 4 \\ 2 \\ 1 \end{bmatrix} \tag{3}$$

The final complexity to compute the value f_n is $O(\log n)$ using binary exponentiation to compute the power of the matrix M. The final complexity is $O(q \log n)$ because there are q terms to be computed.

G Good Subarrays

Author: Humberto Yusta Gómez - Harbour Space University
Category: Ad Hoc 4
Solved by 15 teams

Let's analyze the following algorithm. We start a divide and conquer approach from 1 to n, searching for the maximum. Starting from this maximum, we expand in both directions as long as it remains greater than the others, maintaining the number of even and odd values and checking in each case.

Then, we split and solve the problem recursively from 1 to $k-1$, and from $k+1$ to n where k is the position of the maximum element.

After finishing this algorithm, we would have counted all good subarrays, since we start expanding from the maximum in each time, as long as it is the maximum, so all good subarrays will be considered in this approach.

If we have an array of length n, with k being the position of the maximum, the algorithm will take $\min(k, n-k)$ steps to expand, and will divide the problem into subproblems of size $k-1$ and $n-k-1$. The worst-case scenario is when it splits in half each time, resulting in a complexity of $O(n \log n)$. This can be verified by running a dynamic programming approach that calculates the complexity, or through other mathematical means.

Considering that this algorithm is fast enough, an even simpler implementation is possible. For each element, we expand as long as it remains the largest in the range, maintaining the number of even and odd values, and counting in each case, this will consider exactly the same number of subarrays than the previous approach, hence solving the problem in $O(n \log n)$.

H Hidden Sequence

Author: Anier Velasco Sotomayor – Harbour Space University
Category: Data Structures 3
Solved by 4 teams

Let's first forget about the queries and solve the problem just for the complete array. Let's assume we have an array $c_1, \ldots, c_k$. Let's denote the value of c_1 as x. Then, we can observe that all the other values c_i can be uniquely determined through x:

- $c_1 = x$;
- $b_1 = c_1 + c_2 \to c_2 = b_1 - x$;
- $b_2 = c_2 + c_3 \to c_3 = b_2 - b_1 + x$;
- $\ldots$
- $c_k = b_{k-1} - b_{k-2} + \ldots \mp b_1 \pm x$.

Notice that in the values $c_1, c_3, \ldots$, our first number x appears with a plus sign, while in the values $c_2, c_4, \ldots$, it appears with a minus sign. Each odd value can be represented as $(x + d_i)$, and each even value as $(-x + d_i)$. Since the only thing we can control is the value of x, we can always choose a large enough x so that odd and even numbers have no common elements (the former ones will be positive and the latter ones will be negative). Since x will not affect the answer beyond that point ($x + p = x + q \iff p = q$), we can simply set a very large number for x (for example, 10^{18}) and reconstruct the entire array c using the method described above. After that, the answer to the problem will simply be the number of distinct numbers in c.

Great! Now we know how to solve the problem for the entire array. But what about intervals? The same approach applies! Let's set $c_0 = 10^{18}$, reconstruct all values $c_1, \ldots, c_n$, and the answer to the query (l, r) will be the number of distinct numbers in the interval $[l-1, r]$. This becomes a standard problem that can be solved using a segment tree or Mo's algorithm.

The final time complexity is $O((n+q)\log n)$ with the best implementation.

I Sorting by ASCII

Author: Rubén Alcolea Núñez - Leil Storage
Category: Sorting 2
Solved by 119 teams

This problem is considered the second one easiest of the contest. It's intended to be solved by most of the teams.

The solution implies to sort the strings by the sum of ASCII values of its letters. The only case that needs to be validated is when two strings have the same sum of ASCII values. In that case, the tie is solved by the least index of each string at the original list. This case can be solved using a stable sorting algorithm. The final complexity is $O(n \log n)$.

J Juan and Odd Numbers

Author: Alben Luiz Urquiza Rojas - Universidad de Las Tunas
Category: Arithmetic-Algebra 1
Solved by 155 teams

The odd numbers repeat every 5 odd numbers with a sum of 25.

Solution: $N/5 \times 25$ + the sum of the remaining odd numbers if there are less than 5.

Be careful with overflow... long long ;)

Expected complexity: $O(1)$.

K Good Numbers

Author: Anier Velasco Sotomayor - Harbour Space University
Category: Arithmetics-Algebra, Number Theory, Brute Force 2
Solved by 64 teams

Lemma: If a number x is good, then any multiple of x is good.

Proof: Let x be any good number, and $x \cdot n$ any multiple of it. Let $d_1, d_2, \ldots, d_k$ be the divisors of x. Then, because x is good, $x < d_1 + d_2 + \cdots + d_k$, and by multiplying by n on both sides, we get $x \cdot n < d_1 \cdot n + d_2 \cdot n + \cdots + d_k \cdot n$. QED.

We can clearly see that 12 is a good number, therefore each multiple of 12 is one as well, and that means that we can find a good number in any interval of length at least 12.

Then, we can brute force among the first 12 numbers and the last 12 numbers of the range. For each of those numbers x, we find its divisors in $\mathcal{O}(\sqrt{x})$ time.

Note: As a fact, each multiple of 6 is good except for 6 itself. Hence, in most cases less than 11 numbers are checked in total.

L Distance Graph

Author: Humberto Yusta Gómez - Harbour Space University
Category: Greedy 4
Solved by 1 teams

Let's denote $layer_i$ as the set of vertices that are at a distance i from vertex 1. Let's also denote cnt_i as the number of vertices in $layer_i$.

We can make all possible edges between vertices of adjacent layers. Note that we can not make edges between vertices of the same layer because of the problem restriction, and we can not make edges between vertices of non adjacent layers, since all edges have weight 1 and it would make a shorter path.

So, if cnt_i is fixed for all $1 \leq i \leq k$, then the answer is $\sum_{i=0}^{k-1} cnt_i \cdot cnt_{i+1}$, that is, make all possible edges between vertices at adjacent layers.

Note that, in order to have vertices at distance a from vertex 1, we need at least one vertex at distance $1, 2, ..., a-1$, so $cnt_i \geq 1$ should hold for all $1 \leq i < k$.

If $\sum_{i=0}^{k} \max(a_i, 1) > n$, there is not enough vertices to make a graph that satisfies the constraints. Otherwise, the problem can be reduced to assign $n - \sum_{i=0}^{k} \max(a_i, 1)$ vertices to layers, such that $\sum_{i=0}^{k-1} cnt_i \cdot cnt_{i+1}$ is maximized. In other words, assign free vertices into layers, so that the total number of edges, which is the multiplication of the sizes of each pair of consecutive layers is maximized.

If we add one vertex to $layer_i$, the answer increases by $cnt_{i-1} + cnt_{i+1}$, since we can make all possible edges to vertices of adjacent layers, after this operation, cnt_i will increase by 1.

Let's define $v_i = cnt_{i-1} + cnt_{i+1}$, the previous operation is equivalent to add v_i to the answer and increase v_{i-1} and v_{i+1} by 1.

From this, we can see that it will not be optimal to add vertices to more than two consecutive layers, since a layer only increases its adjacent vertices, and its adjacent vertices also increase it.

Then we can consider for every pair of adjacent layers, adding all free vertices to those two layers, and get the maximum answer.

If we are considering layers x and $x + 1$, we will greedily choose one by one where to put the next free vertex. So at any point, if $v_x \geq v_{x+1}$ we add it to $layer_x$, otherwise to $layer_{x+1}$, adding v_x to the answer and increasing v_{x+1} by 1 (or viceversa). This process can be optimized with summation formulas to do it in $O(1)$ for one pair of layers.

The time complexity of the solution is $O(k)$.

M Minimize the Greatest Value

Author: Ernesto Teruel Velazco – UCI
Category: Binary Search, Dynamic Programming, Greedy 3
Solved by 26 teams

This problem was meant to be solved using different approaches given the small constraints. Below are the solutions proposed by the author.

Solution 1: Binary Search + Brute Force

We can try to solve the problem: "Can you form at most K numbers with the digits, such that the biggest number is at most X?"

Notice that the monotony of the answer to the previous problem is preserved, so we can binary search over X. To solve the problem given a fixed X, we can try all possible permutations of the digits and split them into contiguous subarrays greedily. The time complexity of this solution is $O(N!)$.

Solution 2: Greedy

The greedy solution is based on the assumption that it's optimal to group the digits as evenly as possible. Otherwise, we can always improve the solution by taking one digit from the biggest number and moving it into a smaller one.

In general we can divide the problem in two cases, depending whether K divides N or not.

Case 1: If K divides N, we can create K numbers each one with $\frac{N}{K}$ digits. Let F_i the frequency of the lowest digit, we have two cases:

- Case 1: $F_i \geq K$, we can fill the first digit of each number with digit i. This will ensure that all the numbers are as small as possible.

- Case 2: $F_i < K$, we know that there will be F_i numbers that are smaller than the remaining ones. Set their first digit to i and fill them with the biggest digits that are not yet used. Subsequently, we can solve the problem by considering the remaining $K - F_i$ numbers.

Case 2: If K does not divide N, then we will have some numbers with $\frac{N}{K}$ digits and the remaining with $\frac{N}{K} - 1$ digits. In that case, the biggest digits will be assigned to the numbers with $\frac{N}{K} - 1$ digits and the problem for the numbers with $\frac{N}{K}$ digits will be solved like in the **Case 1**.

This process can be repeated until each digit is used and construct the solution on each step. The time complexity of this solution is $O(N)$.

Solution 3: Dynamic Programming

The dynamic programming solution involves the use of bitmasks. The first step is to sort the digits in non-increasing order to guarantee they are considered in the right order.

Now, let's define the state $\{mask, k\}$ to represent the solution with k numbers and using only the digits present at $mask$. Then, for each transition we will add a new mask and check among all the numbers we can create with the new mask and save the minimum value.

The value of the final solution will be at the state $\{2^n - 1, k\}$, representing the solution to create k numbers using all the digits given in the input. The final complexity of this solution is $O(3^n)$.