Gran Premio de México 2019

Primera Fecha

May 4th, 2019

Contest Solutions

This document contains expected solutions for the 10 problems used during the Contest session.

Problem A – Add and substract

Author: Juan Pablo Marín

First lets get the sum of all differences not considering the restriction of not adding differences if they are -1, 0 or 1.

Looking the formula presented in the problem statement: the kid with list number i will sum each of the following numbers $a_{i+1} - a_i$, $a_{i+2} - a_i$, ..., $a_N - a_i$.

It can be seen that the number the i-th kid has selected will be positive for the first $i-1$ kids and negative in the $N-i$ sums he does, so he contributes a total of $(i-1)*a_i-(N-i)*a_i = (i-1-N+i)a_i = (2*i-N-1)a_i$.

Then we can get the sum of all differences without the restriction calculating the sum:

$$\sum_{i=1}^{N}(2*i-N-1)a_i$$

Now, considering that we don't want to sum differences with values -1, 0 or 1 lets observe that:

$$a_i - a_j = 1 \text{ when } a_j = a_i - 1$$

$$a_i - a_j = -1 \text{ then } a_j = a_i + 1$$

For each ocurrence of the value $a_i - 1$ before a_i we have added 1 to sum, in the same way for each ocurrence of the value $a_i + 1$ we have added -1 to the sum. Then if we know how many times the values $a_i + 1$ and $a_i - 1$ appear before a_i in the list we can remove from the total sum the values we are restricted to use in the sum. So our answer is :

$$\sum_{i=1}^{N}(2*i-N-1)a_i + times(a_i+1, i) - times(a_i-1, i)$$

In the formula the function times(i,j) counts the ocurrences of the value i in the list before index j.

One way to implement times(i,j) is to traverse for all indexes between 1 and $j-1$ and count the number of indexes k where $a_k = i$. This approach has a O(N^2) complexity as we will need to traverse all the indexes for each value of i, wich is slow to get the solution accepted.

In order to improve the computation of times(i,j) we can see that the values the kids can choose are relatively small (between 1 and 10^6) then we can keep track of the times each value i has appeared before index j (times(i,j)) using an array as a table to count the ocurrences increasing the count for each value each time we see it, this approach takes O(1) to update the array and O(1) to query a value. The complexity of the algorithm is O(N) and will run in time to be accepted.

Problem B – Box delivery

Author: Moroni Silverio

Let's first solve the problem considering that the supervisor nor his boss will come any day to the new branch. In this case Jaime has K boxes and N days to bring all boxes to the new branch, this problem is the same as summing K with N terms, and is a problem that is solved with separator.

Lets suppose K = 7 and N = 3 We can add $N-1$ to choose the number of boxes Jaime delivers on each day for example.

```
* *  | * *  | * * *            x_1 = 2, x_2 = 2, x_3 = 3
 x_1 |  x_2 |    x_3
```

Then the problem in this case is to count in how many different ways we can select $N-1$ elements from a set of $K+N-1$ elements and the answer is $\frac{(K+N-1)!}{(K!)(N-1)!}$

We know that each day the supervisor comes, Jaime needs to bring at least one box. Let's suppose the supervisor comes only one day, then, we know Jaime needs to bring one box that day, but he can bring the $K-1$ boxes left in any way he wants in the N, so the problem is the same as the one described above but with the $K-1$ boxes. The same case applies when the boss comes, Jaime needs to bring two boxes that day and we need to count the number of ways Jaime can put the $K-2$ boxes in the N days. The days both the boss and the supervisor comes is the same as when only the boss comes as he already satisfies the contraint given by the supervisor.

Suppose $K' = K - S - 2 * B + T$ where K is the initial number of boxes, S the number of days the supervisor will come, B the number of days the boss will come, and T the number of days both the suppervisor and the boss comes. Then there are $\frac{(K'+N-1)!}{(K'!)(N-1)!}$

To implement this solution is required to consider modular arithmetic in order to deal with the division and the modulus calculation.

Problem C – Connecting cities

Author: Moroni Silverio

This is a graph problem. It is easy to see that vertices in the graph are the cities and roads are the edges.

The task can be expressed in the following way: Given two graphs using the same set of vertices V, $G_1 = (V, E_1)$ and $G_2 = (V, E_2)$ What is the minimum cost of selecting a set $E_x \subseteq E2$ such that the graph $G_x = (V, E_1 \bigcup E_x)$ is connected (i.e it has only one connected component). The cost is the sum of costs building each road in E_x.

To find a solution, we first build the Graph G_1 from the input. If the graph is connected then the set E_x is empty and the answer is : "Thank you, Goodbye".

If G_1 is not connected then it has more than one connected components. The minimal cost for E_x can be determined finding the minimum spanning tree that joins each of the connected components of G_1 as if they were a vertex.

After running kruskal algorithm there are two cases:

- The resulting graph contains more than one connected components, in which case no solution exists and the answer is : "You better hire someone else".
- The resulting graph contains only one connected component, in which case the ouput is the sum of costs of the edges selected by Kruskal algorithm.

Problem D – Determining rally paths

Author: Moroni Silverio

First observation is to note that since all stations are connected with exactly $V-1$ edges then the rally is a tree.

A trivial solution is to traverse the path for each team and concatenate the codes of the vertices visited in the path that exists between E_s and E_f in the order they appear. If we concatenate the code C_x to another code C we get the code $C * 10^{d(C_x)} + C_x$ Where d(C) is the number of digits in C. Doing this for each queried team will not run in time, the function to concatenate a code can be implemented then in O(1) so we need O(V) per query and will have a total running time of O(EV) wich is slow for the problem time limit.

We can divide the path from E_s to E_f in two paths, one that goes from E_s to $LCA(E_s, E_f)$ and one that goes from $LCA(E_s, E_f)$ to E_f, where $LCA(E_s, E_f)$ is the lowest common ancestor for stations E_s and E_f. Lets say $Code(u, v)$ is the concatenation of all codes in the path from station u to station v, then, we can see $Code(E_s, E_f)$ can be obtained considering three paths in the tree:

1. The path from E_s to the root r of the tree.
2. The path from the $LCA(E_s, E_f)$ to the root r of the tree.
3. The path from the $LCA(E_s, E_f)$ to the station E_f.

If we remove the code from the second path to the code of the first path and concatenate the code of the third path we obtain $Code(E_s, E_f)$. To remove the second path from the first path notice that:

$$Code(E_s, r) = Code(E_s, H(E_s, LCA(E_s, E_f))) * 10^{d(Code(LCA(E_s,E_f),r))} + Code(LCA(E_s, E_f), r)$$

Where $H(E_s, LCA(E_s, E_f))$ is the last visited station before reaching $LCA(E_s, E_f)$ in the path from E_s to $LCA(E_s, E_f)$, then

$$Code(E_s, H(E_s, LCA(E_s, E_f))) = \frac{Code(E_s,r) - Code(LCA(E_s,E_f),r)}{10^{d(Code(LCA(E_s,E_f),r))}}$$

Notice that the code of the third path is $Code(LCA(E_s, E_f), E_f)$ to obtain this code we can remove $Code(r, P(LCA(E_s, E_f), E_f))$ to $Code(r, E_f)$ similar as we did before:

$$Code(r, E_f) = Code(r, P(LCA(E_s, E_f))) * 10^{d(Code(LCA(E_s,E_f)),E_f)} + Code(LCA(E_s, E_f), E_f)$$

Where $P(LCA(E_s, E_f))$ is the parent of $LCA(E_s, E_f)$ in the tree, then :

$$Code(LCA(E_s, E_f), E_f) = Code(r, E_f) - Code(r, P(LCA(E_s, E_f))) * 10^{d(Code(LCA(E_s,E_f)))}$$

Notice that the only variable we don't know here is $d(Code(LCA(E_s, E_f)))$ but it can be seen that when we remove a code C_s from the code C we also remove $d(C_s)$ digits from it, then $d(Code(LCA(E_s, E_f))) = d(Code(r, E_f)) - d(Code(r, P(LCA(E_s, E_f))))$

Finally as stated before concatenating $Code(LCA(E_s, E_f), E_f)$ to $Code(E_s, H(E_s, LCA(E_s, E_f)))$ will give us $Code(E_s, E_f)$, then:

$$Code(E_s, E_f) = Code(E_s, H(E_s, LCA(E_s, E_f))) * 10^{d(Code(LCA(E_s,E_f),E_f))} + Code(LCA(E_s, E_f)$$

Note that in order to properly implement this solution it is required to perform all calculations using modular arithmetic as the codes can grow very fast, to do this, notice that there are a division involved so it will be required to multiply using the modular multiplicative inverse with respect to the given modulo instead of dividing.

Problem E – Egyptian binary system

Author: Juan Pablo Marín

The first observation to make is that a number is odd if its last binary digit is 1. We are then interested in counting the number of substrings that the last character is '1'. Also, as there shouldn't be leading '0's (we know this from the statement : all what scientists know is that the data never started with an "eye") the string should start with '1' as well.

Based on the previous observation we can see that for any pair of positions (i, j) if $Si = Sj = 1$ then the substring of S that starts in position i and finishes at j is the binary representation of an odd number. We need to count such pairs.

A way to do this is to keep two counters one for i and one for j for all values of i found in the range $[0, |S|)$ if $Si == 1$ then traverse j in the range $[i, |S|)$ and check if $Sj == 1$ then we found a pair and our result increases by 1. Time complexity of this approach is O(N^2) where N is —S— and it is slow to be accepted in the given time limit.

A more efficient approach is to count how many 1s appear in S, let this number be K. Our answer then is the number of ways you can take a pair from a set of K elements allowing repetitions as the substring from S_i to S_i is an odd number. The solution is then : $\frac{(K)*(K+1)}{2}$ This approach takes O(N), which is enough to run in time .

Problem F – Forecasting rock-paper-scissors

Author: Jonathan Queiroz

Consider the simplified problem in which we only need to determine match results, and do not need to tell the earliest day in which this result could have been predicted. For each of player $x \in \{1, \ldots, N\}$, we implicitly maintain three sets:

- $\texttt{ABOVE}(x)$: the set of players which are known to win against player x;
- $\texttt{SAME}(x)$: the set of players which are known to draw against player x; and
- $\texttt{BELOW}(x)$: the set of players which are known to lose against player x.

These sets may be efficiently maintained using a disjoint-set union (DSU) data structure. At the beginning of the tournament, before any matches take place, $\texttt{SAME}(x) = \{x\}$ and $\texttt{ABOVE}(x) = \texttt{BELOW}(x) = \emptyset$. Whenever we learn that player x draws with player y, the following operations need to be performed:

- Merge $\texttt{ABOVE}(x)$ and $\texttt{ABOVE}(y)$.
- Merge $\texttt{SAME}(x)$ and $\texttt{SAME}(y)$.
- Merge $\texttt{BELOW}(x)$ and $\texttt{BELOW}(y)$.

Similarly, whenever we learn that player x wins against player y, the following operations need to be performed:

- Merge $\texttt{SAME}(x)$ and $\texttt{ABOVE}(y)$.
- Merge $\texttt{BELOW}(x)$ and $\texttt{SAME}(y)$.
- Merge $\texttt{ABOVE}(x)$ and $\texttt{BELOW}(y)$.

Finally, whenever we learn that player x loses to player y, the following operations need to be performed:

- Merge $\texttt{SAME}(x)$ and $\texttt{BELOW}(y)$.
- Merge $\texttt{BELOW}(x)$ and $\texttt{ABOVE}(y)$.
- Merge $\texttt{ABOVE}(x)$ and $\texttt{SAME}(y)$.

Solving queries for the result of a match between players x and y may be done by consecutively checking whether $\texttt{SAME}(x) = \texttt{ABOVE}(y)$ (win for player x), $\texttt{SAME}(x) = \texttt{SAME}(y)$ (draw) and $\texttt{SAME}(x) = \texttt{BELOW}(y)$ (win for player y). If neither of these comparisons is true, the result cannot be determined. This solves the simplified problem.

For the original problem, we also need to determine the earliest day in which match results could have been established. The solution is to use a partially persistent DSU, which allows us to query past versions of the data structure rather than only the latest one. For example, using a partially persistent DSU with union-by-rank/union-by-size (path compression is not feasible), we may solve queries $\texttt{FIND}(x, y, t)$ of the form "did elements x and y belong to the same set after t merges were performed?" in $O(\log n)$ time. A binary search allows us to solve queries $\texttt{FIND-TIME}(x, y)$ of the form "when were elements x and y first merged to the same set?" in $O(\log^2 n)$ time. This in turn allows us to determine the first moment when player x was known to win/draw/lose against player y.

The key idea is to store the time when each merge operation took place. For implementation details, please refer to https://pastebin.com/CtiaZ7MB, which contains a solution running in $O(M \log N + Q \log^2 N)$ time. This is enough to get Accepted.

Alternatively, the operation $\texttt{FIND-TIME}(x, y)$ may be implemented in $O(\log n)$ time by skipping the binary search. This brings the running time down to $O((M + Q) \log N)$.

Problem G – Going to the world finals again

Author: Lina Rosales & Juan Felipe Baquero

We look for all the possible ways of representing X as the sum of consecutive numbers.

So we are looking for the different values of d such that $d+(d+1)+(d+2)+\cdots+(d+k-1)=X$ Where k is the number of days Baker will travel.

Notice that in order to exist a way for X to be represented as the sum of at least two consecutive numbers then X is a trapezoidal number. So the problem can be stated as: in how many ways can X be represented as a trapezoidal number? This value is known as the *politeness* of a number. For every x, the *politeness* of x equals the number of odd divisors of x that are greater than one. One way to see this is that if x has an odd divisor y. Then y consecutive integers centered on x/y have x as their sum.

If a representation has an odd number of terms, x/y is the middle term, while if it has an even number of terms and its minimum value is m it may be extended in a unique way to a longer sequence with the same sum and an odd number of terms, by including the $2m-1$ numbers $-(m-1)$, $-(m-2)$, ..., -1, 0, 1, ..., $m-2$, $m-1$. After this extension, again, x/y is the middle term. By this construction, the polite representations of a number and its odd divisors greater than one may be placed into a one-to-one correspondence, giving a bijective proof of the characterization of polite numbers and politeness.

Solution to the problem is the number of odd divisors of X - 1, since 1 is an odd divisor of X where $d=X$ and Baker wouldn't travel for at least two days.

The number of divisors of a number $X=p_0^{\alpha_0}*p_1^{\alpha_1}*...*p_k^{\alpha_k}=\prod_{i=0}^{k}(\alpha_i+1)$ In order to count only odd divisors we need to remove al factors 2 from X factorization which can be done dividing X by 2 until the resulting number is not a divisor of 2 or just omiting the α value for the factors of 2 in the factorization.

Problem H – Husbands association

Author: Jesus Alejandro Rizo

The first thing to note is that if we sort the tolerance each problem will take from the partner then we can simulate if a problem should be taken or not as described in both scenarios.

So, for the first scenario after the set is sorted in decreasing order, we decide to take the first problem if it's less than T, if we don't take it we continue with the next problem, if we take it then $T = T - prob_i$ and continue until $T = 1$ or after we iterated over all problems.

For the second scenario we apply the same as on the first one but with the list sorted in ascending order.

Time complexity is based on the sorting algorithm, any sorting algorithm that take O(NlogN) will work in time.

Problem I – Inspecting PIN numbers

Author: Juan Pablo Marín

The problem can be divided in two subproblems, one to compute all PINs for all the values possible in the range $[1, 10^5]$, the second problem is to use the computed data as input to answer the queries problem.

For the first part, let's observe that any value n that finishes in trailing zeros can be represented as:

$$n = s * 10^q = s * 2^q * 5^q$$

Where s is the value before the zeroes and k the number of zeroes n has at the end. If $n = K!$ then our pin is $smod 10^5$.

Observe from above formula, that one way to remove the zeroes from the factorial is to not add them to the number in the first place. If we know the values of p_k and q_k, being respectively the number of times 2 appears as a factor in $K!$ and the number of times 5 appears as a factor in the $K!$, then $s = 2^{(p_k - q_k)} * D(K)$ (note that in this formula we are only multiplying values and then we can carry the modulo without problems) where D(K) is the result of multiplying all values between 1 and K not taking any factor 2 nor factors 5.

To implement this part we can keep track of the values for p_k, q_k, and $D(K)$ for all values of K and using fast exponentiation methods compute the PIN values for all possible values of K in O(NlogN).

Once we have this part solved, we can use a succint data structure that properly encodes the data to answer each query in at most O(logN) per query. Some data structures that can solve this type of queries efficiently are wavelet trees and persistent segment trees.

Problem J – Jaimina party invitations

Author: Saraí Ramírez

This is a number theory problem.

It can be seen that in order to cut a sheet into r invitations, all with the same height and width, it is needed to cut the sheet height in a parts and the width in b parts such that $a \times b = r$.

Being $r = p^q$, the only possible ways to create the invitations satisfying all the constraints in the problems is when $a \times b = p^q$ and then $a = p^i$ and $b = p^{q-i}$ for all values of i in $[0, q]$; then there are only $q + 1$ different configurations to cut the sheets.

To sum the perimeters we know a valid configuration has the form $a = p^i$ and $b = p^{q-i}$, in this configuration each rectangle will have $\frac{m}{p^i}$ height and $\frac{n}{p^{q-i}}$ width. Its perimeter will be $2\frac{m}{p^i} + 2\frac{n}{p^{q-i}}$; and as we have p^q rectangles in this configuration the sum of the perimeter of all the invitations in this configuration will be :

$$p^q \left[2\left(\frac{m}{p^i} + \frac{n}{p^{q-i}}\right)\right] = 2\left[mp^{q-i} + np^i\right]$$

Now that we know how to sum the perimeters for a given configuration, lets sum all of them, we know i can take values between 0 and q then we need to sum:

$$\sum_{i=0}^{q} 2\left(mp^{q-i} + np^i\right) = 2\sum_{i=0}^{q}\left(mp^{q-i} + np^i\right) = 2\left(\sum_{i=0}^{q} mp^{q-i} + \sum_{i=0}^{q} np^i\right)$$

Let's focus in $\sum_{i=0}^{q} mp^{q-i}$:

$$\sum_{i=0}^{q} mp^{q-i} = m\sum_{i=0}^{q} p^{q-i} = m\left[p^q + p^{q-1} + p^{q-2} + \cdots + p^1 + p^0\right] = m\sum_{i=0}^{q} p^i$$

In the same way

$$\sum_{i=0}^{q} np^i = n\sum_{i=0}^{q} p^i$$

Substituting these values in our formula, the sum of all perimeters is

$$2\left(\sum_{i=0}^{q} mp^{q-i} + \sum_{i=0}^{q} np^i\right) = 2\left(m\sum_{i=0}^{q} p^i + n\sum_{i=0}^{q} p^i\right) = 2(m+n)\sum_{i=0}^{q} p^i$$

The problem is reduced to the sum of a geometric series which can be written as

$$2(m+n)\left(\frac{p^{q+1}-1}{p-1}\right)$$

There are two important things to consider in the solution, first, that based on possible values for the input it is required to use a fast exponentiation method, binary exponentiation can help. The second one is that we are dividing and obtaining a modulus, it is important to use the multiplicative inverse of $p - 1$ with respect to the modulus to multiply instead of divide.