CROATIAN OPEN COMPETITION IN INFORMATICS

3rd ROUND

SOLUTIONS

Simply imputing the numbers as string, reversing them and comparing does the trick.
Or half a dozen other possible solutions such as modular arithmetic, arrays or more esoteric solutions.

Necesarry skills:
string manipulation

Tags:
ad-hoc

This one is straightforward. All you need to know is how your chosen language rounds output.

Necesarry skills:
knowing your programming language IO routines

Tags:
ad-hoc

By simply counting the number of times each number appears and sorting by those values we arrive at the solution. There are numerous ways we can implement that solution. For start, we can simply sort the numbers in O(N log N) time complexity and for each comparison scan the entire sequence in O(N) complexity.

A better solution is to pre calculate all frequencies and use that data in sorting. Depending on implementation this can range from $O(N \log^2 N)$ to O(N log N).

Necesarry skills:

data structures

Tags:

data strucutres

First, note that greedy strategy finds the optimal solution. Each time a detector detects calls on location i, we will presume that the call is made between the leftmost and rightmost house possible. This means that all detectors between those two houses must detect at least one more call. Naive implementation of this solution can lead to O(N log N + N*C) time complexity and 50% of points.

Let's try to form a better solution. We start by sorting the detectors by position. We now start from the left most detector, and maintain a stack of current calls. Of course at the first detector we add C1 calls to the stack. We now process detectors one by one in order. If the number of calls detected by the current detector is greater then the number of calls on stack, we add more calls to the stack. If it is smaller, we reduce the number of stack. We can now solve the problem by counting the number of times we remove items from stack.

Necesarry skills:

dynamic programming, greedy algorithm

Tags:

greedy algorithm, dynamic programming

For now, forget about the problem we are solving.
Suppose we had a sequence of integers and an algorithm like this:

```
while there are different numbers in the sequence
     select any two different numbers from the sequence
        and erase them
```

For example if the sequence were:

1 2 3 1 2 3 2 3

the algorithm could have done this:

1 2 3 1 2 3 2 3 --> 3 1 2 3 2 3 -> 1 3 2 3 -> 3 3

Note that we could also end up with other sequences if we selected pairs in a different way.
Let candidate be the number that is left in a sequence (3 in the example above).
Let count be the number of numbers left in a sequence (2 in the example above).

The cool thing about the algorithm is that if there is a number that appears more than N/2 times in the sequence it must end up as a candidate no matter the way we select pairs. Intuitively, we don't have enough other numbers to kill all of the candidate numbers.

So we can choose our own way to select pairs. Let's do it recursively like this.
1) Split the sequence S in two halves L and R.
2) Run the recursive algorithm on sequence L to get L.candidate and L.count
3) Run the recursive algorithm on sequence R to get R.candidate and R.count

4) Kill the remaining pairs among L.count + R.count numbers that are left.

The step #4 can be done very efficiently like this:

```
if L.candidate == R.candidate
  S.candidate = L.candidate
  S.count = L.count + R.count
else
  if L.count > R.count
    S.candidate = L.candidate
    S.count = L.count - R.count
  else
    S.candidate = R.candidate
    S.count = R.count - L.count
  end
end
```

So, we can use this idea to build the interval tree, every node containing info (candidate and count) about the subsequence it represents.

We can also query the interval tree to get candidate number for any interval [A, B]. Then we can use binary search to count the number of appearances of the candidate number in the interval and determine if the picture is pretty or not.

Necesarry skills:
advanced data structures, divide and conquer

Tags:
advanced data structures, divide and conquer

With X | Y we denote that X divides Y, ie. there exists k such that k*X = Y.

Let us denote with X_1, X_2, .., X_m duration of events in days. Note that saying:

- "Between dates A and B there were α_1 events 1, α_2 events 2, etc."

is the same as saying:

- "$365 \mid (\alpha_1 X_1+\alpha_2 X_2+...+\alpha_m X_m-(B-A))$".

where B-A is the difference in day between dates A and B. Further, we can split that into:

- "$5 \mid (\alpha_1 X_1+\alpha_2 X_2+...+\alpha_m X_m+A-B)$"
- "$73 \mid (\alpha_1 X_1+\alpha_2 X_2+...+\alpha_m X_m+A-B)$"

(note that 365=5*73). Since 5 and 73 are prime numbers, using Gaussian elimination we can find all possible remainders of X-es when divided by 5 and 73 (separately). Using Chinese remainder theorem we can further determine the remainder of each X when divided by 365.

Literature:

- http://en.wikipedia.org/wiki/Gaussian_elimination
- http://en.wikipedia.org/wiki/Modular_arithmetic
- http://en.wikipedia.org/wiki/Chinese_remainder_theorem

Necesarry skills:
modular arithmetics, Gaussian theorem of elimination, modular inverse

Tags:
discrete mathematics