Fall 23 ICPC Selection - Solution Sketches

(taken from editorials of original problems)

Problem A

(Source: 2022 ICPC Asia-Manila Regional Contest Problem G)

- Note that it is always optimal to choose an s such that $0 \leq s < k$.
- Here, choosing $s = 0$ is equivalent to choosing $s = k$.
- Create an array `buyable` of length k, initialized to all 0.
- We want `buyable[s]` to count the number of purchasable items if we ask the merchant to first appear on this day s.

- Process each item. This item is purchasable after choosing some starting s if and only if there exists a value $v \in [L, R]$ such that $v \bmod k = s$.
- If $R - L + 1 \geq k$, then this item is actually always purchasable, so we apply `buyable[s] += 1` for all s in $[0, k - 1]$
- If $R - L + 1 < k$, then we apply `buyable[s] += 1` for each s in $[L \bmod k, R \bmod k]$.
  - Note that in the case that $L \bmod k > R \bmod k$, this should be broken down into the *two* ranges $[0, R \bmod k]$ and $[L \bmod k, k - 1]$.

- The answer is `max(buyable)`
- This could be solved with a difference array, but that takes $\mathcal{O}(k)$ time and memory, which is too much
- Instead, you can *simulate* a difference array using a `map` or `dict`, keeping track of only the "interesting" points.
- Note that this is also basically just doing a line sweep.
- Running time: $\mathcal{O}(n \lg n)$

Problem B

(Source: 2022 ICPC Asia-Manila Regional Contest Problem F)

- Just implement the process :)
- Helpful observation: Once the civilization is in hegemony, it never escapes it.
- We can implement *killing max* and *killing min* separately and just switch over when hegemony is reached.
- Find the wealthiest nation using a balanced binary search tree or priority queue, which stores (wealth, index) pairs
- Update elements: delete and re-insert, or delete and let the old values be "staled"
- Do something similar to find the least wealth nation, once hegemony is reached
- Not a hassle at all if you know how to pass a custom comparator into your priority queue :))
- How to find the nearest neighbors to the west and east, while accounting for deletions?
- One easy way: Just use pointers!
- Have each faction maintain a pointer to its left and right neighbors, and update these accordingly whenever a faction collapses
- Running time: $\mathcal{O}(n \lg n)$

Problem C

(Source: Anton Trygub Contest 1 (The 1st Universal Cup, Stage 4: Ukraine) Problem F)

We will show that the answer is always 2 or 3. Let's initially color nodes of the first part (from 1 to n) in color 1 and the remaining in color 2.

It's clear that if we don't add any edges, the graph remains 2-colorable (and therefore 3-colorable).

If we add one edge (u, v), then just color node v in color 3, coloring will remain proper.

Now, suppose that we added two edges, and the graph is no longer 3-colorable. Let's consider some cases. Suppose that one of these two edges connects two nodes from different parts. Then, if the other edge is (u, v), we can just color node v in color 3, and the coloring will remain proper. So, we have to add edges, which connect nodes from the same part.

Let these edges be (u_1, v_1) and (u_2, v_2). If these edges share a node, say, $u_1 = u_2$, then we can just color node u_1 in color 3, and the coloring will be proper. Otherwise, let's try coloring some endpoint of each of these two edges in color 3. If we can't get proper coloring this way, it means that all of the edges $(u_1, u_2), (u_1, v_2), (v_1, u_2), (v_1, v_2)$ are present in this graph. And indeed, if such a cycle of length 4 is present in the graph, we can add edges (u_1, v_1) and (u_2, v_2), obtaining a subgraph on 4 nodes u_1, v_1, u_2, v_2, in which each pair of nodes is connected, and which, therefore, isn't 3-colorable.

Even if there is no 4-cycle, we can always add at most 3 edges to make the graph not 3-colorable. It's enough to show that we can choose 2 nodes u_1, v_1 from the first part, and nodes u_2, v_2 from the second part, such that at least 3 edges among edges $(u_1, u_2), (u_1, v_2), (v_1, u_2), (v_1, v_2)$ are present in this graph (then we can add all remaining edges to obtain a complete graph on 4 nodes). Suppose that it doesn't exist. Then there is no edge (u, v) with u in the first part, v in the second, such that there is at least one more edge from u and at least one more edge from v. Then for any edge (u, v), at least one of u, v is a leaf. Then consider any non-leaf node u, it can be connected only to leaves, so u together with these leaves forms a separate connected component, and the graph is not connected. This contradicts the problem statement (graph is told to be connected).

So, answer is 2 if there is a 4-cycle u_1, v_2, v_1, u_2, and 3 otherwise. How to check if there is a 4-cycle?

Here is an algorithm that, for a given node, checks if it's in some 4-cycle in $O(n)$, giving total runtime of $O(n^2)$. Consider node v, and it's neighbors $u_1, u_2, \ldots, u_k$. v is contained in a 4-cycle iff there is a node $v_1 \neq v$, which is connected to at least two nodes among $u_1, u_2, \ldots, u_k$.

Then, just start going through all neighbors of nodes $u_1, u_2, \ldots, u_k$, marking nodes which we meet (except v). If we have to mark some node twice, we found a 4-cycle. This takes $O(n)$ per node.

Problem D

(Source: Codeforces Round 409 (rated, Div. 2, based on VK Cup 2017 Round 2) Problem B)

First, let's check for impossible cases. If there exists a position i such that the i-th character of x is less than the i-th character of y, then it is impossible.

Now, let's assume it's always possible and construct an answer. We can notice that each position is independent, so let's simplify the problem to given two characters a and b with $a >= b$, find a character c such that $min(a, c) = b$. Here, we can choose $c = b$. So, for the original problem, if the answer is possible, we can print y.

Problem E

(source: 2020-2021 ICPC East Central North America Regional Contest (ECNA 2020) Problem J)

It's an implementation problem! Implement the rules and repeat until you can't apply them.

Problem F

(Source: 2011 ACM-ICPC East Central North America (ECNA 2011) Problem I)

```cpp
#include <iostream>
#include <cmath>
#include <stdio.h>
using namespace std;

double dist(double x1, double y1, double x2, double y2)
{
    double dx = x1-x2;
    double dy = y1-y2;
    return sqrt(dx*dx+dy*dy);
}
```

```cpp
bool between(double a, double b, double c)
{
    if (b < c)
        return (b<a && a<c);
    else
        return (c<a && a<b);
}

int main()
{
    double x1, y1, x2, y2, wx1, wy1, wx2, wy2;
    double ans;
    for(int icase=1; ; icase++) {
        cin >> x1 >> y1 >> x2 >>y2;
    if(x1 == 0 && y1 == 0 && x2 == 0 && y2 == 0)
      break;
        cin >> wx1 >> wy1 >> wx2 >>wy2;
        if (wx1 == wx2) {
            if (x1 == x2 || !between(wx1, x1, x2))
                ans = dist(x1, y1, x2, y2);
            else {
                double yint = y1 + (y2-y1)*(wx1-x1)/(x2-x1);
                if (between(yint, wy1, wy2)) {
                    double dist1 = dist(x1,y1,wx1,wy1)+dist(wx1,wy1,x2,y2);
                    double dist2 = dist(x1,y1,wx2,wy2)+dist(wx2,wy2,x2,y2);
                    ans = (dist1<dist2) ? dist1 : dist2;
                }
                else
                    ans = dist(x1, y1, x2, y2);
            }
        }
        else {
            if (y1 == y2 || !between(wy1, y1, y2))
                ans = dist(x1, y1, x2, y2);
            else {
                double xint = x1 + (x2-x1)*(wy1-y1)/(y2-y1);
                if (between(xint, wx1, wx2)) {
                    double dist1 = dist(x1,y1,wx1,wy1)+dist(wx1,wy1,x2,y2);
                    double dist2 = dist(x1,y1,wx2,wy2)+dist(wx2,wy2,x2,y2);
                    ans = (dist1<dist2) ? dist1 : dist2;
                }
                else
                    ans = dist(x1, y1, x2, y2);
            }
        }
    printf("Case %d: %.03f\n", icase, ans/2.0);
    }
}
```

Problem G

(Problem by Victor Reis)

You need to efficiently simulate the process:

1. Initialize two variables: an array to store the counts of each digit in the number so far (let's call it `counts`) and a number to represent the result (let's call it `result`).
2. Start by initializing the `counts` array with the frequency of digits in the initial value, A. You can achieve this by counting the digits in A.
3. Then, iterate N times to update the `counts` array and the `result`. In each iteration:
   a. Calculate the number of digits in each value of the `counts` array and add it to the respective count in `counts`. This represents how many times each digit appeared in the previous number.
   b. Update the `result` by adding the digits to it. To do this, for each digit in the counts array, multiply the result by 10 and add the digit to it. Ensure you apply the modulo M operation at each step to prevent overflow.
4. Print the value of the result