Problem Statement

It has been an hour into a Codeforces contest, when you notice that another contestant in your room has solved problem C using an `unordered_set`. Time to hack!

You know that the hash table has n buckets ($1 \leq n \leq 10^9$). Unfortunately, you do not know the value of n and wish to recover it.

When you insert an integer x into the hash table, it is inserted to the $(x \bmod n) + 1$-th bucket. If there are b elements in this bucket prior to the insertion, this will cause b hash collisions to occur.

By giving k distinct integers $x_1, x_2, \ldots, x_k$ to the interactor ($1 \leq x_i \leq 10^{18}$), you can find out the total number of hash collisions that had occurred while creating a set containing the numbers. However, feeding this interactor k integers in one query will incur a cost of k.

Do note that the interactor creates the hash table by inserting the elements in order into an initially empty set, and a new empty set will be created for each query. In other words, all queries are independent.

Your task is to find the number of buckets n using total cost at most 120 000.

Partial Credit will be given to solutions using higher total cost.

Solution

We can note the a query gives us the count of differences between elements of $\{x_i\}$ that are multiples of n.

The first step of our solution is to find a number x such that the query $\{1, 1 + x\}$ has a positive answer (this implies that x is a multiple of n).

Also in this solution we shall almost always be using our query to detect if there is at least one multiple of n among the differences (ie. just testing the yes/no on whether the answer is positive). The only exception is in some edge-case detection.

First, note that for all n in the valid range, such an x exists in the interval $(5 \cdot 10^8, 10^9]$. This is because the largest multiple of n at most 10^9 is smaller than this then doubling it will give us a bigger multiple.

For a number $L \approx \sqrt{5 \cdot 10^8} \approx 22500$, consider the sets $S_1 = \{1, 2, 3, 4 \ldots, L\}$ and $S_2 = \{5 \cdot 10^8 + L, 5 \cdot 10^8 + 2L, 5 \cdot 10^8 + 3L, \ldots\}$ up to a number greater than 10^9. The differences between sets S_2 and S_1 contains all numbers in the interval $(5 \cdot 10^8, 10^9]$, and thus the the query $S_1 + S_2$ will always be positive (there is no need to even make the query).

We then implement the following halving procedure.
Split the current set S_1 into two halves containing the smallest numbers L_1 and the largest numbers R_1 (note that both L_1, R_1 will be runs of consecutive integers). Similarly split S_2 into L_2 and R_2, noting that both halves will be arithmetic progressions.

We now want to check if there is an edge in the bipartite graph from L_1 to S_2. If there is, then we can recurse setting $S_1 := L_1$ and if there is no edge, then there must be an edge in the bipartite graph from R_1 to S_2. We will abuse notation a little to call such edges between the two halves 'bipartite' edges.

First we can query the set $L_1 \cup S_2$. If this set contains no collision, then we can be sure that there is a collision between R_1 and S_2.
However, if we get a positive answer, then it is unclear if the found collision is between L_1 and S_2 or strictly within one of L_1 or S_2 (This is important because our conclusion that if there is no collision in $L_1 \cup S_2$ there is a collision in $R_1 \cup S_2$ relies on the fact that there was a original collision in $S_1 \cup S_2$ that was strictly between the two parts).
However note that if there is a collision within L_1 then L_1 must fill every bucket modulo n, because L_1 consists of consecutive integers. Thus every element of S_2 will collide with an element of L_1, so there will be a 'bipartite' edge.
Thus, the only case we need to be concerned about is if every collision in $L_1 \cup S_2$ is strictly within S_2. However note that as S_2 is an arithmetic progression with common difference L, the number of collisions within S_2 depends only on $\text{lcm}(n, L) = kL$ (and the size of S_2) and is a strictly decreasing function of k (as long as the answer is positive). Thus we can binary search for the only potential value of k that accounts for all the collisions being inside of S_2. If such a k exists we can query the set $\{1, kL + 1\}$ and if it returns false then we must have at least one 'bipartite' edge. Thus we can test if $L_1 \cup S_2$ contains a bipartite edge and then recurse.
Splitting S_2 into L_2 and R_2 works exactly the same way, using the exact same bipartite edge check.

In total each halving step has cost $\frac{3L}{2} + (\frac{L}{2} + \frac{L}{2}) = \frac{5L}{2}$, and L halves each recursive step.
Thus the total cost for this step is around $5L \approx 112500$.

Once we have a known value x such that $n|x$, we can prime factorize x and for each prime factor of x (with duplicates) try to divide out that prime and see if the division is still a multiple of n (using $\{1, 1 + v\}$ queries). Through this process we will prune out any unnecessary factors of x, to get the exact value for n. This process does a query of cost 2 for each prime factor of x, and thus has cost at most $2 \cdot 30 = 60$ (negligible compared to part 1).

This solution is implemented as `sol.cpp`.

Alternative Solutions

When checking if there are any bipartite edges between A and B, we can query the three sets $A \cup B$, A, B and subtract the latter two values from the first. This counts exactly the edges between A and B, so we can just check if this value is positive.

These bipartite edge checks require twice as much total cost as in the main solution, and so achieves total cost around 220000. However, it is less sensitive to the exact way you choose to halve A and B as we don't require any number theoretic properties to hold.

This solution is implemented as `sol_double.cpp`.

We can also do a slight improvement to the 'bipartite' edge check in the original solution.
We shift down all the values of S_2 by a constant factor such that all values in S_2 are multiples of L, and then on each halving step for S_1 we query $R_1 \cup S_2$ instead (and as such we keep R_1 if the query result is positive).

In this case, if the query has a positive answer there will **always** be a bipartite edge in $R_1 \cup S_2$.

Proof: As before, if there is a collision within R_1 then all buckets modulo n are filled, so there will be a bipartite collision. However in this case if there is a collision within S_2, as S_2 is an arithmetic progression with gap L, S_2 will fill all buckets modulo n that are multiples of $\gcd(n, L)$. In particular, it will fill the bucket $L \pmod n$ which contains L. However, as S_2 contains a collision then in all previous halving rounds we will have kept the R_1. As L is the largest value in the original S_1 then that means that L will be kept in all halving rounds. Thus L is currently in R_1, so there is a bipartite edge between R_1 and S_2 as desired.
This property also works correctly when halving S_2, although in this case it doesn't matter if we query $S_1 \cup L_2$ or $S_1 \cup R_2$ as L will be in S_1 either way.

This solution is implemented as `sol_2.cpp`.

We can also further optimize by noting that our total cost is now not symmetric between $|S_1|$ and $|S_2|$ (it is $2|S_1| + 3|S_2|$, with the condition $|S_1||S_2| > 5 \cdot 10^8$). Thus by AM-GM we can note that the optimal value of L is such that $2L = 3 \cdot \frac{5 \cdot 10^8}{L} \implies L \approx 27386$. Using this value of L brings the total cost of our solution below 110000.

This optimization is included in `sol_2.cpp`.

Another alternative solution is if you don't make the observation that all n will have a multiple in the range $(5 \cdot 10^8, 10^9]$. In this case you can instead make the same sets as before, except with

B starting at $L+1$ instead of $5 \cdot 10^8$ to make the set of differences between A and B include all values from 1 to 10^9. Optimally this requires $L \approx \sqrt{10^9} \approx 31622$. You can combine this with either of the above two solutions to obtain costs of around 158000 or 316000.

Finally instead of creating a 'bipartite graph' we can just create a 'complete graph'- that is a set such that when you query it the answer is positive. Such a such is called a 'Golomb ruler'.

We can then implement the following refinement - given a set S which has a positive result, split it into three sets of equal size A, B, C. We then query the sets $A \cup B$, $A \cup C$. If either of these sets have positive answer then we reduce our set S to that. If neither of those have positive answer we can prove that the set $B \cup C$ must have two elements that are the same mod n, so we can reduce to set $B \cup C$ (without needed to query it). Once we reach a set of size 2 then we have a number that is a multiple of n and we can proceed as in the original solution.

The simplest Golomb ruler that be can create is just the set containing the values from 1 to $L-1$ and the multiple of L for $L \approx \sqrt{10^9} \approx 31622$. This will give us a starting set of size 63244. We can calculate the worst case outcome of our procedure using a dp to take around 250000 cost (the dp is included as `golomb_dp_calc.py`). While it is hard to generate data to hit all of the worst cases, this solution is very sensitive to getting lucky on the first few refinement queries, so random data should give us worst case costs similar to the theoretical worst case.

While the Golomb ruler can be improved further, we can also switch to using the set of size 44000 from before that only covers multiples of the possible values of n. Using this set instead gives us worst case cost around 176000.

Data Generation

As test cases are single integers they should not be hard to generate.

However, breaking solutions on the edge case where n is a multiple of the skip may be harder. For this, you can consider an adaptive solution where you consider multiple possible mods. For each query you can calculate all possible answers and choose one of them using some sort of heuristic (Maybe choose answer 0 first and then choose the largest possible remaining answer if we want to break the "all-multiples" strategy). After choosing an answer we filter to only those n which agree with our answer and then continue.

I have also added my polygon checker and interactor (as `polygon_checker.cpp` and `polygon_interactor.cpp`) although they presumably will need to be rewritten for the IOI sig-graded environment.

Potential Modifications / Subtask Ideas

If you want to limit the maximum number of queries then it's possible to make the cost function $20 + k$ instead of k. This would limit the number of queries made to the order of 10^4 which may be necessary if the interaction was through I/O. However, I would assume the C++ interactor would have low enough overhead such that allowing the user to make order 10^5 queries would be fine.

As noted above, we can also reduce the maximum cost for full score to be 110000.

A potential subtask idea would be to guarantee n to be chosen uniformly at random (or potentially that the generator will not be adaptive/n is fixed in advance). This would allow solutions to pass without considering the edge cases, which could be desired if the penalty for missing the edge cases for the full solution is too high.

A final (obvious) subtask idea would be just to have a smaller upper bound for n.