Problem 3: Solace of Quantum [HackerRank]

By Nalin Ranjan

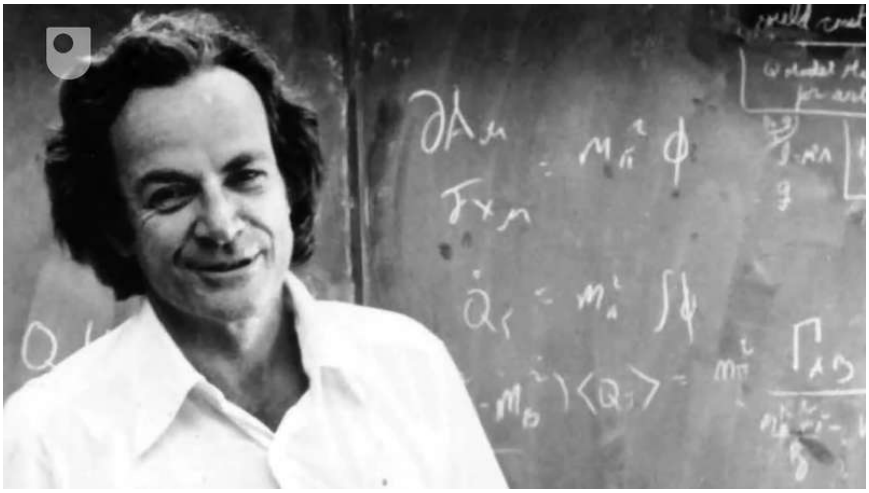
1 Background: Quantum Computing, Quantum Mechanics, and Fermions

(Reading this section isn't necessary for the problem. Still highly recommend it though; it's extremely interesting stuff!) Quantum computing is probably a buzzword you've heard tossed around. You might have heard of Shor's Algorithm for factoring numbers (which could break RSA, the cryptographic backbone of the internet) or Grover's Algorithm (which allows one to search an unstructured database without looking at most of the entries), both of which only work on quantum computers. But what was the original motivation for this totally radical model of computing?

To answer this question, we have to know a little bit of quantum mechanics. Quantum mechanics essentially postulates that all physical systems ultimately fall into one of many *states* that we can describe. The canonical example that's often cited is that of Schrodinger's cat: an evil Erwin Schrodinger has placed a poison capsule inside a box with his cat, but he doesn't know when the capsule is going to detonate. To a quantum physicist, the system (i.e. the cat in the box) is in one of two states: dead (the capsule detonated) or alive (the capsule didn't detonate yet). Now we might not know which state the system is: in this case, we might say the state of the system is a *superposition* of the two states. You can think of a superposition as a distribution over all states that tells us the probability of being in each state. This postulate has some mind-boggling consequences, one of the most important of which is that many of the things that seem on a human scale to be continuous are in fact discrete. If you want to learn more, check out the Feynman Lectures!

It was Richard Feynman who first observed that simulating quantum mechanics on a normal computer was impractical. Why? Because with only a few objects in our system, there could be way too many states to keep track of! One of the simplest multi-particle systems in quantum mechanics is a *fermionic system*. In this system, we have some number of energy levels n, and for each energy level, there is either a *fermion* in that energy level or not. But then there are 2^n possible states that the system could be in, meaning that in order to properly keep track of a superposition, we need to store at worst 2^n probabilities of being in each state! This is infeasible for even $n = 100$. Cue Feynman's brilliant idea: if nature operates

according to quantum mechanics, then can we use natural systems to simulate nature? If we imagine the universe as a quantum computer running a simulation (`universe.exe`), it seems to do pretty well (the universe never lags, for example). Can we harness some of its quantum compute power for our own purposes?



Richard Feynman: Nobel laureate, pioneer in quantum mechanics, father of quantum electrodynamics and quantum computing, first to explain superfluidity of supercooled helium, and avid bongos player. What are you doing with your life?

2 Problem Statement

A fermionic system with n energy levels that isn't in superposition can be represented simply as a bitstring of length n, because of the *Pauli Exclusion Principle*, which roughly says that no two fermions can occupy the same energy level. For example, if we have three energy levels, then the state $|010\rangle$ represents a system which has a fermion in the second energy level, but no fermions in the first or third energy levels. To simulate fermionic systems, we need to be able to simulate sequences of fermionic *raising* and *lowering operators* (don't worry about exactly what they are). Doing so efficiently boils down to being able to do three things efficiently on a state:

1. `query(i)`: Query whether an energy level `i` is occupied (return 1) or not occupied (return 0);

2. `flip(i)`: Change the value at energy level `i` from either occupied (1) to not occupied (0) or vice-versa.
3. `parityUpTo(i)`: Calculate the parity (return 1 if odd, 0 if even) of the sum of the occupation numbers of every energy level up to and including level `i`.

Your job is to design a system that can handle any sequence of any of these three types of queries efficiently. *Assume that in the initial state, every energy level is unoccupied. Further assume that the positions in the state are* ***zero-indexed****, i.e. they start from zero.*

Input

The first line contains two positive integers n and m, separated by the space. Here n is the number of energy levels in the system and m is the number of queries that will be issued. The subsequent m lines will contain two space-separated numbers, which will represent the type of query ($1 =$ `query`, $2 =$ `flip`, $3 =$ `parityUpTo`) and the argument of the query, respectively. See sections below for how queries are formed.

Output

Your program should output answers to queries in the order they are issued. Note that for queries of type 2 (`flip`), your program will not output anything; however, queries of type 2 will impact the output of future type 1 and 3 queries.

Constraints

$1 \leq n \leq 10^4$, $1 \leq m \leq 5 \cdot 10^5$. You can assume that no query will be malformed, i.e. every line will be of the form `"x y"` where `x` $\in \{1, 2, 3\}$ and $0 \leq$ `y` $< n$.

Example

Input:

```
5 6
1 3
2 2
2 3
3 0
3 4
1 2
```

Output:

```
0
0
0
1
```

Explanation: The first two input lines tell us that there will be 5 energy levels and 6 queries. Input line 2 corresponds to the operation `query(3)`. At the beginning the energy level is unoccupied, so the program prints `0` (output line 1). Input lines 3 and 4 correspond to the operations `flip(2)` and `flip(3)`; there is no output from these operations. After these operations, our state is now $|00110\rangle$ (remember zero indexing!). Input line 5 corresponds to the operation `parityUpTo(0)`, which is the parity of the sum of all occupation bits up to and including bit zero; this line results in the program printing `0` again. Input line 6 corresponds to the operation `parityUpTo(4)`, which is the parity of the sum of all occupation bits up to and including bit four; this line results in the program printing `0` ($= (0+0+1+1+0) \mod 2$). The final line of input corresponds to `query(2)`, which is 1 (recall that this asks for the occupation number of the 3rd energy level since we are zero-indexing).

Why doesn't the solution to this problem suffice to simulate fermionic systems? Because we only simulated a single state that *wasn't in superposition*! In reality, a fermionic system could be in a superposition of 2^n different states, where n is the number of energy levels. So we're going to need exponentially many of whatever data structure you're using in your solution!