Problem 3 Solution: Solace of Quantum

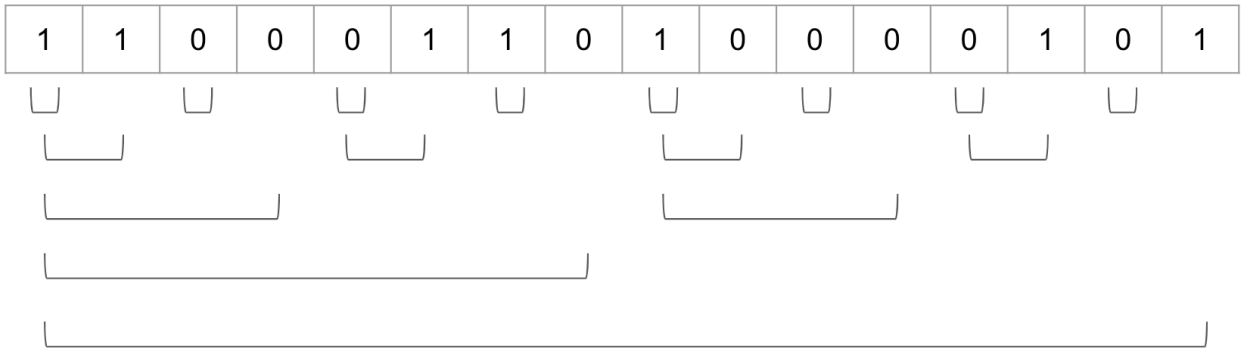
By Nalin Ranjan

This problem is a good example of how optimizing for data structure query time (the time to retrieve some information from it) and update time (the time to change the data structure) can be competing aims. It was inspired by a classic problem in qubit encoding of fermionic Hamiltonians (lol what's that?) to which Sergey Bravyi and Alexei Kitaev presented an elegant solution. The problem tries to coax the non-quantum part of their elegant solution out of you.

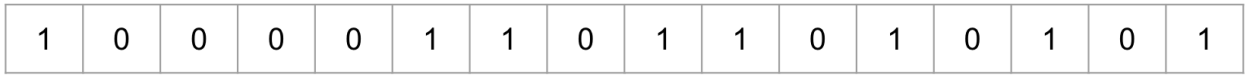
Here, one of the first solutions you probably thought of was just to store an array of length n, where the ith element was just the the ith occupation number. So the state $|11100011\rangle$ would be represented as, well, just the array $[1, 1, 1, 0, 0, 0, 1, 1]$. This is almost good: operations 1 and 2 are ridiculously simple and run in constant time. But handling an `parityUpTo` operation could be really bad, because we have no better option than to start loop from the beginning to the element we want, summing the elements of the array, which is a linear-time operation. Indeed, even the "average case" `parityUpTo` operation (where the argument is somewhere around $n/2$) would be a linear-time operation. A little bit of experimentation will tell you that other really simple heuristics (like storing the partial sums instead) don't work here either.

Let's for now ignore the `query` operation, as we can always just store a separate array that stores each occupation number to answer these queries. The central challenge is how to efficiently support *both* the `flip` and `parityUpTo` operations. It turns out there exists a data structure that is very well-equipped to do so! It's called a *Fenwick Tree*. The name is misleading: a Fenwick Tree is really just a special type of array. We're going to store n elements in our Fenwick Tree and each will store sums of occupation numbers for some subset of the n occupation numbers. The only remaining question, then, is how to decide which occupation numbers each element in the Fenwick Tree will be responsible for tracking.

It turns out that one of the most natural groupings works well. We're going to have every odd-indexed position store just the occupation number at that index, so index 1 actually stores the first occupation number, index 7 stores the seventh occupation number, and so on. For every index that is twice an odd number, we're going to store the sum of the occupation number at that index and the occupation number at the index before. So at index two, we're going to store the sum of occupation numbers 1 and 2, and at index 10 we're going to store the sum of occupation numbers 9 and 10. In general, if an index i is divisible by two k times, then it will store the sum of occupation numbers in the range $[i - 2^k + 1, i]$:



Indexing here starts at one. When implementing it, we can just allocate an array with $n+1$ elements and never use the zeroth element. In the above picture, the sum of occupation numbers in each range (brace) is stored in the *right endpoint* in our Fenwick tree. So our Fenwick tree would look like this:

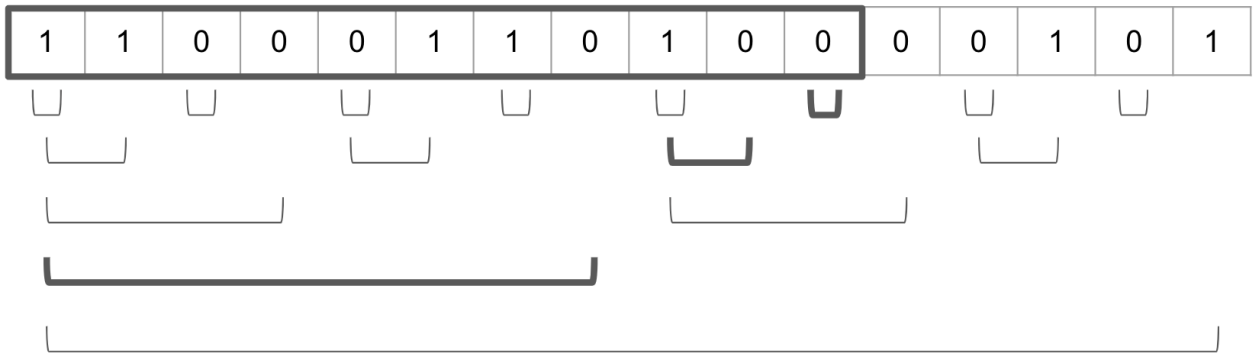


Remember that when we sum numbers, we only care about their parity. Now if we want to flip the value of any occupation number, we'll just have to flip the partial sums of any element in the Fenwick tree that includes that occupation number in the sum it stores. That is, we need to flip the value of each brace sum for each brace containing it. One way to do this if we are flipping some occupation number `i` is

```
set current_index = i
while (current_index <= n) {
     flip fenwick[current_index]
     add last set bit of current_index to current_index
}
```

So for example, if we start with `current_index = 12`, we will flip `fenwick[4]` and add 4 to `current_index`, because $12 = (1100)_2$ and the last set bit is in the fours place. Now that `current_index = 16`, we will flip `fenwick[16]` and add 16 to `current_index`. We will then terminate, because `current_index > 16`.

A very similar procedure can be used to calculate the parity up to a certain index i. We essentially just need to take the disjoint sequence of braces (that decrease in size) that "cover" the indices from 1 to i. For example, with $i = 11$, we want to sum the values of the bolded braces:



And it turns out the implementation of this operation is very similar to the implementation of the `flip` operation: one way to calculate the parity up to and including `i` is

```
set current_index = i
set sum = 0 // This will be the answer
while (current_index > 0) {
     sum = (sum + fenwick[current_index]) % 2
     subtract last set bit of current_index from current_index
}
```

This effectively keeps on summing the value at the current index of the fenwick tree and then zeros out the last set (one) bit of the current index, continuing until we hit zero.

Time Complexity: Every `query` operation is $O(1)$ (constant), since we're just storing the occupation numbers in a separate array. And the `flip` and `parityUpTo` operations each take a number of iterations that is at most the *number of digits* in the binary representation of n, i.e. $O(\log n)$ time.

Space Complexity: We use $O(n)$ memory (the occupation number array and the Fenwick tree). Apart from the factor of two, we can't really hope for better, because you can't store n bits of information in fewer than n bits in general.

Remarks: Moin Mir/Michael Fletcher/Faisal Fakhro and Andrew Tao/Aditya Mehta/Liam Esparraguerra came up with another way to support both the `flip` and `parityUpTo` operations in logarithmic time. Essentially, they used a balanced search tree that would contain the index of occupation bits with value 1. Querying would just involve searching for whether an index was in the tree, and flipping corresponded to adding/deleting an index from the tree. The `parityUpTo` operation can then be supported by a *rank* query, which will return how many elements with smaller index have occupation bit equal to one.

It also came to my attention that the extremely simple solution of just storing each occupation bit in an array and handling the `parityUpTo` operation in linear time sometimes worked in the "fast" languages (C and C++). In retrospect, the test cases were not big enough; there's also a huge gain in speed due to caching that probably made this possible (and explains, for example, why some of the tree-based solutions ended up not meeting the time limit). Nevertheless, if this was your solution, you still got credit for it :)

Plaudits

- Yuping Luo (grad), Danny Chen (grad), and Dingli Yu (grad) for submitting the fastest solution in a blazingly fast 9 minutes and 53 seconds!
- Josh Kolenbrander (COS '23), Kenny Huang (ORF '23), and Andrew Chen (COS '23) for being the fastest undergrad team to solve it in an impressive 20 minutes and 51 seconds!
- Stephane Morel (MAE '25), Akhil Paulraj (MAE '25), and Amanda Wang (ECE '25), for coming up with an extremely optimized version of the linear-time solution. They practically didn't waste a bit, as every occupation number was a bit of a 64-bit integer, and shifting/other bitwise operations made each operation extremely fast!