Rules: Work in groups of 3. Submit one set of solutions per group, and remember to put all your names on it. We'd prefer if you typed your answers, but if your handwriting is neat and legible, written work is fine. You may refer to whatever sources you want, but *CITE* your references! If you get a great idea from somebody, give appropriate credit: it won't lower your grade, and it's the right thing to do. If you have any questions or errors to report, email `ryanw@cs.cmu.edu`.

Deamortization, **or: Incrementing and decrementing a counter quickly.** We saw in class that incrementing a binary counter takes $O(1)$ amortized time, but requires $\Omega(n)$ worst case time (where n is the total number of bits in the counter). We also saw that when considering incrementing and decrementing together, they both require $\Omega(n)$ amortized time. In this problem, we will show how a "redundant" data structure allows one to increment and decrement in $O(1)$ *worst case time.*

The tradeoff will be that we use a funny representation for binary numbers, so we must "correct" the representation when the number is asked to be output. This correction takes $O(n)$ time. (Still, even outputting an n digit number takes $O(n)$ time, so this is not so bad.) The general idea of adding redundancy to data structures to get worst-case time bounds identical to the amortized bounds has been called *deamortization* by fellow CS grad student (and Cornell alum) Maverick Woo.

We promise that there exist very short algorithms for each of these problems. (They can't be too complicated, they're all constant time!)

1 Incrementing in worst case $O(1)$ time.

To tackle this problem, we consider *redundant arithmetic*; namely, binary arithmetic with an additional digit, `2`. The interpretation of a numeral $d_n \cdots d_0$ (with $d_i \in \{\mathtt{0}, \mathtt{1}, \mathtt{2}\}$) is still $d_{n-1} \cdot 2^{n-1} + \cdots + d_0 2^0$, but notice that most numbers have more than one representation now. For example, the number 10 in decimal may be written as `1010`, `0202`, `0210`, and `1002`. (So you can see why we call this "redundant" arithmetic.)

1. Using redundant arithmetic, describe an algorithm for incrementing a counter in $O(1)$ worst case time per increment. The possible inputs to your algorithm are either: (a) a binary numeral (without a 2), or (b) previous output of your algorithm. Observe that numerals like `222` $\cdots$ `222` *will* take more than $O(1)$ time to increment, so your algorithm must avoid outputting certain numerals.

2. As a sanity check to yourself, show the input and output behavior of your algorithm when it is executed six consecutive times, on the initial input `000`. For example, if your algorithm did not use the 2 at all, it would output `001`, `010`, `011`, `100`, `101`, `110` after being run six times.

3. Outline a proof of correctness (since a full proof of correctness could take a very long time for you to write and me to read). That is, state the claims/lemmas you would need to prove in order to prove your algorithm is correct, and sketch proofs for them.

4. Prove that your algorithm does take $O(1)$ worst case time. (Depending on your algorithm, this proof could be very easy.)

Hints: One way to do it takes only two $O(1)$ stages. In the first stage, you "repair" the number, by adjusting 2's in some manner (but only a constant number of 2's, of course). In the second stage, you actually increment the number.

2 Inserting into a heap in $O(1)$ worst case.

Recall the close relationship between binary addition and melding two binomial heaps. This problem illustrates that relationship further: improvement in the runtime of counters implies improvement in the runtime of heap-like data structures.

1. Describe a natural extension of binomial heaps based on the above structure.
2. Prove that insertion of an element in this new structure takes $O(1)$ worst case constant time.

The other relevant operations (deletemin, findmin, meld) should not take more than $O(\log n)$, $O(1)$, and $O(\log n)$ worst case time in your new data structure. (You shouldn't have to prove this; Kozen's lectures should be able to argue it for you if you extended binomial heaps properly.)

3 Incrementing and decrementing in $O(1)$ worst case.

It turns out that not only can you increment in $O(1)$ all the time, but you can increment and decrement in $O(1)$ time as well! To do this, we simply add more redundancy to the data structure: now digits d_i may be chosen over $\{-\texttt{1}, \texttt{0}, \texttt{1}, \texttt{2}\}$.

Describe procedures for incrementing and decrementing a counter using the above. As before, the possible inputs to your algorithms are either (a) a binary numeral or (b) previous output of your algorithms.

Hint: If you did the first problem in an elegant way, this is probably just a straightforward generalization of your incrementing algorithm, except now you "repair" the number by adjusting (a constant number of) either 2's or -1's, depending on whether you want to increment or decrement.

4 Open problem of the week.

Fibonacci heaps are nice, but in practice they can be a real pain to implement. For this reason, they aren't used as much in practice as they should be. So, is there a *simple* data structure which has the same amortized costs per operation as Fibonacci heaps?

Our criterion of what is simple is just what is easily implementable, or more easily implementable than Fibonacci heaps. Perhaps one can use asymptotically fewer pointers, fewer heaps, etc. and get the same amortized costs.