I'll gladly pay you Tuesday for a hamburger today.
— J. Wellington Wimpy

4 Amortized Analysis (February 2)

4.1 Incrementing a Binary Counter

The very first question in Homework Zero asked you to prove that any number could be written in binary. Although most of you (correctly) proved this using strong induction—pulling off either the least significant bit or the most significant bit and inductively converting the remainder—there is a somewhat more complicated proof that only uses weak induction.

Proof: *Base case:* $1 = 2^0$.

Inductive step: Suppose we have a set of distinct powers of two whose sum is n. If we add 2^0 to this set, we get a "set" of powers of two whose sum is $n+1$, but there might be two copies of 2^0. To fix this, as long as there are two copies of any 2^i, delete them both and add 2^{i+1}. The value of the sum is unchanged by this process, since $2^{i+1} = 2^i + 2^i$. Since there is always at most one duplicated exponent, which increases at each iteration, this process eventually terminates. At the end, we have a set of distinct powers of two whose sum is $n+1$. □

Here's a more formal (and shorter!) description of the algorithm. The input B is an array of bits, where $B[i] = 1$ if and only if 2^i appears in the sum.

```
INCREMENT(B):
i ← 0
while B[i] = 1
    B[i] ← 0
    i ← i + 1
B[i] ← 1
```

What's the running time of INCREMENT? Obviously, it depends on the array of bits passed as input. If the first k bits are all 1s, then INCREMENT takes $\Theta(k)$ time. Thus, if the number represented by B is between 0 and n, INCREMENT takes $\Theta(\log n)$ time in the worst case, since the binary representation for n is exactly $\lfloor \lg n \rfloor + 1$ bits long.

4.2 Counting from 0 to n — The Aggregate Method

Now suppose we want to use INCREMENT to count from 0 to n. If we only use the worst-case running time for each call, we get an upper bound of $O(n \log n)$ on the total running time. Although this bound is correct, it isn't the best we can do. The easiest way to get a tighter bound is to observe that we don't need to flip $\Theta(\log n)$ bits *every* time INCREMENT is called. The least significant bit $B[0]$ does flip every time, but $B[1]$ only flips every other time, $B[2]$ flips every 4th time, and in general, $B[i]$ flips every 2^ith time. If we start from an array full of zeros, a sequence of n INCREMENTs flips each bit $B[i]$ exactly $\lfloor n/2^i \rfloor$ times. Thus, the total number of bit-flips for the entire sequence is

$$\sum_{i=0}^{\lfloor \lg n \rfloor} \left\lfloor \frac{n}{2^i} \right\rfloor < \sum_{i=0}^{\infty} \frac{n}{2^i} = 2n.$$

Thus, *on average*, each call to INCREMENT flips only two bits, and so runs in constant time.

This "on average" is quite different from the averaging we did in the last lecture. There is no probability involved; we are averaging over a sequence of operations, not the possible running times of a single operation. This averaging idea is called *amortization*—the *amortized* cost of each INCREMENT is $O(1)$. Amortization is a ~~sleazy~~ clever trick used by accountants to average large one-time costs over long periods of time.

There are several different methods for deriving amortized bounds for a sequence of operations. [CLR] calls the one we just used the *aggregate* method, which is just a fancy way of saying sum up the total cost of the sequence and divide by the number of operations.

> **The Aggregate Method:** *Find the worst case running time* $T(n)$ *of a sequence of* n *operations. The amortized cost of each operation is* $T(n)/n$.

4.3 The Taxation (Accounting) Method

I want my two dollars!
— Johnny Gasparini

The second method we can use to derive amortized bounds is called the *accounting* method in [CLR], but a better name for it might be the *taxation* method. Suppose it costs a dollar to change the value of a bit, so we can measure the running time of our algorithm in dollars. Time is money!

Instead of paying for each bit flip when it happens, the Increment Revenue Service charges a two-dollar*increment tax* whenever we want to set a bit from zero to one. One of those dollars is spent changing the bit from zero to one; the other is stored away as *credit* until we need to reset the same bit to zero. The key point here is that we always have enough credit saved up to pay for the next INCREMENT. The amortized cost of an INCREMENT is the total tax it incurs, which is exactly 2 dollars, since each INCREMENT changes just one bit from 0 to 1.

It is often useful to assign various parts of the tax income to specific pieces of the data structure. For example, for each INCREMENT, we could store one of the two dollars on the single bit that is set for 0 to 1, so that *that* bit can pay to reset itself back to zero later on.

> **Taxation Method 1:** *Certain steps in the algorithm charge you taxes, so that the total money it spends is never more than the total taxes you pay. The amortized cost of an operation is the overall tax charged to you during that operation.*

Perhaps a more optimistic way of looking at the taxation method is to have the bits in the array pay *us* a tax for the privilege of being updated at the proper time. Regardless of whether we change the bit of not, we charge each bit $B[i]$ a tax of $1/2^i$ dollars for each INCREMENT. The total tax we collect is $\sum_{i \geq 0} 2^{-i} = 2$ dollars. Every time $B[i]$ actually needs to be flipped, it has paid us a total of \$1 since the last change, which is just enough for us to pay for the flip.

> **Taxation Method 2:** *Charge taxes to certain items in the data structure at each operation, so that the total money you spend is never more than the total taxes you collect. The amortized cost of an operation is the overall tax you collect during that operation.*

4.4 The Potential Method

The final and most powerful method we'll discuss uses a physics metaphor of "potential energy". Instead of associating costs or taxes with particular operations or pieces of the data structure, we represent prepaid work as *potential* that can be spent on later operations. The potential is a function of the entire data structure.

Let D_i denote our data structure after i operations, and let Φ_i denote its potential. Let c_i denote the actual cost of the ith operation (which changes D_{i-1} into D_i). Then the *amortized* cost of the ith operation, denoted a_i, is defined to be the actual cost plus the change in potential:

$$\boxed{a_i = c_i + \Phi_i - \Phi_{i-1}}$$

So the *total* amortized cost of n operations is the actual total cost plus the total change in potential:

$$\sum_{i=1}^{n} a_i = \sum_{i=1}^{n} (c_i + \Phi_i - \Phi_{i-1}) = \sum_{i=1}^{n} c_i + \Phi_n - \Phi_0.$$

Our task is to define a potential function so that $\Phi_0 = 0$ and $\Phi_i \geq 0$ for all i. Once we do this, the total *actual* cost of any sequence of operations will be less than the total *amortized* cost:

$$\sum_{i=1}^{n} c_i = \sum_{i=1}^{n} a_i - \Phi_n \leq \sum_{i=1}^{n} a_i.$$

For our binary counter example, we can define the potential Φ_i after the ith INCREMENT to be the number of bits with value 1. Initially, all bits are equal to zero, so $\Phi_0 = 0$, and clearly $\Phi_i > 0$ for all $i > 0$, so this is a legal potential function. We can describe both the actual cost of an INCREMENT and the change in potential in terms of the number of bits set to 1 and reset to 0.

$$c_i = \#\text{bits set to } 1 + \#\text{bits reset to } 0$$
$$\Phi_i - \Phi_{i-1} = \#\text{bits set to } 1 - \#\text{bits reset to } 0$$

The amortized cost of the ith INCREMENT is

$$a_i = c_i + \Phi_i - \Phi_{i-1} = 2 \times \#\text{bits set to } 1.$$

Since INCREMENT sets only *one* bit to 1, the amortized cost INCREMENT is 2.

> **The Potential Method:** *Define a potential function for the data structure that is initially equal to zero and is always nonnegative. The amortized cost of an operation is its actual cost plus the change in potential.*

For this particular example, the potential is exactly equal to the total unspent taxes paid using the taxation method, so not too surprisingly, we have exactly the same amortized cost. In general, however, there may be no way of interpreting the change in potential as "taxes".

Different potential functions will lead to different amortized time bounds. The trick to using the potential method is to come up with the best possible potential function. Unfortunately, there is no general technique for doing this other than playing around with the data structure and trying lots of different possibilities.

4.5 Incrementing and Decrementing

Now suppose we wanted do have a binary counter that we could both increment and decrement efficiently, in the amortized sense. A standard binary counter won't work, even in an amortized sense, since alternating between 2^k and 2^{k-1} costs $\Theta(k)$ time per operation.[1]

A nice alternative is represent a number as a pair of bit strings (P, N), where for any bit position i, at most one of the bits $P[i]$ and $N[i]$ is equal to 1. The actual value of the counter is $P - N$. Here are algorithms to increment and decrement our double counter.

```
INCREMENT(P,N):
  i ← 0
  while P[i] = 1
      P[i] ← 0
      i ← i + 1
  if N[i] = 1
      N[i] ← 0
  else
      P[i] ← 1
```

```
DECREMENT(P,N):
  i ← 0
  while N[i] = 1
      N[i] ← 0
      i ← i + 1
  if P[i] = 1
      P[i] ← 0
  else
      N[i] ← 1
```

Now suppose we start from $(0,0)$ and apply a sequence of n INCREMENTS and DECREMENTS. In the worst case, operation takes $\Theta(\log n)$ time, but what is the amortized cost? We can't use the aggregate method here, since we don't know what the sequence of operations looks like.

What about the taxation method? It's not hard to prove (by induction, of course) that after either $P[i]$ or $N[i]$ is set to 1, there must be at least 2^i operations, either INCREMENTS or DECREMENTS, before that bit is reset to 0. So if each bit $P[i]$ and $N[i]$ pays a tax of 2^{-i} at each operation, we will always have enough money to pay for the next operation. Thus, the amortized cost of each operation is at most $\sum_{i \geq 0} 2(\cdot 2^{-i}) = 4$.

We can get even better bounds using the potential method. Define the potential Φ_i to be the number of 1-bits in both P and N after i operations. Just as before, we have

$$\begin{aligned} c_i &= \#\text{bits set to } 1 + \#\text{bits reset to } 0 \\ \Phi_i - \Phi_{i-1} &= \#\text{bits set to } 1 - \#\text{bits reset to } 0 \\ \Longrightarrow \qquad a_i &= 2 \times \#\text{bits set to } 1 \end{aligned}$$

Since each operation sets *at most* one bit to 1, the ith operation has amortized cost $a_i \leq 2$.

[1] At this point in the lecture, a student suggested using *Gray codes*. Gray codes (named after Frank Gray who discovered them in the 1950s) are methods for representing numbers as bit strings so that successive numbers differ by only one bit. For example, here is the four-bit *binary reflected* Gray code for the integers 0 through 15:

$$0000, 0001, 0011, 0010, 0110, 0111, 0101, 0100, 1100, 1101, 1111, 1110, 1010, 1011, 1001, 1000$$

In general, the rule for incrementing a binary reflected Gray code is to invert the bit that would be set from 0 to 1 by a normal binary counter.

In terms of bit-flips, this was the perfect solution; each increment of decrement *by definition* changes only one bit. Unfortunately, *finding* that bit still requires $\Theta(\log n)$ time in the worst case, so the total cost of maintaining a Gray code is actually the same as that of maintaining a normal binary counter.

Exercise: Modify the binary double-counter to support a new operation SIGN, which determines whether the number being stored is positive, negative, or zero, in constant time. The amortized time to increment or decrement the counter should still be a constant.

[Hint: Suppose p is the number of significant bits in P, and n is the number of significant bits in N. For example, if $P = 17 = 10001_2$ and $N = 0$, then $p = 5$ and $n = 0$. Then $p - n$ always has the same sign as $P - N$. Unfortunately, just storing p and n is against the rules; if you could store integers, you wouldn't need to build a counter in the first place!]

Exercise: Suppose instead of powers of two, we represent integers as the sum of Fibonacci numbers. In other words, instead of an array of bits, we keep an array of "fits", where the ith least significant fit indicates whether the sum includes the ith Fibonacci number F_i. For example, the fit string 101110 represents the number $F_6 + F_4 + F_3 + F_2 = 8 + 3 + 2 + 1 = 14$. Describe algorithms to increment and decrement a fit string in constant amortized time. [Hint: Most numbers can be represented by more than one fit string.]