The goode workes that men don whil they ben in good lif al amortised by synne folwyng.
— Geoffrey Chaucer, "The Persones [Parson's] Tale" (c.1400)

I will gladly pay you Tuesday for a hamburger today.
— J. Wellington Wimpy, "Thimble Theatre" (1931)

I want my two dollars!
— Johnny Gasparini [Demian Slade], "Better Off Dead" (1985)

7 Amortized Analysis (October 3)

7.1 Incrementing a Binary Counter

One of the questions in Homework Zero asked you to prove that any number could be written in binary. Although some of you (correctly) proved this using strong induction—pulling off either the least significant bit or the most significant bit and letting the recursion fairy convert the remainder—the most common proof used weak induction as follows:

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 each iteration decreases the number of powers of two in our 'set', this process must eventually terminate. At the end of this process, 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 to add one to a binary numeral. 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
```

We've already argued that INCREMENT must terminate, but how quickly? Obviously, the running time 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.

7.2 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 previous 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; the most common example is calculating uniform payments for a loan, even though the borrower is paying interest on less and less capital over time.

There are several different methods for deriving amortized bounds for a sequence of operations. CLR calls the technique 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 $T(n)$ for a sequence of n operations. The amortized cost of each operation is $T(n)/n$.*

7.3 The Taxation (Accounting) Method

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 us a dollar to toggle 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 or 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.*

In both of the taxation methods, our task as algorithm analysts is to come up with an appropriate 'tax schedule'. Different 'schedules' can result in different amortized time bounds. The tightest bounds are obtained from tax schedules that *just barely* stay in the black.

7.4 The Potential Method

The most powerful method (and the hardest to use) builds on 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.

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

Thus, the amortized cost of the ith INCREMENT is

$$a_i = c_i + \Phi_i - \Phi_{i-1} = 2 \times \#\text{bits changed from 0 to 1}$$

Since INCREMENT changes only *one* bit from 0 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. A good potential function goes up a little during any cheap/fast operation, and goes down a lot during any expensive/slow operation. Unfortunately, there is no general technique for doing this other than playing around with the data structure and trying lots of different possibilities.

7.5 Incrementing and Decrementing

Now suppose we wanted a binary counter that we could both increment and decrement efficiently. 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.

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 binary 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
```

Here's an example of these algorithms in action. Notice that any number other than zero can be represented in multiple (in fact, infinitely many) ways.

$$\begin{array}{ccccccccccccc}
P = 10001 & & P = 1001\mathbf{0} & & P = 1001\mathbf{1} & & P = 1000\mathbf{0} & & P = 10000 & & P = 10000 & & P = 1000\mathbf{1} \\
N = 01100 & \xrightarrow{++} & N = 01100 & \xrightarrow{++} & N = 01100 & \xrightarrow{++} & N = 0100\mathbf{0} & \xrightarrow{--} & N = 0100\mathbf{1} & \xrightarrow{--} & N = 0101\mathbf{0} & \xrightarrow{++} & N = 01010 \\
P - N = 5 & & P - N = 6 & & P - N = 7 & & P - N = 8 & & P - N = 7 & & P - N = 6 & & P - N = 7
\end{array}$$

Incrementing and decrementing a double-binary counter.

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

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

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

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: If P has p significant bits, and N has n significant bits, then $p - n$ always has the same sign as $P - N$. For example, if $P = 17 = 10001_2$ and $N = 0$, then $p = 5$ and $n = 0$. But how do you store p and n??]*

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 fitstring 101110_F represents the number $F_6 + F_4 + F_3 + F_2 = 8 + 3 + 2 + 1 = 14$. Describe algorithms to increment and decrement a single fitstring in constant amortized time. *[Hint: Most numbers can be represented by more than one fitstring!]*

7.6 Aside: Gray Codes

An attractive alternate solution to the increment/decrement problem was independently suggested by several students. *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:

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

The general 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 is the perfect solution; each increment of decrement *by definition* changes only one bit. Unfortunately, it appears that *finding* the single bit to flip 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.

Actually, this is only true of the naïve algorithm. The following algorithm, discovered by Gideon Ehrlich[1] in 1973, maintains a Gray code counter in constant *worst-case* time per increment! The algorithm uses a separate 'focus' array $F[0..n]$ in addition to a Gray-code bit array $G[0..n-1]$.

```
EhrlichGrayInit(n):
  for i ← 0 to n − 1
      G[i] ← 0
  for i ← 0 to n
      F[i] ← i
```

```
EhrlichGrayIncrement(n):
  j ← F[0]
  F[0] ← 0
  if j = n
      G[n − 1] ← 1 − G[n − 1]
  else
      G[j] = 1 − G[j]
      F[j] ← F[j + 1]
      F[j + 1] ← j + 1
```

The EhrlichGrayIncrement algorithm obviously runs in $O(1)$ time, even in the worst case. Here's the algorithm in action with $n = 4$. The first line is the Gray bit-vector G, and the second line shows the focus vector F, both in reverse order:

$$G : 0000, 0001, 0011, 0010, 0110, 0111, 0101, 0100, 1100, 1101, 1111, 1110, 1010, 1011, 1001, 1000$$
$$F : 3210, 3211, 3220, 3212, 3310, 3311, 3230, 3213, 4210, 4211, 4220, 4212, 3410, 3411, 3240, 3214$$

Voodoo! I won't explain in detail how Ehrlich's algorithm works, except to point out the following invariant. Let $B[i]$ denote the ith bit in the *standard* binary representation of the current number. **If $B[j] = 0$ and $B[j-1] = 1$, then $F[j]$ is the smallest integer $k > j$ such that $B[k] = 1$; otherwise, $F[j] = j$.** Got that?

But wait — this algorithm only handles increments; what if we also want to decrement? Sorry, I don't have a clue. Extra credit, anyone?

[1] G. Ehrlich. Loopless algorithms for generating permutations, combinations, and other combinatorial configurations. *J. Assoc. Comput. Mach.* 20:500–513, 1973.