CS 4349 Lecture—November 27, 2017

Main topics for #amortized_analysis.

Prelude

- Homework 11 due November 6th. It has more questions, but they should be easier to answer than the usual ones.
- YOU MUST TURN THIS ONE IN AT THE BEGINNING OF LECTURE SO WE CAN TALK ABOUT IT (or any time Monday).
- A few times this semester I've talked about how some data structure operations take a certain amount of time "on average" even if a few individual operations are actually very slow in the worst case.
- This "average case" analysis of data structures is a common trick used even in some of the standard libraries used in popular languages, but what does it mean?
- On Wednesday, I am going to give some examples of how to analyze a couple particularly useful and interesting real data structures.
- But today, I'm going to formally define how the analysis works using a toy problem and give some tools for doing the analysis.
- And in case you're worried, I won't be testing you on this stuff, although I obviously think it is useful to know, or I wouldn't be teaching it!

Binary Counter

- Suppose we want to implement a binary counter. We'll store an array B[0 .. infinity] of bits where every bit is initially 0. Every time we try to increment the counter, we need to flip individual bits.
- That's accomplished by adding 1 to the least significant bit, if it rolls over we carry the 1 to the next bit and recurse on the more significant bits.
- Or, in pseudocode:

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

- But how long does this take? If the first k bits are all 1s, then increment takes Theta(k) time.
- It takes floor(lg n) + 1 bits to represent a positive integer n. So if B represents an integer between 0 and n, Increment takes Theta(log n) time in the worst case.
- So what if we count from 0 to n by calling Increment n times? The worst-case running time

is O(log n), so doing so takes *at most* O(n log n) time. But in reality, the total time spent in less than that. Today, we're going to look at a few ways of proving that statement.

Summation

- So the simplest way to get a better bound is to observe that Increment doesn't flip all O(log n) bits with every call.
- B[0] if flipped every iteration, but B[1] is only flipper every other iteration, B[2] is flipped every 4th iteration. More generally, B[i] flips every 2^ith iteration.
- So B[i] flips exactly floor{n / 2^i} times.
- If we sum it up, we see the total number of flips is
  - sum_{i=0}^floor{lg n} floor{n/2^i} < sum_{i = 0}^infinity n / 2^i = 2n.
- So "on average" we only perform 2 flips per Increment, even though some increments might use Theta(log n) flips.
- This analysis is call *amortization*: averaging the running time or "cost" of an operation over time. The *amortized* cost of Insertion is O(1).
- Generally, we can claim any amortized cost we want as long as the sum of the amortized costs is at least as high as the sum of the actual costs.
- We just used the *summation method* for analyze Insertion:
  - Let T(n) be the worst-case running time for a sequence of n operations. The amortized cost of each operation is T(n) / n.

Taxation

- Sometimes directly summing costs for certain operations isn't the easiest thing, so we use other tricks inspired by real world situations.
- The first of these is the taxation method. We'll charge a tax to certain operations to help us pay for future ones.
- Let's say the Increment Revenue Service chargers a two dollars increment tax every time we want to flip a bit from 0 to 1.
  - 1 dollar is immediately spent on doing the actual flip from 0 to 1.
  - The other dollar is stored as credit and finally used when that bit gets set back to 0.
- The tax credits will always pay for the bit flips, so we can say the taxes *are* the amortized cost of each operation.
- But we only flip a single bit from 0 to 1 per Increment, so the taxes are 2 per Increment. Again, the amortized cost is 2.
- Taxation Method 1: Certain **steps** in the algorithm charge a tax and taxes pay for all algorithm costs. The amortized cost of an operation is the overall tax charged during an operation.

- There's another taxation scheme we could use: Charge bit B[i] a tax of 1/2^i during *every* Increment.
- By the time we flip B[i], it will have collected $1 in taxes, enough to pay for its flip.
- But the total charged in each operation is sum_{i ≥ 0} 2^i = 2.
- Taxation Method 2: Certain **portions of the data structure** charge a tax in each operation, and taxes pay for all algorithm costs. The amortized cost of an operation is the overall tax charged during an operation.
- Our goal with the taxation method is to come up with a good tax *schedule* to pay for all operations. Different schedules may give different amortized bounds. Our goal is to keep amortized costs low, so we need to stay barely in the black. Meaning no deficits and barely any surpluses.

Charging

- In the taxation method, we collected enough taxes to pay for future operations. This next method is sort of the opposite.
- What we can do instead is to *charge* the cost of some steps in an operation to other steps in the same or an earlier operation.
- Charging Method: Charge the cost of some steps of the algorithm to other steps in the same or an earlier operation. The amortized cost of an operation is the actual running time - charges **to past** operations + charges **from future** operations.
- For example, if an Increment does k bit flips, we charge each of the k - 1 flips from 1 to 0 to the previous flip that set that bit to 1.
- The cost of an Increment is therefore k - (k-1) + 1 = 2.

Potential

- There's one more analysis method I want to talk about. It's certainly more complicated than the previous methods, but hopefully on Wednesday you'll see why its so powerful.
- The idea builds on a physics metaphor. Consider this ball. The faster it moves, the more kinetic energy it has. I can add kinetic energy to the ball by throwing it. I can also add energy by lifting the ball really high into the air. As I lift the ball, I'm add *potential energy* to it that will be converted into kinetic energy as it falls to the ground.
- We can use a similar idea to analyze algorithms. We'll accumulate *potential* during certain operations that can be spent on future operations. Instead of focusing on precisely what steps granted us potential, we'll just store it as a function of the state of the data structure.
- Let D_i be the data structure after i operations.
- Let phi_i be its potential.
- Let c_i be the *actual* cost of the ith operation (changing D_{i-1} to D_i)

- The amortized cost is
  - a_i = c_i + phi_i - phi_{i-1}
- So is this a valid way to define amortized cost? Well, the sum of the amortized costs is
  - 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.
- A potential function is *valid* if phi_i - phi_0 ≥ 0 for all i. If phi is valid, then
  - sum_{i=1}^n a_i ≥ sum_{i=1}^n c_i like we want to see.
- So how do we use potential for our binary counter? Well, we keep taxing or charging to bits flipped to 1. Why don't we just count them instead.
- Let phi_i be the number of bits equal to 1 after the ith Increment.
- phi_0 = 0 and phi_i > 0 for i > 0 so the potential function is valid.
- c_i = # bits changed from 0 to 1 + # bits changed from 1 to 0
- phi_i - phi_{i-1} = # bits changed from 0 to 1 - # bits changed from 1 to 0
- so a_i = c_i + phi_i - phi_{i-1} = 2 x # bit changed from 0 to 1 = 2.
- Potential Method: Define a potential function that is (usually) initially 0 and always non-negative. The amortized cost of an operation is the actual cost plus the change in potential.
- Different potential functions lead to different amortized time bounds.
- Generally, you want the potential to increase a little during actually cheap operations and to decrease a lot during expensive operations. In other words, you want the potential to be high whenever there *may* be an expensive operation coming up.
- And I want to reemphasize that last point. Hopefully on Wednesday, you'll see that you can arrive at different amortized time bounds depending on the analysis. And different sets of bounds for different operations may be consistent. Generally, you want to find a set of bounds that work best for whatever application you have in mind.

Increment and Decrement

- Since we have a little more time, let's consider a slightly more interesting toy example.
- Suppose we want our binary counter to support both increment and decrement. Unfortunately, we can't just flip bits back and forth in B[1 .. infinity]. Alternating between 2^k and 2^k - 1 will take Theta(k) time in *every* operation so there's no better amortized bound.
- Another idea is to instead store two arrays P[1 .. infinity] and N[1 .. infinity] where
  - Most one of P[i] and N[i] equals 1 for all i and
  - if you interpret P and N as binary numbers, the actual value of the counter is P - N.
- So P is the "positive" part and N is the "negative" part.
- The new representation is *not* unique!
- But we do get some fairly simple increment and decrement algorithms based on slight

changes to the increment from earlier.

```
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
```

- So what happens if we start from (0, 0) and do n Increments and Decrements? In the worst case, each operation takes Theta(log n) time, but what is the amortized cost?
- The summation method is really hard to use here, because who knows what things look like after n arbitrary Increments and Decrements.
- But it turns out the potential method is really useful here.
- To use potential functions effectively, we ask, “what does the data structure look like when there may be an expensive operation?” When there are a lot of 1s!
- So let phi_i be the number of 1 bits in both P and N after the ith operation.
- c_i = # bits changed from 0 to 1 + # bits changed from 1 to 0
- phi_i - phi_{i-1} = # bits changed from 0 to 1 - # bits changed from 1 to 0
- so a_i = c_i + phi_i - phi_{i-1} = 2 x # bit changed from 0 to 1 = 2.
- Oh, the amortized cost is still 2!