CS 461, Lecture 11

Jared Saia
University of New Mexico

Today’s Outline

- Midterm Post Mortem
- Amortized Analysis

Midterm Grade Ranges (Revised)

Rough ranges for grades are as follows:

- 90-100 A
- 70-90 B
- 60-70 C
- 0-60 F

Amortized Analysis

‘I will gladly pay you Tuesday for a hamburger today’ - Wellington Wimpy

- In amortized analysis, time required to perform a sequence of data structure operations is averaged over all the operations performed
- Typically used to show that the average cost of an operation is small for a sequence of operations, even though a single operation can cost a lot
Amortized analysis is *not* average case analysis.

- *Average Case Analysis*: the expected cost of each operation
- *Amortized analysis*: the average cost of each operation in the worst case
- Probability is not involved in amortized analysis

### Types of Amortized Analysis

- **Aggregate Analysis**
- **Accounting or Taxation Method**
- **Potential method**

We'll see each method used for 1) a stack with the additional operation MULTIPOP and 2) a binary counter

### Aggregate Analysis

- We get an upper bound $T(n)$ on the total cost of a sequence of $n$ operations. The average cost per operation is then $T(n)/n$, which is also the amortized cost per operation

### Stack with Multipop

- Recall that a standard stack has the operations PUSH and POP
- Each of these operations runs in $O(1)$ time, so let's say the cost of each is 1
- Now for a stack $S$ and number $k$, let's add the operation MULTIPOP which removes the top $k$ objects on the stack
- Multipop just calls Pop either $k$ times or until the stack is empty
Q: What is the running time of Multipop(S,k) on a stack of s objects?
A: The cost is min(s,k) pop operations
If there are n stack operations, in the worst case, a single Multipop can take $O(n)$ time

This analysis is technically correct, but overly pessimistic
While some of the multipop operations can take $O(n)$ time, not all of them can
We need some way to average over the entire sequence of n operations

Let’s analyze a sequence of n push, pop, and multipop operations on an initially empty stack
The worst case cost of a multipop operation is $O(n)$ since the stack size is at most n, so the worst case time for any operation is $O(n)$
Hence a sequence of n operations costs $O(n^2)$

In fact, the total cost of n operations on an initially empty stack is $O(n)$
Why? Because each object can be popped at most once for each time that it is pushed
Hence the number of times POP (including calls within Multipop) can be called on a nonempty stack is at most the number of Push operations which is $O(n)$
Aggregate Analysis

- Hence for any value of $n$, any sequence of $n$ Push, Pop, and Multipop operations on an initially empty stack takes $O(n)$ time.
- The average cost of an operation is thus $O(n)/n = O(1)$.
- Thus all stack operations have an amortized cost of $O(1)$.

Another Example

Another example where we can use aggregate analysis:

- Consider the problem of creating a $k$ bit binary counter that counts upward from 0.
- We use an array $A[0..k-1]$ of bits as the counter.
- A binary number $x$ that is stored in $A$ has its lowest-order bit in $A[0]$ and highest order bit in $A[k-1]$ ($x = \sum_{i=0}^{k-1} A[i] \times 2^i$).

Binary Counter

- Initially $x = 0$ so $A[i] = 0$ for all $i = 0, 1, \ldots, k - 1$.
- To add 1 to the counter, we use a simple procedure which scans the bits from right to left, zeroing out 1’s until it finally finds a zero bit which it flips to a 1.

Increment

```c
Increment(A){
    i = 0;
    while(i<k && A[i]!=1){
        A[i] = 0;
        i++;
    }
    if (i<k)
        A[i] = 1;
}
```
Analysis

- It's not hard to see that in the worst case, the increment procedure takes time $\Theta(k)$
- Thus a sequence of $n$ increments takes time $O(nk)$ in the worst case
- Note that again this bound is correct but overly pessimistic - not all bits flip each time increment is called!

Aggregate Analysis

- In general, for $i = 0, \ldots, \lfloor \log n \rfloor$, bit $A[i]$ flips $\left\lfloor n/2^i \right\rfloor$ times in a sequence of $n$ calls to Increment on an initially zero counter
- For $i > \lfloor \log n \rfloor$, bit $A[i]$ never flips
- Total number of flips in the sequence of $n$ calls is thus
  \[
  \sum_{i=0}^{\lfloor \log n \rfloor} \left\lfloor \frac{n}{2^i} \right\rfloor < n \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n
  \]  

Aggregate Analysis

- In fact, we can show that a sequence of $n$ calls to Increment has a worst case time of $O(n)$
- $A[0]$ flips every time Increment is called, $A[1]$ flips over every other time, $A[2]$ flips over every fourth time, ... 
- Thus if there are $n$ calls to increment, $A[0]$ flips $n$ times, $A[1]$ flips $\lfloor n/2 \rfloor$ times, $A[2]$ flips $\lfloor n/4 \rfloor$ times

Aggregate Analysis

- Thus the worst-case time for a sequence of $n$ Increment operations on an initially empty counter is $O(n)$
- The average cost of each operation in the worst case then is $O(n)/n = O(1)$
The second method is called the accounting method in the book, but a better name might be the taxation method. Suppose it costs us a dollar to do a Push or Pop. We can then measure the run time of our algorithm in dollars (Time is money!)

Like any good government (ha ha) we need to make sure that: 1) our taxes are low and 2) we can use our taxes to pay for all our costs. We already know that our taxes for \( n \) operations are no more than \( 2n \) dollars. We now want to show that we can use the 2 dollars we collect for each push to pay for all the push, pop and multipop operations.

Instead of paying for each Push and Pop operation when they occur, let’s tax the pushes to pay for the pops. I.e. we tax the push operation 2 dollars, and the pop and multipop operations 0 dollars. Then each time we do a push, we spend one dollar of the tax to pay for the push and then save the other dollar of the tax to pay for the inevitable pop or multipop of that item. Note that if we do \( n \) operations, the total amount of taxes we collect is then \( 2n \).

This is easy to show. When we do a push, we use 1 dollar of the tax to pay for the push and then store the extra dollar with the item that was just pushed on the stack. Then all items on the stack will have one dollar stored with them. Whenever we do a Pop, we can use the dollar stored with the item popped to pay for the cost of that Pop. Moreover, whenever we do a Multipop, for each item that we pop off in the Multipop, we can use the dollar stored with that item to pay for the cost of popping that item.
We’ve shown that we can use the 2 tax on each item pushed to pay for the cost of all pops, pushes and multipops.
Moreover we know that this taxation scheme collects at most $2^n$ dollars in taxes over $n$ stack operations.
Hence we’ve shown that the amortized cost per operation is $O(1)$.

Let’s now use the taxation method to show that the amortized cost of the Increment algorithm is $O(1)$.
Let’s say that it costs us 1 dollar to flip a bit.
What is a good taxation scheme to ensure that we can pay for the costs of all flips but that we keep taxes low?

Let’s tax the algorithm 2 dollars to set a bit to 1.
Now we need to show that: 1) this scheme has low total taxes and 2) we will collect enough taxes to pay for all of the bit flips.
Showing overall taxes are low is easy: Each time Increment is called, it sets at most one bit to a 1.
So we collect exactly 2 dollars in taxes each time increment is called.
Thus over $n$ calls to Increment, we collect $2n$ dollars in taxes.

We now need to show that our taxation scheme has enough money to pay for the costs of all operations.
This is easy: Each time we set a bit to a 1, we collect 2 dollars in tax. We use one dollar to pay for the cost of setting the bit to a 1, then we store the extra dollar on that bit.
When the bit gets flipped back from a 1 to a 0, we use the dollar already on that bit to pay for the cost of the flip!
Binary Counter

- We’ve shown that we can use the 2 tax each time a bit is set to a 1 to pay for all operations which flip a bit
- Moreover we know that this taxation scheme collects $2^n$ dollars in taxes over $n$ calls to Increment
- Hence we’ve shown that the amortized cost per call to Increment is $O(1)$

In Class Exercise

- A sequence of Pushes and Pops is performed on a stack whose size never exceeds $k$
- After every $k$ operations, a copy of the entire stack is made for backup purposes
- Show that the cost of $n$ stack operations, including copying the stack, is $O(n)$

In Class Exercise

- A sequence of Pushes and Pops is performed on a stack whose size never exceeds $k$
- After every $k$ operations, a copy of the entire stack is made for backup purposes
- Show that the cost of $n$ stack operations, including copying the stack, is $O(n)$

In Class Exercise

- Q1: What is your taxation scheme?
- Q2: What is the maximum amount of taxes this scheme collects over $n$ operations?
- Q3: Show that your taxation scheme can pay for the costs of all operations