ICS 311 #15: Amortized Analysis

Outline

1. Amortized Analysis: The General Idea
2. Multipop Example
3. Aggregate Analysis
4. Accounting Method
5. Potential Method - *Optional for 2015, but it's the most powerful method*
6. Dynamic Table Example (first look)
7. Other Examples

Readings and Screencasts

- Chapter 17 of Cormen et al. (2015: through 17.2, plus 17.4.1)
- Screencasts 15A, 15B - (also in Laulima and iTunesU)

Amortized Analysis: The General Idea

We have already used *aggregate* analysis several times in this course. For example, when analyzing the BFS and DFS procedures, instead of trying to figure out how many times their inner loops

```
for each v ∈ G.Adj[u]
```

execute (which depends on the degree of the vertex being processed), we realized that no matter how the edges are distributed, there are at most $|E|$ edges, so in aggregate across all calls the loops will execute $|E|$ times.

But that analysis concerned itself only with the complexity of a single operation. In practice a given data structure will have associated with it several operations, and they may be applied many times with varying frequency.

Sometimes a given operation is designed to pay a larger cost than would otherwise be necessary to enable other operations to be lower cost.

Example: Red-black tree insertion. We pay a greater cost for balancing so that future searches will remain $O(\lg n)$.

Another example: Java Hashtable.

- These grow dynamically when a specified load factor is exceeded.
- Copying into a new table is expensive, but copying is infrequent and table growth makes access operations faster.

It is "fairer" to average the cost of the more expensive operations across the entire mix of operations, as all operations benefit from this cost.

Here, "average" means average cost in the worst case (no probability is involved, which greatly simplifies

the analysis).

We will look at three methods. The notes below use Stacks with Multipop to illustrate the methods. See the text for binary counter examples.

(We have already seen examples of aggregate analysis throughout the semseter. We will see examples of amortized analysis later in the semester.)

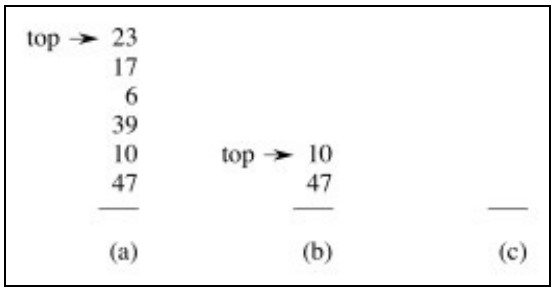
Multipop Example

We already have the stack operations:

- `Push(S, x)`: O(1) each call, so O(n) for any sequence of n operations.
- `Pop(S)`: O(1) each call, so O(n) for any sequence of n operations.

Suppose we add a `Multipop` (this is a generalization of `ClearStack`, which empties the stack):

```
MULTIPOP(S, k)
1  while not STACK-EMPTY(S) and k > 0
2      POP(S)
3      k = k − 1
```



The example shows a `Multipop(S,4)` followed by another where $k \geq 2$.

Running time of `Multipop`:

- Linear in number of `Pop` operations (one per loop iteration)
- Number of iterations of `while` loop is min(s, k), where s = number of items on the stack
- Therefore, total cost = min(s, k).

What is the worst case of a sequence of n `Push`, `Pop` and `Multipop` operations?

Using our existing methods of analysis we can identify a *loose bound.*:

- The most expensive operation is `Multipop`, potentially O(n).
- Therefore, potentially O(n^2) over n operations.

Aggregate Analysis

We can tighten this loose bound by aggregating the analysis over all n operations:

- Each object can only be popped once per time that it is pushed.
- There are at most n `Pushes`, so at most n `Pops`, including those in `Multipop`
- Therefore, total cost = O(n)
- Averaging over the n operations we get O(1) per operation.

This analysis shows O(1) per operation on average in a sequence of n operations without using any probabilities of operations.

See text for example of aggregate analysis of binary counting. An example of aggregate analysis of

dynamic tables is at the end of these notes.

Some of our previous analyses with indicator random variables have been a form of aggregate analysis, e.g., our analysis of the expected number of inversions in sorting, Topic 5 Notes.

Aggregate analysis treats all operations equally. The next two methods let us give different operations different costs, so are more flexible.

Accounting Method

Metaphor:

- View the computer as a coin operated appliance that requires one *cyber-dollar* (CY$) per basic operation.
- The banks are wary of making loans these days, so when an operation is to be performed we must have enough cyber-dollars available to pay for it.
- We are permitted to charge some operations more than they actually cost so we can save enough to pay for the more expensive operations.

Amortized cost = amount we charge each operation.

This differs from aggregate analysis:

- In aggregate analysis, all operations have the same cost.
- In the accounting method, different operations can have different costs.

When an operation is overcharged (amortized cost > actual cost), the difference is associated with *specific objects* in the data structure as *credit*.

We use this credit to pay for operations whose actual cost > amortized cost.

The credit must never be negative. Otherwise the amortized cost may not be an upper bound on actual cost for some sequences.

Let

- c_i = actual cost of ith operation.
- $\hat{c}_i$ = amortized cost of ith operation *(notice the 'hat')*.

Require $\sum_{i=1,n} \hat{c}_i \geq \sum_{i=1,n} c_i$ for all sequences of n operations. That is, the difference between these sums always ≥ 0: we never owe anyone anything.

Stack Example

Whenever we `Push` an object (at actual cost of 1 cyberdollar), we potentially have to pay CY$1 in the future to `Pop` it, whether directly or in `Multipop`.

To make sure we have enough money to pay for the `Pops`, we charge `Push` CY$2.

- CY$1 pays for the push
- CY$1 is prepayment for the object being popped (metaphorically, this CY$1 is stored "on" the object).

Since each object has CY$1 credit, the credit can never go negative.

operation	actual cost	amortized cost
PUSH	1	2
POP	1	0
MULTIPOP	$\min(k, s)$	0

The total amortized cost $\hat{c} = \sum_{i=1,n} \hat{c}_i$ for *any* sequence of n operations is an upper bound on the total actual cost $c = \sum_{i=1,n} c_i$ for that sequence.

Since $\hat{c} = O(n)$, also $c = O(n)$.

Note: we don't actually store cyberdollars in any data structures. This is just a metaphor to enable us to compute an amortized upper bound on costs.

Potential Method (optional this semester)

Instead of credit associated with objects in the data structure, this method uses the metaphor of *potential* associated with the *entire data structure*.

(I like to think of this as potential *energy*, but the text continues to use the monetary metaphor.)

This is the most flexible of the amortized analysis methods, as it does not require maintaining an object-to-credit correspondence.

Let

- D_0 = initial data structure
- D_i = data structure after ith operation
- c_i = actual cost of ith operation.
- $\hat{c}_i$ = amortized cost of ith operation.

Potential Function $\mathbf{\Phi}$: $D_i \rightarrow \Re$, and we say that $\Phi(D_i)$ is the **potential** associated with data structure D_i.

We define the amortized cost $\hat{c}_i$ to be the actual cost c_i plus the change in potential due to the ith operation:

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$

- If at the ith operation, $\Phi(D_i) - \Phi(D_{i-1})$ is positive, then the amortized cost c'_i is an *overcharge* and the potential of the data structure increases.
- On the other hand, if $\Phi(D_i) - \Phi(D_{i-1})$ is negative then c'_i is an undercharge, and the decrease of the potential of the data structure pays for the difference (as long as it does not go negative).

The total amortized cost across n operations is:

$$\sum_{i=1,n} \hat{c}_i = \sum_{i=1,n}(c_i + \Phi(D_i) - \Phi(D_{i-1})) = (\sum_{i=1,n} c_i) + (\Phi(D_n) - \Phi(D_0))$$

(The last step is taken because the middle expression involves a telescoping sum: every term other than D_n and D_0 is added once and subtracted once.)

If we require that $\Phi(D_i) \geq \Phi(D_0)$ for all i then the amortized cost will always be an upper bound on the actual cost no matter which ith step we are on.

This is usually accomplished by defining $\Phi(D_0) = 0$ and then showing that $\Phi(D_i) \geq 0$ for all i. (Note that this is a constraint on Φ, not on $\hat{c}$. $\hat{c}$ can go negative as long as $\Phi(D_i)$ never does.)

Stack Example

Define $\Phi(D_i)$ = number of objects in the stack.

Then $\Phi(D_0) = 0$ and $\Phi(D_i) \geq 0$ for all i, since there are never less than 0 objects on the stack.

Charge as follows (recalling that $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$):

operation	actual cost	$\Delta\Phi$	amortized cost
PUSH	1	$(s+1) - s = 1$ where s = # of objects initially	$1 + 1 = 2$
POP	1	$(s-1) - s = -1$	$1 - 1 = 0$
MULTIPOP	$k' = \min(k, s)$	$(s - k') - s = -k'$	$k' - k' = 0$

Since we charge 2 for each `Push` and there are O(n) Pushes in the worst case, the amortized cost of a sequence of n operations is $O(n)$.

Does it seem strange that we charge `Pop` and `Multipop` 0 when we know they cost something?

- Remember that this is just a way of counting the total cost over a sequence of operations more precisely.
- It is not a claim about the actual cost of a specific procedural call.
- Like with the accounting method, we are guaranteeing that we have just enough credit on hand to pay for the operations when they happen.
- The methods give a tight bound on amortized cost, but with much easier counting than if we had to reason about probability distributions, etc.

Application: Dynamic Tables

There is often a tradeoff between time and space, for example, with hash tables. Bigger tables give faster access but take more space.

Dynamic tables, like the Java Hashtable, grow dynamically as needed to keep the load factor reasonable.

Reallocation of a table and copying of all elements from the old to the new table is expensive!

But this cost is amortized over all the table access costs in a manner analogous to the stack example: We arrange matters such that table-filling operations build up sufficient credit before we pay the large cost of copying the table; so the latter cost is averaged over many operations.

A Familiar Definition

Load factor $\alpha = num/size$, where num = # items stored and $size$ = the allocated size of the table.

For the boundary condition of $size = num = 0$, we will define $\alpha = 1$.

We never allow $\alpha > 1$ (no chaining).

Insertion Algorithm

We'll assume the following about our tables. (See Exercises 17.4-1 and 17.4-3 concerning different assumptions.):

When the table becomes full, we double its size and reinsert all existing items. This guarantees that $\alpha \geq 1/2$, so we are not wasting a lot of space.

```
Table-Insert (T,x)
1   if T.size == 0
2       allocate T.table with 1 slot
3       T.size = 1
4   if T.num == T.size
5       allocate newTable with 2*T.size slots
6       insert all items in T.table into newTable
7       free T.table
8       T.table = newTable
9       T.size = 2*T.size
10  insert x into T.table
11  T.num = T.num + 1
```

Each *elementary insertion* has unit actual cost. Initially $T.num = T.size = 0$.

Aggregate Analysis of Dynamic Table Expansion

Charge 1 per elementary insertion. Count only elementary insertions, since all other costs are constant per call.

c_i = actual cost of ith operation.

- If the table is not full, $c_i = 1$ (for lines 1, 4, 10, 11).
- If full, there are $i - 1$ items in the table at the start of the ith operation. Must copy all of them (line 6), and then insert the ith item. Therefore $c_i = i - 1 + 1 = i$.

A sloppy analysis: In a sequence of n operations where any operation can be $O(n)$, the sequence of n operations is $O(n^2)$.

This is "correct", but inprecise: we rarely expand the table! A more precise account of c_i:

$$c_i = \begin{cases} i & \text{if } i-1 \text{ is exact power of } 2, \\ 1 & \text{otherwise.} \end{cases}$$

Then we can sum the total cost of all c_i for a sequence of n operations:

TABLE-INSERT(T, x)

```
1   if T.size == 0
2       allocate T.table with 1 slot
3       T.size = 1
4   if T.num == T.size
5       allocate new-table with 2 · T.size slots
6       insert all items in T.table into new-table
7       free T.table
8       T.table = new-table
9       T.size = 2 · T.size
10  insert x into T.table
11  T.num = T.num + 1
```

$$\text{Total cost} = \sum_{i=1}^{n} c_i$$
$$\leq n + \sum_{j=0}^{\lfloor \lg n \rfloor} 2^j$$
$$= n + \frac{2^{\lfloor \lg n \rfloor + 1} - 1}{2 - 1}$$
$$< n + 2n$$
$$= 3n$$

$$\sum_{k=0}^{n} x^k = \frac{x^{n+1} - 1}{x - 1}$$

Explain: Why the n? What is the summation counting? Why does the summation start at $j = 0$? Why does it end at $j = \lg n$?

Therefore, the amortized cost per operation = 3: we are only paying a small constant price for the expanding table.

Accounting Method Analysis of Dynamic Table Expansion

We charge CY$3 for each insertion into the table, i.e., the amortized cost of the i-th insertion as $\hat{c}_i = 3$. The actual cost of the i-th insertion is as before:

$$c_i = \begin{cases} i & \text{if } i - 1 \text{ is exact power of } 2, \\ 1 & \text{otherwise.} \end{cases}$$

We must show that we never run out of credit. It's equivalent to proving the following Lemma.

Lemma: *For any sequences of n insertions* $\sum_{i=1,n} \hat{c}_i \geq \sum_{i=1,n} c_i$.

Proof: By using the same equation as above, we know that for any n: $\sum_{i=1,n} c_i \leq 3n$. By the definition of ĉ, we know that: $\sum_{i=1,n} \hat{c}_i = 3n$. The lemma follows. ∎

Intuitively, we charge CY$3 for inserting some item x into the table. Obviously, we are overcharging for each *simple* insertion which costs only CY$1. However, the overcharging will provide enough credit to pay for future copying of items when the table becomes full:

- The first CY$1 pays for the actual cost of inserting x.
- The second CY$1 will pay for the cost of copying x into a new table the next time table becomes full.

Note, the table might need to be expanded more than once after x is inserted, so x might need to be copied more than once. Who will pay for future copying of x? That's where the third CY$1 comes in:

- The third CY$1 will pay for the cost of copying some other item currently in the table that had been copied at least once before.

Let's see why CY$3 is enough to cover all possible future exansions and copying associated with them.

Suppose the capacity of the table is m immediately after an exansion. Then it holds $m/2$ items and no item in the table contains any credits. For any insertion of an item x we charge CY$3. CY$1 pays for the actual insertion of x; we place CY$1 credit on x to pay for the cost of copying it in the future, and we place CY$1 credit on some other item in the table that does not have any credit yet. We will have to insert $m/2$

items before the next expansion of the table. Therefore, by the time the table will get expanded next time (and, consequently, items need to be copied), every item of the table will have CY$1 credit associated with it and this credit will pay for copying that item.

The text also provides the potential analysis of table expansion.

This analysis assumed that the table never shrinks. See section 17.4 for an analysis using the potential method that covers shrinking tables.

Other Examples

Here are some other algorithms for which amortized analysis is useful:

Red-Black Trees

An amortized analysis of Red-Black Tree restructuring (Problem 17-4 of CLRS) improves upon our analysis earlier in the semester:

- Any sequence of m `RB-Insert` and `RB-Delete` operations performs $O(m)$ structural modifications (rotations),
- Thus each operation does **O(1) structural modifications on average**, regardless of the size of the tree!
- An operation still may need to do $O(\lg n)$ recolorings, but these are very simple operations.

Self-Organizing Lists

- Self-organizing lists use a ***move-to-front heuristic***: Immediately after searching for an element, it is moved to the front of the list.
- This makes frequently accessed items more readily available near the front of the list.
- An amortized analysis (Problem 17-5) shows that the heuristic is no more than 4 times worse than optimal.

Splay Trees

- Splay trees are ordinary binary search trees (no colors, no height labels, etc.)
- After every access (every insertion, deletion, or search), the element operated on (or its parent in the case of deletion) is moved towards the top of the tree.
- This movement uses three ***splaying*** operations called "zig", "zig-zig" and "zig-zag".
- Although in the worst case a splay tree can degenerate into an $O(n)$ linked list, amortized analysis shows that the expected case is $O(\lg n)$
- Randomization can be used to make the worst case very unlikely.
- If a single element is accessed at least $m/4$ times where m is the number of operations, then the amortized running time of each of these accesses is $O(1)$.
- Thus, splay-trees self-organize to provide fast access to frequently accessed items.
- This makes them good for locality of reference in memory, but multithreaded access must be implemented carefully.

To Be Continued

Amortized analysis will be used in analyses of

- Graph search (Topic 14, Ch. 22)

- Disjoint set operations (Topic 16, Ch. 21)
- Dijkstra's Algorithm for Shortest Paths (Topic 18, Ch. 24)

Dan Suthers & Nodari Sitchinava
Last modified: Mon Mar 30 13:17:21 HST 2015
Images are from the instructor's material for Cormen et al. Introduction to Algorithms, Third Edition.