Massachusetts Institute of Technology
Debayan Gupta, Aleksander Madry, and Bruce Tidor

6.046J/18.410J
Handout 3

Recitation 3: Amortized and Competitive Analysis

1 Amortized Analysis

1.1 Recap

The main idea behind amortized analysis is to give a tight upper-bound for a sequence of operations that is better than using worst-case analysis on individual operations.

For a given data-structure D, let $0, 1, \ldots n-1$ be a sequence of n operations on the data structure. In amortized analysis, we are not interested in the cost of any individual operation, but rather the total cost of the entire sequence of operations. We can then average over the number of operations to get the amortized cost per operation. Hence an operation has amortized cost $T(n)$ if any sequence of k operations costs at most $kT(n)$.

Note: A common misconception is to conflate amortized analysis for average-case analysis. This is not true. Amortized analysis is in fact a *worst-case* analysis, but bounds a sequence of operations rather than an individual operation.

The three most common methods for amortized analysis are the **Aggregate** Method, the **Accounting** Method, and finally the most powerful, **Potential** Method. We'll examine how each of these methods can be used to analyze the performance of two specific data structures.

1.2 Data Structures

We'll be referencing these two problems throughout the recitation:

1.2.1 Queue using Two Stacks

Recall that a Stack has operations $PUSH(x)$ which adds x to the top of the stack, and $POP()$ which removes and returns the items at the top of the stack (LIFO - last in first out). A Queue supports operations $ENQUEUE(x)$ which adds x to the back of the queue and $DEQUEUE()$ which removes and returns the item at the front of the queue (FIFO - first in first out).

Given two stacks s_1 and s_2, we will implement a Queue data structure as follows:

$ENQUEUE(x)$: Push x onto s_1
$DEQUEUE()$: If s_2 empty, for all items in s_1, pop them out of s_1 and push them onto s_2. Then pop and return item from s_2.

Worst-Case: The worst-case cost of an $ENQUEUE$ operation is clearly $O(1)$ since it makes only 1 push. The worst-case cost of a $DEQUEUE$ operation is $O(n)$, where n is the total number of elements, because it may have to perform $O(n)$ pops to transfer items from s_1 to s_2.

1.2.2 Binary Counter

This data structure is an m-bit binary counter which counts from 0 to $2^m - 1$. It is represented by a list of m binary digits which are all initially set to 0: $[0, 0, 0 \ldots, 0]$. The data structure supports only one operation, $INCREMENT()$ which increments its value by 1. The operation works as follows:

$INCREMENT()$: Start from the right most bit and scan to the left flipping each bit from 1 to 0. When encountering the first 0, flip it to a 1 and terminate. (Note that this correctly increments the binary counter by 1).

Worst-Case: The worst-case cost of the $INCRMENT()$ operation is clearly $O(m)$ since the counter may be in a state $[0, 0, 1, 1, 1, \ldots, 1]$ where there are $O(m)$ 1's to flip to 0

1.3 Aggregate Analysis Method

In Aggregate Analysis, we compute the total cost of n operations by simply adding them up. We can then divide this cost by n to get the amortized cost. This is usually the most straightforward, but often infeasible, method for amortization.

1.3.1 Queue using Two Stacks

Imagine a sequence of n operations (any mixture of $ENQUEUE$'s and $DEQUEUE$'s) on this data structure. Naively, we can bound the cost of this sequence of operations by $O(n^2)$ by looking at the worst-case cost of each operation. However, let's count the number of underlying $PUSH$ and POP operations. Since an element must be added to the queue in order to be removed, the number of $DEQUEUE$ operations is at most the number of $ENQUEUE$ operations. Additionally, once an element gets removed from the Queue, there can be no more operation costs incurred on it.

Hence, for each element x that gets added to the Queue, it costs one $PUSH$ to add it to s_1, one $POP + PUSH$ to move it from s_1 to s_2 and then finally one POP to remove it from s_2. Therefore, for each element that can ever be added to the Queue, there can be at most a constant number of

$PUSH$ and POP operations incurred on it. And since there can be at most n elements added to the Queue, the total cost of any sequence of n operations is $O(n)$.

Therefore, the amortized cost is $O(1)$.

1.3.2 Binary Counter

Let's count exactly how many time each bit gets flipped in the process of n $INCREMENT$ operations. Te least significant bit gets flipped with each $INCREMENT$ call, so it gets flipped a total of n times. The second-least significant bit gets flipped with every other call, so that's a total of $\frac{n}{2}$ flips, and so on. Hence the total number of bit flips is $n + \frac{n}{2} + \frac{n}{4} + ... + \frac{n}{2^{m-1}} \leq 2n$. Therefore, the total cost of all n operations is $O(n)$, and the amortized cost is $O(1)$.

1.4 Accounting Method

The Accounting method works by paying in advance the cost of later operations. Intuitively, you can think of it as a savings system for a bank—every time you do a low-cost operation, you put a few extra coins in the bank (paying in advance), and use them to cover the cost of an occasional expensive operation. Remember, this is not an algorithm, but rather an analysis technique to measure exactly how much we spend over a sequence of operations.

Mathematically, for each operation i, let c_i be the actual cost of the operation. We assign an amortized cost $\hat{c}_i$ (coins) to each operation so that the total amortized cost is an upper bound on the total actual cost, $\sum_i \hat{c}_i \geq \sum_i c_i$.

Note: Make sure your accounting analysis maintains a nonnegative bank balance for any possible sequence of operations (you can't spend what you don't have).

1.4.1 Queue using Two Stacks

Let us assign an amortized cost of 4 coins for the $ENQUEUE$ operation. We will show how doing so leads to an amortized cost of 0 coins for the $DEQUEUE$ operation.

When an element x is added to the queue, it uses 1 coin to pay for the initial $PUSH$ onto s_1. It will store the 3 extra coins for later. Two of these coins will be used if this coin is ever moved from s_1 to s_2 and it will have 1 coin left over. This last coin will be used if it is ever popped from s_2 due to a $DEQUEUE$ call. Hence, this ensures that whenever $DEQUEUE$ is called, the item being popped will have enough credit so that the $DEQUEUE$ operation is free.

Therefore, since we pay at most 4 coins for any operation, the amortized cost is $O(1)$.

1.4.2 Binary Counter

Let's assign a cost of 2 coins for flipping a 0 to a 1 (setting a bit). Hence when the $INCREMENT$ operation is called, whenever it sets a bit from 0 to 1, it will use the first coin to pay for the flip and it will place the second coin on the bit to be used for later. Note that since all the bits start at 0, any bit that is a 1 will have a coin on it. It will use it to pay for itself if it ever gets reset to 0. This way, all the costs are always paid for any sequence of $INCREMENT$ operations. Since we pay at most 2 coins, the amortized cost of $INCREMENT$ is $O(1)$.

1.5 Potential Method

The Potential method is the most powerful of the three amortization methods. In this method, you can think of a "potential energy'' associated with your data structure.

Suppose we perform n operations on our data structure D. We define D_0 to be the initial data structure. Then, for $i = 1, 2, .. n$, let D_i be the state of the data structure immediately after performing operation i on D_{i-1}. We define a potential function $\Phi : \{D_0, D_1, .. D_n\} \to \mathbb{R}$ to be a function which maps each D_i to a real number $\Phi(D_i)$.

Let c_i be the actual cost of operation i. Then, for a given potential function Φ, we define the amortized cost $\hat{c}_i$ of operation i as follows,

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

Or more concisely, $\hat{c}_i = c_i + \Delta\Phi$. Using this definition, we can compute the total amortized cost of n operations as follows,

$$\begin{aligned}\sum_{i=1}^{n} \hat{c}_i &= \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1})) \\ &= \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0)\end{aligned}$$

Therefore, as long as for our given potential has $\Phi(D_n) \geq \Phi(D_0)$ the total amortized cost gives an upper bound on the total actual cost. In practice, since n may not be known before hand, we require $\Phi(D_i) \geq \Phi(D_0)$ for all i. This way, at any given time, the total amortized cost is an upper bound on the total actual cost incurred so far. To further simplify our work, we typically set $\Phi(D_0) = 0$ and simply ensure $\Phi(D_i) \geq 0$ at any time.

The biggest challenge in the potential method is finding a potential function Φ which has small $\Phi(D_n) - \Phi(D_0)$ so that it gives a tight bound on the total actual cost. While this is generally a non trivial problem, the following intuition is helpful.

Intuition: Choose your potential Φ so that $\Delta\Phi$ decreases by a large amount when an expensive operation is performed on your data structure.

We will now apply the potential method to our examples we have been using so far.

1.5.1 Queue using Two Stacks

We define our potential to be $\Phi(D_i) = 2|s_1^i|$, two times the number of elements in s_1 immediately after the i-th operation (s_1^i to refers to s_1 immediately after the i-th operation). The intuition for this choice is that the most expensive operation in our queue is when $DEQUEUE$ has to move all the items out of s_1 into s_2. When this happens, the number of elements in s_1 decreases significantly, therefore decreasing our potential.

First, we must ensure that Φ is a valid potential function. $\Phi(D_0) = 0$ because s_1 starts out empty. $\Phi(D_i) \geq 0$ for any i since the number of items in s_1 can never be negative. This ensures that the amortized cost of n operations is according to Φ is indeed an upper bound on the total actual cost of n operations.

Let's first analyze the $ENQUEUE$ operation. We know that the actual cost is $c_i = 1$ since it only makes one call to $PUSH$ onto s_1. Since the number of elements in s_1 increases by 1, we have the following amortized cost for $ENQUEUE$:

$$\hat{c}_i = c_i + \Delta\Phi = 1 + 2 = 3$$

Now let's analyze $DEQUEUE$. There are two cases for $DEQUEUE$; the first is the easy case where s_2 is non-empty, which gives us an amortized cost of $\hat{c}_i = c_i + \Delta\Phi = 1 + 0 = 1$ since the size of s_1 does not change. In the second case (when s_2 is empty), the actual cost of $DEQUEUE$ is $2|s_1^i| + 1$ since there is a cost of one POP our of s_1 and one $PUSH$ onto s_2 for all items in s_1, and then finally a single POP from s_2. However, when this happens, the number of elements in s_1 decreases by $|s_1^i|$, and our amortized cost is

$$\hat{c}_i = c_i + \Delta\Phi = 2|s_1^i| + 1 - 2|s_1^i| = 1$$

Hence, in both cases, the amortized cost of $DEQUEUE$ is also $O(1)$.

1.5.2 Binary Counter

The most expensive operation is when there is a large number of leading 1's in the counter and $INCREMENT$ has to set them all to 0 before finally flipping a 0 to a 1 and terminating. Therefore, a good guess for a potential function would be the number of 1s in the counter.

We first ensure that Φ is a valid potential function: $\Phi(D_0) = 0$ because our counter starts at 0, and at any given time, $\Phi(D_i) \geq 0$ since our counter only increments. Therefore, the total amortized cost according to Φ gives an upper bound on the total actual cost after n increments.

Suppose the number of leading 1's before operation i was k. Then the actual cost of the i-th $INCREMENT$ is $c_i = k + 1$ since it has to flip all k bits to 0 and then the final bit from 0 to 1. However, in this process the number of 1's in our binary counter goes down by $k - 1$. Therefore, the amortized cost for $INCREMENT$ is,

$$\hat{c}_i = c_i + \Delta\Phi = k + 1 - (k - 1) = 2 = O(1)$$

2 Online Algorithms and Competitive Analysis

2.1 Recap

Let $R = \{r_1, r_2, ...\}$ be a sequence of requests that is processed by an algorithm. An **online** algorithm A is one that processes these requests with no knowledge of future requests (i.e. when processing request r_i, the algorithm does not know any r_j where $j > i$). To evaluate the performance of an online algorithm, we often compare it with an optimal **offline** algorithm OPT which knows the entire sequence of requests at all times. This is known as **competitive analysis**, and we say an algorithm is α-competitive if

$$C_A(R) \leq \alpha \cdot C_{OPT}(R) + c$$

where $C_A(R)$ denotes the cost of algorithm A on request sequence R, and c is some constant. (The smaller the competitive ratio α, the better the algorithm.)

In lecture we saw a competitive analysis of the MTF (move-to-front) self-organizing list. Here, we will explore a different example: the LRU paging algorithm.

2.2 Paging Problem and LRU Algorithm

In the paging problem, we have a two-level memory system composed of a small fast memory unit (RAM) and a large slow memory unit (disk). Each r_i requests a "page" (a chunk of virtual memory) and the fast memory unit can only hold up to k pages. If a requested page is not in fast memory, we incur a "page fault" and must fetch the page from disk. Our cost in this case is the number of page faults.

Formally, each r_i is an arbitrary integer (representing a page) and we are given a set S (initially empty) that can hold up to k distinct integers. If set S contains r_i at the time of the request, then

we incur no cost. If set S does not contain r_i, then we must insert r_i into S (possibly evicting an element in S), incurring a cost of 1.

The LRU (least recently used) algorithm is straightforward: upon page-faulting and inserting a new element into S, evict (if need be) the element in S that was accessed farthest in the past. It turns out that this simple strategy is k-competitive, which is also a lower bound on *any* online paging algorithm (i.e. we can't do better than k-competitive).

2.3 LRU is k-competitive

Some Initial Thoughts: In competitive analysis, we want to compare our algorithm to the *best possible* offline algorithm. Because it is difficult to conceptualize what the best possible algorithm (OPT)[1] would do across an arbitrary request sequence R, let's break the request sequence into chunks where *any* algorithm is guaranteed to fault some number of times. Then, if we can place an upper bound on the faults incurred by our algorithm (A) in any chunk, then we can start to compare OPT and A and possibly determine a competitive ratio.

Specifically, let's break R into contiguous chunks $Q_0, Q_1, Q_2, ...$ such that our LRU algorithm makes at most k faults in Q_0 and exactly k faults in the remaining chunks. Our goal is to prove the following:

Lemma. OPT will make at least 1 fault in each chunk Q_i where $i \geq 1$.

By proving this lemma, we show that for every k faults the LRU algorithm makes, the optimal algorithm must make at least 1. Therefore, the LRU algorithm is k-competitive.

Proof of Lemma:

Let p be the last distinct request before some Q_i. If we can show that Q_i contains requests for k distinct pages that are not p, then by the pidgeonhole principle, any algorithm must make at least 1 page fault in Q_i.

Case 1: Our LRU algorithm faulted on k distinct pages in Q_i.

If none of the faults are p, then we are done. However, if one of them is p, then our algorithm must have evicted p at some point. Since p was the most recently used at the beginning of Q_i, our LRU policy would only have evicted p if there were k different requests between the beginning of Q_i and the eviction of p. Therefore, Q_i contains k distinct pages that are not p.

Case 2: Our LRU algorithm faulted on some page x more than once in Q_i.

After the first fault, x would be the most recently used, and by our LRU policy, there would need to be k requests that are not x between the beginning of Q_i and the eviction of x. This gives $k + 1$ distinct requests in Q_i, at least k of which are not p.

[1] Actually, in the case of the paging problem, the "farthest-in-future" eviction policy is known to be the optimal offline solution, but let's ignore that for now