Amortized Analysis

Jie Wang[1]

University of Massachusetts Lowell
Department of Computer Science

[1] I thank Dr. Zachary Kissel of Merrimack College for sharing his lecture notes with me; some materials presented here are borrowed from his notes.

Overview

1. When measuring time complexity for an algorithm we often measure its worst-case asymptotic runtime.
   - Sometimes we may measure average runtime when an underlying probability distribution on instances is known.
2. When measuring a sequence of operations bounded by a certain mechanism, for example, the number of adds and subtracts must be balanced, it is natural to use amortized analysis.
   - For our purposes we will focus on a sequence of operations performed on data structure, e.g., stack, queue, hash table, counters, etc.

Ideas of Amortized Analysis

Given a sequence of operations on a data structure D, we spread out the cost (runtime) over the sequence.

- Divide some of the cost of inefficient operations across the more efficient operations.
  - This approach would make sense if the inefficient operations occur relatively infrequently within the sequence of operations to be analyzed.
  - In other words, the cost of infrequent operations may be "shared" by efficient operations, providing an average cost of each operation.
- Using the business definition of amortize is another way to think about this process.
  - For example, loans are traditionally amortized, so that one pays the same amount per month over the life of the loan.
  - In reality each month different amounts of money are going to the principal and the interest.

Three Methods

Amortized analysis is an analysis technique (In reality we cannot divide time among operations!).

- i.e., there is no change in computational model.

There are three main methods people use to do amortized analysis:

1. Aggregate method
   - Compute a simple mean.
2. Accounting method
   - Maintain a bank account of credits to pay for the operations.
3. Potential method
   - Give data structures potential energy similar to the notion of potential energy in physics.

Aggregate Analysis

- This is perhaps the most intuitive form of amortized analysis.
- In aggregate analysis we show that a sequence of n operations takes worst-case time $T(n)$ in total.
  - To get the amortized time, simply compute $T(n)/n$, the average time per operation.
  - Note that we assign the *same* cost to every operation.
- Let's try out this method on two data structures:
  1. A stack.
  2. A binary counter

Stack Operations

First consider the main operations on a stack: Push and Pop

- Push and Pop are local operations, and so the worst-case time is $\Theta(1)$.
- Given an n operation sequence of Push and Pop operations, the worst-case time $T(n) = \sum_{i=1}^{n} \Theta(1) = \Theta(n)$.
- Thus, the amortized time per operation is $\Theta(1)$

Now consider a more expensive operation on a stack: Multipop.

- Let S denote a stack and $|S|$ its size.
- Consider three operations: Push, Pop, and Multipop.
  - Multipop takes an argument k (the number of items to pop) and executes the Pop operation on the stack k times:
  - If $k > |S|$, then execute Pop until the stack is empty.
- The worst-case runtime of Multipop is $\min\{k, |S|\}$.

Pops Cannot Exceed Pushes

- We should capitalize more on what we know about a sequence of n stack operations starting from an *empty* stack.
- Observation: An item can only be popped from the stack once.
- Claim: A sequence of n operations, on an initially empty stack, takes $O(n)$ time.
  - The number of $\textsc{Pop}$ operations (including $\textsc{Multipop}$ calls) is at most the number of $\textsc{Push}$ operations.
  - There are, at most, n $\textsc{Push}$ operations so there are at most n stack pops performed for a sequence of n operations.
- This means the amortized cost per operation is $O(n)/n = \Theta(1)$.

Binary Counter

A binary counter has k-bits for a fixed k, similar to the 8-digit odometer in your car.

- INCREMENT is the only operation.
- Since we are restricted to k-bits, the counter can "roll-over" (overflow).

We implement the binary counter as an array of size k:

```
INCREMENT(A[1..k])
i = 1
while i ≤ k  and  A[i] == 1
      A[i] = 0
      i = i + 1
if i ≤ k
      A[i] = 1
```

Analysis

Set the array to 0s initially. Want to find out the runtime of executing n increments.

- **Observation**: While increment $i+1$ causes a number of bits to flip, not every bit flips every time.
- Consider the following binary counter sequence ($k = 4$) with $A[1]$ being in the right-most column:

$$\begin{array}{cccc}
0 & 0 & 0 & 0 \\
0 & 0 & 0 & \mathbf{1} \\
0 & 0 & \mathbf{1} & \mathbf{0} \\
0 & 0 & 1 & \mathbf{1} \\
0 & \mathbf{1} & \mathbf{0} & \mathbf{0} \\
0 & 1 & 0 & \mathbf{1} \\
\vdots & \vdots & \vdots & \vdots \\
1 & 1 & 1 & \mathbf{1}
\end{array}$$

Bit Flips

- In a sequence of n operations, $A[1]$ flips every operation.
- $A[2]$ flips every other operation; i.e., $\lfloor \frac{n}{2} \rfloor$ in n operations.
- $A[3]$ flips every $2^2 = 4$ operations.
- Similarly, $A[i]$ flips every 2^{i-1} operations. Thus,

$$
\begin{aligned}
T(n) &= \sum_{i=1}^{k} \left\lfloor \frac{n}{2^{i-1}} \right\rfloor = \sum_{i=0}^{k-1} \left\lfloor \frac{n}{2^i} \right\rfloor && \text{(Change of indices)} \\
&\leq \sum_{i=0}^{\infty} \frac{n}{2^i} && \text{(Upperbound series)} \\
&= n \sum_{i=0}^{\infty} \left(\frac{1}{2} \right)^i && \text{(Geometric series)} \\
&= 2n \in O(n)
\end{aligned}
$$

- Taking an average to get $O(n)/n = \Theta(1)$ amortized time per INCREMENT operation.

Accounting Method

A business way of thinking about amortized analysis:

- Assign every operation an amortized cost in credits.
    - This may be more than or less than the actual cost.
    - This is often referred to as the *charge* for the operation.
- We put any overages in a "piggy bank".
    - An overage occurs when the amortized cost of an operation exceeds the actual cost of an operation.
- If we don't have enough credits, from the amortized cost, to cover the actual cost of the operation we have a deficit.
    - To cover the deficit, we extract the required credits from the piggy bank.
- How to assign amortized costs to operations is important.
    - Need to guarantee that regardless of the sequence of operations the amortized costs will be an upper bound of the actual cost.

Example

Consider the stack that supports MULTIPOP.

- To use the accounting method we must assign an appropriate amortized cost to each operation.

Operation	**Actual Cost**	**Amortized Cost**
PUSH	1	2
POP	1	0
MULTIPOP	$\min\{k, \lvert S \rvert\}$	0

- That is, we think ahead: For each element pushed into the stack, we will also budget the cost of popping it to the push, leaving the cost of element-popping for free.

Does It Work?

- It does.
- One can only pop elements that have been pushed onto the stack.
- Regardless of which MULTIPUSH and PUSH is used, removing an item requires constant time.
- Each item pushed to the stack charges 1 credit, which leaves 1 credit for the piggy bank to pop the item being pushed.
  - Every pop require us to take one credit out of the piggy bank.
- This means a sequence of n operations costs $\Theta(n)$, implying an amortized cost of $\Theta(1)$ for every operation.

Binary Counter

- Need to think of the two sub-operations for INCREMENT:
  1. set bit i; i.e., set bit i to 1.
     - The actual cost of this operation is 1.
  2. clear bit i; i.e., set bit bit i to 0.
     - The actual cost of this operation is 1.
- What can we do for amortized cost?
  - The set operation will have amortized cost 2, one for setting it, and one for clearing it.
    - Note: A bit cannot be cleared without setting it first.
  - The clear operation will have amortized cost 0.

Binary Counter Continued

- Setting a bit is charged 2 credits, one for the operation and one goes to the piggy bank.
- Clearing a bit uses the one credit overcharged for setting the bit from the piggy bank.
- Since we can never have a negative number of 1s, the balance of the piggy bank is never below zero.
- The cost of clearing the bits in the **while** loop for INCREMENT is covered by the money in the piggy bank.
- the cost to set a bit at the index of INCREMENT requires a cost of 2.
- Taking these fact together we arrive at a cost of $O(n)$ for n INCREMENT operations thus the average cost of an INCREMENT operation is $\Theta(1)$.

Potential Method

Look at a data structure as a whole.

- Assign a *potential energy* to a given state of the data structure.
- Every operation puts the data structure into a different state that either increases or decreases the potential energy.
- Denote by D_i the state of a data structure after the i^{th} operation.
- Denote by $\Phi(D_i)$ the potential energy of the data structure in state D_i.
    - This is a real value.
    - The function Φ is referred to as the potential function.
- Denote the amortized cost of the i^{th} operation by $\hat{c}_i$.
    - Formally, $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$, where c_i is the actual cost of the i^{th} operation.

The Total Amortized Cost for n Operations

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

- If $\Phi(D_n) \geq \Phi(D_0)$, then the total amortized cost is an upper bound of the actual cost.
- In practice, $\Phi(D_i) \geq \Phi(D_0)$ for all i.
  - This guarantees we store up enough energy in advance.
  - Traditionally, $\Phi(D_0) = 0$.

Potential Functions

There are a wide range of potential functions that can be used for the analysis.

- The choice really depends on how tight of a bound we want.

Let's look at the stack again.

- Define the potential function to be the number of items in stack S.
  - This naturally gives us $\Phi(D_0) = 0$.
  - Notice that stacks cannot have negative potential.
    - $\Phi(D_i) \geq 0$ for all i.
    - The fact $\Phi(D_i) \geq 0$ for all i means that $\Phi(D_i) \geq \Phi(D_0)$ for all i. Thus conserving sufficient energy.

Amortized Cost of Operations

- PUSH has amortized cost 2:
  - The actual cost c_i is 1.
  - The change in potential is a difference of 1.
    - $\Phi(D_i) - \Phi(D_{i-1}) = (|S| + 1) - |S| = 1.$
- MULTIPOP has amortized cost 0:
  - The actual cost of the operation C_i is $\min\{|S|, k\}$.
  - The change in potential is a difference of $-\min\{|S|, k\}$.
    - $\Phi(D_i) - \Phi(D_{i-1} = (|S| - \min\{|S|, k\}) - |S| = -\min\{|S|, k\}.$
- POP has amortized cost 0:
  - The actual cost c_i is 1.
  - The change in potential is a difference of -1.
    - $\Phi(D_i) - \Phi(D_{i-1}) = (|S| - 1) - |S| = -1.$

Binary Counter

- Let $\Phi(D_i)$ be the number of 1s in the counter after operation i.
- Then $\Phi(D_0) = 0$ since we start the counter with 0s.
- Let t_i be the number of cleared bits after operation i.
  - This means the actual cost c_i is $t_i + 1$ since we have to set one bit.
- Denote by b_i the number of 1s after the i-th INCREMENT operation.
  - If $b_i = 0$ all bits were cleared.
    - This implies $b_{i-1} = t_i = k$ (only when the binary counter was full can this happen).
    - Note: $b_i = b_{i-1} - t_i = 0$.
  - If $b_i > 0$, then
    - $b_i = b_{i-1} - t_i + 1$

Binary Counter Continued

- The potential difference for the i-th INCREMENT operation is $1 - t_i$:
  - $\Phi(D_i) - \Phi(D_{i-1}) = (b_{i-1} - t_i + 1) - b_{i-1} = 1 - t_i$.
- The amortized cost of INCREMENT is $(t_i + 1) + (1 - t_i) = 2$.
- If one wanted, one could use the potential method to argue that the bound holds even if the counter does *not* start at zero.