Notes on Amortized Analysis

Geoffrey Matthews

May 15, 2018

Amortized analysis

- Analyze a *sequence* of operations on a data structure.
- **Goal:** Show that although some operations may be expensive, *on average* the cost per operation is small.
- Average is not over a distribution of inputs, but over a sequence of operations.
- No probability is involved: *Average* cost in the *worst* case.
- We look at three methods of calculating:
  1. aggregate analysis
  2. accounting method
  3. potential method
- And two simple examples:
  1. stack with multipop
  2. binary counter
- And a more interesting example:
  - dynamic tables

Stack operations

$\textsc{Push}(S, x)$: $O(1)$
$\textsc{Pop}(S)$: $O(1)$

$\textsc{Multipop}(S, k)$

```
while S is not empty and k > 0
    POP(S)
    k = k - 1
```

(a)	(b)	(c)
top → 23		
17		
6		
39		
10	top → 10	
47	47	

$\textsc{Multipop}(S, 4)$ $\textsc{Multipop}(S, 7)$

Running time of Multipop:

- Linear in # of Pop operations.
- Let each Push/Pop cost 1.
- # iterations of **while** loop is $\min(s, k)$
  - where $s = \#$ of objects in stack.
- Total cost $= \min(s, k)$

Worst-case analysis without amortization

- Sequence of n Push, Pop, and Multipop operations.
- May have up to n Push operations.
- So worst-case there are n items on the stack.
- Therefore, worst-case cost of a Multipop operation is $O(n)$.
- Have n operations, each of which could be Multipop.
- Therefore, worst-case cost of sequence of n operations is $O(n^2)$.

Something wrong with worst-case analysis

- There's clearly something wrong with this analysis.
- What is actual worst-case number of PUSHs and POPs as a function of n?
- But how can we get a more accurate worst-case analysis?
- We need to consider how the operations interact with each other.
- We need to keep an account of how much time is spent in each one, because that affects the time spent in the others.

Aggregate analysis

- **Observations**
  - Each object can be popped only once per time that it's pushed.
  - Have $\leq n$ PUSHs, therefore $\leq n$ POPs, including those in MULTIPOP.
  - Therefore, total cost $= O(n)$.
  - Average over n operations is $= O(1)$ per operation on average, including those in MULTIPOP.
- This is called **aggregate analysis**.
  - No probability involved.
  - Showed worst-case $O(n)$ for entire sequence.
  - Therefore, $O(1)$ per operation on average.

Binary counter

Counter value	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	Total cost
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11
8	0	0	0	0	1	0	0	0	15
9	0	0	0	0	1	0	0	1	16
10	0	0	0	0	1	0	1	0	18
11	0	0	0	0	1	0	1	1	19
12	0	0	0	0	1	1	0	0	22
13	0	0	0	0	1	1	0	1	23
14	0	0	0	0	1	1	1	0	25
15	0	0	0	0	1	1	1	1	26
16	0	0	0	1	0	0	0	0	31

- Bits that flip upon increment shaded.
- Total cost of flipping bits at right.
- Total cost always less than twice number of increments.

Binary counter

- k-bit binary counter $A[0..k-1]$ of bits.
- $A[0]$ is the least significant bit.
- Counts upward from 0.
- Value of counter is
$$\sum_{i=0}^{k-1} A[i] \cdot 2^i$$
- Initially counter is 0, so $A[0..k-1] = 0$.
- To increment, add 1 (mod 2^k):

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

Worst case analysis of binary counter

- Each call could flip k bits.
- n increments is $O(nk)$.

Aggregate analysis of binary counter

- Not every bit flips every time.

bit	flips how often	times in n INCREMENTS
0	every time	n
1	$1/2$ the time	$\lfloor n/2 \rfloor$
2	$1/4$ the time	$\lfloor n/4 \rfloor$
	$\vdots$	
i	$1/2^i$ the time	$\lfloor n/2^i \rfloor$
	$\vdots$	
$i \geq k$	never	0

Total number of flips

$$
\begin{aligned}
\sum_{i=0}^{k-1} \left\lfloor n/2^i \right\rfloor &< n \sum_{i=0}^{k-1} (1/2)^i \\
&= n \frac{(1/2)^k - 1}{1/2 - 1} \\
&= n \frac{1 - (1/2)^k}{1 - 1/2} \\
&< n \left(\frac{1}{1/2} \right) \\
&= 2n
\end{aligned}
$$

- n INCREMENTs costs $O(n)$.
- Average cost per operation $O(1)$.

Accounting Method and Potential Method

- Aggregate method works when we can add up all operations.
- More complex operations need a more sophisticated method.
- Two approaches:
- **Accounting method:**
  - assign charges to each operation
  - some operations charged more than they cost
  - others, charged less, can use accrued credit
- **Potential method:**
  - prepaid work is "potential energy"
  - energy is assigned to data structures as a whole
  - some operations increase potential energy
  - some operations can release potential energy to reduce costs
  - most flexible of the amortized analysis methods

Accounting method

- **Amortized cost** = amount we charge
- Amortized cost must always be $\geq$ actual cost
- When amortized cost $>$ actual cost, store the difference *on specific objects* in the data structure as **credit**.
- When we have credit, we have accounted for expenses not yet accrued
- Use credit later to pay for operations whose actual cost $>$ amortized cost.
- Differs from aggregate analysis:
  - In the accounting method, different operations can have different costs.
  - In aggregate analysis, all operations have the same cost.
- Credit must never go negative.
  - Otherwise we have a sequence of operations for which amortized cost is not an upper bound on actual cost.
  - Amortized cost would tell us nothing.

Accounting method costs

$$c_i = \text{actual cost of } i\text{th operation}$$
$$\hat{c}_i = \text{amortized cost of } i\text{th operation}$$

Require

$$\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$$

Total credit stored

$$\sum_{i=1}^{n} \hat{c}_i - \sum_{i=1}^{n} c_i$$

must never be negative.

Accounting method amortized analysis of stack operations

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

Intuition:

- When pushing an object, pay \$2
- \$1 pays for the PUSH
- \$1 is prepayment for it being popped by POP or MULTIPOP
- Since each object has \$1 credit, the credit can never go negative.
- Total amortized cost, $O(n)$, is an upper bound on total cost.
- Worst cast amortized cost is $2n = O(n)$.

Accounting method amortized analysis of binary counter

- Charge \$0 to set a bit to 0
- Charge \$2 to set a bit to 1
  - \$1 pays for setting the bit to 1
  - \$1 prepayment for setting it back to 0
  - Have \$1 credit for every 1 in the counter
  - Therefore credit ≥ 0
- Amortized cost of INCREMENT:
  - Cost of resetting bits to 0 is paid by credit.
  - At most 1 bit is set to 1.
  - Amortized cost is always ≤ 2.
  - For n operations amortized cost is $O(n)$.

The Potential Method

- Like the accounting method, but think of the credit as the *potential* stored with the entire data structure.
- Accounting method stores credit with specific objects.
- Potential method stores potential in the data structure as a whole.
- Can release potential to pay for future operations.
- Most flexible of the amortized analysis methods.

Potential function

$$
\begin{aligned}
D_i &= \text{data structure after the } i\text{th operation} \\
D_0 &= \text{initial data structure} \\
c_i &= \text{actual cost of } i\text{th operation} \\
\hat{c}_i &= \text{amortized cost of the } i\text{th operation}
\end{aligned}
$$

Potential function: $\Phi : D_k \to \mathbb{R}$
$\Phi(D_i)$ is the *potential* associated with the data structure D_i.

$$
\begin{aligned}
\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\
&= c_i + \Delta\Phi(D_i)
\end{aligned}
$$

The amortized cost is the *increase in potential* due to the ith operation.

Total amortized cost

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

- If we require that $\Phi(D_i) \geq \Phi(D_0)$ for all i, then the amortized cost is always an upper bound on the actual cost.
- In practice:

$$\begin{aligned}\Phi(D_0) &= 0 \\ \Phi(D_i) &\geq 0 \qquad \text{for all } i\end{aligned}$$

Amortized analysis of stack operations using the potential method

$$\begin{aligned}\Phi &= \#\text{ of objects in the stack}\\ &= \#\text{ of \$1 bills in the accounting method}\\ \Phi(D_0) &= 0\end{aligned}$$

Since $\#$ of objects in stack is always ≥ 0,

$$\Phi(D_i) \geq 0 = \Phi(D_0) \qquad\qquad \text{for all } i$$

operation	actual cost	$\Delta\Phi$
PUSH	1	$(s+1) - s = 1$
POP	1	$(s-1) - s = -1$
MULTIPOP	$k' = \min(k, s)$	$(s-k') - s = k'$

Therefore the amortized cost of a sequence of n operations is $O(n)$.

Amortized analysis of binary counter: potential method

- $\Phi = b_i = \#$ of 1's after ith INCREMENT
- Suppose ith operation resets t_i bits to 0.
- $c_i \leq t_i + 1$, since it resets t_i bits and sets ≤ 1 bit to 1.
- If $b_i = 0$, the ith operation reset all k bits and didn't set one, so
$$b_{i-1} = t_i = k \Rightarrow b_i = b_{i-1} - t_i = 0$$
- If $b_i > 0$ the ith operation reset t_i bits, set one, so
$$b_i = b_{i-1} - t_i + 1$$
- Either way
$$b_i \leq b_{i-1} - t_i + 1$$
- Therefore
$$\Delta\Phi(D_i) \leq (b_{i-1} - t_i + 1) - b_{i-1} = 1 - t_i$$
$$\hat{c}_i = c_i + \Delta\Phi(D_i) \leq (t_i + 1) + (1 - t_i) = 2$$
- If counter starts at 0, $\Phi(D_0) = 0$.
- Therefore, amortized cost of n operations is $O(n)$.

Dynamic Tables

- Nice application of amortized analysis.
- Suppose you have a table, maybe a hash table, maybe a heap.
- Details of table organization not important.
- We will assume insertion and deletion take $O(1)$.
- You don't know in advance how many items will be stored in it.
- When it fills, you must reallocate a larger table and copy all the items into the new table.
- When it gets sufficiently small, you *might* want to reallocate with a smaller size.
- How can you do this so it doesn't mess up the efficiency of your table?
- Does it turn $O(1)$ (hash) or $O(\lg n)$ (heap) into $O(n)$, since in worst case we have to copy all n elements into new array?

Dynamic Table Goals

1. $O(1)$ amortized time per operation.
2. Unused space always $\leq$ constant fraction of allocated space.

- **Load factor** $\alpha = num/size$ where $num = \#$ items stored, $size =$ allocated size.
- Never allow $\alpha > 1$
- Keep $\alpha >$ constant fraction (goal 2).

Table expansion

- First we consider only expansion.
- When table becomes full, double its size and reinsert all existing items.
- Each time we actually insert an item, it's an **elementary insertion**.

```
TABLE-INSERT(T, x)
if T.size == 0                                   // empty?
    allocate T.table with 1 slot
    T.size = 1
if T.num == T.size                               // expand?
    allocate new-table with 2 · T.size slots
    insert all items in T.table into new-table
    free T.table
    T.table = new-table
    T.size = 2 · T.size
insert x into T.table
T.num = T.num + 1
```

Running time

- Charge 1 per elementary insertion.
- Count only elementary insertions.
  - All other costs are constant per cell.
- c_i = actual cost of ith operation
- If not full, $c_i = 1$
- If full, insert $i - 1$ items plus one more, $c_i = i$.
- n operations, worst case:

$$c_i = O(n)$$
$$n \text{ operations} = O(n^2)$$

Aggregate analysis

- Of course, we don't *always* expand:

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

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

- Aggregate analysis: the amortized cost per operation is 3.

Accounting method

- Charge \$3 per elementary insertion of x:
  - \$1 pays for x's insertion.
  - \$1 pays for x's move in the future.
  - \$1 pays for some other item to be moved.
- Suppose we've just expanded, $size = m$.
- $size = 2m$ after next expansion.
- Assume that the expansion used up all the credit, so that there's no credit stored after the expansion.
- Will expand again after another m insertions.
- Each insertion will put \$1 on one of the m items that were in the table just after expansion, and will put \$1 on the item inserted.
- Have \$$2m$ of credit by next expansion, when there are $2m$ items to move.
- Just enough to pay for expansion, with no credit left over!
- Credit always ≥ 0.

Potential method

$$\Phi(T) = 2 \cdot T.num - T.size$$

- Initially, $num = size = 0$.

$$\Phi = 0$$

- Just after expansion, $size = 2 \cdot num$

$$\Phi = 0$$

- Just before expansion, $size = num$

$$\Phi = num$$

  we have enough potential to pay for moving all items.
- Need $\Phi \geq 0$ always.

$$\begin{array}{ccccc} size \geq & num & \geq & size/2 & \Rightarrow \\ & 2 \cdot num & \geq & size & \Rightarrow \\ & \Phi & \geq & 0 & \end{array}$$

Amortized cost of ith operation

$$num_i = num \text{ after } i\text{th operation}$$
$$size_i = size \text{ after } i\text{th operation}$$
$$\Phi_i = \Phi \text{ after } i\text{th operation}$$

- If no expansion:

$$size_i = size_{i-1}$$
$$num_i = num_{i-1} + 1$$
$$c_i = 1$$

Then we have

$$\begin{aligned}
\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\
&= 1 + (2 \cdot num_i - size_i) - (2 \cdot num_{i-1} - size_{i-1}) \\
&= 1 + (2 \cdot num_i - size_i) - (2(num_i - 1) - size_i) \\
&= 1 + 2 = 3
\end{aligned}$$

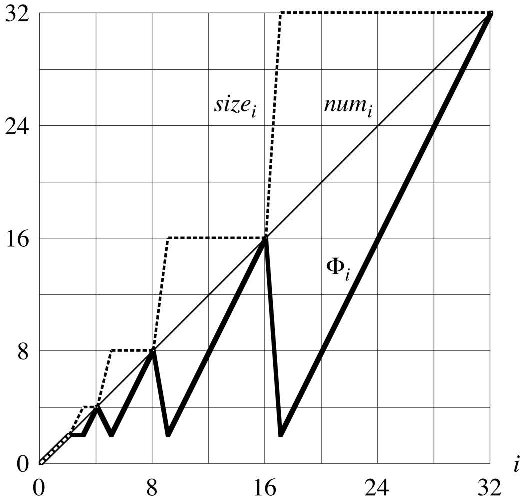
Amortized cost of ith operation

- If expansion:

$$\begin{aligned} size_i &= 2 \cdot size_{i-1} \\ size_{i-1} &= num_{i-1} = num_i - 1 \\ c_i &= num_{i-1} + 1 = num_i \end{aligned}$$

Then we have

$$\begin{aligned} \hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\ &= num_i + (2 \cdot num_i - size_i) - (2 \cdot num_{i-1} - size_{i-1}) \\ &= num_i + (2 \cdot num_i - 2(num_i - 1)) - (2(num_i - 1) - (num_i - 1)) \\ &= num_i + 2 - (num_i - 1) \\ &= 3 \end{aligned}$$



- As we insert items, the potential builds up until we have enough to pay for moving all items, when the potential drops back to zero.

Expansion and contraction

When α drops too low, contract the table.

- Allocate a new, smaller one.
- Copy all items.

Still want:

- α bounded from below by a constant
- amortized cost of $O(1)$

"Obvious strategy"

- Double size when inserting into a full table ($\alpha = 1$).
- Halve size when deletion would make table less than half full ($\alpha < 1/2$).
- Then would always have $1/2 \le \alpha \le 1$.
- Unfortunately, suppose we fill the table, then:

$$\begin{array}{rcl} \text{insert} & \Rightarrow & \text{double} \\ \text{two deletes} & \Rightarrow & \text{halve} \\ \text{two inserts} & \Rightarrow & \text{double} \\ \text{two deletes} & \Rightarrow & \text{halve} \\ \text{two inserts} & \Rightarrow & \text{double} \\ & \cdots & \end{array}$$

- Not performing enough operations in between expansion and contraction to pay for the next one.

Simple solution

- Double when full ($\alpha = 1$).
- Halve size when $\alpha = 1/4$.
- Immediately after expansion *or* contraction, $\alpha = 1/2$.
- Always have $1/4 \leq \alpha \leq 1$

Intuition

- Want to make sure we perform enough operations in between consecutive expansions/contractions to pay for the change in table size.
- Need to delete half of the items before contraction.
- Need to double the number of items before expansion.
- Either way, the number of operations between expansions and contractions is at least a constant fraction of the number of items copied.

$$\Phi(T) = \begin{cases} 2 \cdot T.num - T.size & \text{if } \alpha \geq 1/2 \\ T.size/2 - T.num & \text{if } \alpha < 1/2 \end{cases}$$

T empty $\Rightarrow \Phi = 0$

$\alpha \geq 1/2 \Rightarrow num \geq size/2 \Rightarrow 2 \cdot num \geq size \Rightarrow \Phi \geq 0$

$\alpha \leq 1/2 \Rightarrow num < size/2 \Rightarrow \Phi \geq 0$

Further intuition

- Φ measures how far from $\alpha = 1/2$ we are.
- $\alpha = 1/2 \Rightarrow \Phi = 2 \cdot \mathit{num} - 2 \cdot \mathit{num} = 0$
- $\alpha = 1 \Rightarrow \Phi = 2 \cdot \mathit{num} - \mathit{num} = \mathit{num}$
- $\alpha = 1/4 \Rightarrow \Phi = \mathit{size}/2 - \mathit{num} = 4 \cdot \mathit{num}/2 - \mathit{num} = \mathit{num}$
- Therefore, when we double or halve, we have enough potential to pay for moving all *num* items.

Further intuition

- Potential increases linearly between $\alpha = 1/2$ and $\alpha = 1$.
- Potential increases linearly between $\alpha = 1/2$ and $\alpha = 1/4$.
- Since α has different distances to go to get to 1 or $1/4$, starting from $1/2$, rate of increase of Φ differs.
- For α to go from $1/2$ to 1:
  - *num* increases from *size*$/2$ to *size*
  - Φ increases from 0 to *size*
  - Φ needs to increase by 2 for each item inserted.
  - That's why the coefficient of 2 in the formula for Φ.
- For α to go from $1/2$ to $1/4$:
  - *num* decreases from *size*$/2$ to *size*$/4$.
  - Φ increases from 0 to *size*$/4$
  - Thus, Φ needs to increase by 1 for each item deleted.
  - That's why the coefficient of -1 in the formula for Φ.

Eight cases for calculating amortized costs

- insert *vs.* delete
- $\alpha \geq 1/2$ *vs.* $\alpha < 1/2$
- *size* changes *vs.* *size* doesn't change

Insert, $\alpha \geq 1/2$, with or without expansion

- Same analysis as before.
- $\hat{c}_i = 3$

Insert, $\alpha_{i-1} < 1/2$, no expansion

- $\alpha_i < 1/2$

$$\begin{aligned}
\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\
&= 1 + (size_i/2 - num_i) - (size_{i-1}/2 - num_{i-1}) \\
&= 1 + (size_i/2 - num_i) - (size_i/2 - (num_i - 1)) \\
&= 0
\end{aligned}$$

- $\alpha_i \geq 1/2$

$$\begin{aligned}
\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\
&= 1 + (2 \cdot num_i - size_i) - (size_{i-1}/2 - num_{i-1}) \\
&= 1 + (2(num_{i-1} + 1) - size_{i-1}) - (size_{i-1}/2 - num_{i-1}) \\
&= 3 \cdot num_{i-1} - \frac{3}{2} \cdot size_{i-1} + 3 \\
&= 3 \cdot \alpha_{i-1} size_{i-1} - \frac{3}{2} \cdot size_{i-1} + 3 \\
&< \frac{3}{2} \cdot size_{i-1} - \frac{3}{2} \cdot size_{i-1} + 3 \\
&= 3
\end{aligned}$$

Insert, $\alpha < 1/2$, expansion

- Cannot happen.

Insert

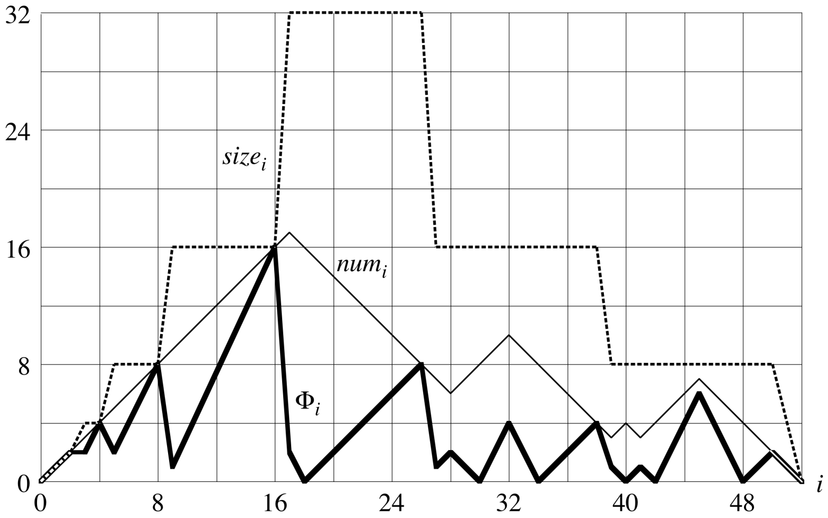
expansion	$\alpha \geq 1/2$	$\hat{c}_i = 3$
no expansion	$\alpha \geq 1/2$	$\hat{c}_i = 3$
expansion	$\alpha < 1/2$	impossible
no expansion	$\alpha_{i-1} < 1/2, \alpha_i < 1/2$	$\hat{c}_i = 0$
no expansion	$\alpha_{i-1} < 1/2, \alpha_i \geq 1/2$	$\hat{c}_i = 3$

- Therefore, in all cases, the amortized cost of insertion is ≤ 3.

Delete

contraction	$\alpha < 1/2$	$\hat{c}_i = 1$
no contraction	$\alpha < 1/2$	$\hat{c}_i = 2$
contraction	$\alpha \geq 1/2$	impossible
no contraction	$\alpha_{i-1} \geq 1/2, \alpha_i \geq 1/2$	$\hat{c}_i = -1$
no contraction	$\alpha_{i-1} \geq 1/2, \alpha_i < 1/2$	$\hat{c}_i = 2$

- In all cases the amortized cost is ≤ 2.



32
24
16
8
0
0
8
16
24
32
40
48
i
$size_i$
num_i
Φ_i