Lecture 2: Amortized Analysis

October 11, 2019

What is amortized analysis?

- It estimates the average cost (=running time) of an operation as part of a sequence of operations on a data-structure.
- The goal is to show that the average cost of an operation is small, if one averages over a sequence of operations, even though a single operations within the sequence might be expensive.

Common techniques for amortized analysis

- The aggregate method: determines an upper bound $T(n)$ on the total cost of a sequence $op_1 \, op_2 \, \ldots \, op_n$ of n operations.
  - the amortized cost of each operation is $T(n)/n$; it is an upper bound of the average cost per operation.
- The accounting method:
  - Distinguishes different types of operations; operations of different types may have different amortized costs.
  - Some ops. may be overcharged as "prepaid credit"/object in the data structure, which is used later in the sequence to pay for ops. that are charged less than they actually cost.
- The potential method: determines the amortized cost of each operation (like accounting method) and may overcharge ops. early on to compensate for undercharges later. The credit is maintained as *potential energy* of the data structure instead of credit associated with individual objects within the data structure.

The 3 techniques for amortized analysis will be examined on 2 examples:

- a stack with the additional operation MULTIPOP which pops several object at once.
- a binary counter which counts from 0 using only one operation, INCREMENT.

Remark

- for analysis purposes, some attributes, such as the charge value *x.credit* may be assigned to an object *x*. These attributes are for analysis purposes only, and should not show up in the code.

1. The aggregate method

- Computes the worst-case time $T(n)$ needed to perform a sequence of n operations in total.
- The average cost, or amortized cost, per operation is $T(n)/n$.
- This cost applies to each operation, even if there are several types of operations in the sequence.
- The other two methods (accounting method and potential method) may assign different amortized costs to different types of operations.

Fundamental stack operations:

- PUSH(S, x): pushes object x onto stack S.
- POP(S): pops the top of stack S and returns the popped object.

- Both operations run in $O(1)$ time $\Rightarrow$ it is ok to assign cost 1 to both operations.
- $\Rightarrow$ the total cost of a sequence of n PUSH and POP operations is $n \Rightarrow$ the actual running time for n operations is $\Theta(n)$.

Fundamental stack operations:

- PUSH(S, x): pushes object x onto stack S.
- POP(S): pops the top of stack S and returns the popped object.

- Both operations run in $O(1)$ time $\Rightarrow$ it is ok to assign cost 1 to both operations.
- $\Rightarrow$ the total cost of a sequence of n PUSH and POP operations is $n \Rightarrow$ the actual running time for n operations is $\Theta(n)$.

What happens if we extend stacks with the operation MULTIPOP(S, k)? MULTIPOP(S, k) behaves as follows:

- removes the k top objects of stack S, if S has $\geq k$ elements
- pops the entire stack S if it contains less than k elements.

Assumption: STACKEMPTY(S) returns TRUE if S is an empty stack, and FALSE otherwise.

MULTIPOP(S, k)
1 **while** not STACKEMPTY(S) and $k \neq 0$
2   POP(S)
3      $k := k - 1$

Total cost of MULTIPOP(S, k) on a stack S with s elems. is $\min(s, k)$.

$\Rightarrow$ the running time of this operation is a linear function of $\min(s, k)$.

Assume a sequence of n PUSH, POP, and MULTIPOP operations on an initially empty stack.

- The worst-case cost of a MULTIPOP operation in this sequence is $O(n)$, since the stack size is at most n.
- $\Rightarrow$ the worst-case cost of any stack operation is $O(n)$
- $\Rightarrow$ the worst-case cost of the sequence of n operations is $O(n^2)$, since we may have $O(n)$ MULTIPOP operations costing $O(n)$ each.
- This bound is not tight; **we shall do better.**

- Each object can be popped at most once for each time it is pushed $\Rightarrow$ the number of times that POP can be called on a nonempty stack, including calls with MULTIPOP, is $\leq$ the number of PUSH operations, which is at most n.

$\Rightarrow$ for any n, any sequence of n PUSH, POP, and MULTIPOP operations takes $O(n)$ time.

- The amortized cost of an operation is the average $O(n)/n = O(1)$.

Remark. The amortized-case analysis **did not use any probabilistic argument**. We actually showed a worst case bound $O(n)$ on a sequence of n operations. The amortized cost is the average cost in such a sequence of n operations, that is, $O(1)$.

A *k*-bit binary counter counts upward from 0.

- Is represented by an array $A[0..k-1]$ of bits, where $A.length = k$.
- The lowest-order bit is stored in $A[0]$, and the highest-order bit is stored in $A[k-1]$. Thus, $x = \sum_{i=0}^{k-1} A[i] \cdot 2^i$.
- Initially, $x = 0$, thus $A[0] = A[1] = \ldots = A[k-1] = 0$.

The following procedure adds 1 modulo 2^k to the counter:

INCREMENT(*A*)
1 $i := 0$
2 **while** $i < A.length$ and $A[i] = 1$
3      $A[i] := 0$
4      $i := i + 1$
5 **if** $i < A.length$
6   $A[i] := 1$

The cost of INCREMENT is linear in the number of bits flipped.

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	1	1	1	1	1	26
16	0	0	0	1	0	0	0	0	31

An 8-bit binary counter as its value goes from 0 to 16 by a sequence of 16 INCREMENT operations. Bits that flip to achieve the next value are shaded. The running cost for flipping bits is shown at the right. Note that the total cost is never more than twice the total number of INCREMENT operations.

A straightforward worst-case runtime analysis yields a bound that is not tight

- A single execution of INCREMENT takes time $\Theta(k)$ in the worst case, when all elements of A are 1.
- $\Rightarrow$ a sequence of n INCREMENT operations on an initially 0 counter takes time $O(nk)$ in the worst case.

The analysis can be tightened if we notice that, in a sequence of n INCREMENT operations, $A[0]$ flips n times, $A[1]$ flips $\lfloor n/2 \rfloor$ times, . . . ,$A[i]$ flips $\lfloor n/2^i \rfloor$ times, etc. For $i > \lfloor \log_2 n \rfloor$, bit $A[i]$ never flips. $\Rightarrow$ the total number of flips in the sequence is

$$\sum_{i=0}^{\lfloor \log_2 n \rfloor} \left\lfloor \frac{n}{2^i} \right\rfloor < n \sum_{i=0}^{\infty} \frac{1}{2^i} = 2 \cdot n.$$

- The worst time for the sequence is $O(n)$
- $\Rightarrow$ the amortized cost is $O(n)/n = O(1)$.

2. The accounting method

Main ideas

- We assign different amortized costs to different operations. Some operations have amortized cost more or less than they actually cost.

- When an operation's amortized cost exceeds its actual cost, the difference is assigned to specific objects in the data structure as **credit**.

- Credit can be used later on to help pay for operations whose amortized cost is less than their actual cost. This is very different from the aggregate method, where all operations have the same amortized cost.

- Total credit associated with the data structure := amount by which total amortized cost incurred exceeds total actual cost incurred.

- Total amortized cost of a sequence of operations should be always $\geq$ total actual cost $\Leftrightarrow$ total credit should be always ≥ 0.

Let's assume the following amortized costs:

Operation	Amortized cost	Actual cost
PUSH	2	1
POP	0	1
MULTIPOP(k)	0	$\min(s, k)$

where s is the number of elements stored in the stack.

We will show that we can pay for any sequence of stack operations by charging the amortized costs.

Analogy: Stack $\leftrightarrow$ stack of plates with 1\$ on top of each plate.

- Actual cost of PUSH/POP operations = 1\$.
- Amortized cost of PUSH = 2\$ = 1\$ actual charge + 1\$ credit on the plate being pushed.
- Amortized cost of POP = 0\$ = 1\$ credit from the plate being popped - 1\$ actual cost charged.
- By the same reasoning, the amortized cost of MULTIPOP should be 0\$.

Start with an empty stack.

- For any sequence of n PUSH, POP, and MULTIPOP operations, the total cost is $\leq$ total amortized cost.
  - Total amortized cost of n operations is $\leq 2 \cdot n = O(n)$
  - $\Rightarrow$ total cost is $O(n) \Rightarrow$ amortized cost is $O(n)/n = O(1)$.

- Let's study the cost of n INCREMENT operations of a binary counter, starting from 0:
    - Running time = number of bits flipped $\Rightarrow$ we assign a unit of cost (1\$) to the flipping of a bit.

Let's assume the following amortized costs:

Operation	Amortized cost	Actual cost
flip $0 \rightarrow 1$	2	1
flip $1 \rightarrow 0$	0	1

Intuition: At any point in time, every 1 in the counter has 1\$ credit on it; to reset it to 0 we need not charge anything because we pay with the credit on the bit.

- The number of 1 bits in the counter is always $\geq 0 \Rightarrow$ the total credit is always ≥ 0.
- $\Rightarrow$ total amortized cost of n INCREMENT operations is $\leq 2n$, thus it is $O(n)$.
- The total actual cost is $\leq$ the total amortized cost $\Rightarrow$ total actual cost is also $O(n)$.

The potential method
How does it work? (1)

- represents prepaid work as potential energy, or just potential, that can be released to pay for future operations.
- The potential is associated with the data structure as a whole rather than with specific objects within the data structure.

How does it work?

- It starts with an initial data structure D_0 on which n operations are performed.
- For each $1 \leq i \leq n$, let c_i be the actual cost of the i-th operation, and D_i the data structure resulted after the i-th operation.
- We assume given a potential function Φ that maps any data structure D_i to a value $\Phi(D_i) \in \mathbb{R}$, called the potential associated with D_i.

- The amortized cost of the i-th operation w.r.t. Φ is

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

  that is, the actual cost of the i-th operation plus the increase in potential due to the operation.

- The total amortized cost of a sequence of n operations is

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

$\Rightarrow$ if we define Φ such that $\Phi(D_n) \geq \Phi(D_0)$, then the total amortized cost $\sum_{i=1}^{n} \hat{c}_i$ is $\geq$ total actual cost $\sum_{i=1}^{n} c_i$.

In practice, we don't know n, therefore we define Φ such that $\Phi(D_i) \geq \Phi(D_0)$ for all i.

- This condition guarantees that every performed operation can be paid in advance.

It is often convenient to define $\Phi(D_0) = 0$ and then show that $\Phi(D_i) \geq 0$ for all i.

- **Intuition:**
  - If $\Phi(D_i) - \Phi(D_{i-1}) > 0$ then the amortized cost $\hat{c}_i$ represents an *overcharge* to the i-th operation $\Rightarrow$ the potential of the data structure increases.
  - If $\Phi(D_i) - \Phi(D_{i-1}) > 0$ then the amortized cost $\hat{c}_i$ represents an *undercharge* to the i-th operation $\Rightarrow$ the actual cost of the operation is paid by the decrease in the potential.

The potential method for stack operations

- Consider again stacks with PUSH, POP and MULTIPOP operations, and let D_0 be the empty stack.
- Define $\Phi(D_i) :=$ number of elements in D_i. Then $\Phi(D_0) = 0$ and $\Phi(D_i) \geq 0$ for all $i \Rightarrow$ total actual cost $\leq$ total amortized cost.
- We recall that the actual cost of every stack operation is 1, thus $c_i = 1$ for all i.
- Consider D_{i-1} has s elements.
  - If the i-th operation is PUSH then $\Phi(D_i) - \Phi(D_{i-1}) = (s + 1) - s = 1$, therefore the amortized cost of the i-th operation is $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + 1 = 2$.
- If the i-th operation is MULTIPOP and $k' = \min(k, s)$, then $\Phi(D_i) - \Phi(D_{i-1}) = -k'$, therefore in this case $\hat{c}_i = c_i - k' = k' - k' = 0$.
- Similarly, the amortized cost for POP at step i is $\hat{c}_i = 0$.

$\Rightarrow$ total amortized cost of the sequence of n ops. is $\leq 2 \cdot n$, thus $O(n)$, hence total actual cost is also $O(n)$.

The potential method for a binary counter

- Define $\Phi(D_i) := b_i :=$ the number of 1s in counter D_i.
- If the i-th operation resets t_i bits to 0, then
  - the actual cost of this operation is $c_i \leq t_i + 1$ because, in addition to resetting t_i bits to 0, INCREMENT sets at most one bit to a 1.
  - $\Rightarrow$ the potential difference is

    $$\Phi(D_i) - \Phi(D_{i-1}) \leq (b_{i-1} - t_i + 1) - b_{i-1} = 1 - t_i.$$

  - $\Rightarrow$ the amortized cost is

    $$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) \leq 1 + t_i + (1 - t_i) = 2.$$

- If the sequence of n INCREMENTs starts at zero, then $\Phi(D_0) = 0$. Since $\Phi(D_i) \geq 0$ for all i, we have

$$\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} \hat{c}_i \leq 2n$$

$\Rightarrow$ worst-case cost is $O(n)$.

1. If a MULTIPUSH operation were included in the set of stack operations, would the $O(1)$ bound on the amortized cost of stack operations continue to hold?

2. A sequence of n operations is performed on a data structure. The i-th operation costs i if i is an exact power of 2, and 1 otherwise. Use an aggregate method of analysis to determine the amortized cost per operation.

3. Suppose we wish not only to increment a counter but also to reset it to zero (that is, set all its bits to 0). Show how to implement a counter as a bit vector so that any sequence of n INCREMENT and RESET operations takes time $O(n)$ on an initially zero counter. (HINT: Keep a pointer to the high-order 1.)

4. Redo Exercise 2 using a potential method of analysis.

5. Show that if a DECREMENT operation were included in the k-bit counter example, then n operations could cost as much as $\Theta(nk)$ time.

- Chapter 18: *Amortized Analysis* of
    T. H. Cormen, C. E. Leiserson. R. L. Rivest. *Introduction to Algorithms*. The MIT Press, 2000.