Lecture 7

Analysis of algorithms: Amortized Analysis

January 2014

What is amortized analysis?

Amortized analysis: set of techniques (Aggregate method, Accounting method, Potential method) for proving upper (worst-case) bounds for the running time of an algorithm, usually involving a sequence of operations on some data structure.

What is amortized analysis?

Amortized analysis: set of techniques (Aggregate method, Accounting method, Potential method) for proving upper (worst-case) bounds for the running time of an algorithm, usually involving a sequence of operations on some data structure.

- Comparison with worst-case time analysis:
  Worst-case running time of a sequence of n operations = $n \cdot t$ where t is worst-case running-time of any operation in the sequence
  - Technically correct, but sometimes overly pessimistic (as we will see)

What is amortized analysis?

Amortized analysis: set of techniques (Aggregate method, Accounting method, Potential method) for proving upper (worst-case) bounds for the running time of an algorithm, usually involving a sequence of operations on some data structure.

- Comparison with worst-case time analysis:
  Worst-case running time of a sequence of n operations = $n \cdot t$ where t is worst-case running-time of any operation in the sequence
  - Technically correct, but sometimes overly pessimistic (as we will see)
- Main idea of amortized analysis: Some operations have high worst-case running time, but we can show that the worst case does not occur every time.

There are 3 main techniques for amortized analysis:

1. the summation (or aggregate) method,
2. the accounting method, and
3. the potential method.

We will examine them on two 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, INCR.

There are 3 main techniques for amortized analysis:

1. the summation (or aggregate) method,
2. the accounting method, and
3. the potential method.

We will examine them on two 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, INCR.

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

The Summation (or Aggregate) Method

- Computes an upper bound $T(n)$ on the total cost of a sequence of n operations.
- 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.

Remark

We are not making an “average case” argument about inputs. We are still talking about *worst-case performance*.

Common techniques for amortized analysis

- The aggregate method: determines an upper bound $T(n)$ on the total cost of a sequence of n operations. The amortized cost per operation is then $T(n)/n$.
- The accounting method: overcharges some operations early in the sequence, storing the overcharge as "prepaid credit" on specific objects in the data structure. The credit is used later in the sequence to pay for operations that are charged less than they actually cost.
- The potential method: determines the amortized cost of each operation (like the accounting method) and may overcharge operations early on to compensate for undercharges later. This method maintains the credit as the *potential energy* of the data structure instead of associating the credit with individual objects within the data structure.

Example 1: a stack with PUSH/POP/MULTIPOP

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 $n \cdot 1 = n = \Theta(n)$.

Example 1: a stack with PUSH/POP/MULTIPOP

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 $n \cdot 1 = n = \Theta(n)$.

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.

Example 1: a stack with Push/Pop/MultiPop

Worst-case running time analysis

Assumption: StackEmpty(S) returns True if S is an empty stack, and False otherwise.

```
MultiPop(S, k)
while not StackEmpty(S) and k ≠ 0
        Pop(S)
        k := k − 1
```

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

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

Assume a sequence $\mathcal{S}$ 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 $\mathcal{S}$ 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.

$$\mathcal{S} = \underbrace{op_1,\ op_2,\ \ldots,\ op_n}_{\text{contains at most } n \text{ PUSH and } n \text{ POP operations}}$$

$\Rightarrow$ for any n, any sequence $\mathcal{S}$ of n PUSH, POP, and MULTIPOP operations takes $T(n) = O(n)$ time.

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

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, $O(1)$.

Example 2: Binary counter

A binary counter counts upward from 0.

- Is represented by a sequence $A[0..]$ of bits.
- The lowest-order bit is stored in $A[0]$, and the highest-order bit is stored in $A[k-1]$. Thus, $x = \sum_{i \geq 0} A[i] \cdot 2^i$.
- Initially, $x = 0$, thus $A[i] = 0$ for all i

The following procedure increments the counter by 1:

```
INCR(A)
i := 0
while A[i] == 1
      A[i] := 0
      i := i + 1
A[i] := 1
```

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

Counter value	...	A[3]	A[2]	A[1]	A[0]	Total cost
0	...	0	0	0	0	0
1	...	0	0	0	**1**	1
2	...	0	0	**1**	**0**	$2+1=3$
3	...	0	0	1	**1**	$1+3=4$
4	...	0	**1**	**0**	**0**	$3+4=7$
5	...	0	1	0	**1**	$1+7=8$
⋮	...					

NOTE: The total cost is never more than twice the total number of INCR operations.

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

Counter value	...	A[3]	A[2]	A[1]	A[0]	Total cost
0	...	0	0	0	0	$0 \leq 2 \cdot 0$
1	...	0	0	0	**1**	$1 \leq 2 \cdot 1$
2	...	0	0	**1**	**0**	$2 + 1 = 3 \leq 2 \cdot 2$
3	...	0	0	1	**1**	$1 + 3 = 4 \leq 2 \cdot 3$
4	...	0	**1**	**0**	**0**	$3 + 4 = 7 \leq 2 \cdot 4$
5	...	0	1	0	**1**	$1 + 7 = 8 \leq 2 \cdot 5$
$\vdots$	...					

NOTE: The total cost is never more than twice the total number of INCR operations.

Assume $\mathcal{S} = \underbrace{\text{INCR}, \text{INCR}, \ldots, \text{INCR}}_{n \text{ times}}$

$A[0]$ 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 $\mathcal{S}$ 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 running time of one operation in $\mathcal{S}$ is $O(\log n)$
- The worst running time for $\mathcal{S}$ is $O(n)$
  $\Rightarrow$ the amortized cost is $O(n)/n = O(1)$.

The Accounting Method

- Assigns 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.
- 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 INCR 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 $0 \to 1$	2	1
flip $1 \to 0$	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 INCR 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 $O(n)$.

- 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 potential method
How does it work? (2)

- 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 payed 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 payed by the decrease in the potential.

Let D_0 be the initial 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 $O(n)$.

- 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

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

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

Exercises

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.

References

- Chapter 18: *Amortized Analysis* of

  T. H. Cormen, C. E. Leiserson. R. L. Rivest. *Introduction to Algorithms*. The MIT Press, 2000.