Amortized Analysis

Chapter 17

Amortized analysis

Amortized analysis is a method of algorithm analysis where the time to perform a sequence of operations is averaged over all of the operations performed.

It can show that the average cost of an operation is small even though a single operation within the sequence might be expensive.

Amortized analysis guarantees the average performance of each operation in the worst case.

Average-case analysis studies the worst performance in the average case.

Average-case analysis involves input distributions and probabilities; amortized analysis does not.

Amortized analysis

We study three types of amortized analysis:

1. The aggregate method. Here we determine the total cost T(n) of n operations and conclude that the average cost is T(n) / n.
2. The accounting method. Here we overcharge some operations early in the sequence, storing the overcharge as "prepaid credit" on specific data items. This credit is used on later operations to make them appear to cost less.
3. The potential method. Like the accounting method except that the prepaid credit is stored as a "potential energy" of the data structure as a whole.

Aggregate method

Show that the total worst-case cost of n operations is T(n). Conclude that the amortized cost, or the average worst-case cost, is T(n) / n.

As an example, we study some stack operations:

S.push(x)	pushes object x onto stack S
S.pop()	pops the top of stack S and returns the popped object.
S.multipop(k)	removes the top k objects of stack S or pops the entire stack if it contains fewer than k objects.

Stack operations

S.push(x) and S.pop() are standard. S.multipop() is implemented as:

```
MULTIPOP(k)
        while not EMPTY() and k ≠ 0
            POP()
            k = k – 1
```

(Note that I'm using object-oriented conventions in my pseudocode here, assuming that MULTIPOP, EMPTY, POP, etc. are member functions of the stack S.)

Analyzed by itself, MULTIPOP takes O(min(k, s)) time, where s is the number of elements in the stack.

A sequence of operations

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

PUSH and POP take O(1) time apiece. A single MULTIPOP could take O(n) time, so the time for the sequence is easily bounded by $O(n^2)$.

We can do better. Each object can be popped at most once for each time it is pushed. The number of POPs, including the POPs within MULTIPOP, are at most the number of PUSH operations. Thus, over all calls to MULTIPOP, only O(n) time is taken. This leads to a total time of O(n).

A sequence of operations

In aggregate analysis, we simply divide this total time by n to get the amortized cost of an operation: O(n) / n = O(1). That is, it takes constant amortized time for any of our operations.

We have just shown an average running time of a stack operation is O(1) with no probabilistic reasoning. We used the basic definition of an average instead.

Incrementing a Binary Counter

Another example of aggregate analysis occurs when implementing a k-bit binary counter.

We use an array A[0..k-1] of bits. A[0] is the lowest-order bit and A[k-1] is the highest-order, so the number x represented by the counter is

$$\sum_{i=0}^{k-1} 2^i \, A[i]$$

The counter starts with x = 0, which means all A[i] are 0.

Incrementing a Binary Counter

```
INCREMENT(A)                                   // note: length[A] = k
        i = 0
        while i < length[A] and A[i] = 1
                A[i] = 0
                i = i + 1
        if i < length[A]
                A[i] = 1
```

The cost of a call to INCREMENT is linear in the number of bits it flips. (I.e. it is linear in the number of array entries that it writes.)

A single call to INCREMENT takes $\Theta(k)$ in the worst case.

Incrementing a Binary Counter

But it's not the case that every call to INCREMENT flips all k bits. So let's look at a sequence of n calls to INCREMENT:

A[0] flips each call
A[1] flips every second call
A[2] flips every fourth call
A[i] flips every 2^i calls.

So the total amount of bits flipped for n calls to INCREMENT is:

$$\sum_{i=0}^{k-1} \left\lfloor \frac{n}{2^i} \right\rfloor \leq \sum_{i=0}^{k-1} \frac{n}{2^i} < \sum_{i=0}^{\infty} \frac{n}{2^i} = n \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n$$

0000
0001
0010
0011
0100
0101
0110
0111
1000

So the average number of bits flipped per call is at most 2n / n = 2. Since the work is proportional to the number of bits flipped, the average work is O(1).

The Accounting Method

We assign different charges to different operations: some operations are charged more or less than their actual cost. The amount we charge an operation is called its amortized cost.

When an operation's amortized cost is larger than its actual cost, the extra cost is assigned to specific objects in the data structure as credit. Credit is used to help pay for operations whose amortized cost is smaller than its actual cost.

The total amortized cost of all operations must always be an upper bound on the total actual cost. This will happen if credit on an item never goes negative, on any sequence of operations. Picking amortized costs for operations is the tricky part.

Stack Operations: The Accounting Method

operation	actual cost	amortized cost
PUSH	1	2 (leave credit on pushed item)
POP	1	0
MULTIPOP	min(k, s)	0

The 1-unit credit on each PUSHed item is prepayment for the cost of POPping it. When we POP an item, we use the 1-unit credit on the item to pay the actual cost of the POP. If we MULTIPOP, the actual cost is O(min(k, s)) but we remove min(k, s) items from the stack. We use the min(k, s) units of credit stored on the removed items to pay for the actual costs of the MULTIPOP.

The total amortized cost of a sequence of n operations is therefore 2 times the number of PUSH operations, or at most 2n. Thus the total actual cost is at most 2n, for an average of O(1) per operation.

Binary Counter: The Accounting Method

In INCREMENT, we flip bits.

flip bit to	actual cost	amortized cost
1	1	2 (credit stored on bit flipped)
0	1	0

We start off with all bits 0 and no credit on them. Therefore, any bit whose value is 1 will have a 1-unit credit stored on it. Any INCREMENT operation flips one bit to a 1 and some $m \geq 0$ bits to a 0. We pay two units for the bit flipped to a 1, but the actual cost of the bits flipped to zero is paid for by the credit stored on those bits. Therefore each INCREMENT operation has an amortized cost of 2, which is O(1).

The Potential Method

In the potential method, we store the prepaid work as a potential energy (referred to simply as potential) on the entire data structure.

Suppose we have an initial data structure D_0 and perform n operations on it. For i=1 to n, let c_i be the actual cost of operation i, and D_i be the data structure that results after operation i. Let Φ be the potential function in use.

Then the amortized cost of operation i is:

$$\widehat{c_i} = c_i + \Phi(D_i) - \Phi(D_{i-1})$$

The Potential Method

If we sum this up over all n operations, we get:

$$\sum_{i=1}^{n} \widehat{c_i} = \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1}))$$

$$= (\sum_{i=1}^{n} c_i) + \Phi(D_n) - \Phi(D_0)$$

So if we design a potential function with $\Phi(D_n)$ always greater than or equal to $\Phi(D_0)$, the summed amortized cost $\sum_{i=1}^{n} \widehat{c_i}$ is greater than or equal to the summed actual cost $\sum_{i=1}^{n} c_i$.

It is conventional to define $\Phi(D_0) = 0$ and show that all $\Phi(D_i) \geq 0$.

Choice of a potential function often involves tradeoffs.

Stack Operations: The Potential Method

We define the potential function Φ to be the number of objects in the stack.

Since we start with an empty stack, $\Phi(D_0) = 0$.

At any point in time, $\Phi(D_i) \geq 0$.

- For a PUSH operation i, $\Phi(D_i) - \Phi(D_{i-1}) = 1$, so its amortized cost is $c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1+1 = 2$.
- For a MULTIPOP operation i that pops $k' = \min(k, s)$ elements, the actual cost of the operation $c_i = k'$, but the potential difference $\Phi(D_i) - \Phi(D_{i-1}) = -k'$. Thus the amortized cost of MULTIPOP is $k' - k' = 0$.
- For a POP operation i, we have $c_i = 1$ and $\Phi(D_i) - \Phi(D_{i-1}) = -1$, for an amortized cost of 0.

The amortized cost of a stack operation is O(1).

Binary Counter: The Potential Method

We define the potential function Φ to be the number of bits equal to 1 in the counter. $\Phi(D_0) = 0$.

For an INCREMENT operation i, let t_i be the number of bits of the counter that get set to 0.

Actual cost is at most $t_i + 1$ (at most 1 bit gets set to 1).

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

So the amortized cost is

$$c_i + \Phi(D_i) - \Phi(D_{i-1})$$

$$\leq (t_i + 1) + (1 - t_i) = 2$$

Stack Operations with Backup

Suppose we have a stack with the operations PUSH, POP, and COPY. A sequence of PUSH and POP operations are performed on the stack, with a COPY operation automatically inserted after every k PUSH and/or POP operations. COPY makes a copy of the entire stack for backup purposes. The size of the stack never exceeds k. Show that the cost of n PUSH and POP operations, including the automatic COPY operations, takes time O(n) by assigning suitable amortized costs to the operations.

(Problem 17.2-1)

Stack Operations with Backup

The stack size could be as large as k, so a COPY operation's actual cost is k. We would like its amortized cost to be 0, so we need k units of credit to pay for it.

PUSH and POP both have an actual cost of 1. Since there are k PUSH and POP operations inbetween each COPY operation, if we get 1 credit from each of them, we will have enough credit to pay for the COPY.

So we set the amortized cost of PUSH and POP to 2 apiece. One of that cost goes to pay the actual cost, and 1 is placed as credit on the stack. Then when a COPY comes around, we have the credit on the stack to pay for it.

Stack Operations with Backup

Exercise:

Can we do the same problem when the stack also has a MULTIPOP operation? What would be the amortized cost of each operation?

Powers of 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. What is the amortized cost per operation?

aggregate method

The cost of the n operations is

$$\sum_{i=1}^{n} 1 + \sum_{j=0}^{\lfloor \log n \rfloor} (2^j - 1) =$$

$$n + \sum_{j=0}^{\lfloor \log n \rfloor} 2^j - \sum_{j=0}^{\lfloor \log n \rfloor} 1 =$$

$$n + \sum_{j=0}^{\lfloor \log n \rfloor} 2^j - (\lfloor \log n \rfloor + 1) =$$

$$n - (\lfloor \log n \rfloor + 1) + 2^{\lfloor \log n \rfloor + 1} - 1 =$$

Powers of 2

$$n - \lfloor \log n \rfloor - 2 + 2 \cdot 2^{\lfloor \log n \rfloor} <$$
$$n + 2 \cdot 2^{\lfloor \log n \rfloor} \leq$$
$$n + 2n =$$
$$3n.$$

Amortized cost per operation: less than 3

Reading

We won't go over the rest of chapter 17 (dynamic tables), but please read it.