Amortization

Tufts University

Warm-up Question

How much time does it take to INCREMENT() a bit counter?

0	1	1	1	0	0	1	1	1	1

$$\Downarrow$$

0	1	1	1	0	1	0	0	0	0

Cheat of the day

$$\sum_{i=0}^{\infty} \frac{1}{2^i} = 1 + \frac{1}{2} + \frac{1}{4} + \ldots = 2$$

Introducing amortized analysis

Formal version of *average* runtime

Used when we repeatedly call same operations

- Some operations will be expensive
- others will be cheap

How to combine average with big O and Ω?

Back to bit counter

Say counter has $\log_2 n$ bits

Start at 0. Call $\textsc{Increment}()$ n times

Runtime is $\Theta(c_i)$ where c_i is number of changed bits

Worst-case runtime of one $\textsc{Increment}()$?

$c_i = \#$bits (from $011\ldots1$ to $10\ldots0$)

Lemma

The total time used in n executions of $\textsc{Increment}()$ is $\Theta(n)$

Brute force math

0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	0	1	0
0	0	0	0	0	0	0	0	1	1
0	0	0	0	0	0	0	1	0	0
0	0	0	0	0	0	0	1	0	1
0	0	0	0	0	0	0	1	1	0

$\vdots$

1	1	1	1	1	1	1	1	1	1

How many changes in the lowest order digit? n changes

What about second lowest order digit? $n/2$ changes

And highest order digit? one change

Total number of changes $= \sum_{i=0}^{\log n} \frac{n}{2^i} = n \sum_{i=0}^{\log n} \frac{1}{2^i} < 2n$

$\Rightarrow$ Runtime needed by the n operations is $O(n)$

Amortized runtime

Lemma

The total time used in n executions of INCREMENT*() is* $\Theta(n)$

A single operation could take $\Theta(\#bits)$, but most need $\Theta(1)$

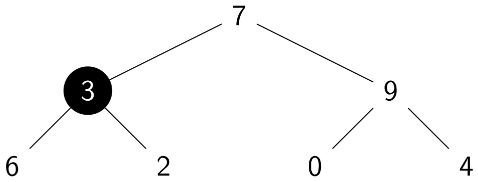
On "average" each execution needs $O(1)$~~On "average" each execution needs $O(1)$~~

Amortized cost of INCREMENT() over n operations is $O(1)$

$\Leftrightarrow$ The n executions will need $n \cdot O(1)$ time in the worst case

For short, we say INCREMENT has $O(1)$ amortized cost

Remember bottom up heap construction?



Insert all numbers at once. Fix from leaves upwards

At each node we have to **sink** root $\Rightarrow \Theta(\log n_i)$

> n_i = number of elements in the sub-heap

Sometimes n_i is big but often n_i is small

> n sink operations need $\Theta(n)$ time in total
> $\Rightarrow \Theta(1)$ amortized runtime

Potential method

General technique for computing amortized runtime

- Main idea: set a goal g runtime per operation
- Let c_i cost of i-th operation
- Compare c_i against our goal

  If $c_i < g$ we *gain* potential

  If $c_i > g$ we *lose* potential
- Goal: always maintain positive potential

  $\Rightarrow$ Average runtime is gn − remaining potential

Definitions

Key point: define a potential function Φ to the DS

In the bit vector: Potential Φ(bit vector) = # of 1s in vector

c_i cost of i-th operation (in bit vector, c_i = # of bits changed)

ϕ_i = potential after i-th operation

$\hat{c}_i = c_i + \phi_i - \phi_{i-1}$

Counter	c_i	Φ	$\hat{c}_i$
00000			
00001			
00010			
00011			
	$\vdots$		

Lemma

If $\phi_0 = 0$ *and* $\Phi \geq 0$, *then* $\sum_i \hat{c}_i \geq \sum_i c_i$

Proof.

$\sum_i \hat{c}_i = \sum_i (c_i + \phi_i - \phi_{i-1}) = \sum_i (c_i) + \phi_n - \phi_0 \geq \sum_i c_i$ $\square$

Potential method

We know $\sum_i \hat{c}_i \geq \sum_i c_i$

Goal: show $\hat{c}_i = O(1)$

$\Rightarrow \sum_i c_i \leq \sum_i \hat{c}_i \leq n \cdot O(1) = O(n)$

Lemma

In the bit vector problem, it holds that $\hat{c}_i = 2$ *for any* $i \geq 0$

Proof.

Counter

$X =$

Rest of vector	0	1	1	1

sequence of 1s

$X + 1 =$

Rest of vector	1	0	0	0

$\square$

Summary

1. Defined a potential function Φ, representing cost of structure
2. Defined amortized cost $\hat{c}_i$ (real cost + change in potential)

   Real cost varies wildly but amortized remains consistent

3. Show that runtime= $\sum_i c_i \leq \sum_i \hat{c}_i$
4. Give an upper bound on $\hat{c}_i$
5. Combine above steps to obtain amortized bound

In order to use the potential you should

- Define a potential function Φ

  $\Phi(\text{start}) = 0,\ \Phi \geq 0$

- Give an upper bound on $\hat{c}_i$

Accountant method

Alternative version for bounding amortized runtime:

- Use a virtual currency:

  User **deposits** coins when invoking an operation
  To compute we must **withdraw** coins

- Each coin can be used to do $O(1)$ computations

- Golden rule: never run out of coins

Example 3: Dynamic Array

Simple data structure

- Data pointer + 2 counters (current size/max capacity)
- Insertion is often fast (first empty slot) $\Rightarrow \Theta(1)$
- If full, we **double** size of array. Copy everything and insert
  - $\Rightarrow \Theta(n)$ runtime if this happens
- Let's prove that amortized insertion time is $O(1)$

Accountant approach

Two operations involved in n insertions

- Insert: Insertion when array has space. User deposits 3 coins in each invocation
- Resize: Grow array when full. Implicitly invoked. 0 coins deposited

Checking balance

- Let's look at INSERTION
  - 3 coins are deposited
  - Insertion needs $O(1)$ number of operations
    - Check if space, insert, increase counter
    - $O(1)$ operations $\Rightarrow$ 1 coin withdrawn
    - Net gain: 2 coins. We associate them to the inserted element
- What about EXPAND?
  - 0 coins are deposited. Must withdraw coins to pay for runtime
  - **Key property**: only when array full.
    - Items in second half have 2 coins saved each.
    - $\Rightarrow$ one coin per element in array!
    - Withdraw 1 coin for element we copy onto bigger array
    - $\Rightarrow$ Use all coins but never go negative
    - After expansion half of the array is full

Big picture

Theorem

n insertions in a dynamic array will need $\Theta(n)$ time in total.
(or insertion in a dynamic array uses $\Theta(1)$ amortized time).

User deposits 3 coins per $\textsc{Insert}$ (0 on $\textsc{Expand}$)

n operations invoked $\Rightarrow 3n$ coins deposited

Coins withdrawn each time computer spends time

Never go negative balance $\Rightarrow$ at most $3n$ coins withdrawn

Each coin is $O(1)$ operations $\Rightarrow$ at most $3n \cdot O(1)$ time spent

Food for thought

Showed that insertion on DA needs $\Theta(1)$ amortized

What did we change? Nothing!

Amortized analysis just changes... analysis

Same algorithm, just better bounds

Brute force counting Just count. Needs critical idea

Potential Associate a Potential to data structure

Change in potential *flattens* amortized runtime

Accounting Associate runtime to coins

Keep track of deposit/withdrawals

Never run out of coins