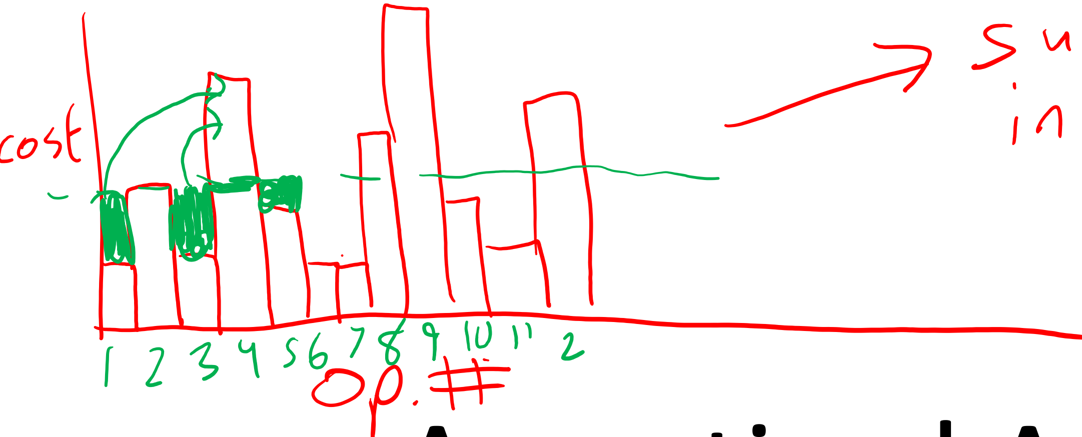
Announcements

~~Amortized Analysis: Example~~ Announcement

- HW6 out tonight, due May 1

~~do~~ Goal: find sum (average) of costs for sequence of operations

- direct way: add up the costs!
  $x_1 + x_2 + \cdots + x_n$, where x_i is the cost of the i^{th} op.
- slightly less direct way: break user-level operations into atomic ops, count # times each atomic op. occurs
  - banking method gives upper bound on cost



sum/ integral of this cost function

Amortized Analysis

$$\text{average} = \frac{\text{sum}}{\#\text{operations}}$$

sometimes, we can do this through formulas

other times, we need a different technique

Amortized Analysis: Definition

- The amortized cost of a sequence of *n* operations is the ***average cost per operation, for the worst-case sequence***

-or, cost for the entire sequence of operations as a function of # ops in sequence

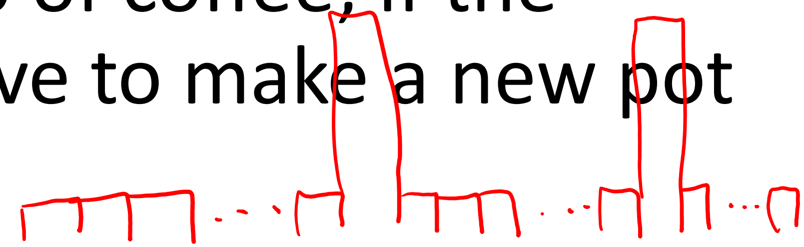
for both, as # ops $\rightarrow \infty$

Amortized Analysis: Basic Idea

- Keep a bank:
  - When you perform a cheap operation, put the savings in the bank
  - When you perform an expensive operation, use your savings to pay for it
- Just make sure that it never goes negative!

In-Class Exercise: Getting a Cup of Coffee

- Suppose there is a coffee-pot in some room
- People come and pour a cup of coffee; if the coffee-pot is empty, they have to make a new pot of coffee before they pour.
- Each pot contains 10 cups.
- Pouring a cup takes 1 unit of time
- Brewing a new pot takes 10 units of time
- What is the worst-case running time for getting one cup of coffee? The amortized running time?



average cost as # ops $\to \infty$ ~~to~~ is 2 units of time

In-Class Exercise: Binary Counter

0011111
0100000

- Suppose you have a binary array
- You are using it as a counter:
  - In each increment, increase value by 1
  - E.g., [0, 0, 1, 1, 1, 0, 1] -> [0, 0, 1, 1, 1, 1, 0]
- If there are n increments, and in the worst case, each might require changing all values, running time = $O(n^2)$ $O(n \log n)$

In-Class Exercise: Binary Counter

- Suppose you have a binary array
- You are using it as a counter:
  - In each increment, increase value by 1
  - E.g., [0, 0, 1, 1, 1, 0, 1] -> [0, 0, 1, 1, 1, 1, 0]
- Suppose that it costs $1 to flip a bit
- Keep a "bank" for each bit
- Suppose whenever you flip a bit from 0 to 1, you pay $2: $1 to flip, $1 to that bit's bank, 1 → 0 flip, take from bank
- Perform amortized running-time analysis

In-Class Exercise: Binary Counter

1. What is the algorithm for incrementing a binary counter?
   - start at rightmost bit, if 1, flip to 0 and move left
   - continue til we find a 0, flip to 1 and stop

2. How many 0 → 1 flips can be in an increment? How many 1 → 0 flips?

   1 → $2 → $1 for flip, $1 for bank

   unlimited

   actual cost of $1

   amortize cost of $0

3. If a bit is currently set to 1, what was the last thing done to it? How much is in its bank?

   $1

   0 → 1 flip

Binary Counter: Another Solution

- Over n increments, how often do we flip the first bit? n

  1 bit 0

  1 bit 1

- How often do we flip the second bit? $n/2$
- How often do we flip the k^{th} bit? $n/2^k$

first

- What is the total running time?

$$(x_1 + x_2 + x_3 + \cdots + x_n)$$

cost of I1, cost of I2

$$\left(n + \frac{n}{2} + \frac{n}{2^k} + \cdots = 2n\right)$$

I1, I2, I3, …, IN

I2, I4, I6

In-Class Exercise: Binary Counter v2

- Suppose you have a different binary counter
- But now the cost to flip the k^{th} bit is 2^k (bit on the right is bit 0)
- What is the total running time? (average)
- (Hint: modify our second analysis for simple binary counter)

#increments	
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000

~~value~~ string in binary counter

Binary Counter: Another Solution

cost to flip k^{th} bit is 2^k bit 0

- Over n increments, how often do we flip the first bit? cost? $n \cdot 2^0 = n$
- How often do we flip the second bit? cost? $n/2 \cdot 2 = n$
- How often do we flip the k^{th} bit? cost?

$$\left(\frac{n}{2^k}\right) \cdot \left(2^k\right) = \underline{n}$$

- What is the total running time?

$$n + n + \dots + n = O(n \log n)$$

B0 B1 Bit # log n average = log n

In-Class Exercise: Binary Counter v2

Dictionaries

$11 = 1\cdot(2^0) + 1\cdot(2^1) + 0\cdot(2^2) + 1\cdot(2^3)$

set

A3 A1 A0

$11 \equiv 1011$

- Implement a "dictionary" as a collection of arrays

list

- Array *i* has length 2^i
- An array can be full or empty
- Each array is sorted, but no relationships between elements in different arrays

convert k to its binary representation, check where 1s are

```
A0:   [5]
A1:   [4,8]
A2:   empty
A3:   [2, 6, 9, 12, 13, 16, 20, 25]
```

Suppose we have k elements, which arrays are used?

k = 11, arrays used = {A0, A1, A3}

Dictionaries

- Question: Why can data of any size *n* be stored in such a structure? binary representation
- A lookup can be done with binary search on each array

Dictionaries

- Question: Why can data of any size n be stored in such a structure?
- A lookup can be done with binary search on each array:
  - $O(\log 1) + O(\log 2) + O(\log 4) + \ldots + O(\log n/2)$
    $= O(\log^2 n)$

Dictionaries

- Suppose we want to insert value 10
- Create length-1 array A_{temp} with "10"
- Check if A_0 is empty: if it is, then just use that
- If A_0 is not empty, call "merge" on A_0 with A_{temp}
- Check if A_1 is empty: if it is, then just use that
- If A_1 is not empty, call "merge" with it
- Check if A_2 is empty: if it is, then just use that
- If A_2 is not empty, "merge" with it
- …

$|A_0| = 1$

$|A_{temp}| = 1$

→ array size = 4

Dictionaries

equivalent to flipping bit 0

$A_{temp} = [10]$; cost = \$1 = 2^0

@4

$11 = 1011$

$12 = 1100$

A0: ~~[5]~~

A1: ~~[4, 8]~~

A2: ~~empty~~ [4, 5, 8, 10]

A3: [2, 6, 9, 12, 13, 16, 20, 25]

$merge(A0, A_{temp}) = [5, 10]$: cost \$2 = 2^1

$merge([5, 10], A1) = [4, 5, 8, 10]$: cost \$4 = 2^2

$merge(X, Y) = O(k)$

$|X| = |Y| = k$

Dictionaries: Costs

– What is cost of n insertions?

A temp

- Suppose creating initial array of size 1 costs \$1
- Merging two arrays of size *m* costs \$2m
- ~~How much did the previous insert cost?~~

1011

100

$2^2 \; 2^1 \; 2^0$

$2^2 \; 2^1 \; 2^0$

In-Class Exercise: Dictionaries

$1011 \rightarrow 1100$

- Use the binary counter v2 problem to figure out the amortized running time of inserting *n* elements into the dictionary

BCv2: cost to flip bit k was 2^k

dictionary: cost to merge with array A_k?

A_k has 2^k elements, so if we merge with it, cost is $2 \cdot 2^k$

identical to BCv2, average cost ~~of~~ for insertion is $O(\log n)$

In-Class Exercise: Dictionaries

In-Class Exercise: MultiPop

- We are building a stack-like data structure that supports the following operations:
  - Push - user, atomic
  - Pop - user, atomic
  - MultiPop(k): Pops top-k elements instead of 1 (up to size of stack). Implemented as a series of Pop operations. - user

Push, Pop cost $1

Push Pop Push Push MP Push ...

In-Class Exercise: MultiPop

- Analyze running time for a sequence of *n* Push, Pop, MultiPop operations
- Standard analysis:
  - Push takes O(1)
  - Pop takes O(1)
  - MultiPop(k) takes O(k)
  - Worst case: *n* MultiPop operations = O(n^2) for *n* operations = O(*n*) per operation on average

single MP is O(n) in worst case

In-Class Exercise: MultiPop

- Do an amortized analysis instead!

Push: \$2 → \$1 pays for push
→ \$1 to bank

Pop: \$0 → take \$1 from bank

* MultiPop: \$0 → take \$k from bank

worst-case seq.: $\boxed{\$2/\text{operation}}$

or \$2n total