Amortized Analysis

Given a data structure and some operations, let $T(n)$ be the sum of upper bounds on worst case runtimes for n operations

Amortized cost is $T(n)/n$ (basically average)

- doesn't involve probability/expectation - always use worst case
- takes into accoutn that cost of operation is not always the same
- Comes in two *flavours*:
    - aggregate (i.e. brute force)
        - count all costs over all n executions, then divide by n
    - accounting method
        - if c_i is actual cost of operation i, $\hat{c}_i$ is how much we (over)charge
        - $\sum \hat{c}_i - \sum c_i \geq 0$

Example - binary counter

Aggregate method

Using a bit array A, $k = |A|$,

```
def Increment(A):
    i = 0
    while i < |A| and A[i] == 1:
        A[i] = 0
        i = i + 1
    if i < |A|:
        A[i] = 1
```

Naively, the runtime for `Increment(A)` is $\in \mathcal{O}(k)$, so calling it n times results in a total runtime of $T(n) \in \mathcal{O}(nk)$, so amortized cost is $nk/n = k$

However, using more careful aggregate analysis:

for n increments:

- $A[0]$ flips n times
- $A[1]$ flips $\lfloor n/2 \rfloor$ times
- $A[2]$ flips $\lfloor n/4 \rfloor$ times
- in general, $A[i]$ flips $\lfloor n/2^i \rfloor$ times

$$T(n) = \sum_{i=0}^{k-1} \lfloor n/2^i \rfloor$$

$$\leq \sum_{i=0}^{k-1} n/2^i$$

$$\leq n \sum_{i=0}^{\infty} 2^{-i}$$

$$= 2n$$

So the amortized cost is $2n/n = 2$

Accounting method

Define:

- c_i as the actual cost of operation i
- $\hat{c}_i$ as how much we (over)charge for operation i
  - overcharge so that we charge in advance, then don't charge later

Idea:

- charge $\$$ for flipping $0 \to 1$
- charge another $\$$ for flipping $1 \to 0$ *later*