Amortized Analysis

- amortized analysis is the evaluation of the average cost over a sequence of operations on a data structure
- the average cost maybe small although a single operation can be expensive
- it is not the cost for the average case and doesn't involve probability analysis

1. Aggregate Analysis

- the amortized cost is T(n)/n where T(n) is the worst case
- it applies to any operation in a sequence of n operations
 - o operations can be different types

Insertion to a dynamic array

- ullet items can be inserted at a given index with O(1) if the index is present in the array
- if not, then the array double in size and the cost is not longer constant

$$c_i = 1 + egin{cases} i-1 & ext{if } i-1 ext{ is power of 2} \ 0 & ext{otherwise} \end{cases}$$

ullet if we insert n elements then

$$rac{\sum_{i=1}^n c_i}{n} \leq rac{n + \sum_{j=1}^{\lfloor \lg{(n-1)}
floor} 2^j}{n} = rac{O(n)}{n}$$

notice that

$$\sum_{j=0}^a 2^j = 2^0 + \ldots + 2^a = 2^{a+1} - 1$$

$$\sum_{j=1}^{\lfloor \lg{(n-1)}
floor} 2^j = 2^{\lfloor \lg{(n-1)}
floor+1} - 1 - 1$$

ullet note that we subtract with another -1 because we start at j=1

$$egin{align*} \sum_{j=1}^{\lfloor \lg{(n-1)}
floor} 2^j &= 2*2^{\lfloor \lg{(n-1)}
floor} - 2 \ &= 2*(n-1) - 2 \ &= O(n) \end{aligned}$$

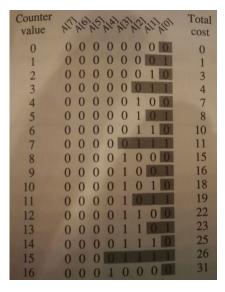
Stack Operations (multipop)

- push(s, x) pushes x onto s in O(1)
- pop(s, x) in O(1)
- multipop(s, k) pops k top elements from s if the size $\geq k$ otherwise it pops all elements \circ at most O(n)
- what is the amortized cost of a sequence of n push, pop, multipop operations?
 - \circ the size of the stack is n

- of or any n, the cost of a sequence of n of these operations is O(n) (as we can't pop more than n)
- amortized cost is O(n)/n = O(1)

Binary Counter

- ullet A[0...k-1] is an array denoting a k-bit binary counter that starts at 0
 - \circ adding 1 to A[i] flips it
 - $\circ \;\;$ if A[i]=1 then it yields a carry to A[i+1]



- notice that A[0] flips n times
 - $\circ \ A[1]$ flips n/2 times
 - $\circ \ A[2] \ {
 m flips} \ n/4 \ {
 m times}$

$$\sum_{i=0}^{\lfloor \lg n \rfloor} \lfloor \frac{n}{2^i} \rfloor < n \sum_{i=0}^{\infty} 1/2^i = n \times \frac{1}{1-\frac{1}{2}} = 2n$$

• amortized cost is O(n)/n = 1

2. Accounting Method

- ullet for different operations, we "charge" a specific amount \hat{c}_i different than their actual costs c_i
 - o can be less or more
- when amortized cost is more than the actual then
 - o we store the excess credit into the object
 - o credit is stored for future use when the amortized cost is less than the actual
- how do we assign amortized costs?
 - the total amortized cost *must* be an upper bound on the actual cost

$$\sum_{i=1}^n \hat{c}_i \geq \sum_{i=1}^n c_i$$

o thus the total credit in the data structure is

$$\sum_{i=1}^n \hat{c}_i - \sum_{i=1}^n c_i \geq 0$$

Stack Operations

- actual costs
 - o push is 1
 - o pop is 1
 - \circ multipop is $\min(k,s)$ where s is the length of S
- amortized cost
 - o push is 2
 - o pop is 0
 - o multipop is 0
- analysis
 - each object in the stack has 1 "coin" of credit on it because it costs 1 to push and 1 gets saved
 - the total credit for a stack is going to be nonnegative as we can never pop more than what the stack has

3. Potential Method

- similar to the *accounting method*, however instead of storing credit, we store "potential"
- the potential is stored with the entire data structure instead of just a single object
- ullet c_i is the actual cost
- ullet D_i is data structure after the ith operation to D_{i-1}
- ullet $\phi(D_i)$ is the potential associated with D_i
- \hat{c}_i is the amortized cost of the ith iteration and is defined as $\hat{c}_i = c_i + \phi(D_i) + \phi(D_{i-1})$
- the total amortized cost is

$$\sum_{i=0}^n \hat{c}_i = \sum_{i=1}^n [c_i + \phi(D_i) - \phi(D_{i-1})] = \sum_i^n c_i + \phi(D_n) - \phi(D_0)$$

Stack Operations

Let the potential of a stack ϕ be the *number of elements* in the stack.

Binary Counter

Let the potential of the counter ϕ be the *number of the 1's* in the counter.