Motivation for amortized analysis

```
DFS ( G graph, v ∈ V(G))
    [ M an array of marked vertices   M[v] = 1
                                      M[u] = 0 ∀u
      P array of Parents              P[v] = v
                                      P[u] = Null
    S a stack = [v]
O(n) [While S ≠ ∅ Do
        t = top(S)
        While M[ firstnbhr(t)] = 1 Do
            delete firstnbhr of t
        if neighbors(t) ≠ ∅ :
            u = firstnbhr of t
            M[u] = 1   P[u] = t
            S.push(u)
        else S.pop()
```

O(deg(t)) = O(n)

sloppy analysis = $O(n^2)$

Sum amount of work done every time a vertex x is at top of the stack - we're doing a total of $O(\deg(x))$ work.

$\Rightarrow$ complexity is $O\left(\sum_{x \in V} \deg(x)\right)$

$= O(n+m)$

Def amortized analysis averages the time required to perform a sequence of data structure operations over all the operations performed.

3 main techniques of doing this analysis:

- aggregate analysis
- accounting method
- potential method

I aggregate analysis

total cost is $T(n)$ find

average cost $\frac{T(n)}{n}$

consider operations on a stack S

- pop from stack - ~~return~~
  - remove top element & return it if $S \neq \emptyset$ & otherwise return Null
- push - push an element onto top of stack
- multi-pop(k) - remove top k elements of the stack & return them if $|S| \geq k$ otherwise remove all elts of the stack & delete them.

Q what is the complexity of a sequence of n push, pop & multipop operations?

push - $O(1)$
pop - $O(1)$
multipop - $O(k) = O(n)$ because stack always has $\leq n$ elts.

trivial bound is

$$O(n \cdot n)$$

↑ # of operations; ↑ worst case on each operation

$$= O(n^2)$$

- we can't have n operations of multipop each costing $O(n)$ complexity because

# of pops total in all multipops + individual pops put together is at most the total # of pushes

$T_1(n) :=$ amt of work in pushes

$T_2(n) =$ amt of work in pops

$T_2(n) \leq T_1(n)$

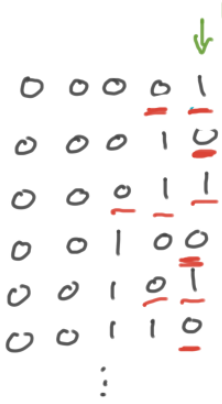
total amount of work is $O(n)$

+ amortized cost of each step is $O(1) = O(\frac{T(n)}{n})$

Note This is not probabilistic

we have worse-case bound on amt of work to do.

k-bit binary counter




```
Increment (A)
    i = 0
    while i < A.length() + A[i] = 1
        A[i] = 0
        i++
    if i < A.length()
        A[i] = 1
```

Sloppy analysis on run-time

$O(k)$ work $*$ w/ n updates

(w/ $n \leq 2^k$)

$O(k \cdot n)$ work total

we are certainly not doing $O(k)$ work at each step - in fact we do just one bit-flip every other increment.

How much work is done for the i^{th} bit over the series of n operations?

bit 1 flips every single time

$= n$ flips

bit 2 flips $n/2$ times

bit 3 flips $n/4$ times

i^{th} bit flips $\left\lfloor \frac{n}{2^i} \right\rfloor$

$=$ total cost for i^{th} bit

$\Rightarrow$ amortized cost for i^2 bit

$$\left\lfloor \frac{n}{2^i} \right\rfloor \cdot \frac{1}{n} \leq \frac{1}{2^i}$$

$\Rightarrow$ amortized cost for a single increment of the counter

$$= \sum_{i=1}^{k} \text{amortized cost of increment for } i^{th} \text{ bit}$$

$$\leq \sum_{i=1}^{k} \frac{1}{2^i} \leq 1$$

$\Rightarrow$ Cost for n updates is $O(n)$

II accounting method

assign a cost to each subroutine (possibly more or less than the actual complexity theoretic costs)

- if operation requires less work than the assigned cost, we "bank" whatever work is not used
- if an operation requires more work than the assigned cost, we make-up the shortfall from the excess stored in the bank.
- Trick assign "correct" costs to each subroutine to balance everything ou

Stack operations

	complexity	cost assigned
push	1	2
pop	1	0
multipop(k)	k	0

We need to show that in any series of operations, we always have enough stored in the bank to allow us to make up any shortfall

Pop after a series of n operations

The amount extra stored in the bank = $|S|$

Pf by induction on n

$n = 1$

push - stack has length 1
we have 1 in bank ✓

pop / multi-pop 0 work, $S = \emptyset$
+ 0 in bank ✓

assume we have a sequence of $n+1$ operations + by induction after first n of them, bank has $|S|$ in it.

3 cases for $n+1^{st}$ operation:

- push → stack has size +1 + bank also has +1
- pop
- multipop. → to do operation, take k out of the bank + we take k off stack. maintaining

maintaining parity between bank & stack.

Cor bank always sufficient resources to perform any sequence of operation

Complexity of a series of operations is

$\leq$ worst case cost of a series of operations

$\leq \sum_{i=1}^{n} 2 = 2n = O(n)$

Ex use accounting method to analyze incrementing a binary counter

Hint pick the right cost to charge so that at each pass we only charge constant

```
increment (A)
    i = 0
    while i < A.length() + A[i] = 1
charge 0 ->     A[i] = 0
                i ++
charge 2 -> if  i < A.length()
                A[i] = 1
```

flip to 1 - cost of 2

flip to 0 - cost of 0

to prove correctness, we still need to show that the reserve in the bank always covers # switches to 0 we have to do.

to do this prove by induction:

bank = # of ones in the string

specifically, this is ≥ 0 & has enough to flip the first j instances of 1 to be 0

III Potential method

let D_i be the state of our data structure after i steps

Def a potential function $\Phi(D_i) \in \mathbb{N}$

$$\hat{c}_i := c_i + \Phi(D_i) - \Phi(D_{i-1})$$

↑ cost of arriving at D_i from D_{i-1}

cancel out

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

This is what we want to calculate – cost of arriving at the final instance D_n

We want an upper bound on $\sum c_i$

if we additionally choose our potential function Φ s.t. $\Phi(D_0) = 0$

- $\Phi(D_0) = 0$
- $\Phi(D_i) \geq 0 \quad \forall i$

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

intuitively, this is similar to accounting method $\$$

- if $\Phi(D_i) > \Phi(D_{i-1})$ we're putting additional "money" into the bank
- if $\Phi(D_i) < \Phi(D_{i-1})$ we've withdrawn from bank to make up any shortfalls

Stack operations

$\Phi(D_i)$:= # items in the stack

$\Phi(D_0) = 0$

$\Phi(D_i) \geq 0 \quad \forall i$

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$

if i^{th} operation

- push
- multipop

i^{th} operation is a push

$$\hat{c}_i = 1 + \left(\overline{\Phi}(D_i) + 1\right) - \overline{\Phi}(D_{i-1})$$

$$= 2$$

if i^{th} operation is instead a multipop

$$\hat{c}_i = k + \overline{\Phi}(D_i) - \overline{\Phi}(D_{i-1})$$

$$= k + \left(\overline{\Phi}(D_{i-1}) - k\right) - \overline{\Phi}(D_{i-1})$$

$$= 0$$