Introduction

Today we begin studying how to calculate the total time of a sequence of operations as a whole. (As opposed to each operation individually.)

Why and when we care:

- you have an algorithm that uses a certain data structure
- some operation takes a long time, but not a problem because...
- you just need the total time to be good
Multi-pop stack operations:

- push(x): implementation as usual, $\Theta(1)$ time
- pop(): implementation as usual, $\Theta(1)$ time
- multipop(i): calls pop() up to i times, $\Theta(i)$ worst case time (probably silly, but on purpose to show something)

Start from empty and perform $m$ operations. What is the total time?
Multi-Pop Stack Cost (naïve)

Start from empty and perform $m$ operations. What is the total time?

A naïve calculation may get a gross overestimate:

1. each operation may take $O(i)$ time
2. $i$ may be near $m$ because stack size may be near $m$
3. total: $m$ operations take $O(m^2)$ time

You may see what’s wrong with it:
Multi-Pop Stack Cost (naïve)

Start from empty and perform $m$ operations. What is the total time?

A naïve calculation may get a gross overestimate:

1. each operation may take $O(i)$ time
2. $i$ may be near $m$ because stack size may be near $m$
3. total: $m$ operations take $O(m^2)$ time

You may see what’s wrong with it:

- either multipop happens too rarely to matter
- or multipop happens so often, stack size stays small

How to calculate a better answer?
Multi-Pop Stack Cost (clever)

Starting from empty, perform $m$ operations:

1. at most $m$ pushes, $m$ pops: total $\Theta(m)$
2. stack size is always at most $m$
3. all multipops together cannot pop more than $m$ times: total $\Theta(m)$
4. total: $m$ operations take $\Theta(m)$ time, worst case
Multi-Pop Stack Cost (clever)

Starting from empty, perform $m$ operations:

1. at most $m$ pushes, $m$ pops: total $\Theta(m)$
2. stack size is always at most $m$
3. all multipops together cannot pop more than $m$ times: total $\Theta(m)$
4. total: $m$ operations take $\Theta(m)$ time, worst case

You may be tempted to say:

- each operation takes $\Theta(1)$ time in some sense

The next slide tells you the magic word for that!
Amortized Time

$m$ operations take $\Theta(m)$ total time, worst case.

Each operation takes $\Theta(1)$ amortized time.

This name is inspired by obvious analogies with paying for mortgages etc.
Amortization Method #0: Aggregate

Aggregate method: like what I just did about multi-pop stacks.

1. at most $m$ pushes, $m$ pops: total $\Theta(m)$
2. stack size is always at most $m$
3. all multipops together cannot pop more than $m$ times: total $\Theta(m)$
4. total: $m$ operations take $\Theta(m)$ time, worst case

In general: holistically calculate total worst-case time.

Not much of a method. Mainly ad-hoc arguments and tailor-making to each situation.
Amortization Method #1: Banker/Accounting

Banker/Accounting method:

Using multi-pop stacks again, imagine:

- each operation receives 2 bitcoins cyber-dollars
- push spends 1 cyber-dollar; pop does too
- multipop(i) spends \( \min(i, size) \) cyber-dollars
- if surplus: save up for future spending
- if deficit: spend past savings

Prove: always have non-negative savings. (Next slide.)

Conclude: each operation takes \( O(2) \) amortized time (i.e., what it receives).
Prove: Multi-Pop Stack Savings ≥ 0

- push: $1 surplus, non-negative savings
  imagine attaching this 1 dollar to the item pushed
- pop: surplus, non-negative savings
- multipop(i): spending = number of items popped
  So just spend the money attached to those items
  ⇒ non-negative savings
Banker/Accounting Method: Summary

Banker/Accounting method: Imagine:

- each operation receives some cyber-dollars
  this will become amortized time
- each operation spends some cyber-dollars
  this should equal worst-case time
- if surplus: save up for future spending
- if deficit: spend past savings

Prove: always have non-negative savings.

Suggestion: Imagine a way to distribute savings over data. This helps with deficit cases in the proof.
Amortization Method #2: Physicist/Potential

Physicist/Potential method:

Using multi-pop stacks again:

Choose “potential” $\Phi_i = \text{stack size after } i \text{ operations.}$

Let $a_i = \text{operation } i \text{’s time} + \Phi_i - \Phi_{i-1}$

$$\sum_{i=1}^{m} a_i = \sum_{i=1}^{m} \text{operation } i \text{’s time} + \Phi_i - \Phi_{i-1}$$

$$= \text{total time} + \Phi_m - \Phi_0$$

$$\geq \text{total time}$$

Can use $a_i$ as amortized time (does not underestimate).

- push: amortized time $= 2$
- pop: amortized time $= 0$
- multipop(i): amortized time $= 0$
Physicist/Potential Method: Summary

Physicist/Potential method:

Choose “potential” $\Phi_i = a$ function of data structure state after $i$ operations.

Prove: $\Phi_m \geq \Phi_0$.

Then amortized time = operation $i$’s time + $\Phi_i - \Phi_{i-1}$.
One More Example: Binary Counter Increment

Put a $k$-bit number in an array $C$ of $k$ bits. LSB at $C[0]$. Initially all 0’s.

increment():
    i := 0
    while i < C.length && C[i]=1:
        C[i] := 0
        i := i + 1
    if i < C.length:
        C[i] := 1

(For this example: modifying a bit takes $\Theta(1)$ time.)

Up to $k$ bits could be already 1. Increment takes $\Theta(k)$ time worst case. What about a sequence of $m$ increments?
Binary Increment: Aggregate Method

- $C[0]$ is modified $m$ times
- $C[1]$ is modified $\lfloor m/2 \rfloor$ times
- $C[2]$ is modified $\lfloor m/4 \rfloor$ times
- $C[i]$ is modified $\lfloor m/2^i \rfloor$ times

Total number of modifications:

$$\sum_{i=0}^{k-1} \left\lfloor \frac{m}{2^i} \right\rfloor < \sum_{i=0}^{\infty} \frac{m}{2^i} = 2 \cdot m$$

$m$ increments take $O(m)$ total time. Amortized time $O(1)$. 
Binary Increment: Accounting Method

Distribution of savings: attach $1 to each bit storing 1.

Base case: initially, $0 savings, no bit stores 1.

Induction step:
If each bit storing 1 has $1 saved before increment:

- increment receives $2
- change some bits from 1 to 0: spend their attached dollars (does not use the received $2)
- may change a bit from 0 to 1: spend $1, save $1

Then each bit storing 1 has $1 saved after increment.

Savings $\geq$ how many bits store 1’s, non-negative.
Amortized time $O(2)$, i.e., $O(1)$. 
Binary Increment: Potential Method

Choose $\Phi_i$ = number of bits storing 1’s after $i$ increments.
Check: $\Phi_m \geq \Phi_0$ because $\Phi_0 = 0$.
Can use $i$th increment’s time + $\Phi_i - \Phi_{i-1}$ for amortized time.

Amortized time: at each increment:

- Say, $t$ bits are changed from 1 to 0.
- In addition, may change 1 bit from 0 to 1.
- This takes time $\leq t + 1$.
- $\Phi_i \leq \Phi_{i-1} - t + 1$

Amortized time $\leq (t + 1) + (1 - t) = 2$. 
Trailers

Tutorial: A more important example than today’s!

Next lecture: Amortized time of union-find!