CSC263 Week 7

Announcements

Problem Set 3 is due this Tuesday!

Midterm graded and will be returned on Friday during tutorial (average 60%)

Amortized Analysis

- Often, we perform **sequences** of operations on data structures
- Define **"worst-case sequence complexity"** of a sequence of m operations as:

  maximum total time over all sequences of m operations

Similar to worst-case time for one operation

- Worst-case sequence complexity is at most:

  m(worst-case complexity of any operation)

- But is it really always that bad?

Amortized analysis

- We do amortized analysis when we are interested in the total complexity of a **sequence** of operations.
  - Unlike in average-case analysis where we are interested in a **single** operation.

- The ***amortized sequence complexity*** is the "**average**" cost per operation **over the sequence**.
  - But unlike average-case analysis, there is **NO** probability or expectation involved.

For a sequence of m operations:

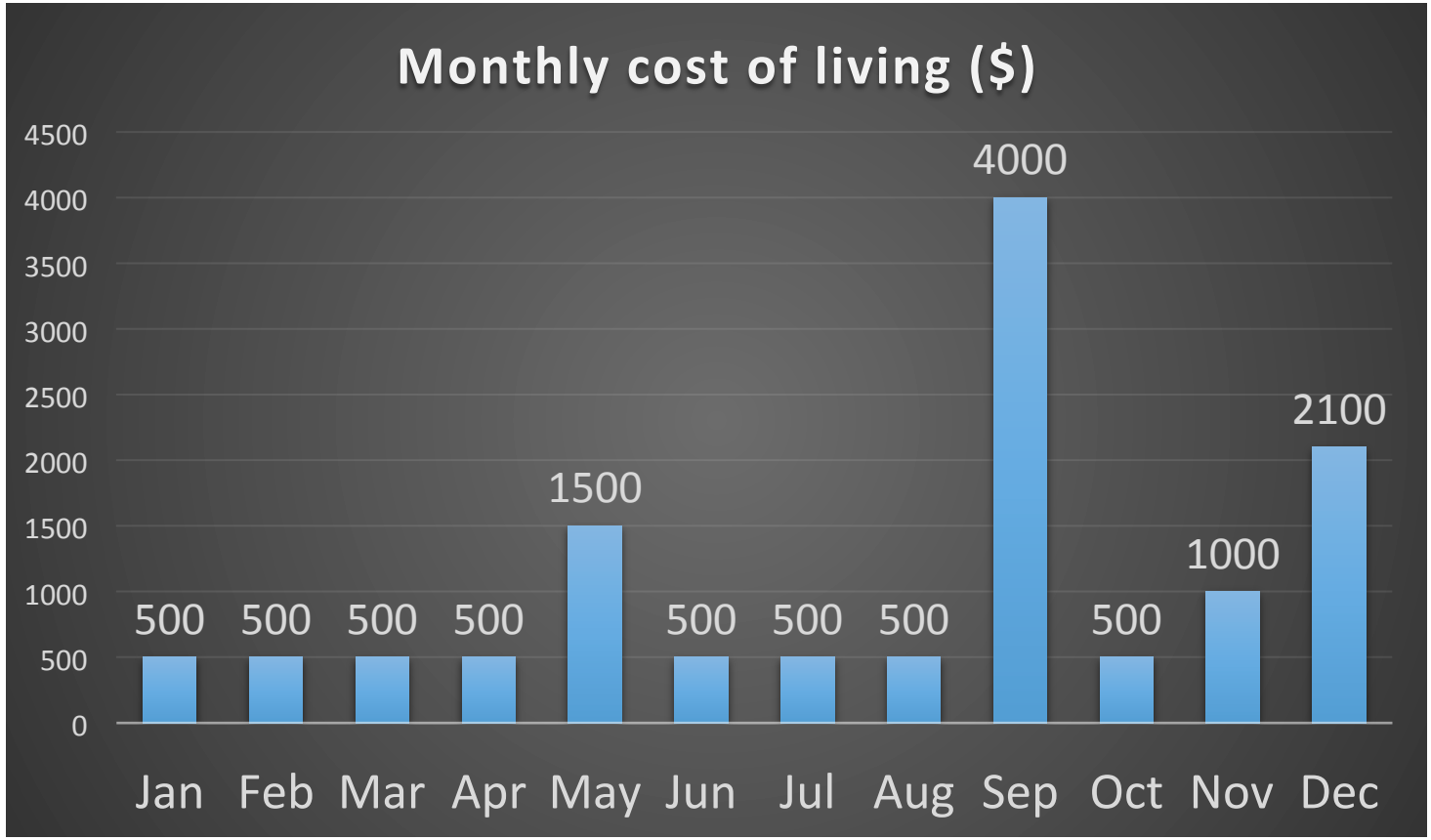
Amortized sequence complexity

$$= \frac{\text{worst-case } \textbf{sequence} \text{ complexity}}{m}$$

The MAXIMUM possible ***total*** cost of among all possible sequences of m operations

Amortized analysis

- Real-life intuition: Monthly cost of living, a sequence of 12 operations



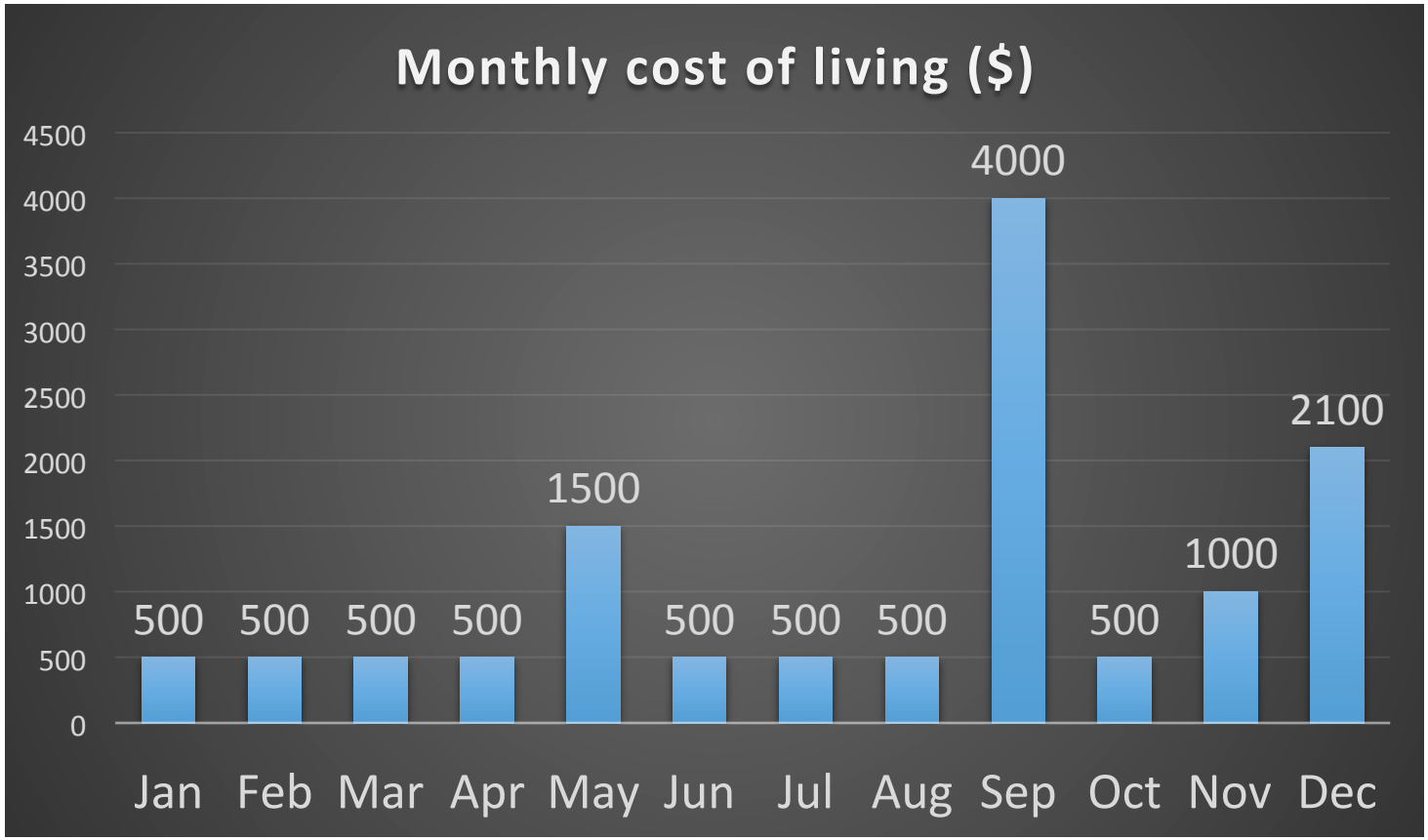
Methods for amortized analysis

- Aggregate method
- Accounting method
- Potential method (skipped, read Chapter 17 if interested)

Aggregate method

What is the amortized cost per month (operation)?

Just **sum up** the costs of all months (operations) and **divide** by the number of months (operations).



Aggregate method: sum of all months' spending is $126,00, divided by 12 months

– the amortized cost is $1,050 per month.

Binary Counter

- Sequence of k bits (k fixed)
- Single operation INCREMENT: add 1 (in binary)
- Cost of one INCREMENT: number of bits that need to change

Binary Counter

	0000
Add 1	0001
Add 1	0010
Add 1	0011
Add 1	0100
Add 1	0101
Add 1	0110
Add 1	0111
Add 1	1000

Binary Counter

Initially	0000	Cost
Add 1	0001	1
Add 1	0010	2
Add 1	0011	1
Add 1	0100	3
Add 1	0101	1
Add 1	0110	2
Add 1	0111	1
Add 1	1000	4

Binary Counter

- If we do n increments, the worst case complexity of any increment is log n.
- A naïve analysis would then say that the worst case complexity of n increments is (n log n)
- But it is never this bad since the worst case only happens once!

Aggregate Method for Binary Counter

Amortized cost of sequence of n INCREMENT operations:

bit	changes	total number of changes
0	every operation	n
1	every 2 operations	n/2
2	every 4 operations	n/4
...	...	...
i	every 2^i operations	$n/2^i$

Total number of bit flips during sequence = Σ_i (flips of bit i)

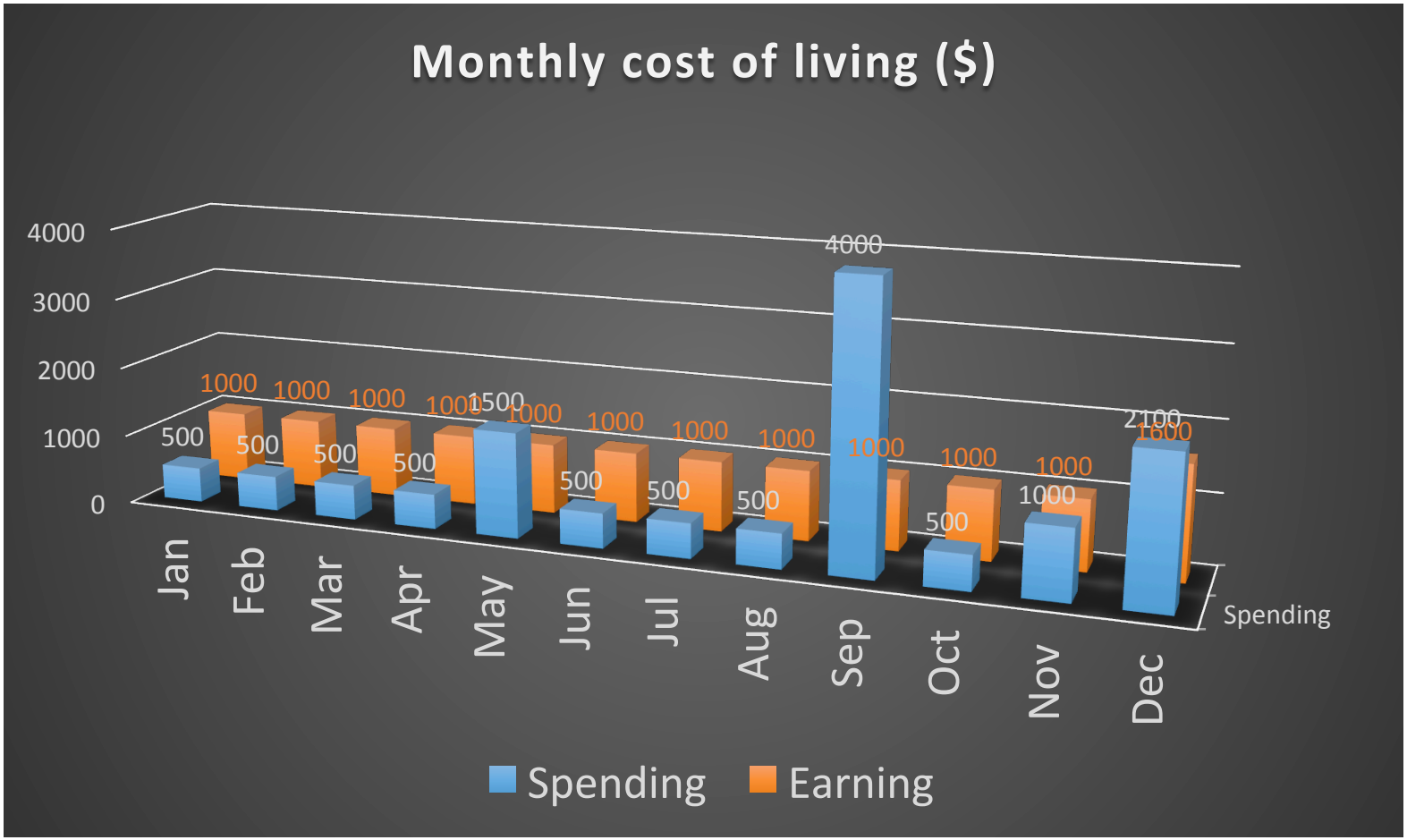
$= n + n/2 + \ldots + n/2^{\log n} = n(1 + ½ + ¼ + \ldots + 1/2^{\log n}) \leq 2n$

So amortized cost $\leq 2n/n = 2$ for each operation!

Accounting method

Instead of calculating the average spending, we think about the cost from a **different angle**, i.e.,

How much money do I need to **earn** each month in order to **keep living**? That is, be able to pay for the spending every month and **never become broke**.



Accounting method: if I **earn** $1,000 per month from Jan to Nov and earn $1,600 in December, I will never become broke (assuming earnings are paid at the beginning of month).

So the **amortized cost**: $1,000 from Jan to Nov and $1,600 in Dec.

Aggregate vs Accounting

- Aggregate method is easy to do when the cost of each operation in the sequence is concretely defined.

- Accounting method is more interesting
  - It works even when the sequence of operation is not concretely defined
  - It can obtain more refined amortized cost than aggregate method (different operations can have different amortized cost)

Accounting Method

- Find a charge (some number of time units charged per operation) such that:
  the sum of the charges is an upper bound on the total actual cost
- Like maintaining a bank account
  Low cost operations charged a bit more than their actual amount
  the surplus is deposited in the account for later use
- Analogy: Rahul earns 2K per month. Typically he spends 2K. On good months, he spends < 2K, and surplus goes in the bank to pay for the bad (expensive) months.
- Charges must be set high enough so that the balance is always positive.
- But if set too high: upper bound will be >> the (worst case) total actual cost.
- Goal: just scrape by -- Set charges as low as possible so that bank account is always positive

Accounting Method

- We want to show amortized cost is, say $5
- Assign a charge for each operation
- When charge > actual cost, leftover amount is assigned as credit (usually to specific elements in data structure)
- When an operation's charge < actual cost, use some stored credit to "pay" for excess cost.
- For this to work, need to argue that credit is never negative

If we have more than one operation, we can assign different charges to each one

Accounting Method for Binary Counter

- Charge each operation $2

  $1 to flip 0 → 1 (only one bit flips from 0 to 1)

  used stored credits to pay for flips 1 → 0

  $1 credit -- store with the bit just changed to 1

- **Credit Invariant:** At any step each bit of the counter that is equal to 1 will have $1 credit

Accounting Method for Binary Counter

Credit Invariant: At any step each bit of the counter that is equal to 1 will have $1 credit

Proof by induction:

- Initially counter is 0 and no credit
- Induction step: assume true up to some value of x and now consider next increment

Case 1: x = b .. b b 0 1 … 1 → b .. b b 1 0 … 0

(i least significant bits are 1, i+1st bit is 0)

i+1 = actual cost: use i credits to pay for i flips 1→0

use 1 out of 2 to pay for 0→1,

use 1 out of 2 for credit on the new "1"

Accounting Method for Binary Counter

Credit Invariant: At any step each bit of the counter that is equal to 1 will have $1 credit

Proof by induction:

- Initially counter is 0 and no credit
- Induction step: assume true up to some value of x and now consider next increment

Case 2: x = 1 1 … 1 → 0 0 … 0

(all bits are 1)

actual cost is k

use k credits to pay for k flips 1→0

extra $2 isn't needed.

Accounting Method for Binary Counter

Credit Invariant: At any step each bit of the counter that is equal to 1 will have $1 credit

- Thus invariant is always true
- So total charge for sequence is upper bound on total cost.
- Total charge = 2n so amortized cost per operation =2

NOTE: you need the invariant in order to show that the credit is always positive

Amortized Analysis on **Dynamic Arrays**

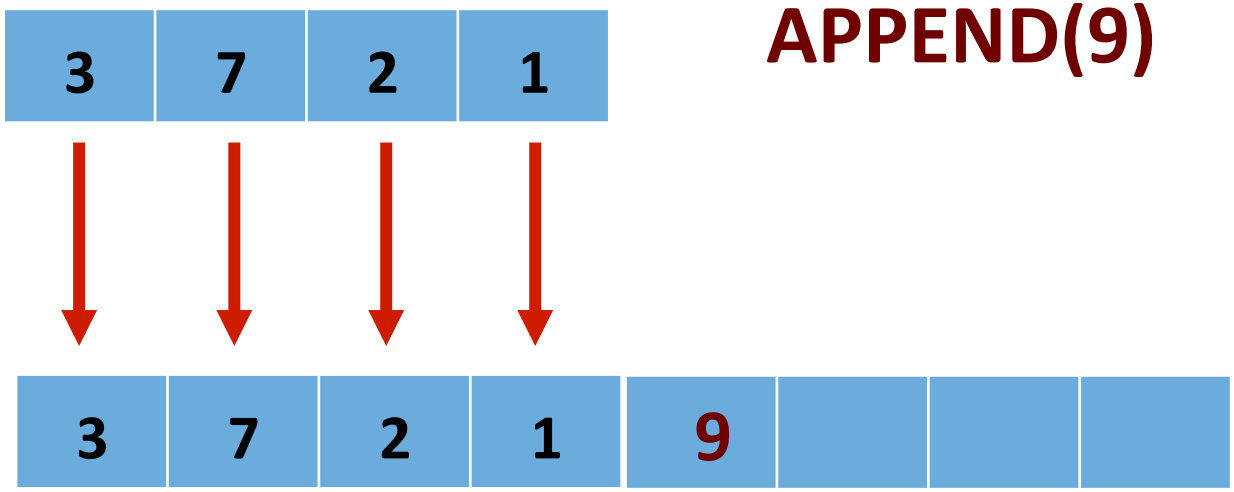
Problem description

- Think of an **array** initialized with a **fixed** number of slots, and supports **APPEND** and **DELETE** operations.
- When we APPEND too many elements, the array would be **full** and we need to **expand** the array (make the size larger).
- When we DELETE too many elements, we want to **shrink** to the array (make the size smaller).
- Requirement: the array must be using **one contiguous block** of memory all the time.

How do we do the **expanding** and **shrinking**?

One way to **expand**

- If the array is full when APPEND is called
  - Create a new array of **twice** the size
  - Copy the all the elements from old array to new array
  - Append the element



Amortized analysis of **expand**

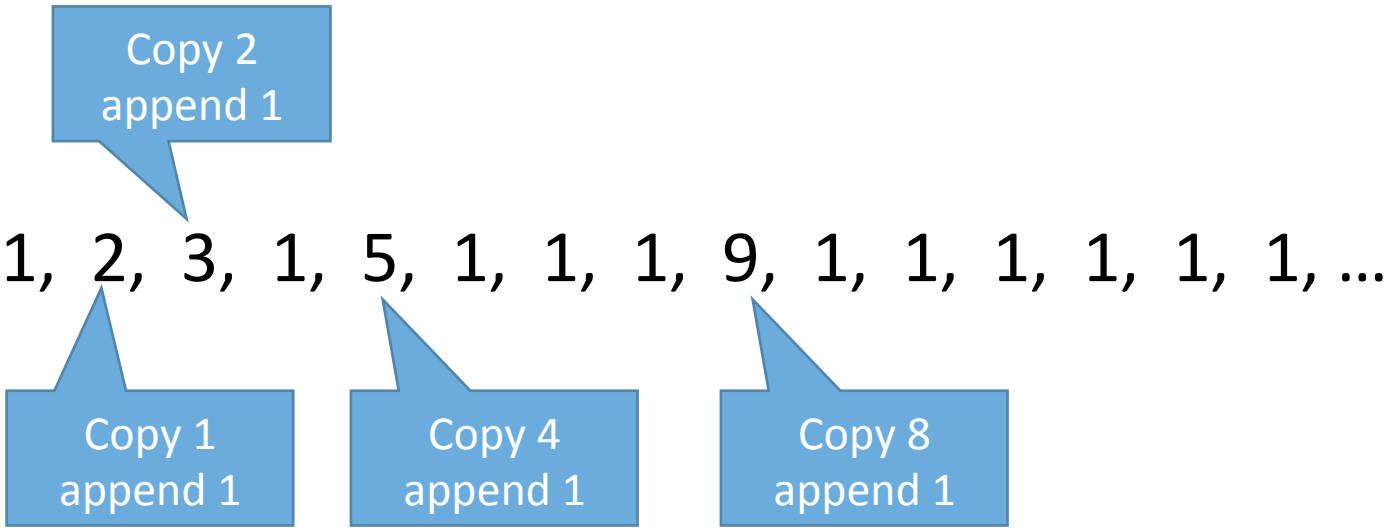
Now consider a dynamic array initialized with size 1 and a sequence of ***m*** APPEND operations on it.

Analyze the amortized cost per operation

Assumption: only count array assignments, i.e.,
append *an element and* ***copy*** *an element*

Use the **aggregate** method

The cost sequence would be like:



Cost sequence concretely defined, sum-and-divide can be done, but we want to do something more interesting…

Use the **accounting** method!

*How much money do we need to **earn** at each operation, so that all future costs can be paid for?*

*How much money to earn for **each APPEND'ed element** ?*

$1 ?

$2 ?

$3 ?

$log m ?

$m ?

First try: charge $1 for each append

This $1 (the “append-dollar”) is spent when appending the element.

But, when we need to copy this element to a new array (when expanding the array), we don’t have any money to pay for it --

BROKE!

This makes sense since the total cost of n appends is greater than n

Next try: charge $2 for each append

$1 (the “append-dollar”) will be spent when appending the element

$1 (the “copy-dollar”) will be spent when copying the element to a new array

What if the element is copied for a **second** time (when expanding the array for a second time)?

BROKE!

Third try: charge $3 for each append

$1 (the “append-dollar”) will be spent when appending the element

$1 (the “copy-dollar”) will be spent when copying the element to a new array

$1 (the “recharge-dollar”) is used to **recharge** the old elements that have spent their “copy-dollars”.

So one dollar stored to pay for my copy, and one for a friend

NEVER BROKE!

$1 (the “recharge-dollar”) is used to **recharge** the old elements that have used their “copy-dollar”.

Old elements who have used their “copy-dollars”

New elements each of whom **spares** $1 for recharging one old element’s “copy-dollar”.

There will be enough new elements who will spare **enough money** for **all** the old elements, because the way we expand – **TWICE the size**

Third try: charge $3 for each append

$1 (the “append-dollar”) to pay for append

$1 (the “copy-dollar”) as credit to pay for copy

$1 (the “recharge-dollar”) as credit to pay for friends’ copy

Credit invariant:

Each element in 2nd half of array has $2 credit

Third try: charge $3 for each append

$1 (the “append-dollar”) to pay for append

$1 (the “copy-dollar”) as credit to pay for copy

$1 (the “recharge-dollar”) as credit to pay for friends’ copy

Credit invariant:

Each element in 2nd half of array has $2 credit

Base case: no elements in array so true

Third try: charge $3 for each append

$1 (the "append-dollar") to pay for append

$1 (the "copy-dollar") as credit to pay for copy

$1 (the "recharge-dollar") as credit to pay for friends' copy

Credit invariant:

Each element in 2nd half of array has $2 credit

Inductive step.

Case 1: array not full

$1 to append, $2 stored on new item

Third try: charge $3 for each append

$1 (the "append-dollar") to pay for append

$1 (the "copy-dollar") as credit to pay for copy

$1 (the "recharge-dollar") as credit to pay for friends' copy

Credit invariant:

Each element in 2nd half of array has $2 credit

Inductive step.

Case 2: Array full; make new array

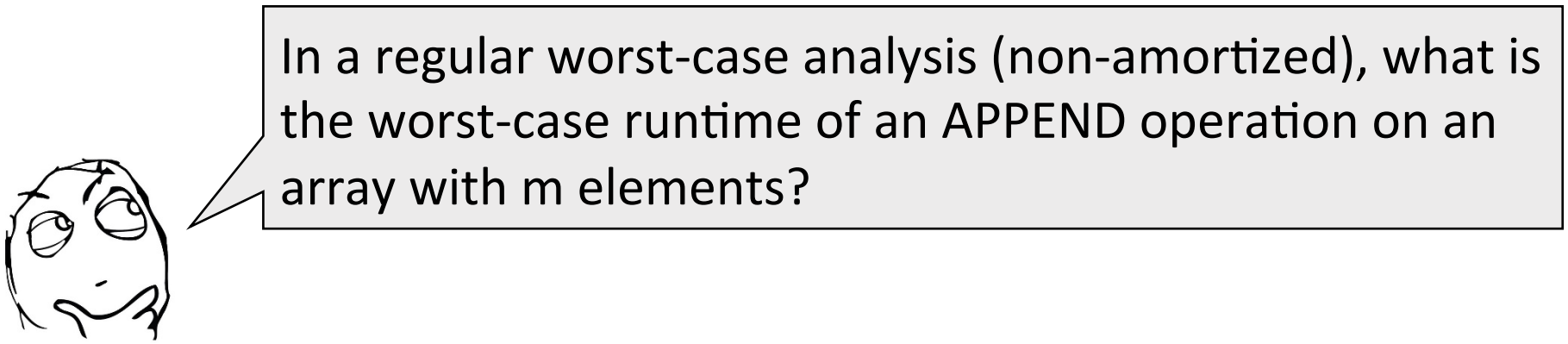
Copy all items using stored credit

Add new item ($1) plus $2 credit

So in all cases credit invariant is maintainted

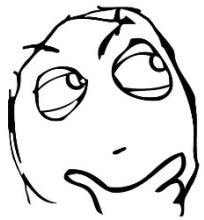
If we charge $3 for each APPEND it is enough units to pay for all costs in any sequence of APPEND operations (starting with an array of size 1)

In other words, for a sequence of ***m*** APPEND operations, the amortized cost per operations is **3**, which is in O(1).



By performing the amortized analysis, we showed that "**double the size when full**" is a good strategy for expanding a dynamic array, since it's amortized cost per operation is in O(1).

In contrast, "**increase size by 100 when full**" would not be a good strategy. **Why?**



Takeaway

Amortized analysis provides us valuable insights into what is the proper strategy of expanding dynamic arrays.

Expanding and Shrinking dynamic arrays

A bit trickier…

First thing that comes to mind…

When the array is ½ full after DELETE, create a new array of half of the size, and copy all the elements.

Consider the following sequence of operations performed on a **full** array with ***n*** element…

APPEND, DELETE, APPEND, DELETE, APPEND, …

Θ(n) amortized cost per operation since every APPEND or DELETE causes allocation of new array.

NO GOOD!

The right way of shrinking

When the array is ¼ full after DELETE, create a new array of ½ of the size, and copy all the elements.

Charge $3 per APPEND

$2 per DELETE

- 1 append/delete-dollar
- 1 copy-dollar
- 1 recharge-dollar

The array, after shrinking…

Elements who just spent their copy-dollars

Array is half-empty

Before the **next expansion**, we need to **fill** the empty half, which will spare enough money for copying the green part.

Before the **next shrinking**, we need to **empty** half of the green part, which will spare enough money for copying what's left.

Credit Invariant for Dynamic Arrays: Append and Delete

Credit Invariant:

In an array of size 2^k there are at least $2^k/4$ elements.

Elements in rightmost half have $2 stored.

Empty slots inleftmost half have $1 stored

Proof

Base case: First operation is an insert

$1 to pay for append, $2 stored

Credit Invariant for Dynamic Arrays: Append and Delete

Credit Invariant:

In an array of size 2^k there are at least $2^k/4$ elements.

Elements in rightmost half have $2 stored.

Empty slots in bottom half have $1 stored

Proof

Inductive Step: four cases

(a) append without overflow

(b) append with overflow

(c) delete without shrinking

(d) delete with shrinking

Credit Invariant for Dynamic Arrays: Append and Delete

Credit Invariant:

In an array of size 2^k there are at least $2^k/4$ elements.

Elements in rightmost half have $2 stored.

Empty slots in bottom half have $1 stored

Inductive Step: four cases

(a) append without overflow:

```
                    2                                  2  2
a1:  a  b  c  d  e  X  X  X      →      a  b  c  d  e  f  X  X

                 1  1                          1
a2:  a  b  X  X  X  X  X  X      →      a  b  c  X  X  X  X  X
```

Credit Invariant for Dynamic Arrays: Append and Delete

Credit Invariant:

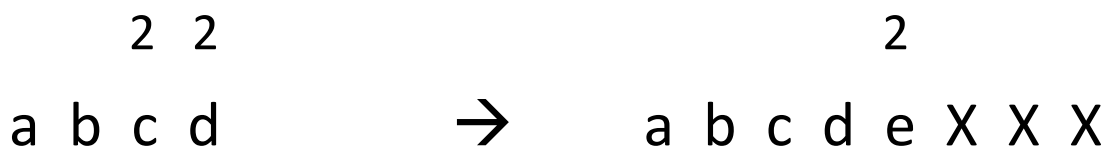
In an array of size 2^k there are at least $2^k/4$ elements.

Elements in rightmost half have $2 stored.

Empty slots in bottom half have $1 stored

Inductive Step: four cases

(b) append with overflow:



credit pays to copy old stuff

$3: store $2, and use $1 to pay for new copy

Credit Invariant for Dynamic Arrays: Append and Delete

Credit Invariant:

In an array of size 2^k there are at least $2^k/4$ elements.

Elements in rightmost half have $2 stored.

Empty slots in bottom half have $1 stored

Inductive Step: four cases

(c) delete no shrinking:

c1:

					2	2			
a	b	c	d	e	f	X	X	→	

					2			
a	b	c	d	e	X	X	X	

c2:

			1						
a	b	c	X	X	X	X	X	→	

					1	1		
a	b	X	X	X	X	X	X	

$2 charge: $1 for delete, $1 for credit

Credit Invariant for Dynamic Arrays: Append and Delete

Credit Invariant:

In an array of size 2^k there are at least $2^k/4$ elements.

Elements in rightmost half have $2 stored.

Empty slots in bottom half have $1 stored

Inductive Step: four cases

(d) delete with shrinking:

1 1 1

a b c X X X X X → a b X X X X X X → a b X X

First delete c, $1 to delete, $1 stored on new blank spot

Then shrink – dollars stored will pay for copy of all guys to left

So, overall credit invariant maintained

Summary: In a dynamic array, if we expand and shrink the array as discussed (double on full, halve on ¼ full) then:

For any sequence of APPEND or DELETE operations, $3 per APPEND and $2 per DELETE is enough money to pay for all costs in the sequence.

Therefore the amortized cost per operation of any sequence is upper-bounded by 3, i.e., O(1).

Next week

Graphs!