Amortized Analysis: Dynamic Arrays

Fatemeh Panahi
Department of Computer Science
University of Toronto
CSC263-Fall 2017
Lecture 7

Today

- Amortized Analysis
  - Aggregate Method
  - Accounting Method
- Dynamic Arrays

Reading Assignments 

Chapter 17

Running time

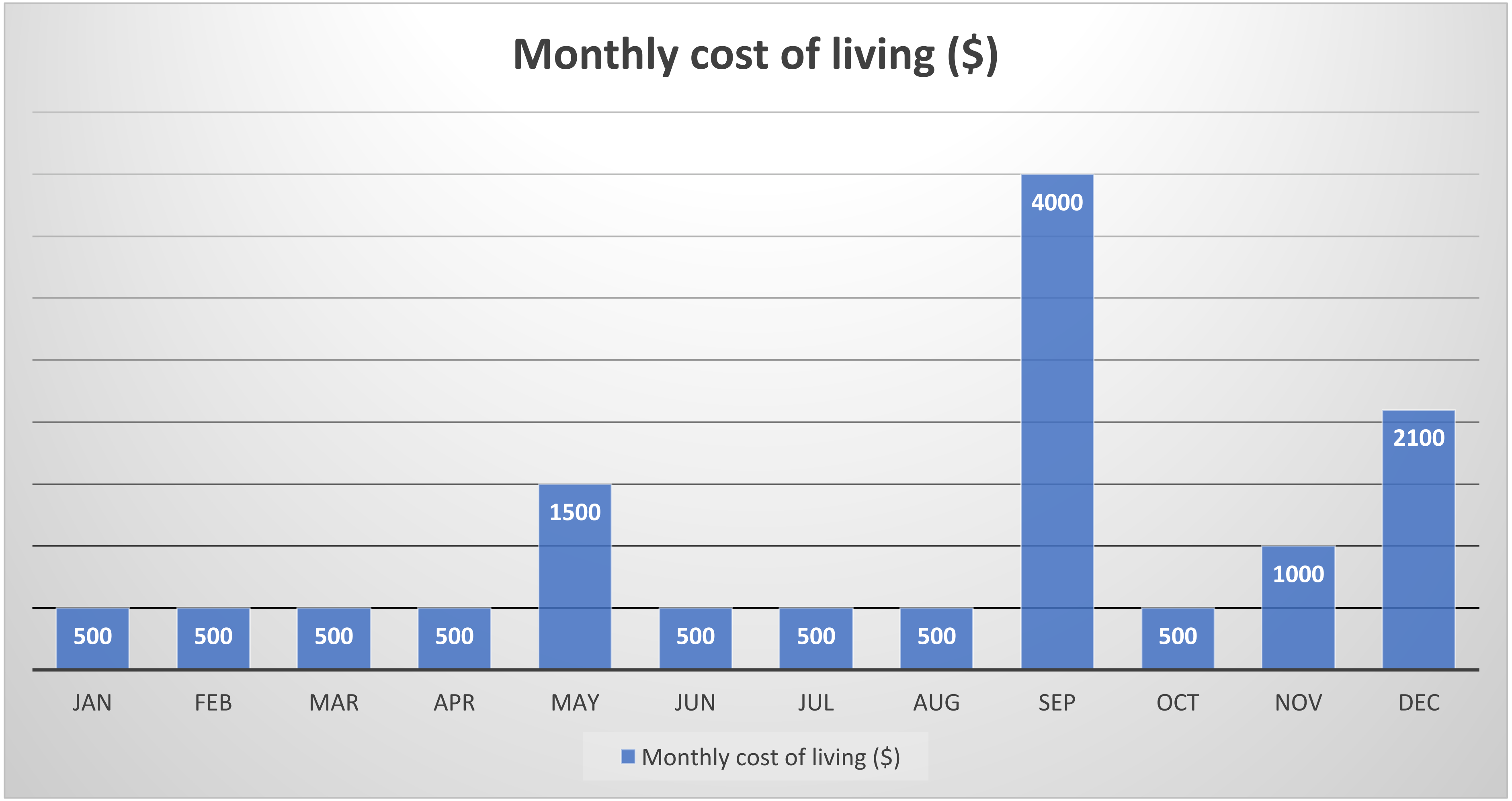
- Best case
- Average case
- Worst case
- Expected running time
- Amortized cost

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.

Real life intuition

Monthly cost of living, a sequence of 12 operations



Example

You are to maintain a collection of lists and support the following operations.

- insert(item, list): insert item into list (cost = 1).
- sum(list): sum the items in list, and replace the list with a list containing one item that is the sum (cost = length of list).

Cost of an insert operation and the amortized cost of a sum operation.

4							

Insert(4)

Example

You are to maintain a collection of lists and support the following operations.

- insert(item, list): insert item into list (cost = 1).
- sum(list): sum the items in list, and replace the list with a list containing one item that is the sum (cost = length of list).

Cost of an insert operation and the amortized cost of a sum operation.

4	7						

Insert(7)

Example

You are to maintain a collection of lists and support the following operations.

- insert(item, list): insert item into list (cost = 1).
- sum(list): sum the items in list, and replace the list with a list containing one item that is the sum (cost = length of list).

Cost of an insert operation and the amortized cost of a sum operation.

4	7	2					

Insert(2)

Example

You are to maintain a collection of lists and support the following operations.

- insert(item, list): insert item into list (cost = 1).
- sum(list): sum the items in list, and replace the list with a list containing one item that is the sum (cost = length of list).

Cost of an insert operation and the amortized cost of a sum operation.

4	7	2	6				

Insert(6)

Sum()

Example

You are to maintain a collection of lists and support the following operations.

- insert(item, list): insert item into list (cost = 1).
- sum(list): sum the items in list, and replace the list with a list containing one item that is the sum (cost = length of list).

Cost of an insert operation and the amortized cost of a sum operation.

19							

Insert(6)

Sum()

Sum()

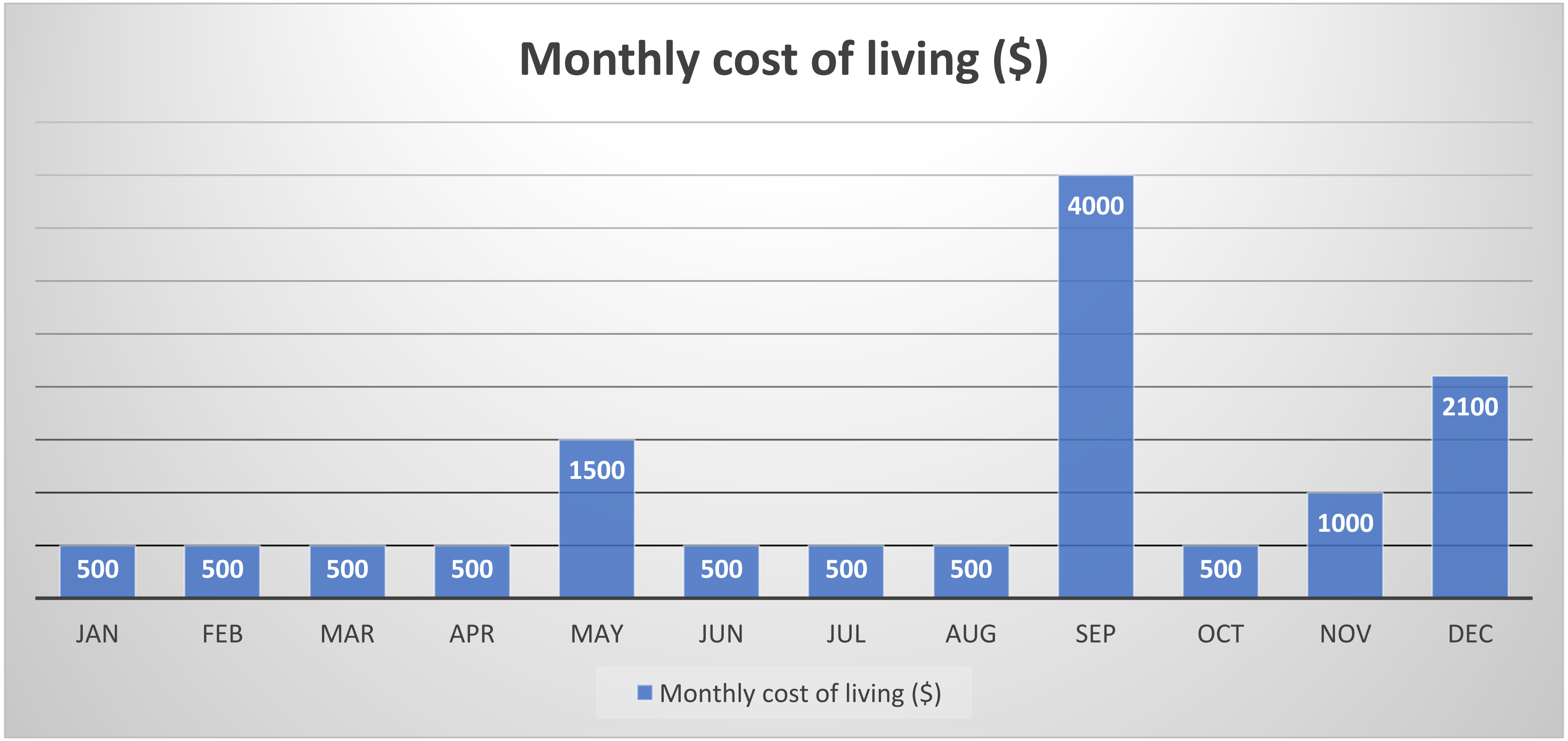
Methods for amortized analysis

- 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). Amortized cost = $\frac{\$\ 12600}{12} = 1050$



Aggregate method

For a sequence of m operations, let $\mathrm{T(m)} = \text{worst–case complexity of m operations}$

Amortized sequence complexity $= \dfrac{\mathrm{T(m)}}{\mathrm{m}}$

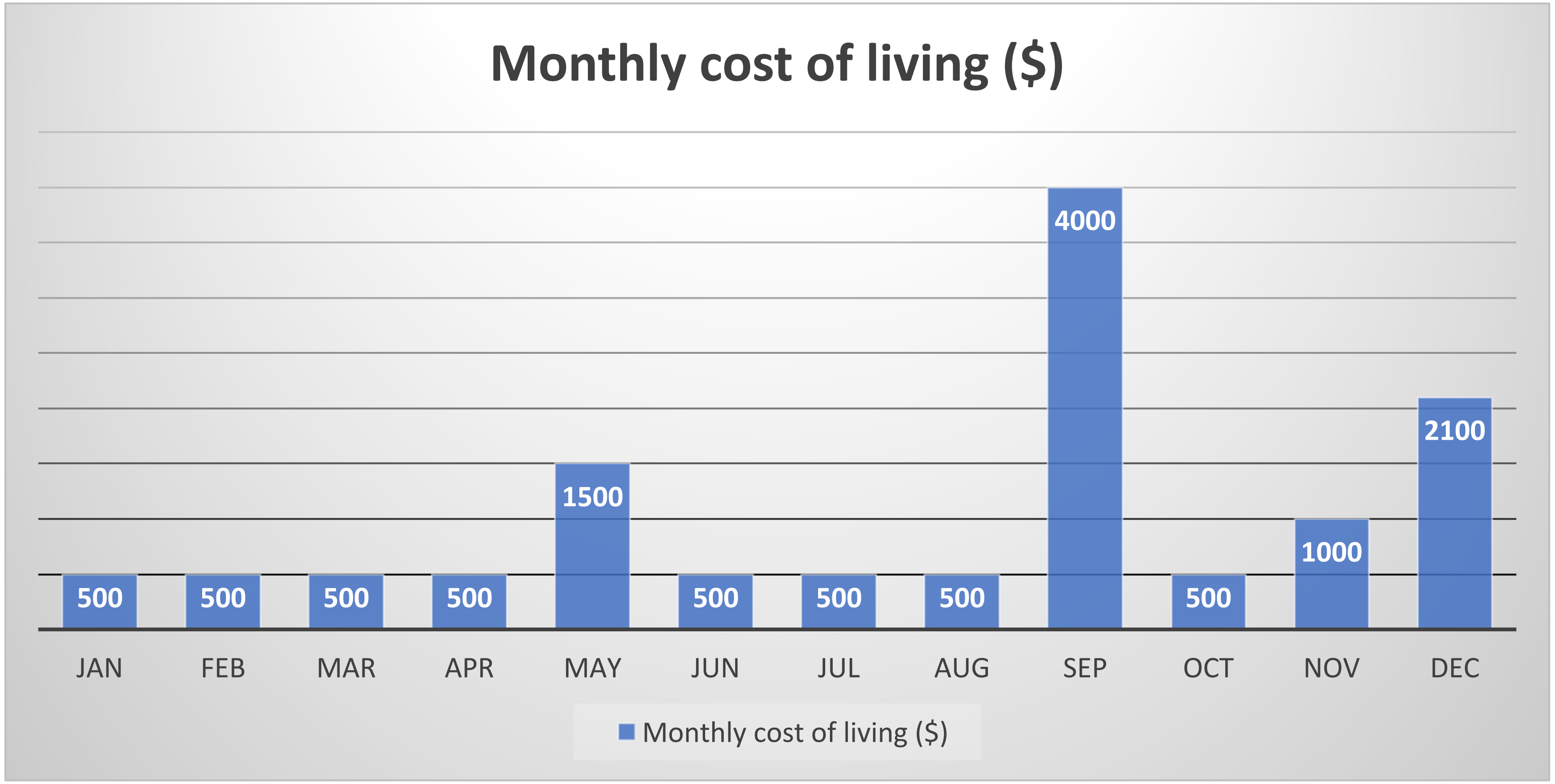
The MAXIMUM possible ***total*** cost of among all possible sequences of m operations

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

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

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.

Accounting method

- We assign differing charges to different operations.
- We call the amount we charge an operation **its amortized cost**.
- When an operation's amortized cost exceeds its actual cost, we assign the difference to specific objects in the data structure as credit.

- the actual cost of the operation i: c_i
- the amortized cost of the operation i: $\widehat{c_i}$.

$$\sum_{i=1}^{n} \widehat{c_i} \geq \sum_{i=1}^{n} c_i$$

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

Example: Incrementing a binary counter

- Implementing a k-bit binary counter that counts upward from 0.

$A = [0,1, \dots n]$

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

A	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1
2	0	0	0	0	0	0	1	0
3	0	0	0	0	0	0	1	1
4	0	0	0	0	0	1	0	0
5	0	0	0	0	0	1	0	1
6	0	0	0	0	0	1	1	0
7	0	0	0	0	0	1	1	1
8	0	0	0	0	1	0	0	0
9	0	0	0	0	1	0	0	1
10	0	0	0	0	1	0	1	0
11	0	0	0	0	1	0	1	1
12	0	0	0	0	1	1	0	0
13	0	0	0	0	1	1	0	1
14	0	0	0	0	1	1	1	0
15	0	0	0	0	1	1	1	1
16	0	0	0	1	0	0	0	0

Example: Incrementing a binary counter

- Implementing a k-bit binary counter that counts upward from 0.

$A = [0,1, \ldots n]$

INCREMENT(A)

$i = 0$
while $i < A.length$ and $A[i] == 1$
  $A[i] = 0$
  $i = i + 1$
if $i < A.length$
  $A[i] = 1$

A	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1
2	0	0	0	0	0	0	1	0
3	0	0	0	0	0	0	1	1
4	0	0	0	0	0	1	0	0
5	0	0	0	0	0	1	0	1
6	0	0	0	0	0	1	1	0
7	0	0	0	0	0	1	1	1
8	0	0	0	0	1	0	0	0
9	0	0	0	0	1	0	0	1
10	0	0	0	0	1	0	1	0
11	0	0	0	0	1	0	1	1
12	0	0	0	0	1	1	0	0
13	0	0	0	0	1	1	0	1
14	0	0	0	0	1	1	1	0
15	0	0	0	0	1	1	1	1
16	0	0	0	1	0	0	0	0

$A[0]$ flips n times
$A[1]$ flips $\lfloor n/2 \rfloor$ times
$A[1]$ flips $\lfloor n/4 \rfloor$ times
…
$A[i]$ flips $\lfloor n/2^i \rfloor$ times

The total number of flips

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

Total amortized cost
$= T(n)/n = O(1)$

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

Amortized Analysis on Dynamic Arrays

Problem description

- Think of an **array** initialized with a **fixed** number of slots, and supports **APPEND** and **DELETE** operations.

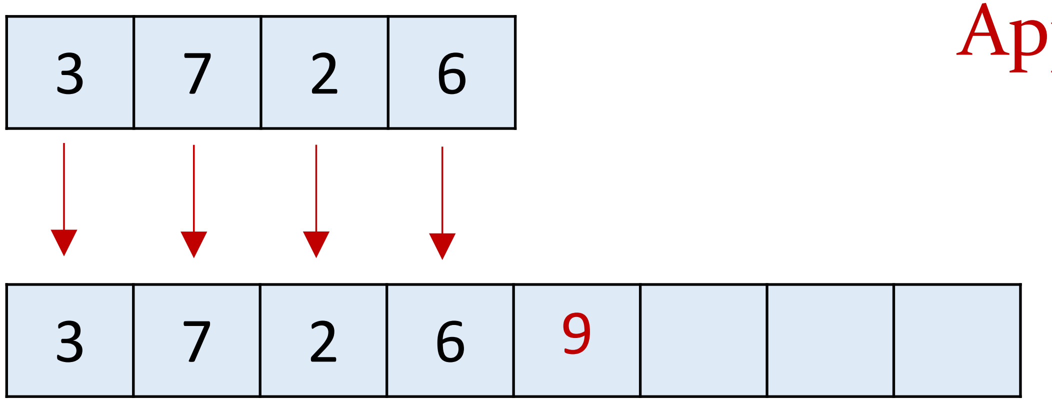
- When we APPEND too many elements, the array would be **full** and we need to **expand** the array (make the size larger).
- When we DELETE too many elements, we want to **shrink** to the array (make the size smaller).
- Requirement: the array must be using **one contiguous block** of memory all the time.

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

One way to expand

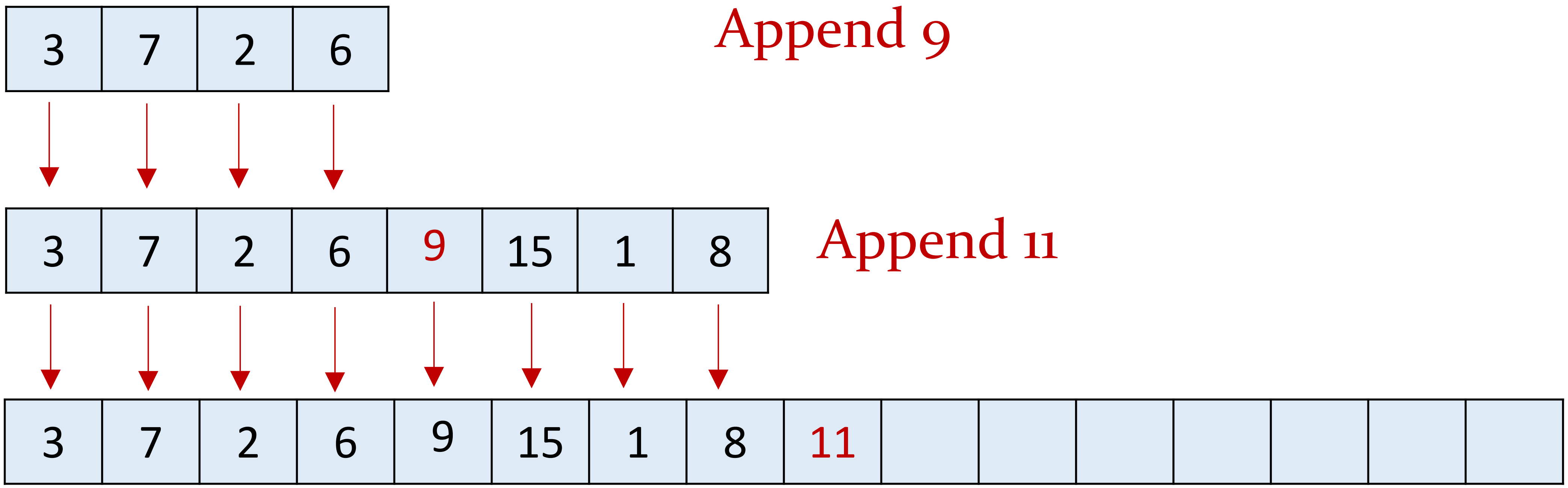
- If the array is full when APPEND is called
    - Create a new array of **twice** the size
    - Copy the all the elements from old array to new array
    - Append the element

Append 9



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

What is the cost for **copying** the elements when an element is appended?
What is the cost for **appending** the elements when an element is inserted?

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	…	31	32	33
Copy	-	1	2	-	4	-	-	-	8	-	-	-	-	-	-	-	16	-	-	-	…	-	-	32
Append	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	…	1	1	1
total	1	2	3	1	5	1	1	1	9	1	1	1	1	1	1	1	17	1	1	1	…	1	1	33

$$c_i = \begin{cases} i+1 & \textit{if } i \textit{ is power of } 2 \\ 1 & \textit{otherwise} \end{cases}$$

Use the aggregate method

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	…	31	32	33
Copy	-	1	2	-	4	-	-	-	8	-	-	-	-	-	-	-	16	-	-	-	…	-	-	32
Append	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	…	1	1	1
total	1	2	3	1	5	1	1	1	9	1	1	1	1	1	1	1	17	1	1	1	…	1	1	33

n operation of append operation.

$Let\, m = \log n \rightarrow n = 2^m$

Cost for copies: $1 + 2 + \ \dots 2^m = \sum_{i=0}^{m} 2^i = 2^{m+1} = 2n$

Cost for appends: $\sum_{i=0}^{n} 1 = n$

Total cost = $3n$

Amortized cost = $3n/n = 3$

Use the accounting method

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

*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 ?

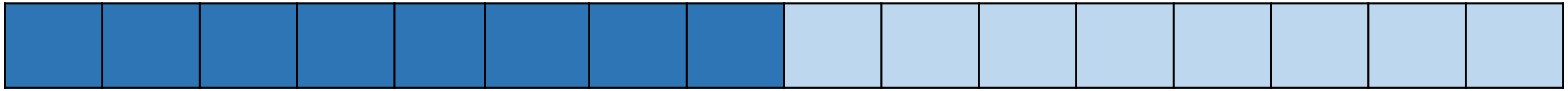
\$2 ?

\$3 ?

\$log m ?

\$m ?

Amortized cost 1$?



Earn $1 for each appended element

- This $1 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 any money to pay for it.

Amortized cost 2$?

1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Earn $2 for each appended element

- $1 (the "append-dollar") will be spent when appending the element.

Amortized cost 2$?

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Earn $2 for each appended element

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

How about the elements that have been copied more than one time?

Amortized cost 3$?

2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2

Earn $3 for each appended element

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

Amortized cost 3$?

1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Earn $3 for each appended element

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

Use the accounting method!

0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1

Earn $3 for each appended element

- The elements in the first half have been copied twice.

1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0

- The elements in the first quarter have been copied three times.

0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0

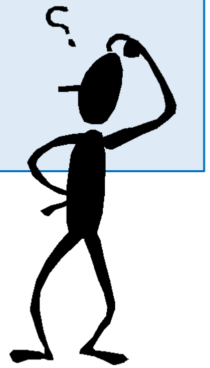
NEVER BROKE!

Use the accounting method!

If we earn \$3 upon each APPEND it is enough money to pay for all costs in the sequence of APPEND operations.

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

In a regular worst-case analysis (non-amortized), what is the worst-case runtime of an APPEND operation on an array with m elements?



Amortized Analysis on Shrinking Dynamic Arrays

First idea

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

x	x	x	x	x	x	x	x	x							

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!

Better solution

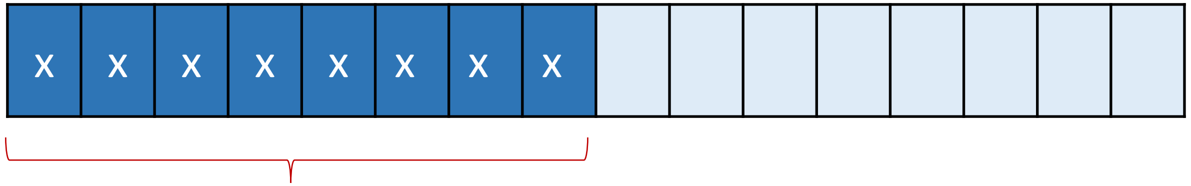
When the array is ¼ full after DELETE, create a new array of ½ of the size, and copy all the elements.

Earning \$3 per APPEND and \$3 per DELETE would be enough for paying all the cost.

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

Shrinking cost

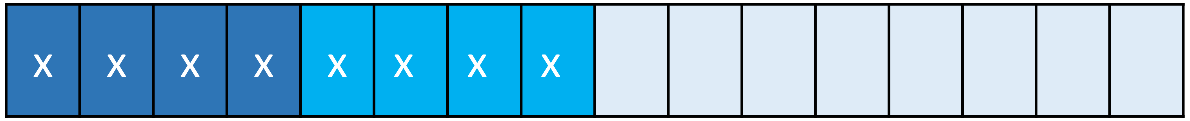
The array, after shrinking…



Array is half-empty

Elements who just spent their copy-dollars

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, which will spare enough money for copying what's left.

Summary

- In a dynamic array, if we expand and shrink the array as discussed (double on ½ full, halve on 1/4 full)…
- For any sequence of APPEND or DELETE operations, earning $3 per operation 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., $O(1)$.

Questions