COMP_SCI 214: Data Structures and Algorithms

Amortized Analysis

PROF. SRUTI BHAGAVATULA

Announcements

- Exam 2 on Thursday
- I won't be taking clarifying questions during the exam for two big reasons:
  - It's super distracting for the students around you especially if I have to squeeze between rows and then talk
  - Students who get clarifications are at an advantage to those who didn't hear what I said
- If you really need to come ask me something, I'll ask that you come down to talk to me so that us talking doesn't distract your neighbors.

Recall throughout the quarter

- Insertions into arrays (dynamic arrays)
  - Recall, we said: if we double the array every time we reach capacity, we can get $O(1)$ insertion amortized
- Hash tables with resizing to maintain constant load factor
  - Recall, we said: if we keep resizing the hash table to keep the load factor constant, we can get $O(1)$ average and amortized operations
- Both pertain to a sequence of operations

Sequences of operations

- We have a data structure on which we perform a sequence of k operations
- Normal complexity analysis tells us that the cost of the sequence is bounded by k times the worst-case complexity of the operation

Dynamic arrays

- Growing by double every time the array reaches capacity is very efficient
  - Why? We'll see today!
- First, let's see why growing by just one element is inefficient

A naïve implementation (1/2)

```
class BadDynArray[T]:
    let data: VecC[T]

    def __init__(self):
        self.data = []

    def len(self):
        return self.data.len()

    def get(self, index):
        return self.data[index]

    def set(self, index, element):
        self.data[index] = element
```

Store data in an array.
Keep no spare capacity.

Most functions just call array ops.

A naïve implementation (2/2)

```
class BadDynArray[T]:
    let data: VecC[T]

    …

    def push_back(self, element):
        let new_data = [ None; self.len() + 1 ]
        for i, v in self.data:
            new_data[i] = v
        new_data[self.len()] = element
        self.data = new_data
```

Create new vector with one extra space.
Copy elements over.

Naïve representation complexities

- `get/set/len` are $O(1)$
- `push_back` is $O(n)$!

- **Quiz:** What is the number of steps to build an n-element array, starting with an empty array and doing n `push`es?
  1. n
  2. n^2
  3. n^3

Naïve representation complexities

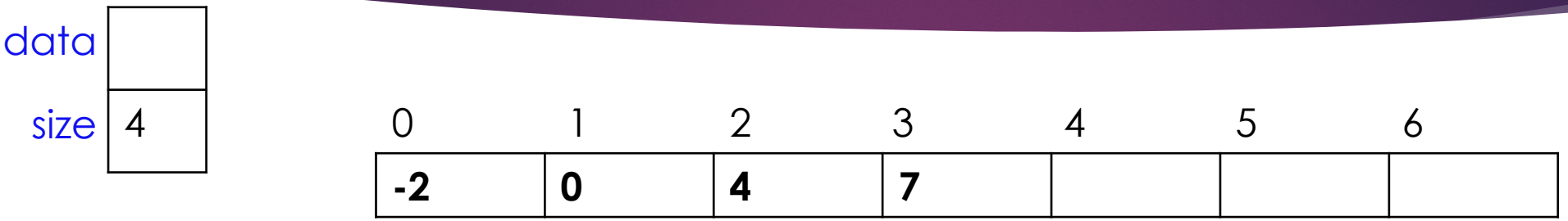
- `get/set/len` are $O(1)$
- `push_back` is $O(n)$!

- **Quiz:** What is the number of steps to build an n-element array, starting with an empty array and doing n `push`es?

1. n
2. **n^2**
3. n^3

$$\sum_n n = n^2$$

Instead…Leave extra space in the array



- # elements in dynamic array != vector length
  - Store size separately
- When we add an element, there may be spare capacity already
- When we grow, grow enough to keep some spare capacity
  - Doubling each time means we'll run out of space less and less frequently

Implementation (1/3)

```
class DynArray[T]:
  let data: VecC[OrC(T, NoneC)] # could be unused space
  let size: nat?

  def __init__(self, initial_capacity: nat?):
    self.data = [None; initial_capacity]
    self.size = 0

  def len(self):
    return self.size

  def capacity(self) -> nat?:
    return self.data.len()

  …
```

Not much to see here.

Implementation (2/3)

```
class DynArray[T]:
  …

  def get(self, index):
    self._bounds_check(index)
    return self.data[index]

  def set(self, index, element):
    self._bounds_check(index)
    self.data[index] = element

  def _bounds_check(self, index):
    if index >= self.size:
      error('DynArray: out of bounds')
```

Need to check that we're not accessing unused spots!

Implementation (3/3)

```
class DynArray[T]:
  …

  def push_back(self, element):
    self._ensure_capacity(self.size + 1)
    self.data[self.size] = element
    self.size = self.size + 1

  def _ensure_capacity(self, cap):
    if self.capacity() < cap:
      cap = max(cap, 2 * self.capacity())
      let new_data = [ None; cap ]
      for i, v in self.data:
        new_data[i] = v
      self.data = new_data
```

Double when we need to grow.

Time complexities

- `get/set/len` are $O(1)$
- `push_back` is still $O(n)$ in the worst case!

- **Quiz:** What is the number of steps to build an n-element array, starting with an empty array and doing n `push`es?
  1. n
  2. n^2
  3. n^3
  4. It's complicated

Time complexities

- `get/set/len` are $O(1)$
- `push_back` is still $O(n)$ in the worst case!

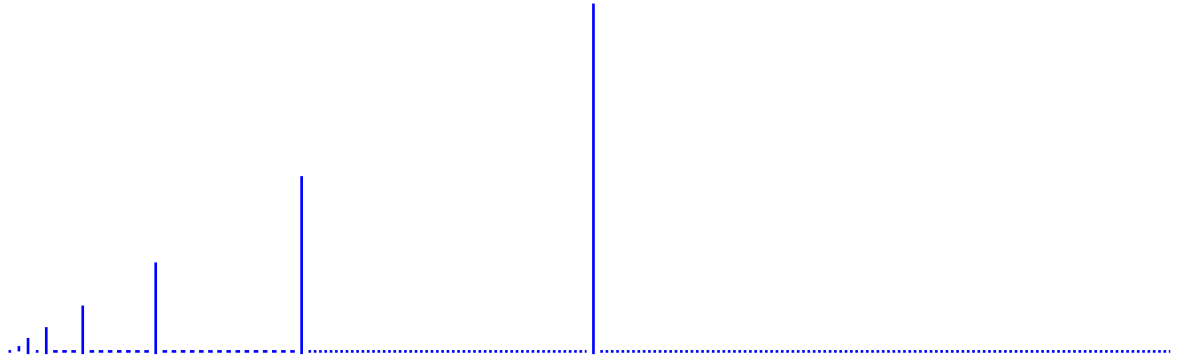
- **Quiz:** What is the number of steps to build an n-element array, starting with an empty array and doing n `push`es?
  1. n
  2. n^2
  3. n^3
  4. **It's complicated**

It would be n^2 if each `push` always the worst case.

But will it always be?

The peculiar thing about `push_back`

- Most of the time it's cheap
- But occasionally, we need to grow (which is expensive)



- **X axis**: push #
- **Y axis**: time it takes to do push #x

"Modes" of `push_back`

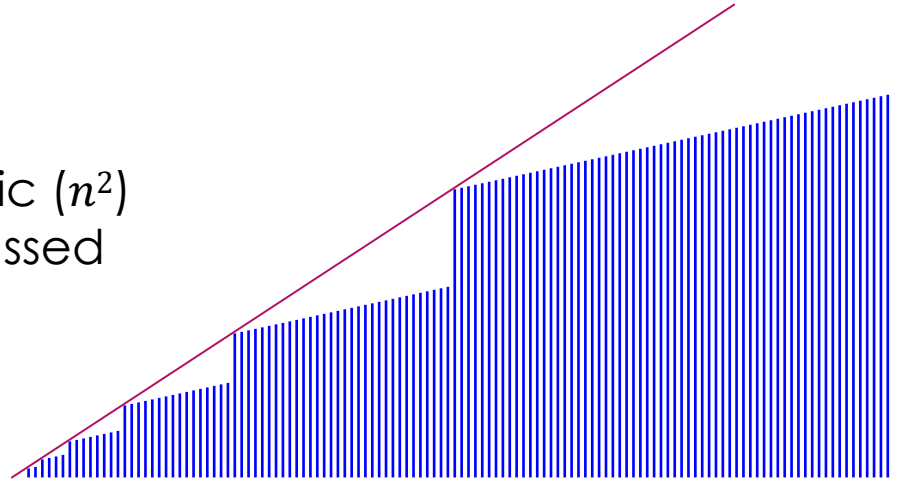
- `push_back` has two "modes"
  1. There is spare capacity: set array element, $O(1)$
  2. There is no spare capacity: double the size, $O(n)$
  - As we go from 0 to n elements, mix of both modes

- When we double, get space for **many** cheap pushes!

- Now let's consider the work of doing pushes as a sequence of operations

Cumulative time

It's linear!
Not quadratic (n^2)
like we discussed
before



- **X axis**: push #
- **Y axis**: cumulative time it takes to do push #x

Sequences of operations

- We have a data structure on which we perform a sequence of k operations
- Normal complexity analysis tells us that the cost of the sequence is bounded by k times the worst-case complexity of the operations
- The overall actual cost of the sequence may be much less
  - $actual\_cost = \sum_{i=0}^{k} cost\_of\_operation\_i$
- How is this actual cost across the sequence a linear function?
  - Let's see how this works out in math

Dynamic array aggregate analysis

- Suppose we create a new array and push n times
- How can we show linear total time?
- Let s_i be the size of the array at step i
- Let c_i be the cost of the ith push

$$c_i = \begin{cases} 2(i-1)+1 & \text{if } i-1 \text{ is a power of 2 (we're full)} \\ 1 & \text{otherwise (there's space)} \end{cases}$$

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	2	4	4	8	8	8	8	16	16	16
c_i	1+1	2+1	4+1	1	8+1	1	1	1	16+1	1	1

Adding it up

- Let d_i be the cost of doubling (when we double)
- $d_i = c_i$ -1 (since the cost includes the actual push of the new element)
- Then,

$$\sum_{i=1}^{n} c_i = \sum_{i=1}^{n} (1 + d_i)$$

$$= n + \sum_{i=1}^{n} d_i$$

d_i is almost always 0 since we rarely double

$$= n + \sum_{i=0}^{\log_2 n \text{ times}} 2^i$$

The number of times we double is $\log_2 n$

$$= n + \left(1 + 2 + 4 + \ \ldots + \frac{n}{4} + \frac{n}{2} + n\right) < 3n$$

Cost of whole sequence is linear

Awesome, but…

- We only showed the whole sequence is $O(n)$ instead of $O(n^2)$
- How did we say $O(1)$ amortized cost before?

Sequences of operations

- We have a data structure on which we perform a sequence of k operations
- Normal complexity analysis tells us that the cost of the sequence is bounded by k times the worst-case complexity of the operations
- The overall actual cost of the sequence may be much less
  - $actual\_cost = \sum_{i=0}^{k} cost\_of\_operation\_i$
- Amortized cost is the overall actual cost divided by the length of the sequence
  - Amortized cost = $actual\_cost \ / \ k$
  - Average of the actual cost of each operation over the sequence

Amortized cost

The overall actual cost divided by the length of the sequence

- As if every operation in the sequence cost the same amount
- Just looking at the worst-case complexity is too pessimistic
  - Particularly when the worst-case occurs infrequently
  - Only tells us about 1 operation in isolation

An analogy to saving up money

- **Starting philosophy:** You use money wisely if you never take any vacations or spend large sums
  - The best case of using money wisely is not taking vacations
  - The worst case is taking vacations constantly (every week or month)
- This implies that if you ever take a vacation, you are not spending money wisely; seems flawed and impossible
- **Modified philosophy:** If you save up money all year to take one giant vacation at the end of the year, you use money wisely
  - Save up for expensive operations

But wait, we've seen "average" before!

- Recall Quicksort
  - Average-case complexity: $O(n \log n)$
  - What were we averaging over?
    - The possible inputs to the algorithm for a single run of the algorithm
    - Average over a probability distribution over inputs

- **Average-case complexity** has to do with chance
  - There is a very low chance that we'll get a bad pivot, but it might happen
- **Amortized cost** is actual cost averaged over a series of operations

When to use amortized analysis?

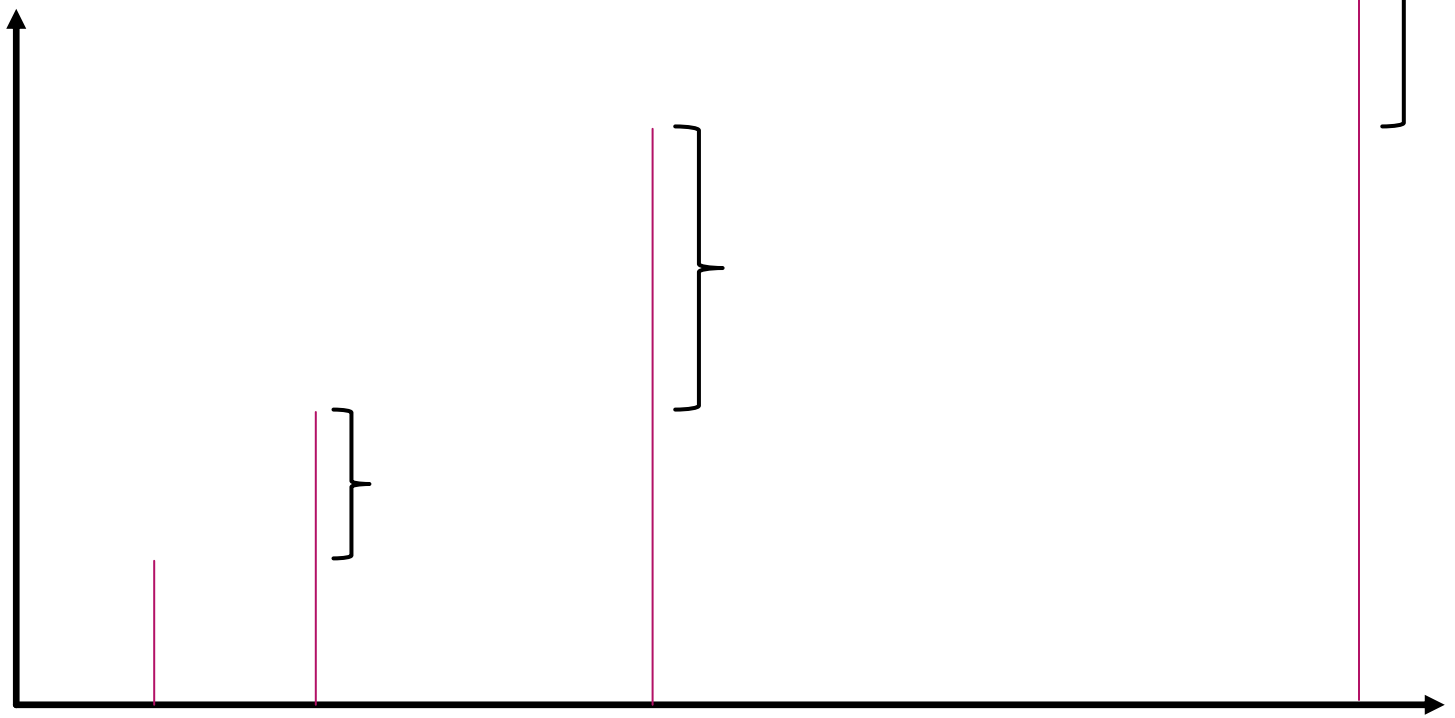
- You have a sequence of k operations on a data structure
- We expect the actual cost of the whole sequence to be much less than k times the worst-case complexity of the operations
  - A few operations are expensive
  - Many are cheap

Do we *have* to double?

- Any multiplicative growth works!
  - Tripling each time
  - Growing by 1.5x (common in practice)
  - Etc.
  - All amortized $O(1)$; just different constant factors
- Adding a constant amount of space doesn't work
  - Not even adding 1 million each time we grow
  - Why is this?

What multiplicative growth looks like

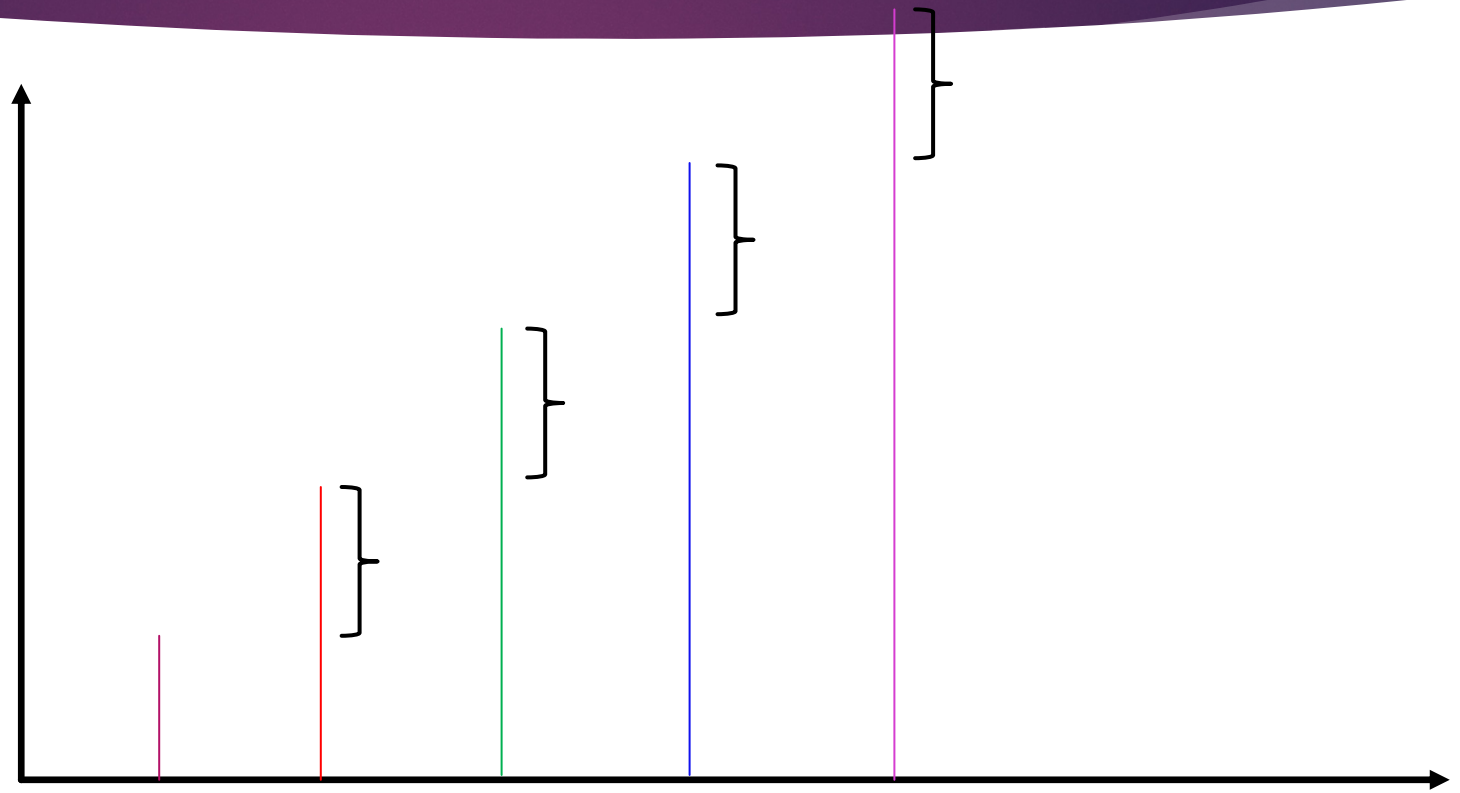
If you collapse each expensive operation to a straight line to its right, every point on the x-axis gets an average constant cost



With multiplying sizes, expensive operations become less and less frequent
Resizing every 100, 200, 400, 800, 1600, 3200, 6400, 128,000, 256,00, etc. –th insert

What constant additive growth looks like

If we collapse each line to its right, cost at each point on the x-axis may accumulate multiple costs. Cannot get constant cost per operation across all operations.



Hash table revisited

- Average-case complexity of $O(1)$ depended on a good hash function and fixed #entries of the hash table
  - Load factor (#entries/capacity) stays constant or low
  - Complexity of operations = $O(load\ factor)$
- But hash tables in the wild will have more and more inserts
  - Load factor becomes unwieldy and $O(1)$ disappears

Hash tables revisited

- How do we get that $O(1)$ always?
  - Resize the hash table each time the load factor nears a threshold (say, 1)
  - Re-hash all elements to the new hash table
- Load factor will always stay roughly constant with the resizing happening infrequently
- Same amortized cost guarantees as for dynamic array