# DSC 190

## Data Structures & Algorithms

Lecture 2 | Part 1

**Recap**

# Arrays vs. Linked Lists

- Last time, we reviewed two ways of storing sequential data: **arrays** and **linked lists**.

- **Arrays** support constant time access, but are slow to grow.
  arr[42]

- **Linked lists** are fast to grow but slow to access.

# Motivation

- Can we have the best of both worlds?
- $\Theta(1)$ time access like an array.
- $\Theta(1)$ time append like a linked list.
- **Yes!** (sort of)

# DSC 190

## Data Structures & Algorithms

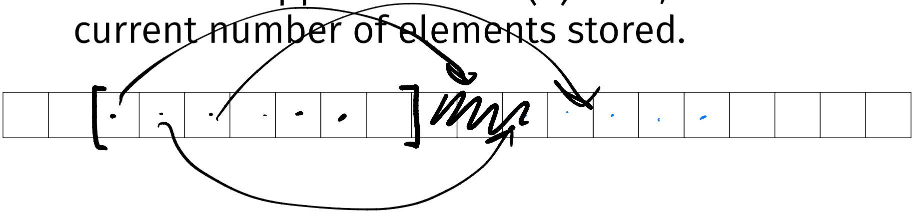
Lecture 2 | Part 2

## Dynamic Arrays

# Why are arrays slow to grow?

- Appending to an array requires:[1]
  1. allocating a new chunk of memory; and
  2. copying the entire array to the new chunk.

- Thus each append takes $\Theta(k)$ time, where $k$ is current number of elements stored.



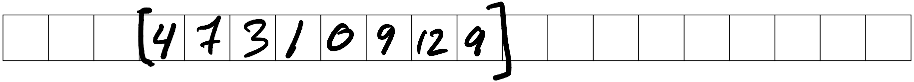
[1]There are some subtleties here. See: `https://youtu.be/5J6UlEdvDSk`

4, 7, 3

# The Idea

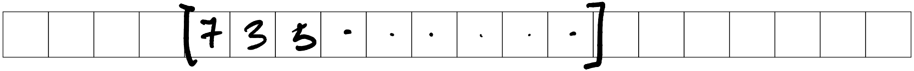
- Allocate a larger **underlying array** than initially needed.
  - Some “empty space” at end of array to “grow into”.

- Only need to allocate more memory when we run out of empty space.



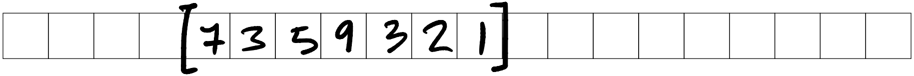
# Physical Size vs. Logical Size

- Our array will have two “sizes”.
- **Physical size**: the size of the underlying array. 9
  - I.e., the number of “slots” that have been allocated.
- **Logical size**: the number of elements currently being stored. 3
  - I.e., the number of “slots” being used.



# Appending

- If there is **empty space** (logical < physical), just insert the element into first empty slot in Θ(1) time (**fast**).
- If there is **no empty space** (logical = physical), grow the underlying array in Θ(*k*) time, then insert the element (**slow**).



# Intuition

- Most appends are fast: $\Theta(1)$ time.
- Some appends are slow: $\Theta(k)$ time.
- If slow appends are **rare enough**, the “typical” time of an append will be close to $\Theta(1)$.

# Dynamic Arrays

- This data structure is called a **dynamic array**.
- Fast access (it's just an array), and fast appends (most of the time).
- The big remaining question: how much do we grow the array when we run out of space?
- The right strategy makes all the difference.

# “Typical” Time

- Our goal is to design a strategy to minimize the “typical” time of an append.
- What do we mean by “typical”, exactly?
- Up next, a new way of measuring “typical” time: **amortized time complexity**.

# DSC 190

## Data Structures & Algorithms

Lecture 2 | Part 3

# Amortized Analysis

# Goal

- Measure the “typical” time taken by an operation:
  - most of the time, the operation is fast;
  - but sometimes, the operation is slow.

- Idea: “spread” the cost of the slow operations over the many fast operations.

# Amortization

- **Amortization** means spreading out the cost of something over time.

- E.g., buying a car:
  - **Up-front cost**: \$30,000
  - **Amortized cost over 60 months**: \$500/month

# Example: UCSD Parking

- Parking costs \$7 per day (for faculty).
- Every 100 days, you forget to pay and get a \$80 ticket.
- The “amortized cost” of parking is:

$$\frac{\text{total cost}}{\text{total days}} = \frac{\$700 + \$80}{100} = \$7.80$$

# Amortized Analysis

- **Amortized analysis** is a way of measuring the “typical” time of an operation in a sequence.
- **Idea:** spread the cost of the slow operations over the many fast operations.
- **Approach:** compute total time of operations, divide by number of operations.[2]

[2] Related to average case analysis, but not quite the same.

# Computing Amortized Time

- The **amortized time** of *n* operations is:

$$T_{\text{amort}}(n) = \frac{\text{total time taken by all operations}}{n}$$

- An equivalent strategy is to separately analyze the “fast” and “slow” operations (ops):

$$T_{\text{amort}}(n) = \frac{(\text{total time of fast ops}) + (\text{total time of slow ops})}{n}$$

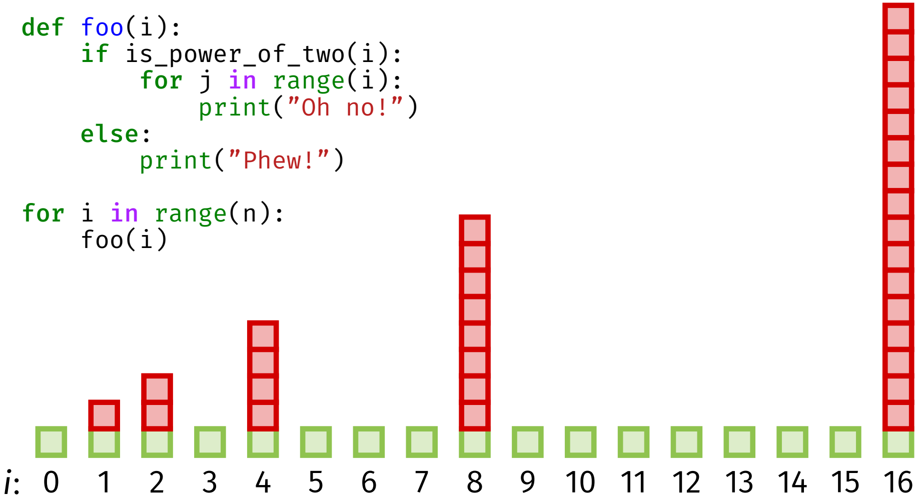
# Example: foo

```
def foo(i):
    if is_power_of_two(i):
        for j in range(i):
            print("Oh no!")
    else:
        print("Phew!")

for i in range(n):
    foo(i)
```

$\Theta(i)$

$\Theta(1)$

```python
def foo(i):
    if is_power_of_two(i):
        for j in range(i):
            print(”Oh no!”)
    else:
        print(”Phew!”)

for i in range(n):
    foo(i)
```



# Example: `foo`

```
def foo(i):
    if is_power_of_two(i):
        for j in range(i):
            print("Oh no!")
    else:
        print("Phew!")

for i in range(n):
    foo(i)
```

- Start by computing total time taken by "slow" calls.

| slow call # | # iters. |
|---|---|
| 1 | 1 |
| 2 | 2 |
| 3 | 4 |
| ⋮ | ⋮ |
| $k$ | $2^{k-1}$ |

# Example: `foo`

```
def foo(i):
    if is_power_of_two(i):
        for j in range(i):
            print("Oh no!")
    else:
        print("Phew!")

for i in range(n):
    foo(i)
```

- Start by computing total time taken by "slow" calls.

| slow call # | # iters. |
|---|---|
| 1 | 1 |
| 2 | 2 |
| 3 | 4 |
| $\vdots$ | $\vdots$ |
| $k$ | $2^{k-1}$ |

```python
def foo(i):
    if is_power_of_two(i):
        for j in range(i):
            print("Oh no!")
    else:
        print("Phew!")

for i in range(n):
    foo(i)
```

## Exercise

Out of the $n$ calls to `foo`, (roughly) how many are **slow**?

$\log_2 n$

# Example: `foo`

| slow call # | # iters. |
|---|---|
| 1 | 1 |
| 2 | 2 |
| 3 | 4 |
| ⋮ | ⋮ |
| $k$ | $2^{k-1}$ |
| $\log_2 n$ | $2^{(\log_2 n)-1}$ |

- The total time taken over all **slow** calls is:

  $$1 + 2 + 4 + \ldots + 2^{k-1} + \ldots + 2^{\log_2(n)-1}$$

- This is a geometric sum.

# Recall: Geometric Sum

- A **geometric sum** is a sum of the form:

$$1 + r + r^2 + \ldots + r^{k-1} + \ldots + r^n = \sum_{k=0}^{n} r^k$$

$2^5 - 1 = 31$

$1 + 2 + 4 + 8 + 16 = 31$

- There is a formula for this sum:

$$\sum_{k=0}^{n} r^k = \frac{1 - r^{n+1}}{1 - r}$$

$r = 2 \quad n = 4$

$\frac{1 - 2^5}{1 - 2} \quad 2^5 = 32$

# Example: `foo`

- Recall our geometric sum for the total time taken by the **slow** calls:

$$1 + 2 + 4 + \ldots + 2^{k-1} + \ldots + 2^{\log_2(n)-1} = \sum_{k=0}^{\log_2(n)-1} 2^k$$

- Using the formula on the previous slide with $r = 2$ and $n = \log_2(n) - 1$, we get:

$$\sum_{k=0}^{\log_2(n)-1} 2^k = \frac{1 - 2^{\log_2(n)}}{1 - 2} = 2^{\log_2(n)} - 1 = n - 1 = \Theta(n)$$

# Example: `foo`

```
def foo(i):
    if is_power_of_two(i):
        for j in range(i):
            print("Oh no!")
    else:
        print("Phew!")

for i in range(n):
    foo(i)
```

- The total time taken by the **slow** calls is $\Theta(n)$.

```
def foo(i):
    if is_power_of_two(i):
        for j in range(i):
            print("Oh no!")
    else:
        print("Phew!")

for i in range(n):
    foo(i)
```

**Exercise**

What is the total time taken by all of the **fast** calls to `foo`? $\Theta(n)$

# Example: `foo`

```
def foo(i):
    if is_power_of_two(i):
        for j in range(i):
            print("Oh no!")
    else:
        print("Phew!")

for i in range(n):
    foo(i)
```

- Out of the $n$ calls to `foo`, $\Theta(\log_2 n)$ calls are "slow".
- So $\Theta(n - \log n) = \Theta(n)$ calls are "fast".
- Each fast call takes $\Theta(1)$ time.
- Total time taken by fast calls: $\Theta(n) \times \Theta(1) = \Theta(n)$.

# Example: `foo`

- Amortized time:

$$T_{\text{amort}}(n) = \frac{(\text{total time of fast calls}) + (\text{total time of slow calls})}{n}$$
$$= \frac{\Theta(n) + \Theta(n)}{n}$$
$$= \Theta(1)$$

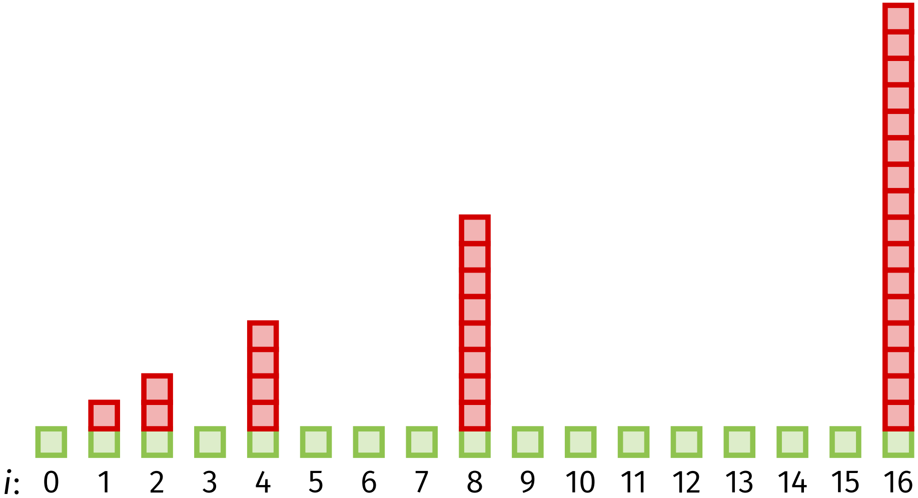
- The amortized time of `foo` is $\Theta(1)$ per call.

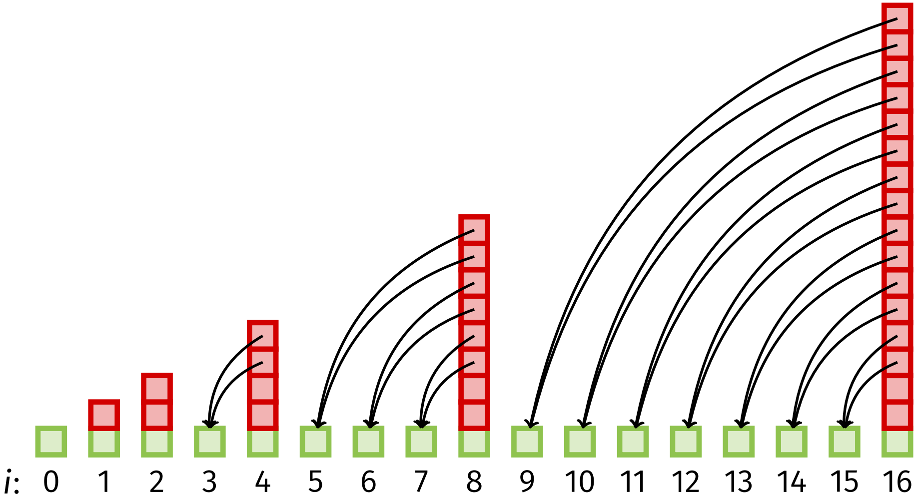
# In other words...

- Some calls to `foo` are fast, taking $\Theta(1)$.
- Some calls to `foo` are slow, taking $\Theta(n)$.
- But the slow calls are rare enough that the amortized (“typical”) cost per call is $\Theta(1)$.

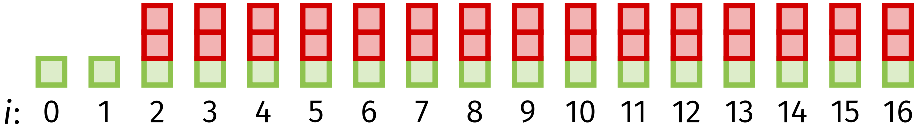
# Another View

- The cost of the **slow** iterations can be “spread over” the previous **fast** calls.
- This works because the **slow** calls are rare enough.

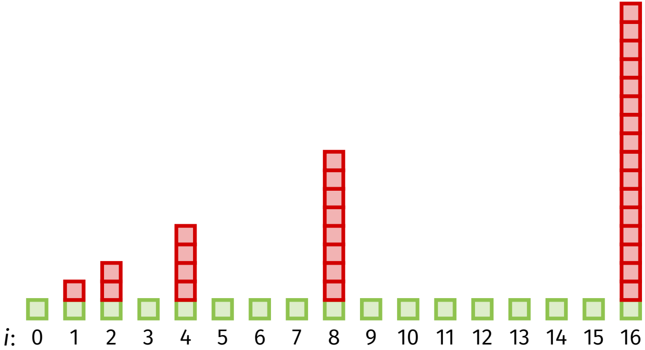

i: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16



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


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

# Observation

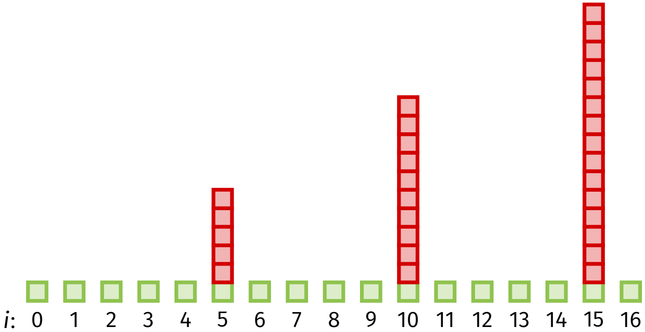



- Observation: the **slow** calls are get slower, but they also get **rarer**.
  - Twice as bad, but half as frequent.
  - Their increased cost is spread over a larger gap.

# On the other hand...

```
def bar(i):
    if is_divisible_by_five(i):
        for j in range(i):
            print(”Oh no!”)
    else:
        print(”Phew!”)

for i in range(n):
    foo(i)
```

# Observation




- Observation: the **slow** calls are get slower, but are **not getting rarer**.
  - Will lead to $\Theta(n)$ amortized cost, instead of $\Theta(1)$.

# DSC 190

## DATA STRUCTURES & ALGORITHMS

Lecture 2 | Part 4

# Growth Strategies for Dynamic Arrays

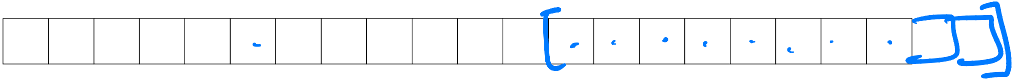
# Amortized Analysis of Dynamic Arrays

- What is the amortized cost of `append` on a dynamic array?
- It depends on the **growth strategy**.

# Attempt #1: Linear Growth

- Initially allocate $S$ slots.
- When we run out, grow physical size to $2S$ slots.
- When we run out again, physical size to $3S$.
- etc.

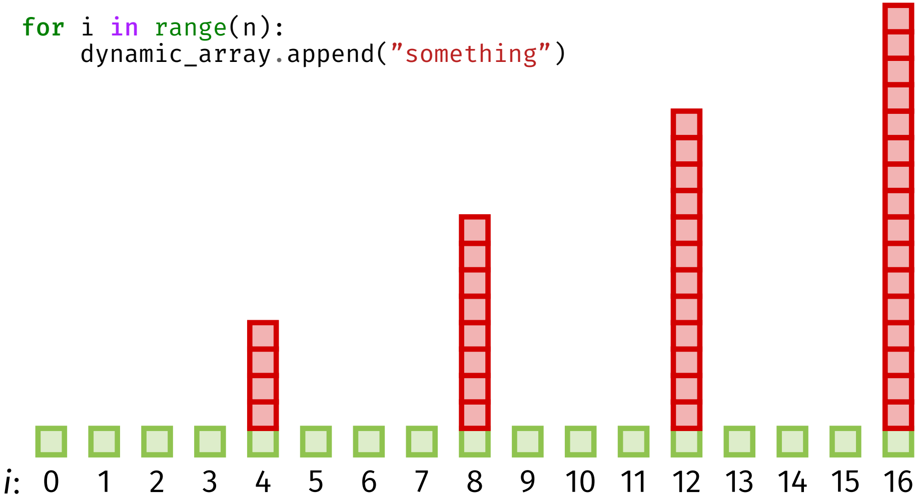
# Example



# Analysis

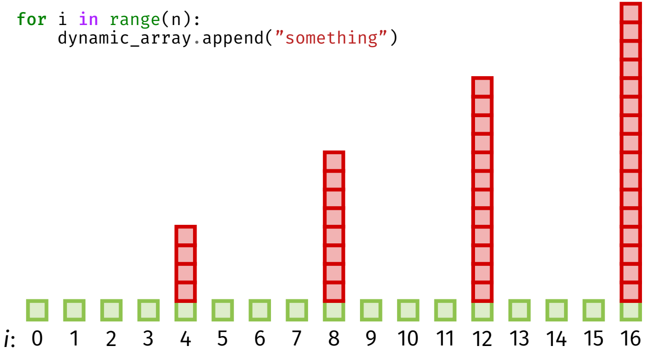
- Every $S$th append is **slow**, taking time $\Theta(k)$, where $k$ is the number of elements stored.
- All other appends are **fast**, taking time $\Theta(1)$.

```
for i in range(n):
    dynamic_array.append("something")
```



# Attempt #1: Linear Growth

```
for i in range(n):
    dynamic_array.append("something")
```




- The **slow calls** are getting worse (linearly), but are **not** getting rarer.
  - This will lead to a **linear** amortized cost.

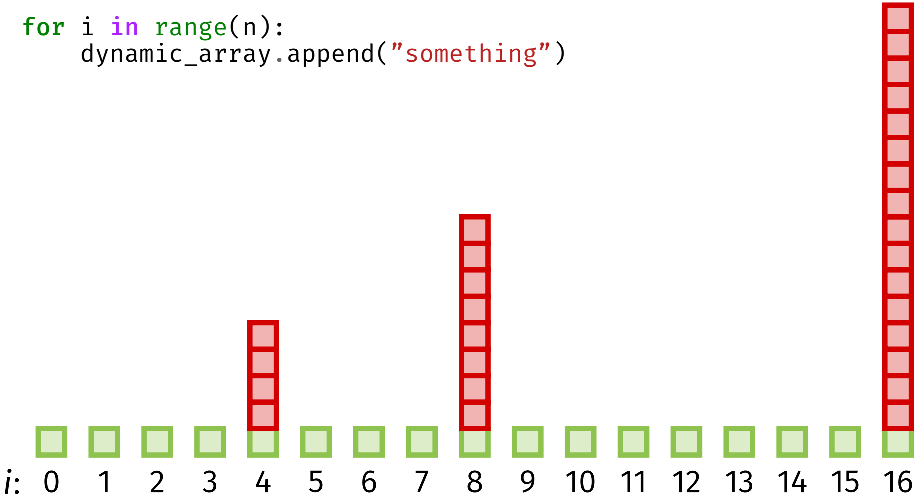
# Attempt #2: Geometric Growth

- Initially allocate $S$ slots.
- When we run out, *double* the physical size.
- When we run out again, double it again.
- etc.

# Example

| | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | | | | | | | | | | | |

```
for i in range(n):
    dynamic_array.append("something")
```



# Informal Analysis

- The **slow calls** are getting slower (geometrically), but are **getting rarer**!
- This will lead to an amortized cost of $\Theta(1)$.

# In general...

- We have used a **growth factor** of $\gamma = 2$.
- In general, we can use any $\gamma > 1$.
- Next up: a formal analysis of the amortized cost for a general $\gamma$.

# DSC 190

## DATA STRUCTURES & ALGORITHMS

Lecture 2 | Part 5

## Formal Analysis of Dynamic Arrays

# Amortized Time Complexity

- The **amortized** time for an append is:

$$T_{\text{amort}}(n) = \frac{\text{total time for } n \text{ appends}}{n}$$

- We'll see that $T_{\text{amort}}(n) = \Theta(1)$ when geometric resizing is used with any growth factor $\gamma > 1$.

# Amortized Analysis

total time for *n* appends

=

total time for **non-growing** appends

+

total time for **growing** appends

fast

slow

# Counting Growing Appends

- Want to calculate time taken by growing appends.
- First: how many appends caused a resize?
  - $\beta$: initial physical size
  - $\gamma$: growth factor

# Counting Growing Appends

- Suppose initial physical size is $\beta = 512$, and $\gamma = 2$
- Resizes occur on append #:

$$512, 1024, 2048, 4096, \ldots$$

- In general, resizes occur on append #:

$$\beta\gamma^0, \beta\gamma^1, \beta\gamma^2, \beta\gamma^3, \ldots$$

# Counting Growing Appends

- In a sequence of $n$ appends, how many caused the physical size to grow?
- Simplification: Assume $n$ is such that $n$th append caused a resize. Then, for some $x \in \{0, 1, 2, \ldots\}$:

$$n = \beta\gamma^x$$

- If $x = 0$ there was 1 resize; if $x = 1$ there were 2; etc.

# Counting Growing Appends

- Solving for $x$:

$$x = \log_\gamma \frac{n}{\beta}$$

- Check: without assumption, $x = \lfloor \log_\gamma \frac{n}{\beta} \rfloor$
- Number of resizes is $\lfloor \log_\gamma \frac{n}{\beta} \rfloor + 1$

# Counting Growing Appends

- Number of resizes is $\lfloor \log_\gamma \frac{n}{\beta} \rfloor + 1$
- Check with $\gamma = 2$, $\beta = 512$, $n = 400$
  - Correct # of resizes: 0
- Check with $\gamma = 2$, $\beta = 512$, $n = 1100$
  - Correct # of resizes: 2

# Time of Growing Appends

- How much time was taken across all appends that caused resizes?
- Assumption: resizing an array with physical size $k$ takes time $ck = \Theta(k)$.
  - $c$ is a constant that depends on $\gamma$.

# Time of Growing Appends

- Time for first resize: $c\beta$.
- Time for second resize: $c\gamma\beta$.
- Time for third resize: $c\gamma^2\beta$.
- Time for $j$th resize: $c\gamma^{j-1}\beta$.
- This is a **geometric progression**.

# Time of Growing Appends

- Time for $j$th resize: $c\gamma^{j-1}\beta$.
- Suppose there are $r$ resizes.
- Total time:

$$c\beta \sum_{j=1}^{r} \gamma^{j-1} = c\beta \sum_{j=0}^{r} \gamma^{j}$$

# Recall: Geometric Sum

- From before:

$$\sum_{k=0}^{n} r^k = \frac{1 - r^{n+1}}{1 - r}$$

# Time of Growing Appends

- Total time:

$$c\beta \sum_{j=0}^{r} \gamma^j = c\beta \frac{1 - \gamma^{r+1}}{1 - \gamma}$$

# Time of Growing Appends

- Remember: in $n$ appends there are $r = \lfloor \log_\gamma \frac{n}{\beta} \rfloor + 1$ resizes.

- Total time:

$$c\beta \frac{1 - \gamma^{r+1}}{1 - \gamma} = c\beta \frac{1 - \gamma^{\lfloor \log_\gamma \frac{n}{\beta} \rfloor + 2}}{1 - \gamma}$$

$$= \Theta(n)$$

# Amortized Analysis

total time for $n$ appends

=

total time for **non-growing** appends

+

$\Theta(n)$ ← total time for **growing** appends

# Time of Non-Growing Appends

- In a sequence of $n$ appends, how many are **non-growing**?

$$n - \left( \lfloor \log_\gamma \frac{n}{\beta} \rfloor + 1 \right) = \Theta(n)$$

- Time for one such append: $\Theta(1)$.
- Total time: $\Theta(n) \times \Theta(1) = \Theta(n)$.

# Amortized Analysis

total time for $n$ appends

=

$\Theta(n)$ ← total time for **non-growing** appends

+

$\Theta(n)$ ← total time for **growing** appends

# Amortized Time Complexity

- The **amortized** time for an append is:

$$T_{\text{amort}}(n) = \frac{\text{total time for } n \text{ appends}}{n}$$

$$= \frac{\Theta(n)}{n}$$

$$= \Theta(1)$$

# Dynamic Array Time Complexities

- Retrieve $k$th element: $\Theta(1)$
- Append/pop element at end:
  - $\Theta(1)$ best case
  - $\Theta(n)$ worst case (where $n$ = current size)
  - $\Theta(1)$ amortized
- Insert/remove in middle: $O(n)$
  - May or may not need resize, still $O(n)$!

# DSC 190

## Data Structures & Algorithms

Lecture 2 | Part 6

## Practicalities

# Advantages

- Great cache performance (it’s an array).
- Fast access.
- Don’t need to know size in advance of allocation.

# Downsides

- Wasted memory.
- Expensive deletion in middle.

# Implementations

- Python: `list`
- C++: `std::vector`
- Java: `ArrayList`

(notebook posted on dsc190.com)

[3, "justin", pd.DataFrame]

## Exercise

Why do we need `np.array`? Python's `list` is a dynamic array, isn't that better?

# In defense of `np.array`

- Memory savings are one reason.
- Bigger reason: using Python's `list` to store numbers does not have good **cache** performance.