Please log onto https://edstem.org/us/courses/21257/discussion/ to submit live lecture questions

Please log onto PollEv.com/champk to answer the daily lecture participation question

Lecture 8: Recurrences and Intro to Hashing

CSE 373: Data Structures and Algorithms

Warm Up!

Please fill out the Poll at- pollev.com/champk

What's the theta bound for the runtime function for this piece of code?

```
public void method1(int n) {
    if (n <= 100) {
        System.out.println(":3");
    } else {
        System.out.println(":D");
        for (int i = 0; i<16; i++) {
            method1(n / 4);
        }
    }
}
```

$$T(n) = \begin{cases} \textit{constant work} & \text{if } n \le 100 \\ 16T\left(\frac{n}{4}\right) + \textit{constant work} & \text{otherwise} \end{cases}$$

a = 16, b = 4, c = 0

$\log_4 16 = 2$ $2 > 0$

$T(n) \in \Theta\left(n^{\log_b a}\right)$

$\Theta\left(n^{\log_4 16}\right) = \boldsymbol{\Theta}\left(\boldsymbol{n^2}\right)$

Master Theorem

$$T(n) = \begin{cases} d & \text{if } n \text{ is at most some constant} \\ aT\left(\frac{n}{b}\right) + f(n) & \text{otherwise} \end{cases}$$

Where $f(n)$ is $\Theta(n^c)$

If $\log_b a < c$ then $T(n) \in \Theta(n^c)$

If $\log_b a = c$ then $T(n) \in \Theta(n^c \log n)$

If $\log_b a > c$ then $T(n) \in \Theta\left(n^{\log_b a}\right)$

Announcements

Exercise 1 – Algorithm Analysis – Due Friday April 15th

Project 1 – Deques – Due Wednesday April 13th

Project 2 is out! Due Wednesday April 27th

- 2 week assignment, PLEASE PLEASE PLEASE START NOW

For real, though, it will take you 2 weeks, do not wait until next week to start

Midterm goes out Friday April 29th

Office Hours FYI

- TAs have been instructed to only spend 15 min with each student
- Attending in person makes things go faster
- For online office hours please use the Ed board post to queue

Questions

Modeling Recursive Code

Recurrence to Big-Θ

$$T(n) = \begin{cases} 2 & \text{if } n < 3 \\ 2T\left(\frac{n}{3}\right) + n & \text{otherwise} \end{cases}$$

It’s still really hard to tell what the big-O is just by looking at it.

But fancy mathematicians have a formula for us to use!

Master Theorem

$$T(n) = \begin{cases} d & \text{if } n \text{ is at most some constant} \\ aT\left(\frac{n}{b}\right) + f(n) & \text{otherwise} \end{cases}$$

Where $f(n)$ is $\Theta(n^c)$

If $\log_b a < c$ then $T(n) \in \Theta(n^c)$

If $\log_b a = c$ then $T(n) \in \Theta(n^c \log n)$

If $\log_b a > c$ then $T(n) \in \Theta\left(n^{\log_b a}\right)$

a=2 b=3 and c=1

$y = \log_b x$ *is equal to* $b^y = x$

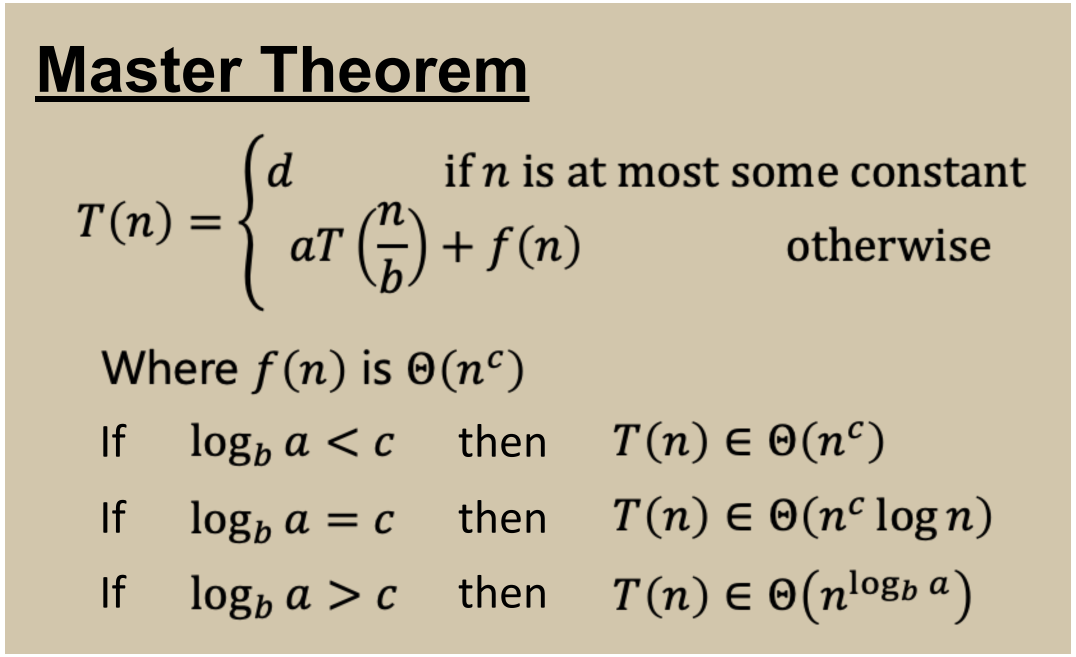
$\log_3 2 = x \Rightarrow 3^x = 2 \Rightarrow x \cong 0.63$

$\log_3 2 < 1$

We’re in case 1

$T(n) \in \Theta(n)$

Understanding Master Theorem



Master Theorem

$$T(n) = \begin{cases} d & \text{if } n \text{ is at most some constant} \\ aT\left(\frac{n}{b}\right) + f(n) & \text{otherwise} \end{cases}$$

Where $f(n)$ is $\Theta(n^c)$

If $\log_b a < c$ then $T(n) \in \Theta(n^c)$

If $\log_b a = c$ then $T(n) \in \Theta(n^c \log n)$

If $\log_b a > c$ then $T(n) \in \Theta\left(n^{\log_b a}\right)$

- A measures how many recursive calls are triggered by each method instance
- B measures the rate of change for input
- C measures the dominating term of the non recursive work within the recursive method
- D measures the work done in the base case

The log of a < c case

- Recursive case does a lot of non recursive work in comparison to how quickly it divides the input size
- Most work happens in beginning of call stack
- Non recursive work in recursive case dominates growth, n^c term

The log of a = c

- Recursive case evenly splits work between non recursive work and passing along inputs to subsequent recursive calls
- Work is distributed across call stack

The log of a > c case

- Recursive case breaks inputs apart quickly and doesn’t do much non recursive work
- Most work happens near bottom of call stack

Recursive Patterns

Pattern #1: Halving the Input

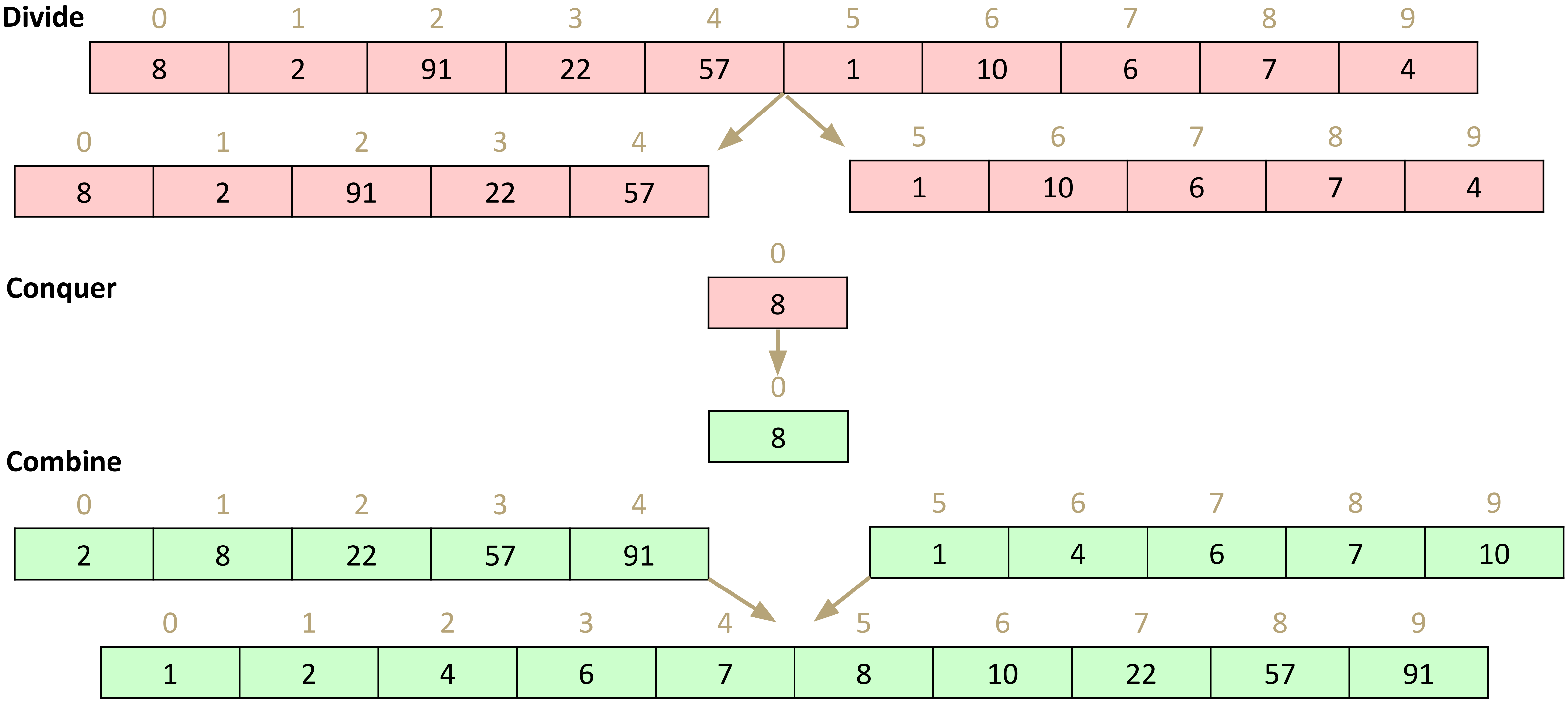
Binary Search Θ(logn)

Pattern #2: Constant size input and doing work

Merge Sort

Pattern #3: Doubling the Input

Merge Sort



Divide

0	1	2	3	4	5	6	7	8	9
8	2	91	22	57	1	10	6	7	4

0	1	2	3	4
8	2	91	22	57

5	6	7	8	9
1	10	6	7	4

Conquer

0
8

0
8

Combine

0	1	2	3	4
2	8	22	57	91

5	6	7	8	9
1	4	6	7	10

0	1	2	3	4	5	6	7	8	9
1	2	4	6	7	8	10	22	57	91

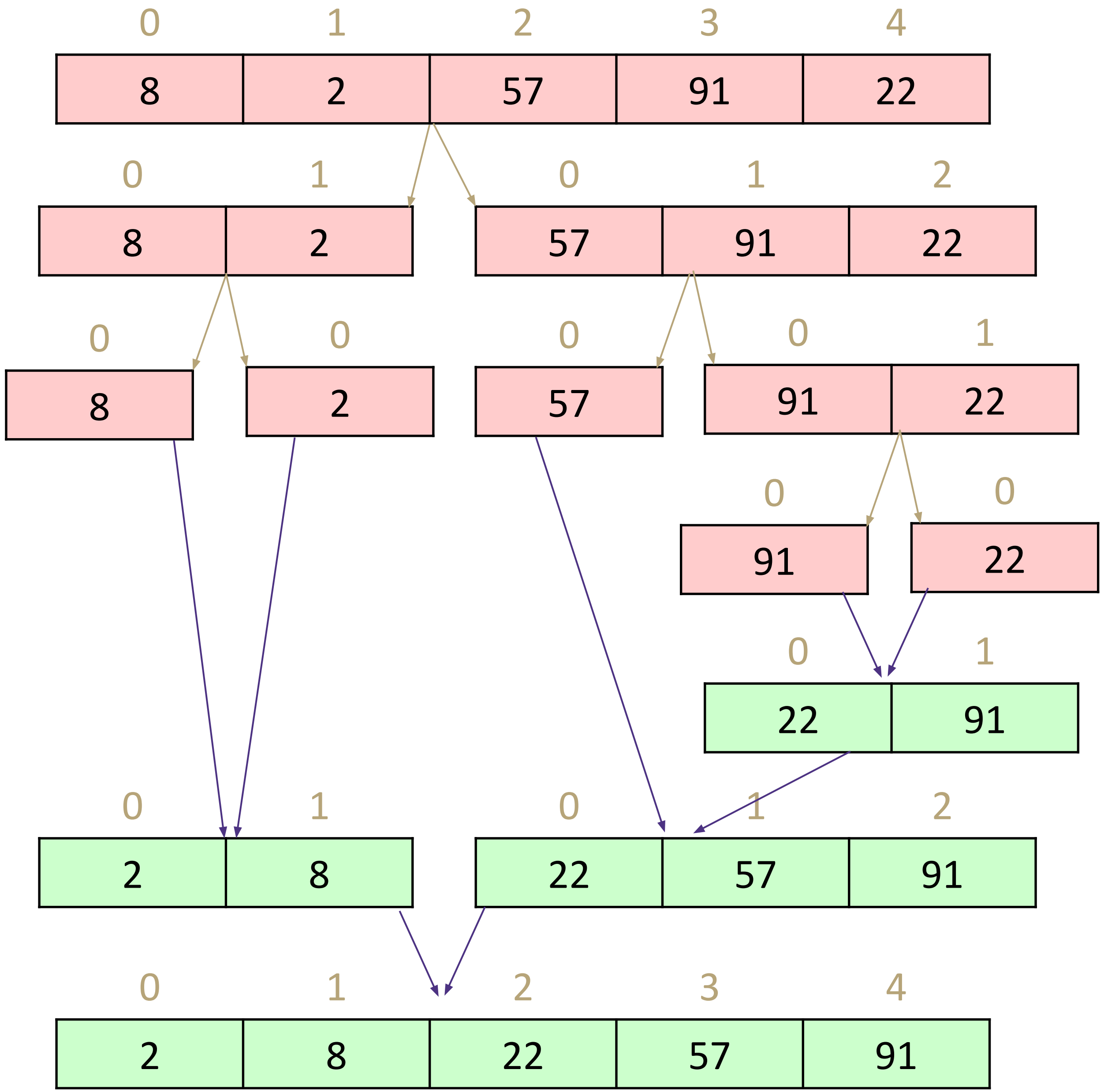
Merge Sort

```
mergeSort(input) {
   if (input.length == 1)
      return
   else
      smallerHalf = mergeSort(new [0, ...,
mid])
      largerHalf = mergeSort(new [mid + 1,
...])
      return merge(smallerHalf, largerHalf)
}
```

$$T(n) = \begin{cases} 1 \text{ if } n <= 1 \\ 2T(n/2) + n \text{ otherwise} \end{cases}$$

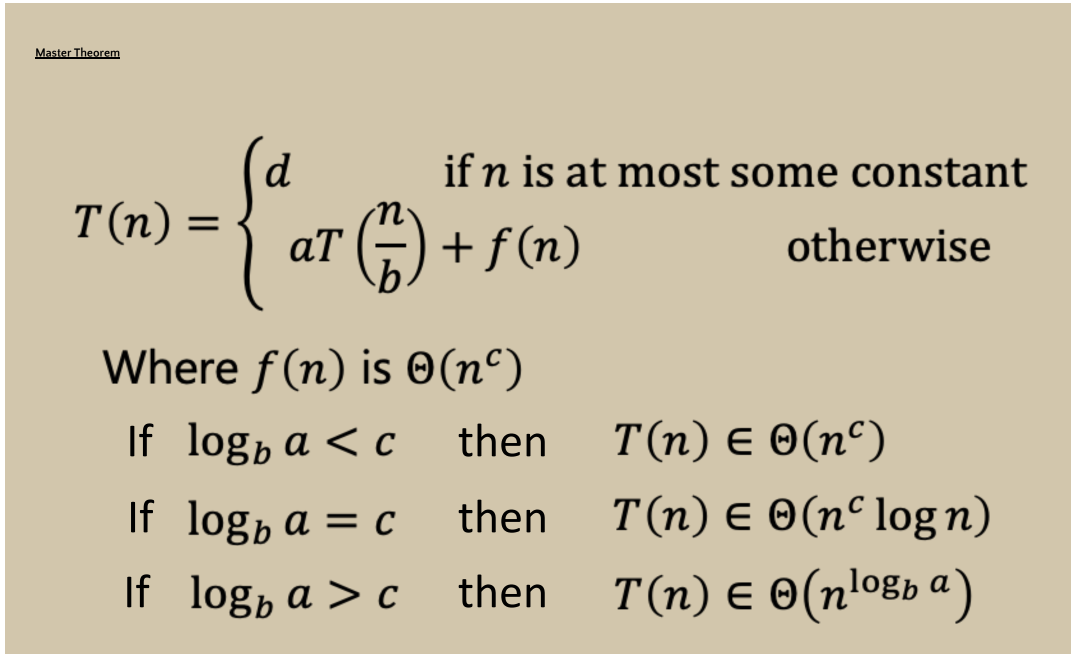
Pattern #2 – Constant size input and doing work

Take a guess! What is the Big-O of worst case merge sort?



Merge Sort Recurrence to Big-Θ

$$T(n) = \begin{cases} 1 \text{ if } n <= 1 \\ 2T(n/2) + n \text{ otherwise} \end{cases}$$



Master Theorem

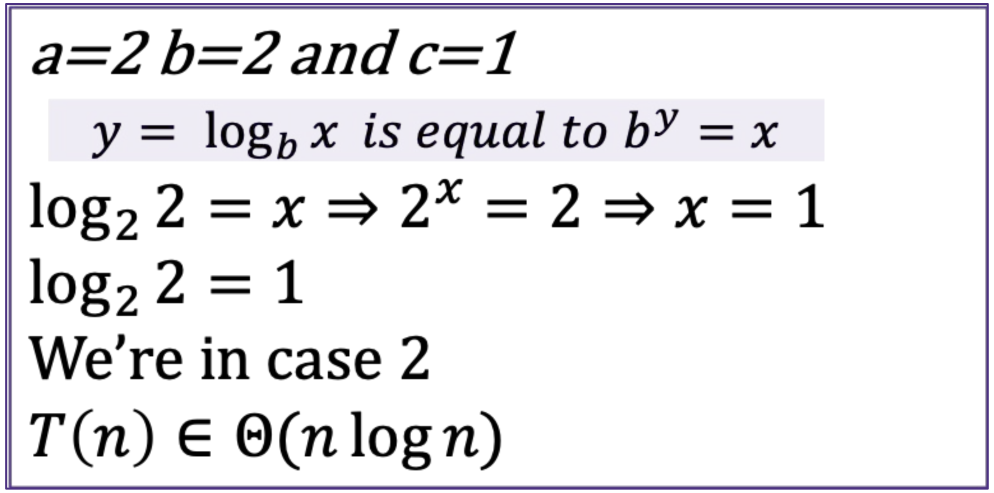
$$T(n) = \begin{cases} d & \text{if } n \text{ is at most some constant} \\ aT\left(\frac{n}{b}\right) + f(n) & \text{otherwise} \end{cases}$$

Where $f(n)$ is $\Theta(n^c)$

If $\log_b a < c$ then $T(n) \in \Theta(n^c)$

If $\log_b a = c$ then $T(n) \in \Theta(n^c \log n)$

If $\log_b a > c$ then $T(n) \in \Theta\left(n^{\log_b a}\right)$



a=2 b=2 and c=1

$y = \log_b x$ *is equal to* $b^y = x$

$\log_2 2 = x \Rightarrow 2^x = 2 \Rightarrow x = 1$

$\log_2 2 = 1$

We're in case 2

$T(n) \in \Theta(n \log n)$

Recursive Patterns

Pattern #1: Halving the Input

Binary Search Θ(logn)

Pattern #2: Constant size input and doing work

Merge Sort Θ(nlogn)

Pattern #3: Doubling the Input

Calculating Fibonacci

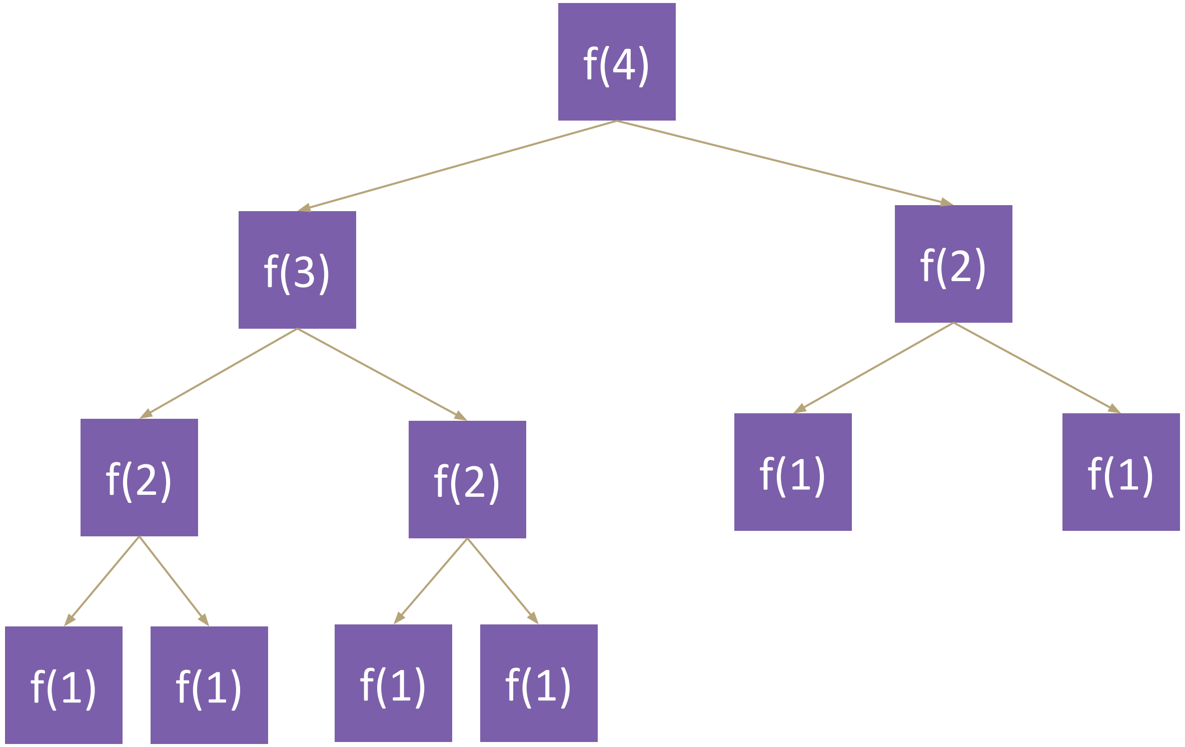
Calculating Fibonacci

```
public int fib(int n) {
    if (n <= 1) {
        return 1;
    }
    return fib(n-1) + fib(n-2);
}
```

- Each call creates 2 more calls
- Each new call has a copy of the input, almost
- Almost doubling the input at each call

Almost

Pattern #3 – Doubling the Input



Calculating Fibonacci Recurrence to Big-Θ

```
public int f(int n) {
    if (n <= 1) {
        return 1;
    }
    return f(n-1) + f(n-2);
}
```

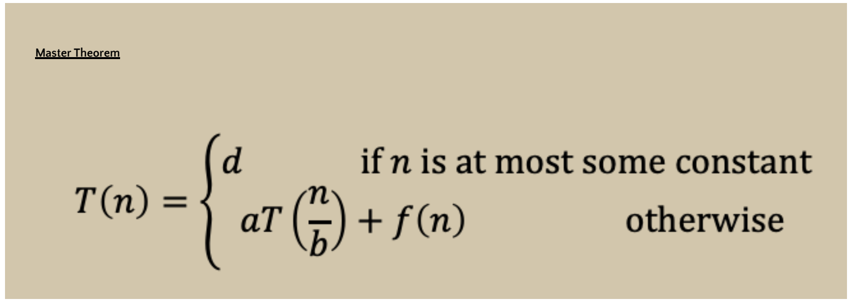
d

$2T(n\text{-}C_1) + C_2$

$$T(n) = \begin{cases} d \text{ when } n \leq 1 \\ 2T(n-C_1) + C_2 \quad \text{otherwise} \end{cases}$$

Finish the recurrence, what is the model for the recursive case?

Can we use master theorem?



Master Theorem

$$T(n) = \begin{cases} d & \text{if } n \text{ is at most some constant} \\ aT\left(\frac{n}{b}\right) + f(n) & \text{otherwise} \end{cases}$$

Uh oh, our model doesn't match that format…

Can we intuit a pattern?

T(1) = d

T(2) = 2T(2-1) + c = 2(d) + c

T(3) = 2T(3-1) + c = 2(2(d) + c) + c = 4d + 3c

T(4) = 2T(4-1) + c = 2(4d + 3c) + c = 8d + 7c

T(5) = 2T(5-1) + c = 2(8d + 7c) + c = 16d +25c

Looks like something's happening but it's tough

Maybe geometry can help!

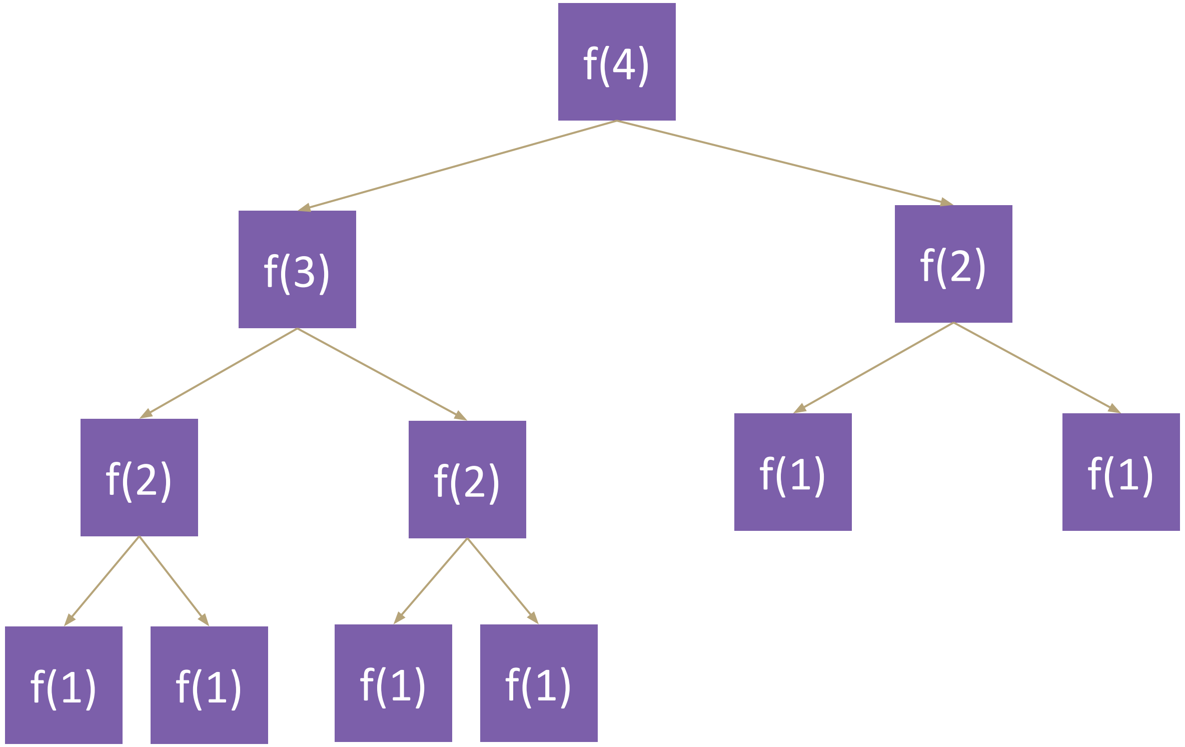
Calculating Fibonacci Recurrence to Big-Θ

How many layers in the function call tree?

How many layers will it take to transform "n" to the base case of "1" by subtracting 1

For our example, 4 -> Height = n

$$T(n) = \begin{cases} d \text{ when } n \leq 1 \\ 2T(n-1) + c \text{ otherwise} \end{cases}$$



How many function calls per layer?

Layer	Function calls
1	1
2	2
3	4
4	8

How many function calls on layer k?

$\mathbf{2^{k-1}}$

How many function calls TOTAL for a tree of k layers?

$\mathbf{1 + 2 + 3 + 4 + \ldots + 2^{k-1}}$

Calculating Fibonacci Recurrence to Big-Θ

Patterns found:

How many layers in the function call tree? n

How many function calls on layer k? 2^{k-1}

How many function calls TOTAL for a tree of k layers?

$1 + 2 + 4 + 8 + \ldots + 2^{k-1}$

Total runtime = (total function calls) x (runtime of each function call)

Total runtime = $(1 + 2 + 4 + 8 + \ldots + 2^{k-1})$ x (constant work)

$$1 + 2 + 4 + 8 + \ldots + 2^{k-1} = \sum_{i=1}^{k-1} 2^i = \frac{2^k - 1}{2 - 1} = 2^k - 1$$

Summation Identity

Finite Geometric Series

$$\sum_{i=1}^{k-1} x^i = \frac{x^k - 1}{x - 1}$$

$$\boldsymbol{T(n) = 2^n - 1 \in \Theta(2^n)}$$

Recursive Patterns

Pattern #1: Halving the Input

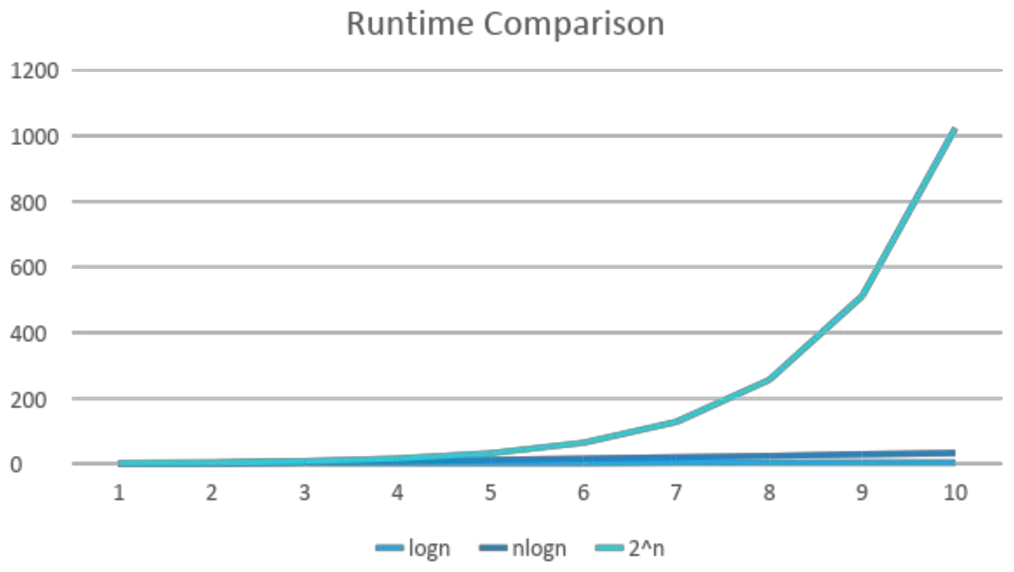
Binary Search Θ(logn)

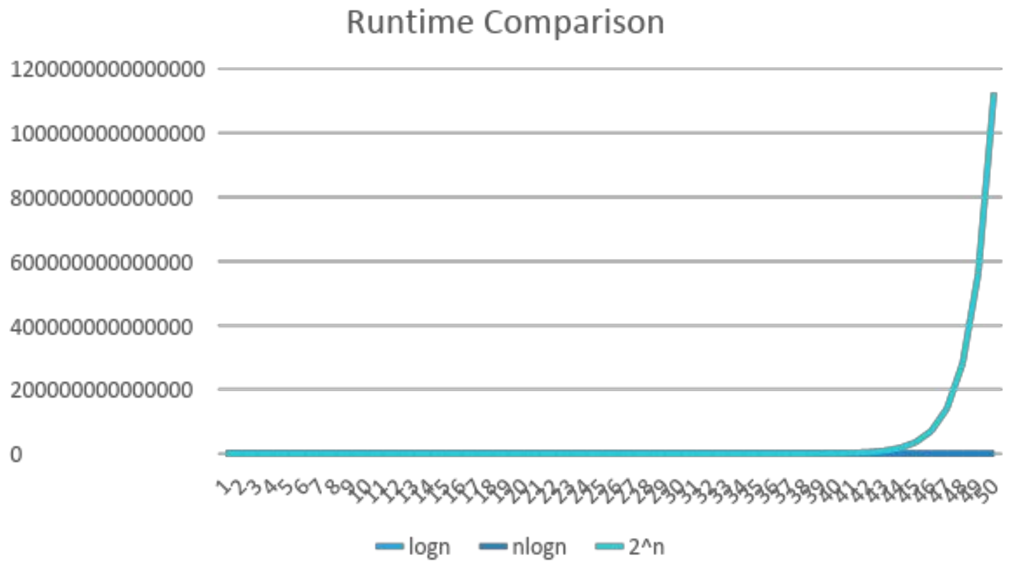
Pattern #2: Constant size input and doing work

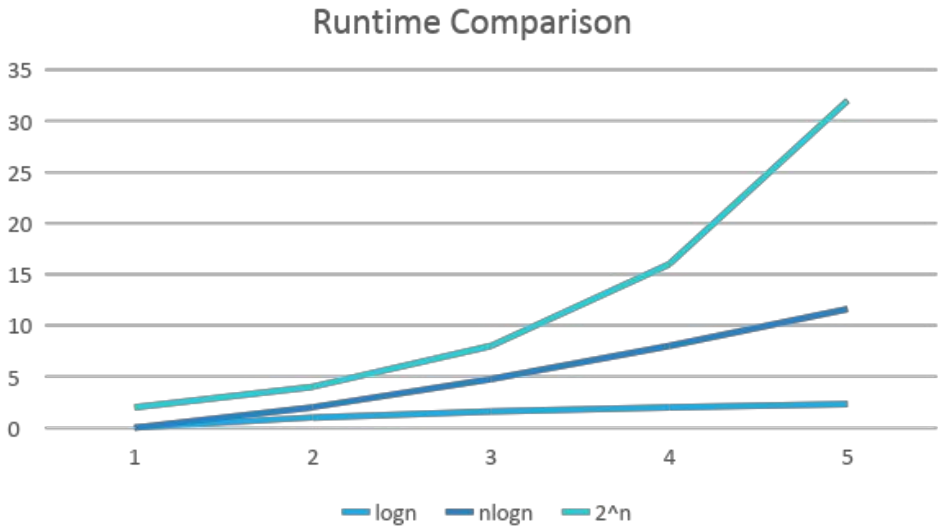
Merge Sort Θ(nlogn)

Pattern #3: Doubling the Input

Calculating Fibonacci Θ(2^n)







Questions

Intro to Hashing

Dictionaries (aka Maps)

Every Programmer's Best Friend

You'll probably use one in almost every programming project.

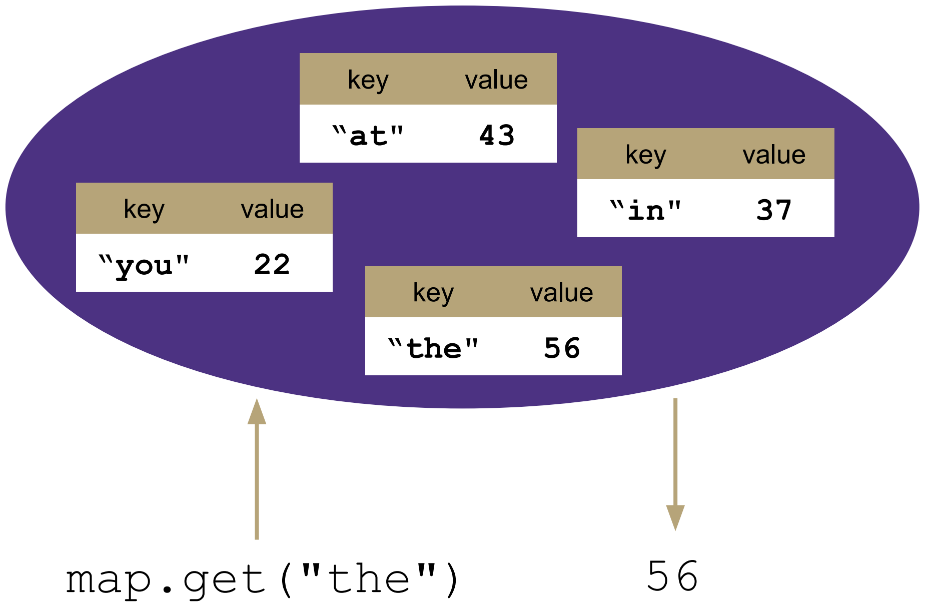
- Because it's hard to make a big project without needing one sooner or later.

```
// two types of Map implementations supposedly covered in CSE 143
Map<String, Integer> map1 = new HashMap<>();
Map<String, String> map2 =  new TreeMap<>();
```

Review: Maps

map: Holds a set of distinct *keys* and a collection of *values*, where each key is associated with one value.

- a.k.a. "dictionary"



`map.get("the")` 56

Dictionary ADT

state

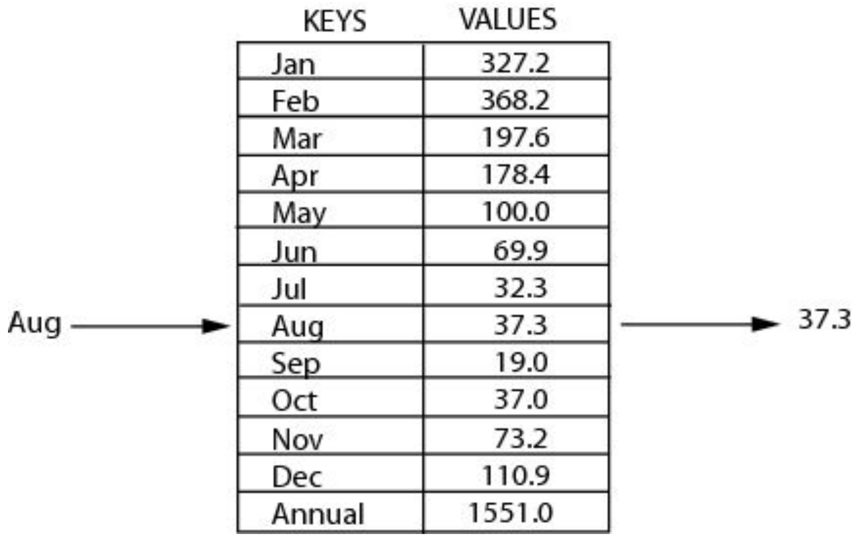
Set of items & keys
Count of items

behavior

put(key, item) add item to collection indexed with key
get(key) return item associated with key
containsKey(key) return if key already in use
remove(key) remove item and associated key
size() return count of items

supported operations:

- **put**(*key*, *value*): Adds a given item into collection with associated key,
  - **if the map previously had a mapping for the given key, old value is replaced.**
- **get**(*key*): Retrieves the value mapped to the key
- **containsKey**(key): returns true if key is already associated with value in map, false otherwise
- **remove**(*key*): Removes the given key and its mapped value



KEYS	VALUES
Jan	327.2
Feb	368.2
Mar	197.6
Apr	178.4
May	100.0
Jun	69.9
Jul	32.3
Aug	37.3
Sep	19.0
Oct	37.0
Nov	73.2
Dec	110.9
Annual	1551.0

Implementing a Map with an Array

Map ADT

state

Set of items & keys
Count of items

behavior

put(key, item) add item to collection indexed with key
get(key) return item associated with key
containsKey(key) return if key already in use
remove(key) remove item and associated key
size() return count of items

ArrayMap<K, V>

state

`Pair<K, V>[] data`

behavior

put find key, overwrite value if there. Otherwise create new pair, add to next available spot, grow array if necessary
get scan all pairs looking for given key, return associated item if found
containsKey scan all pairs, return if key is found
remove scan all pairs, replace pair to be removed with last pair in collection
size return count of items in dictionary

```
containsKey('c')
get('d')
put('b', 97)
put('e', 20)
```

0	1	2	3	4
('a', 1)	('b', 97)	('c', 3)	('d', 4)	('e', 20)

Big O Analysis – (if key is the last one looked at / not in the dictionary)

put()	O(N) linear
get()	O(N) linear
containsKey()	O(N) linear
remove()	O(N) linear
size()	O(1) constant

Big O Analysis – (if the key is the first one looked at)

put()	O(1) constant
get()	O(1) constant
containsKey()	O(1) constant
remove()	O(1) constant
size()	O(1) constant

Implementing a Map with Nodes

Map ADT

state

Set of items & keys
Count of items

behavior

put(key, item) add item to collection indexed with key

get(key) return item associated with key

containsKey(key) return if key already in use

remove(key) remove item and associated key

size() return count of items

LinkedMap<K, V>

state

front
size

behavior

put if key is unused, create new with pair, add to front of list, else replace with new value

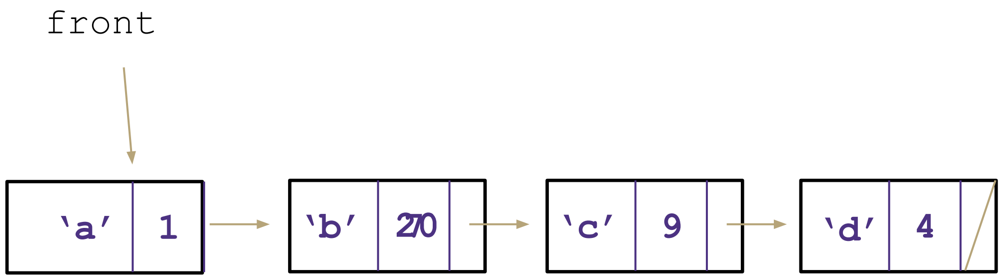
get scan all pairs looking for given key, return associated item if found

containsKey scan all pairs, return if key is found

remove scan all pairs, skip pair to be removed

size return count of items in dictionary

```
containsKey('c')
get('d')
put('b', 20)
```



Big O Analysis – (if key is the last one looked at / not in the dictionary)

put()	O(N) linear
get()	O(N) linear
containsKey()	O(N) linear
remove()	O(N) linear
size()	O(1) constant

Big O Analysis – (if the key is the first one look at)

put()	O(1) constant
get()	O(1) constant
containsKey()	O(1) constant
remove()	O(1) constant
size()	O(1) constant

Can we do better?

Let's simplify the problem we're working with + combine it with some facts about arrays.

Problem Simplification: only worry about supporting integer keys

Array Facts: accessing (`data[i]`) or updating an element (`data[i] = …`) at a given index takes `Theta(1)` runtime.

If we store the Key-Value pairs at the `data[key]` then we don't have to do any looping to find it. For example consider `` `containsKey` `` or `` `get` `` -- we can just jump directly to `data[key]` to figure out the return answer.

DirectAccessMap<Integer, V>

state

```
Data[]
size
```

behavior

```
put put item at given index
get get item at given index
containsKey if data[] null at
index, return false, return
true otherwise
remove nullify element at index
size return count of items in
dictionary
```

indices	0	1	2	3	4	5	6	7	8	9
data				(3, Sherdil)						

```
put(3, "Sherdil");
get(3);
```

Can we do better? -- Direct Access Map impl.

```
public void put(int key, V value) {
    this.array[key] = value;
}

public boolean containsKey(int key) {
    return this.array[key] != null;
}

public V get(int key) {
    return this.array[key];
}

public void remove(int key) {
    this.array[key] = null;
}
```

DirectAccessMap<Integer, V>

state

Data[]

size

behavior

put put item at given index

get get item at given index

containsKey if data[] null at index, return false, return true otherwise

remove nullify element at index

size return count of items in dictionary

Operation		Array w/ indices as keys
put(key,value)	best	$\Theta(1)$
	worst	$\Theta(1)$
get(key)	best	$\Theta(1)$
	worst	$\Theta(1)$
containsKey(key)	best	$\Theta(1)$
	worst	$\Theta(1)$

Direct Access Map tradeoffs:

- what’s a benefit of using DirectAccessMap?
- what’s a bad thing when using DirectAccessMap?

- ☹ wasted space
  - what if we want to store two key: 0 and 99999999999? Our current setup would just be wasting all that array space in-between
- ☹ only integer keys
  - kind of annoying that we could only have this for ints, but **being able to quickly go from the key to the array index is super valuable because it’s array lookups are fast (constant time)**. When we can just jump to the right position, we avoid the looping that ArrayMap/LinkedMap had to do where you might have to loop and look at every element. We’ll keep this core idea of ”knowing the index” and jumping there right away for all the versions of the dictionaries we talk about today.
- ☺ super fast though: $\Theta(1)$ runtime for everything

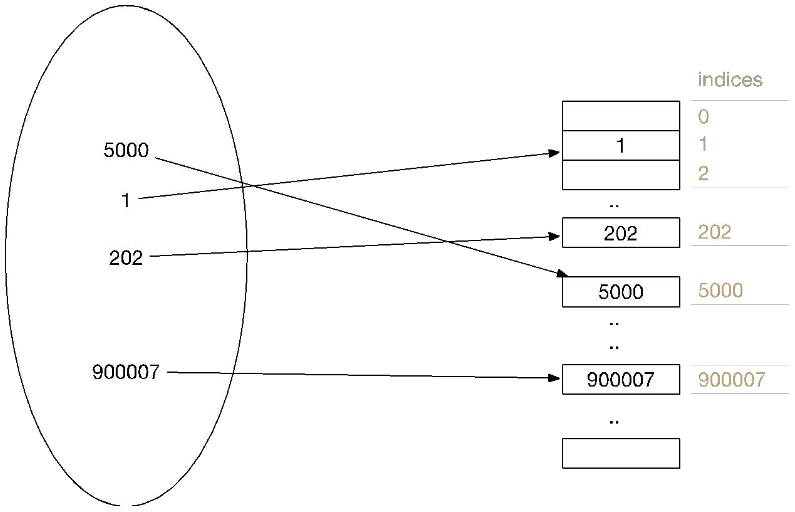
Can we do this for any integer?

Idea 1:

Create a GIANT array with every possible integer as an index

Problems:

- Can we allocate an array big enough?
- Super wasteful

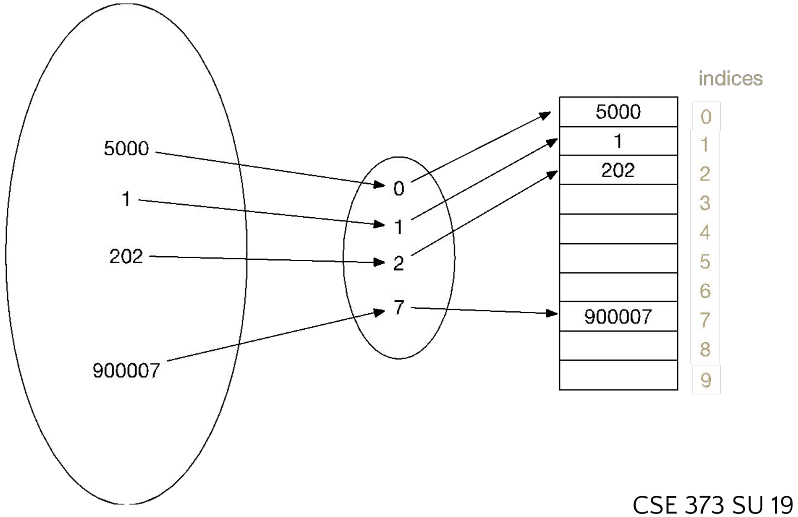


Idea 2:

Create a smaller array, but create a way to translate given integer keys into available indices. Way less wasteful space-wise.

Problem:

- How can we pick a good translation?



Hash functions: translating a piece of data to an int

Hash function definition

A hash function is any function that can be used to map data of arbitrary size to fixed-size values.

In our case: we want to translate int keys to a valid index in our array. If our array is length 10 but our input key is 500, we need to make sure we have a way of mapping that to a number between 0 and 9 (the valid indices for a length 10 array). This mapping that we decide on is a **hash function**.

One simple thing we can do (and that you will do when you implement this in your project):

Hash function: take your key and % it by the length of the array.

ex: key is 500, and array is length 10 – if you take 500 % 10, you will get the number 0, so we'd just plop 500 and it's value at index 0.

“review”: Integer remainder with % “mod”

The `%` operator computes the remainder from integer division.

`14 % 4` is `2`

```
      3
4 ) 14
    12
     2
```

`218 % 5` is `3`

```
     43
5 ) 218
    20
     18
     15
      3
```

Equivalently, to find `a % b` (for `a,b > 0`):

```
while(a > b-1)
   a -= b;
return a;
```

Applications of `%` operator:

- Obtain last digit of a number: `230857 % 10` is `7`
- See whether a number is odd: `7 % 2` is `1`, `42 % 2` is `0`
- Limit integers to specific range: `8 % 12` is `8`, `18 % 12` is `6`

Limit keys to indices within array

For more review/practice, check out https://www.khanacademy.org/computing/computer-science/cryptography/modarithmetic/a/what-is-modular-arithmetic

First Hash Function: % table size

indices	0	1	2	3	4	5	6	7	8	9
elements	“foo”	“biz”				“bar”			“bop”	

```
put(0, “foo”);  0 % 10 = 0
put(5, “bar”);  5 % 10 = 5
put(11, “biz”)  11 % 10 = 1
put(18, “bop”); 18 % 10 = 8
```

Implement First Hash Function

```
public void put(int key, int value) {
    data[hashToValidIndex(key)] = value;
}

public V get(int key) {
    return data[hashToValidIndex(key)];
}

public int hashToValidIndex(int k) {
   return k % this.data.length;
}
```

Note: % is just a math operator like +, -, /, *, so it's constant runtime

SimpleHashMap<Integer>

state

Data[]

size

behavior

put mod key by table size, put item at result

get mod key by table size, get item at result

containsKey mod key by table size, return data[result] == null remove mod key by table size, nullify element at result

size return count of items in dictionary

Operation		Array w/ indices as keys
put(key,value)	best	$\Theta(1)$
	worst	$\Theta(1)$
get(key)	best	$\Theta(1)$
	worst	$\Theta(1)$
containsKey(key)	best	$\Theta(1)$
	worst	$\Theta(1)$

Questions?

things we talked about:

- review of ArrayMap + LinkedMap
- DirectAccessMap
- % as a hash function andSimpleHashMap

First Hash Function: % table size

indices	0	1	2	3	4	5	6	7	8	9
elements	“:(”	“biz”				“bar”			“bop”	

```
put(0, “foo”);   0 % 10 = 0
put(5, “bar”);   5 % 10 = 5
put(11, “biz”)   11 % 10 = 1
put(18, “bop”);  18 % 10 = 8
put(20, “:(”);   20 % 10 = 0
```

→ Collision!

Hash Obsession: Collisions

Collision: multiple keys translate to the same location of the array

Future big idea: the fewer the collisions, the better the runtime! (we'll see this when we figure out that resolving these leads to worse runtime)

Two questions:

1. When we have a collision, how do we resolve it?
2. How do we minimize the number of collisions?

Roadmap for lecture content today

- Maps/Dictionary review
- DirectAccessMap
  - a map implemented with an array with only integer keys
- SimpleHashMap
  - a more flexible version of DirectAccessMap that uses a hash function on the key of interest to figure out where it is in the array
- **SeparateChainingHashMap**
  - fixes some limitations of the above Maps while still being very fast (in-practice).
  - It's what you'll implement in project 2 / what Java's official HashMap does -- it's the back-bone data structure that powers so many Java programs and that you will definitely use if you keep programming. Get hyped!

Strategies to handle hash collision

There are multiple strategies. In this class, we'll cover the following ones:

1. Separate chaining
2. Open addressing
   - Linear probing
   - Quadratic probing
   - Double hashing

Separate chaining

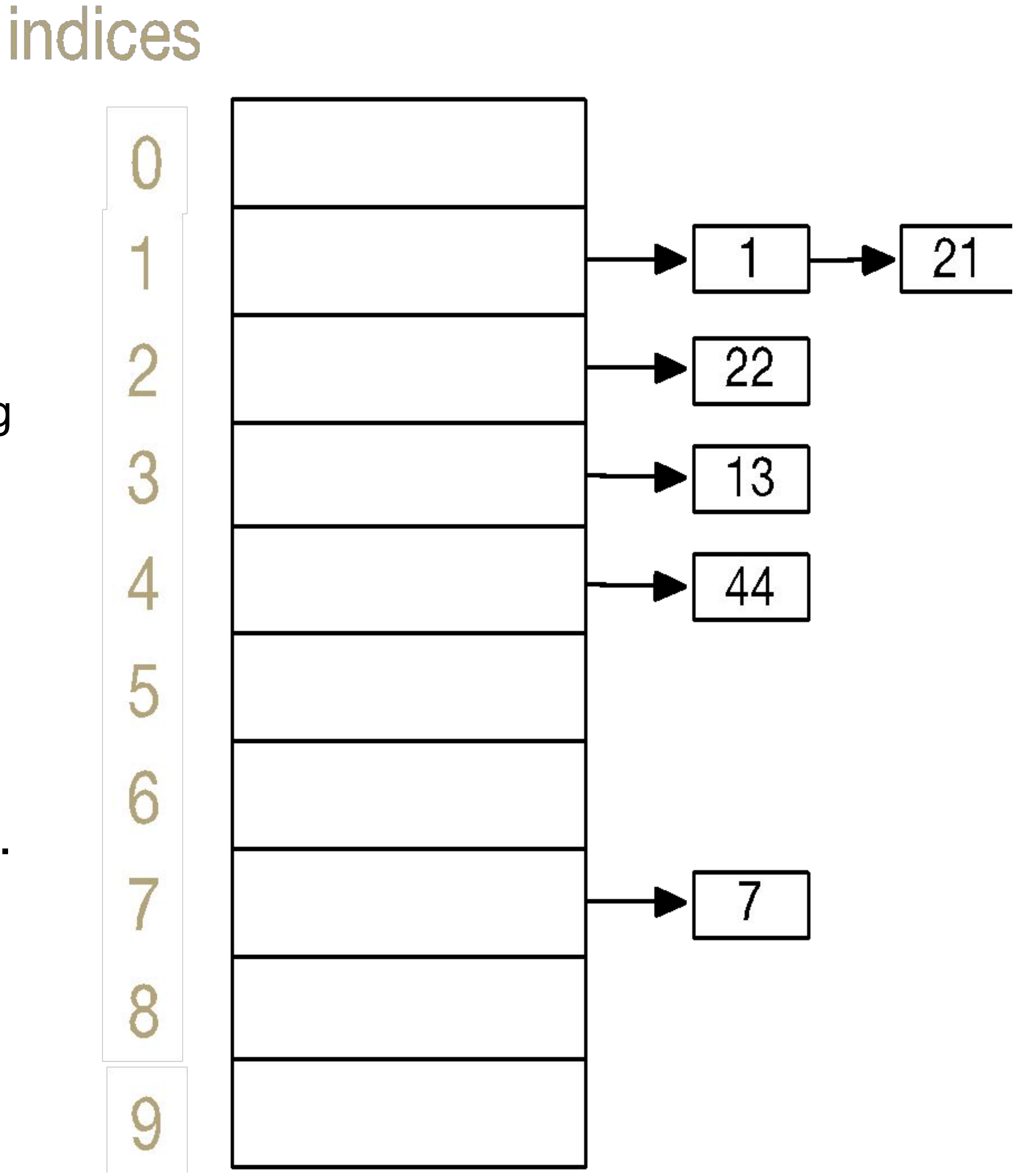
Solution 1: Separate Chaining

Each index in our array represents a “bucket”. When an item x hashes to index h:

- If the bucket at index h is empty: create a new list containing x
- If the bucket at index h is already a list: add x if it is not already present

in other words:

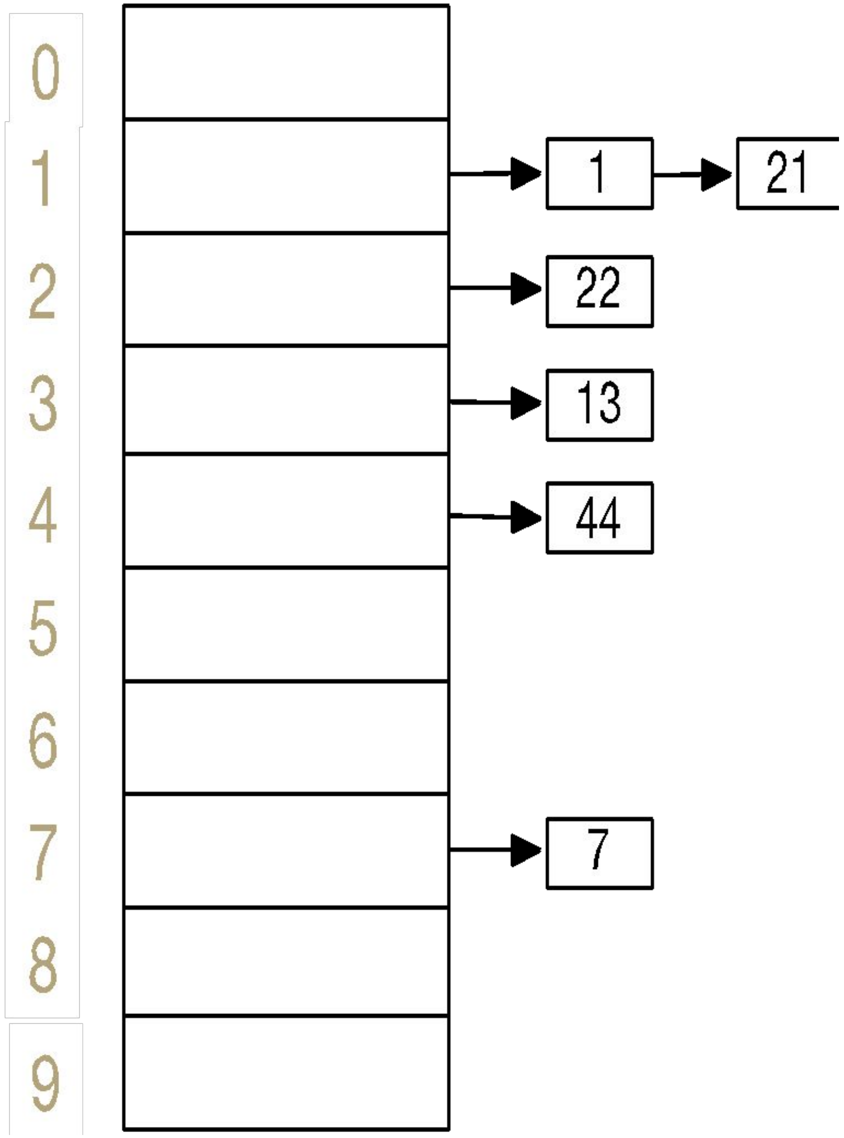
If multiple things hash to the same index, then we’ll just put all of those in that same index bucket. Often, you’ll see the data structure chosen is a linked-list like structure.



Separate chaining

Reminder: the implementations of put/get/containsKey are all very similar, and almost always will have the same complexity class runtime

```
// some pseudocode
public boolean containsKey(int key) {
    int bucketIndex = key % data.length;
    loop through data[bucketIndex]
     return true if we find the key in
     data[bucketIndex]
    return false if we get to here (didn't
  find it)
}
```

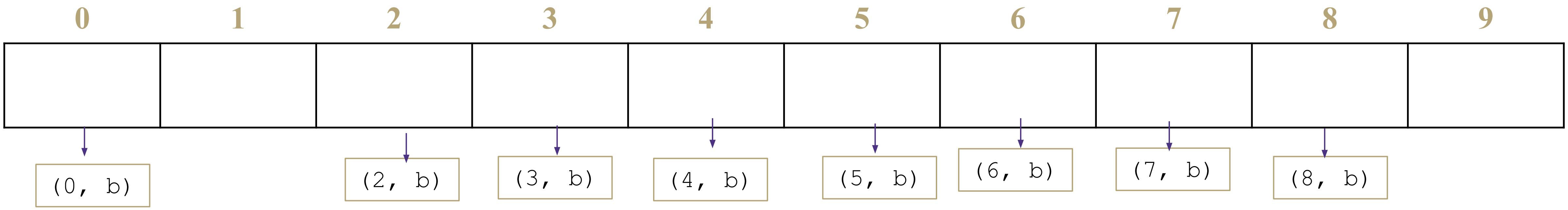


runtime analysis

Are there different possible states for our Hash Map that make this code run slower/faster, assuming there are already n key-value pairs being stored?

Yes! If we had to do a lot of loop iterations to find the key in the bucket, our code will run slower.

A best case situation for separate chaining



0	1	2	3	4	5	6	7	8	9
(0, b)		(2, b)	(3, b)	(4, b)	(5, b)	(6, b)	(7, b)	(8, b)	

It’s possible (and likely if you follow some best-practices) that everything is spread out across the buckets pretty evenly. This is the opposite of the last slide: when we have minimal collisions, our runtime should be less. For example, if we have a bucket with only 0 or 1 element in it, checking containsKey for something in that bucket will only take a constant amount of time.

We’re going to try a lot of stuff we can to make it more likely we achieve this beautiful state ☺️.

In-practice situations for separate chaining

Generally we can achieve something close to the best case situation from the previous slide and maintain our Hash Map so that every bucket only has a small constant number of items. There may be some outliers that have slightly more buckets, but generally if we follow all the best practices, the runtime will still be Θ(1) for most cases!

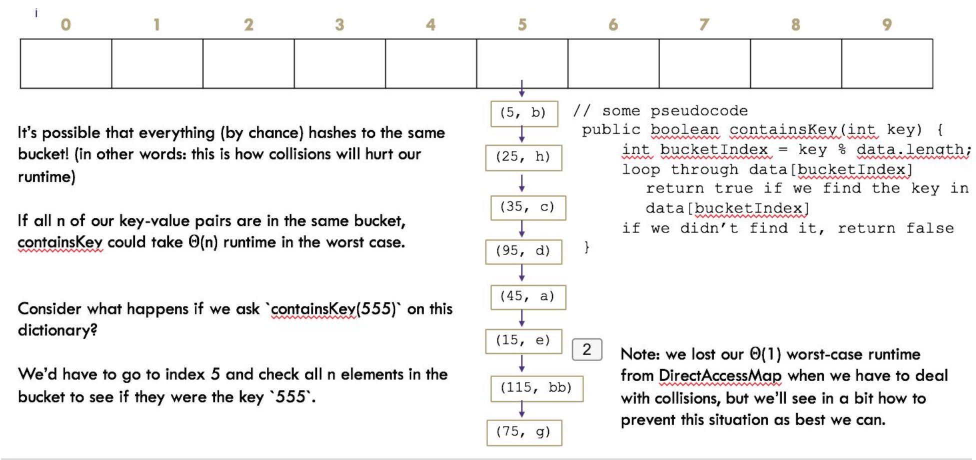
(The worst case is still Θ(n) but again, we'll try really hard to prevent that)

Operation		Array w/ indices as keys
put(key,value)	best	Θ(1)
	In-practice	Θ(1)
	worst	Θ(n)
get(key)	best	Θ(1)
	In-practice	Θ(1)
	worst	Θ(1)
remove(key)	best	Θ(1)
	In-practice	Θ(1)
	worst	O(n)

Reminder: the in-practice runtimes are assuming an even distribution of the keys inside the array and following of best-practices to ensure the average chain length is low.

Best practices (pay attention to this for the hw)

- what about resizing?
  - for data structures like ArrayMap or ArrayList or ArrayStack we had to resize when we’re full just because we couldn’t store any more things! But our Separate Chaining Hash Map is a little bit different: we aren’t ever ***forced*** to resize our main array, since the buckets are flexible size.



It’s possible that everything (by chance) hashes to the same bucket! (in other words: this is how collisions will hurt our runtime)

If all n of our key-value pairs are in the same bucket, containsKey could take Θ(n) runtime in the worst case.

Consider what happens if we ask `containsKey(555)` on this dictionary?

We’d have to go to index 5 and check all n elements in the bucket to see if they were the key `555`.

```
// some pseudocode
 public boolean containsKey(int key) {
     int bucketIndex = key % data.length;
     loop through data[bucketIndex]
        return true if we find the key in
        data[bucketIndex]
     if we didn’t find it, return false
 }
```

Note: we lost our Θ(1) worst-case runtime from DirectAccessMap when we have to deal with collisions, but we’ll see in a bit how to prevent this situation as best we can.

It turns out we still want to resize “every so often” to make sure the average/expected length of each bucket is a small number.

Consider what happens if we had the array length 10 like on the left, but had 100 key-value pairs?

Assuming our in-practice niceness (not-worst case) you would expect on average each of the 10 buckets has about 10 key-value pairs in it.

What happens if we stick with the same size array but add 100 more key-value pairs? Each bucket gets about 10 more –key-value pairs and the runtime is getting worse and worse.

Best practices (pay attention to this for the hw)

It turns out we still want to resize "every so often" to make sure the average/expected length of each bucket is a small number.

Consider what happens if we had the array length 10 like on the left, but had 100 key-value pairs?

Assuming our in-practice niceness (not-worst case) you would expect on average each of the 10 buckets has about 10 key-value pairs in it.

What happens if we stick with the same size array but add 100 more key-value pairs? Each bucket gets about 10 more –key-value pairs and the runtime is getting worse and worse.

The pattern we're getting to is that the expected runtime is approximately: # of pairs / array.length (AKA n / c where n is the number of elements and c is the number of possible chains). If array.length is fixed for your whole program, then this is an order-n runtime, but if the array.length also increases (because you re-size) and you redistribute out the values evenly across the buckets, you can keep your runtime low. In particular, **if you resize when when your n / c ratio increases to about 1, you're expected to have 1 element or fewer in each bucket at all times.** (do this on your homework).

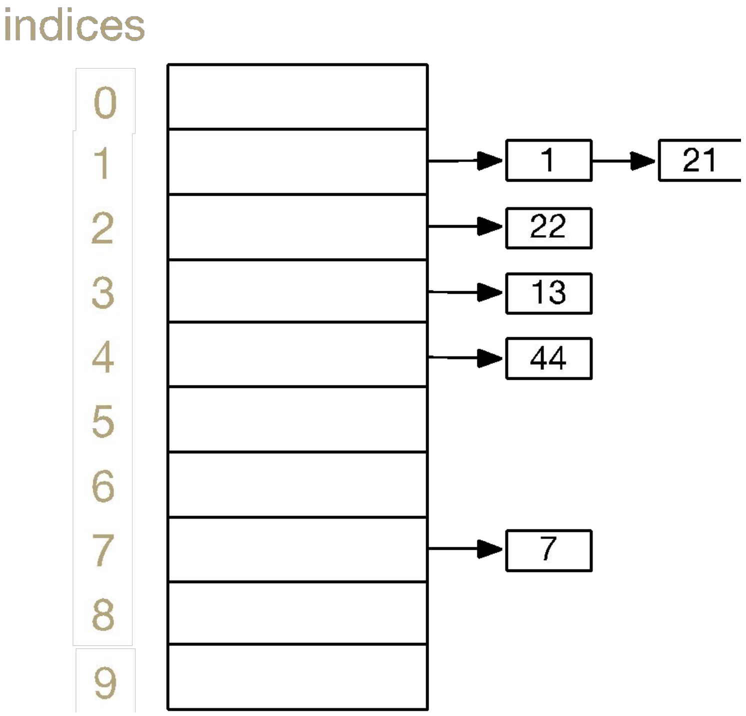
Tip: make sure you re-hash (re-distribute) your keys by the new array length after re-sizing so they don't get clustered in the old array length range.

Lambda + resizing rephrased

To be more precise, the in-practice runtime depends on λ, the current average chain length.

However, if you resize once you hit that 1:1 threshold, the current λ is expected to be less than 1 (which is a constant / constant runtime, so we can simplify to O(1)).

Operation		Array w/ indices as keys
put(key,value)	best	O(1)
	In-practice	O(λ)
	worst	O(n)
get(key)	best	O(1)
	In-practice	O(λ)
	worst	O(n)
remove(key)	best	O(1)
	In-practice	O(λ)
	worst	O(n)



"In-Practice" Case:
Depends on average number of elements per chain

Load Factor λ

If n is the total number of key-value pairs

Let c be the capacity of array

Load Factor λ = $\frac{n}{c}$

What about non integer keys?

Hash function definition
A **hash function** is any <u>function</u> that can be used to map <u>data</u> of arbitrary size to fixed-size values.

Let’s use define another hash function to change stuff like Strings into ints!

Best practices for designing hash functions:

Avoid collisions

- The more collisions, the further we move away from O(1+λ)
- Produce a wide range of indices, and distribute evenly over them

Low computational costs

- Hash function is called every time we want to interact with the data

(Before we % by length, we have to convert the data into an int)

Implementation 1: Simple aspect of values

```
public int hashCode(String input) {
   return input.length();
}
```

Pro: super fast

Con: lots of collisions!

Implementation 2: More aspects of value

```
public int hashCode(String input) {
   int output = 0;
   for(char c : input) {
      out += (int)c;
   }
   return output;
}
```

Pro: still really fast

Con: some collisions

Implementation 3: Multiple aspects of value + math!

```
public int hashCode(String input) {
   int output = 1;
   for (char c : input) {
      int nextPrime = getNextPrime();
      out *= Math.pow(nextPrime, (int)c);
   }
   return Math.pow(nextPrime, input.length());
}
```

Pro: few collisions

Con: slow, gigantic integers

Java's hashCode (relevant for project)

- Luckily, most of these design decisions have been made for us by smart people. All objects in java come with a `hashCode()` method that does some magic (see previous slide for the not-magic version) to turn any object type (like String, ArrayList, Point, Scanner) into an integer. These hashCodes are designed to distribute pretty evenly / not have lots of collisions, so we use them as the starting point for determining the bucket index.

- high level steps to figure out which bucket a key goes into
  - call the key.hashCode() to get an int representation of the object
  - % by the array table length to convert it to a valid index for your hash map

Best practices for an nice distribution of keys recap

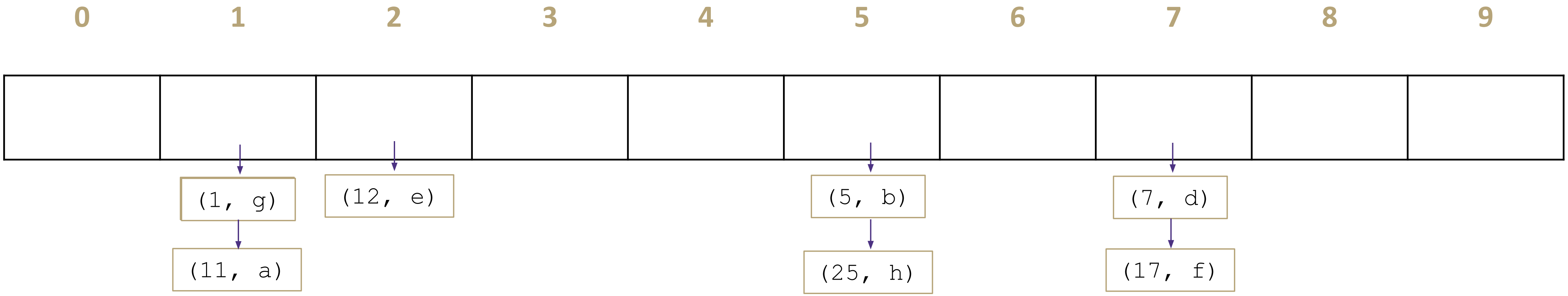
- resize when lambda (number of elements / number of buckets) increases up to 1
- when you resize, you can choose a the table length that will help reduce collisions if you multiply the array length by 2 and then choose the nearest prime number
- design the hashCode of your keys to be somewhat complex and lead to a distribution of different output numbers

Practice

Consider an IntegerDictionary using separate chaining with an internal capacity of 10. Assume our buckets are implemented using a LinkedList where we append new key-value pairs to the end.

Now, suppose we insert the following key-value pairs. What does the dictionary internally look like?

(1, a) (5,b) (11,a) (7,d) (12,e) (17,f) (1,g) (25,h)



0 1 2 3 4 5 6 7 8 9

(1, g)

(11, a)

(12, e)

(5, b)

(25, h)

(7, d)

(17, f)

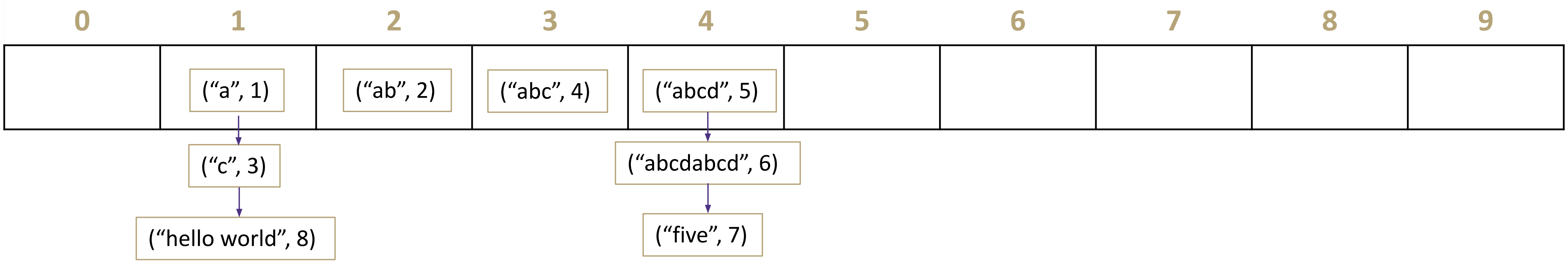
Practice

Consider a StringDictionary using separate chaining with an internal capacity of 10. Assume our buckets are implemented using a LinkedList. Use the following hash function:

```
public int hashCode(String input) {
   return input.length() % arr.length;
}
```

Now, insert the following key-value pairs. What does the dictionary internally look like?

("a", 1) ("ab", 2) ("c", 3) ("abc", 4) ("abcd", 5) ("abcdabcd", 6) ("five", 7) ("hello world", 8)



0 1 2 3 4 5 6 7 8 9

("a", 1)

("c", 3)

("hello world", 8)

("ab", 2)

("abc", 4)

("abcd", 5)

("abcdabcd", 6)

("five", 7)

Java and Hash Functions

Object class includes default functionality:

- equals
- hashCode

If you want to implement your own hashCode you should:

- Override BOTH hashCode() and equals()

If a.equals(b) is true then a.hashCode() == b.hashCode() **MUST** also be true

That requirement is part of the Object interface.
Other people's code will assume you've followed this rule.

Java's HashMap (and HashSet) will assume you follow these rules and conventions for your custom objects if you want to use your custom objects as keys.