香港科技大学(广州)
THE HONG KONG
UNIVERSITY OF SCIENCE AND
TECHNOLOGY (GUANGZHOU)

Design and Analysis of Algorithms

Jing Tang | DSAA 2043 Fall 2024

Hashing

Outline

- **Hash tables** are another sort of data structure that allows fast INSERT/DELETE/SEARCH.
  - like self-balancing binary trees
  - The difference is we can get better performance in expectation by using randomness.
- **Hash families** are the magic behind hash tables.
- **Universal hash families** are even more magical.

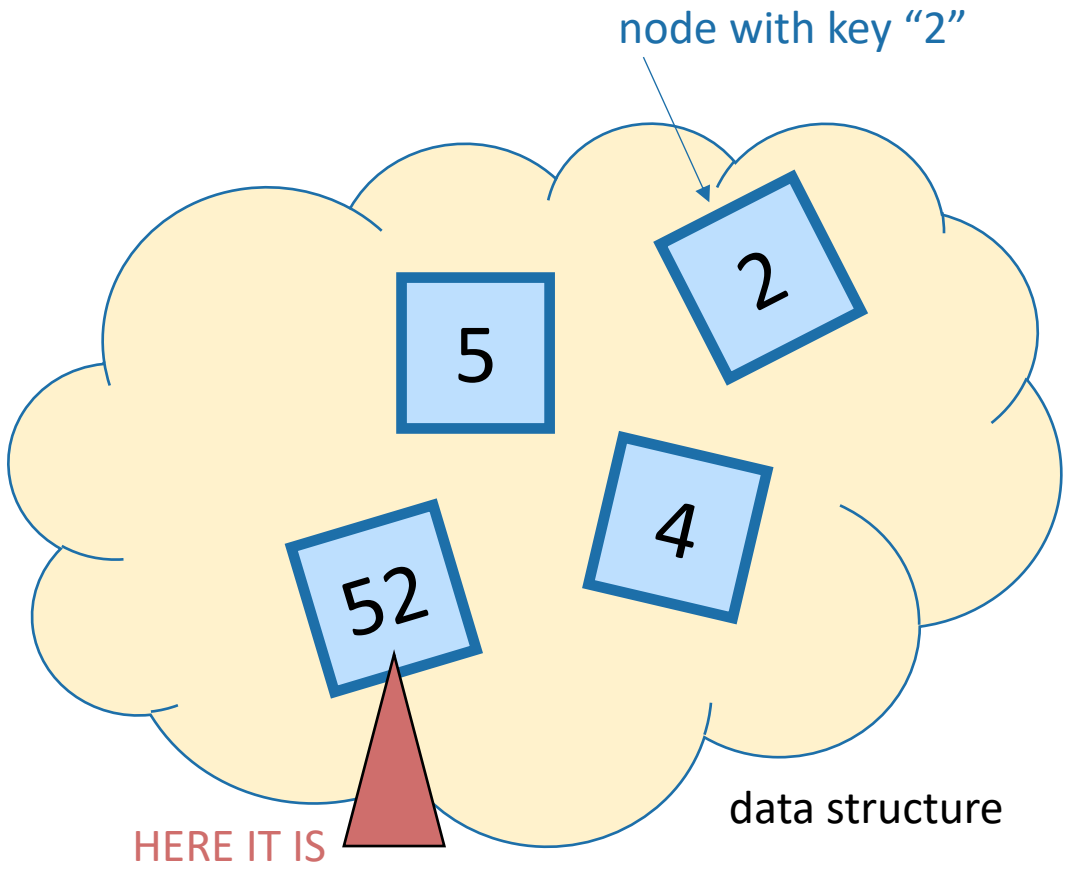
Goal

- We want to store nodes with keys in a data structure that supports fast INSERT/DELETE/SEARCH.

- INSERT

- DELETE

- SEARCH



Last time

- Self balancing trees:
  - O(log(n)) deterministic INSERT/DELETE/SEARCH

#prettysweet

Today:

- Hash tables:
  - O(1) expected time INSERT/DELETE/SEARCH
- Worse worst-case performance, but often great in practice.

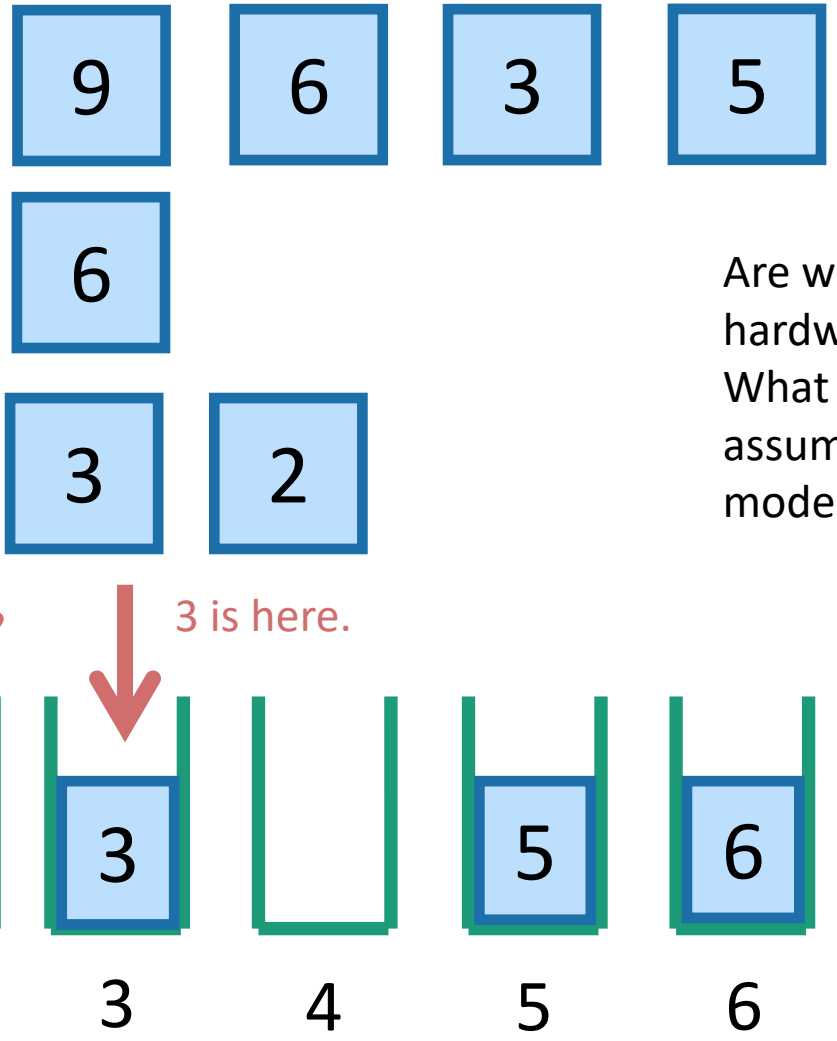
#evensweeterinpractice

eg, Python's `dict`, Java's `HashSet/HashMap`, C++'s `unordered_map`

Hash tables are used for databases, caching, object representation, …

One way to get O(1) time

- Say all keys are in the set {1,2,3,4,5,6,7,8,9}.
- INSERT:
- DELETE:
- SEARCH:

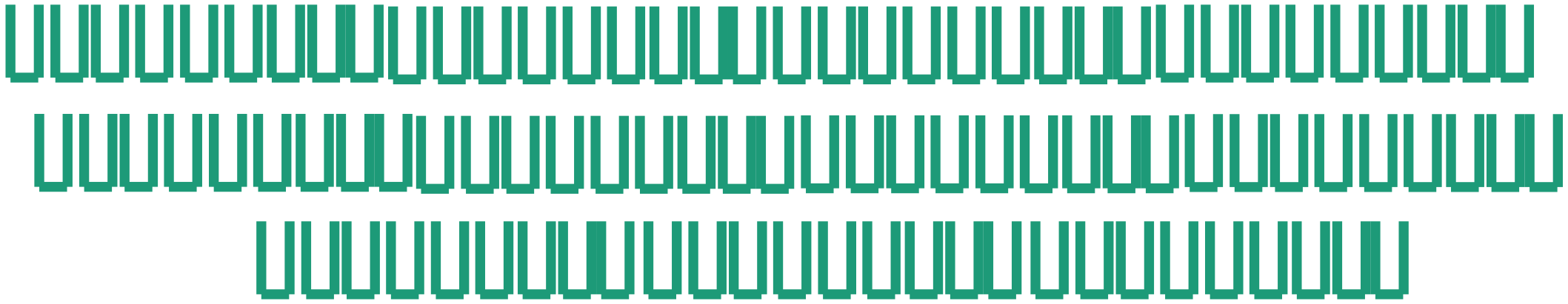


This is called "direct addressing"

Are we delegating to hardware/memory? What are the assumptions behind our model of computation?

That should look familiar

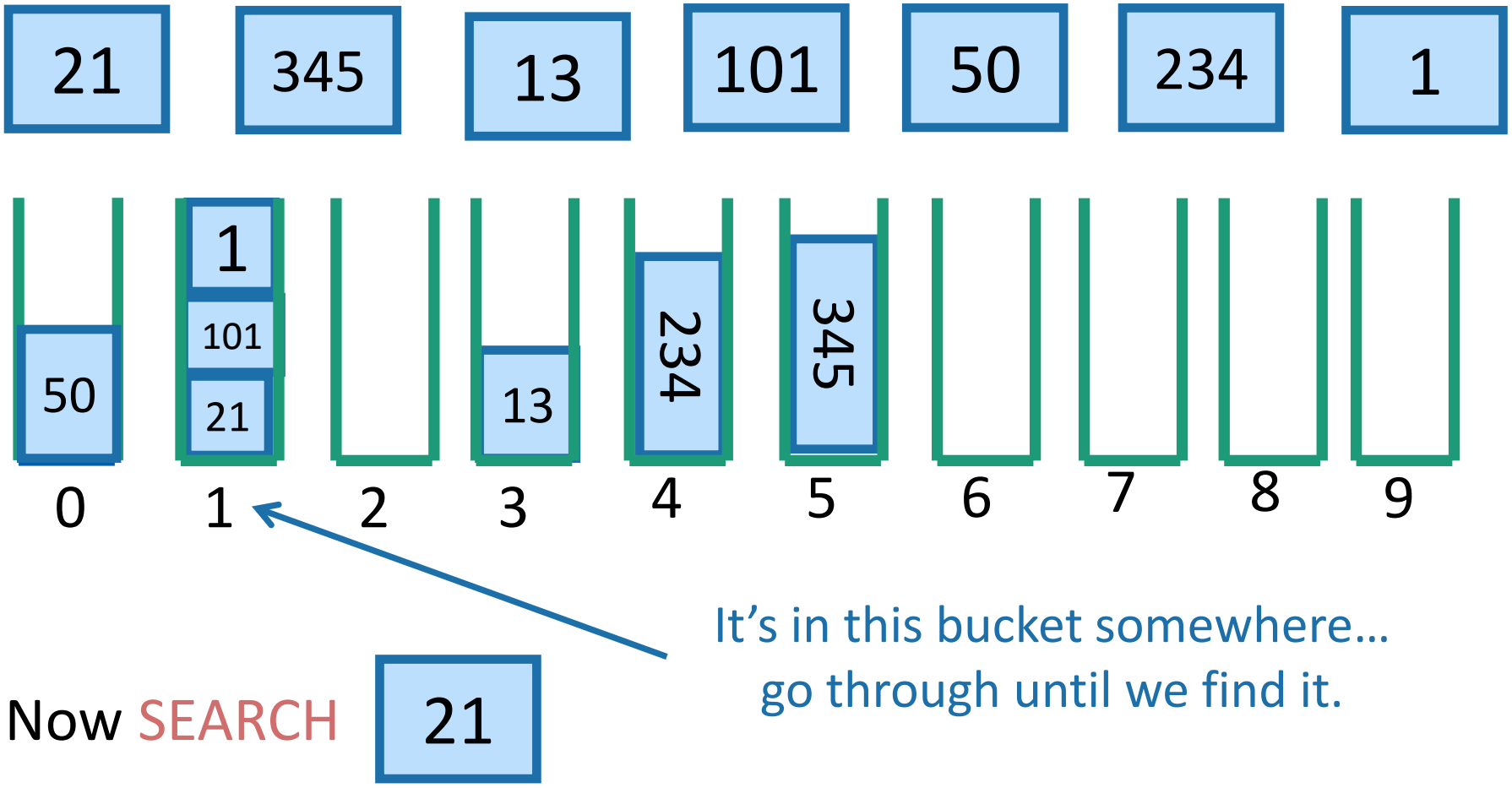
- If the keys may come from a "universe" U = {1,2, …., 10000000000}, it takes a lot of space.



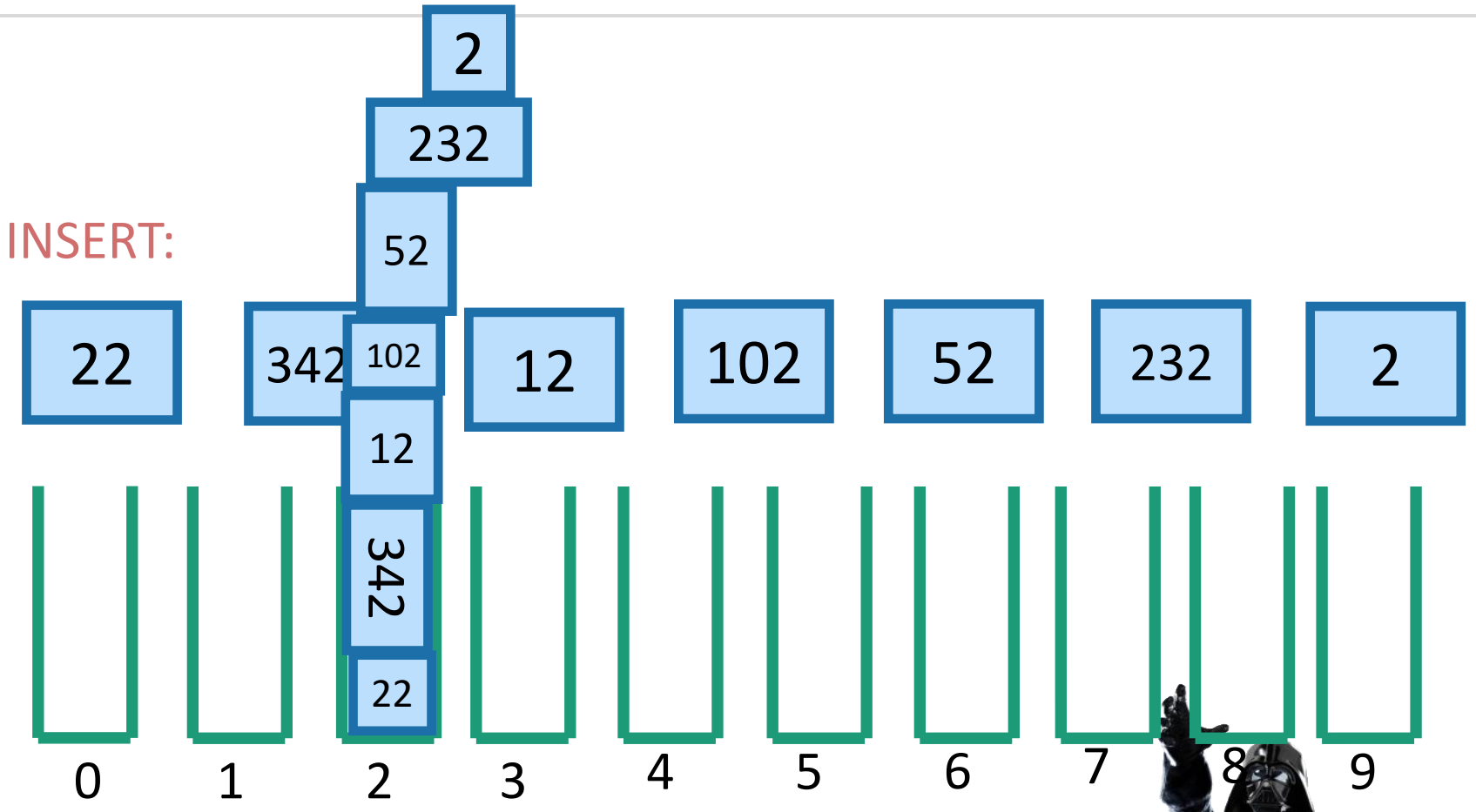
Solution?

Put things in buckets based on one digit

INSERT:



Problem…



Now SEARCH 22

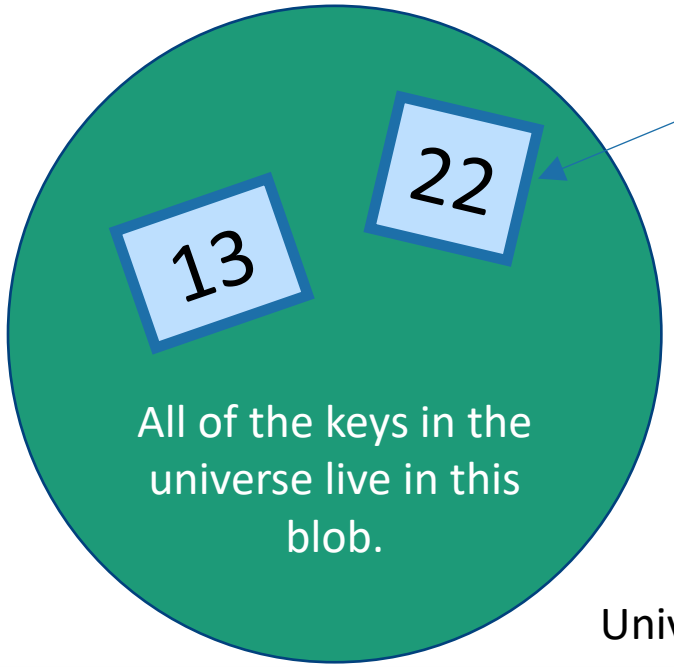
….this hasn't made our lives easier…

Hash tables

- That was an example of a hash table.
  - not a very good one, though.
- We will be **more clever** (and less deterministic) about our bucketing.
- This will result in fast (expected time) INSERT/DELETE/SEARCH.

But first! Terminology.

- U is a *universe* of size M.
  - M is really big.
- But only a few (at most n) elements of U are ever going to show up.
  - M is waaaayyyyyyy bigger than n.
- But we don't know which ones will show up in advance.



Example: U is the set of all strings of at most 280 ascii characters. (128^{280} of them).

The only ones which I care about are those which appear as trending hashtags on twitter. #hashinghashtags

There are way fewer than 128^{280} of these.

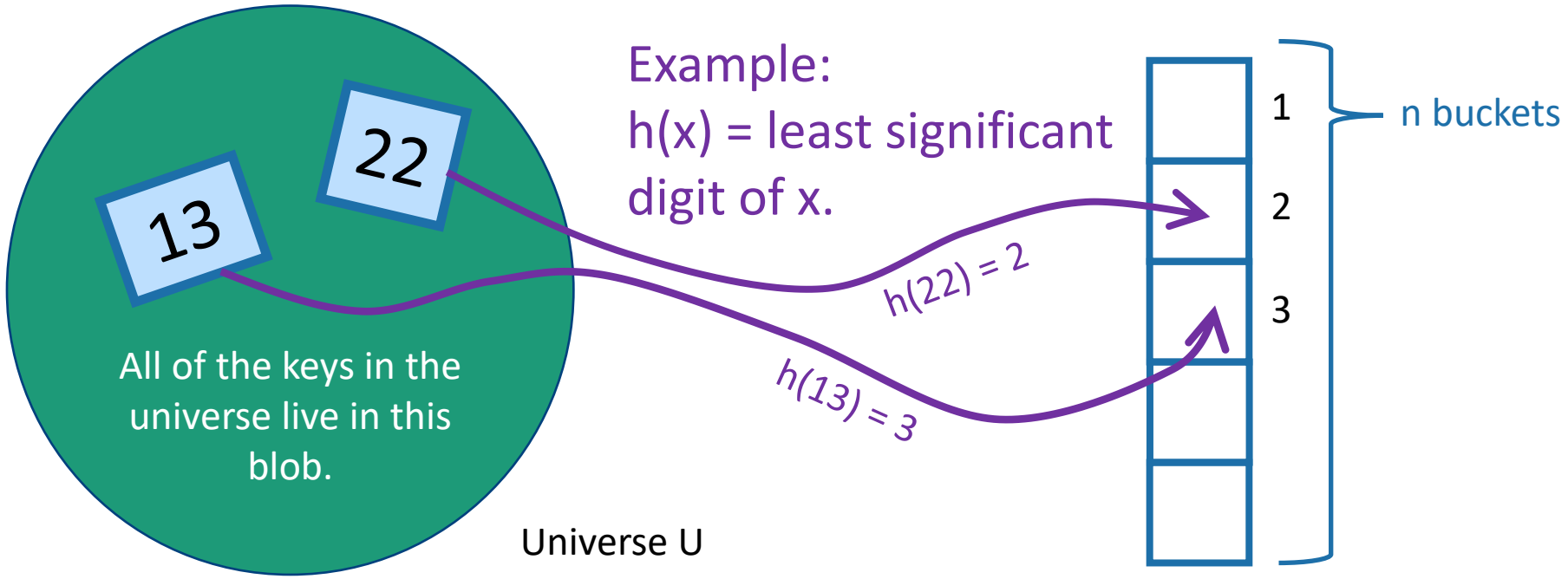
Hash Functions

- A *hash function* $h: U \to \{1, \ldots, n\}$ is a function that maps elements of U to buckets 1, ..., n.

For this lecture, we are assuming that the number of things that show up is the same as the number of buckets, both are n.

This doesn't have to be the case, although we do want:

#buckets = O(#things which show up)



Hash Tables (with chaining)

- Array of n buckets.
- Each bucket stores a linked list.
  - We can insert into a linked list in time O(1)
  - To find something in the linked list takes time O(length(list)).
- A hash function $h: U \to \{1, \ldots, n\}$.
  - For example, h(x) = least significant digit of x.

INSERT:

13 22 43 9

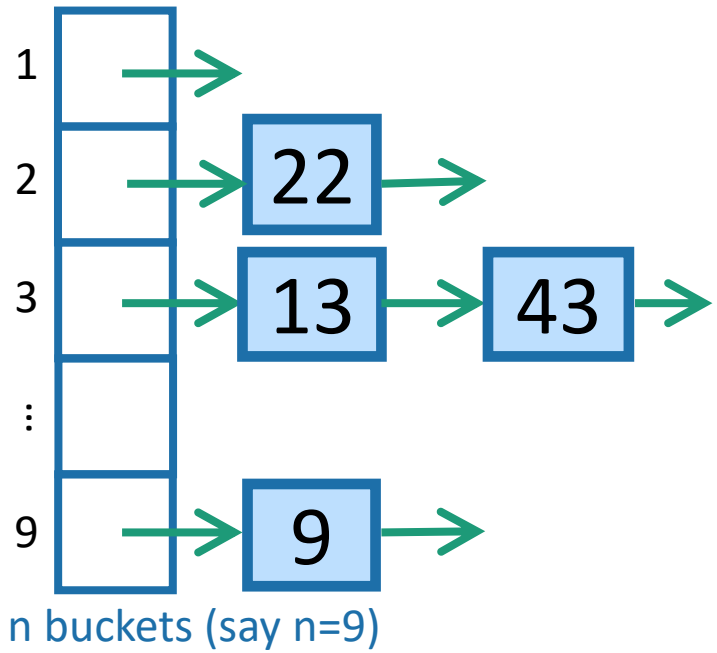
SEARCH 43:

Scan through all the elements in bucket h(43) = 3.

DELETE 43:

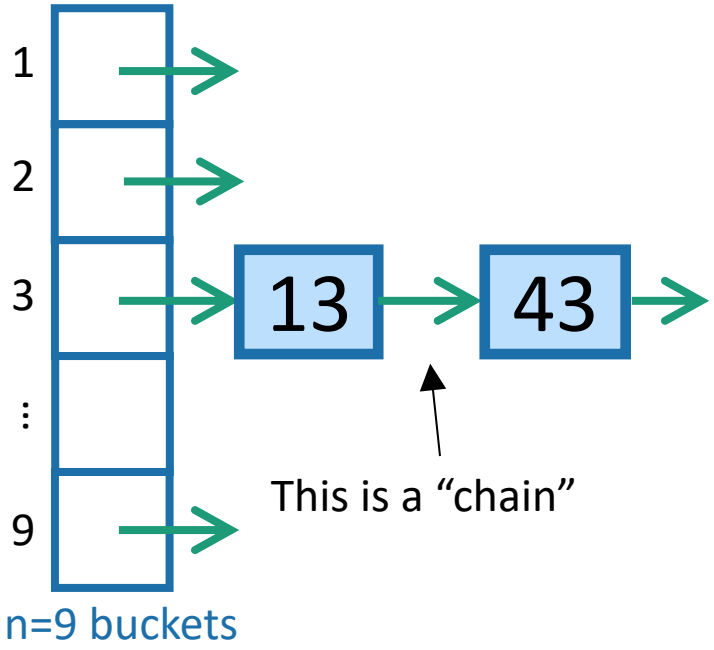
Search for 43 and remove it.

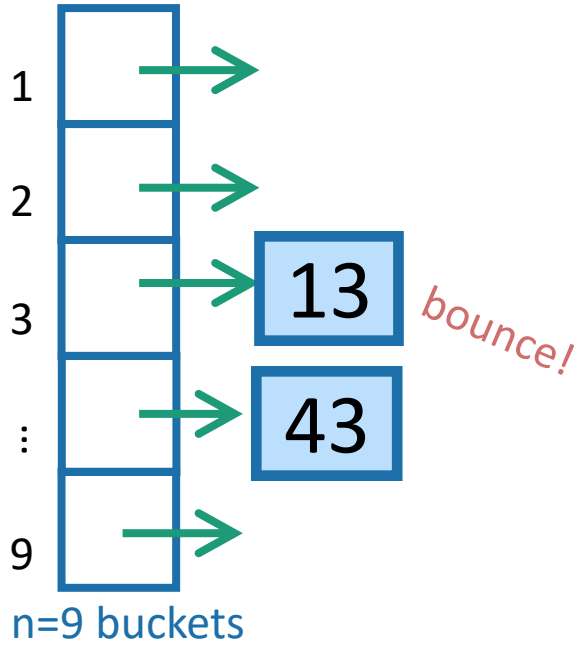
For demonstration purposes only!
This is a terrible hash function! Don't use this!



Aside: Hash tables with open addressing

- The previous slide is about hash tables with chaining.
- There's also something called "open addressing"
- You don't need to know about it for this class.





\end{Aside}

Hash Tables (with chaining)

- Array of n buckets.
- Each bucket stores a linked list.
  - We can insert into a linked list in time O(1)
  - To find something in the linked list takes time O(length(list)).
- A hash function $h: U \to \{1, \ldots, n\}$.
  - For example, h(x) = least significant digit of x.

INSERT:

13 22 43 9

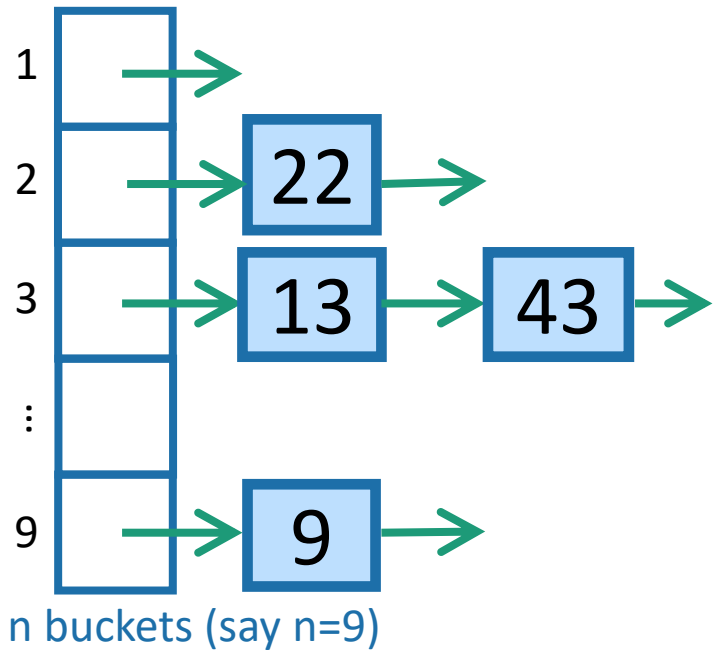
SEARCH 43:

Scan through all the elements in bucket h(43) = 3.

DELETE 43:

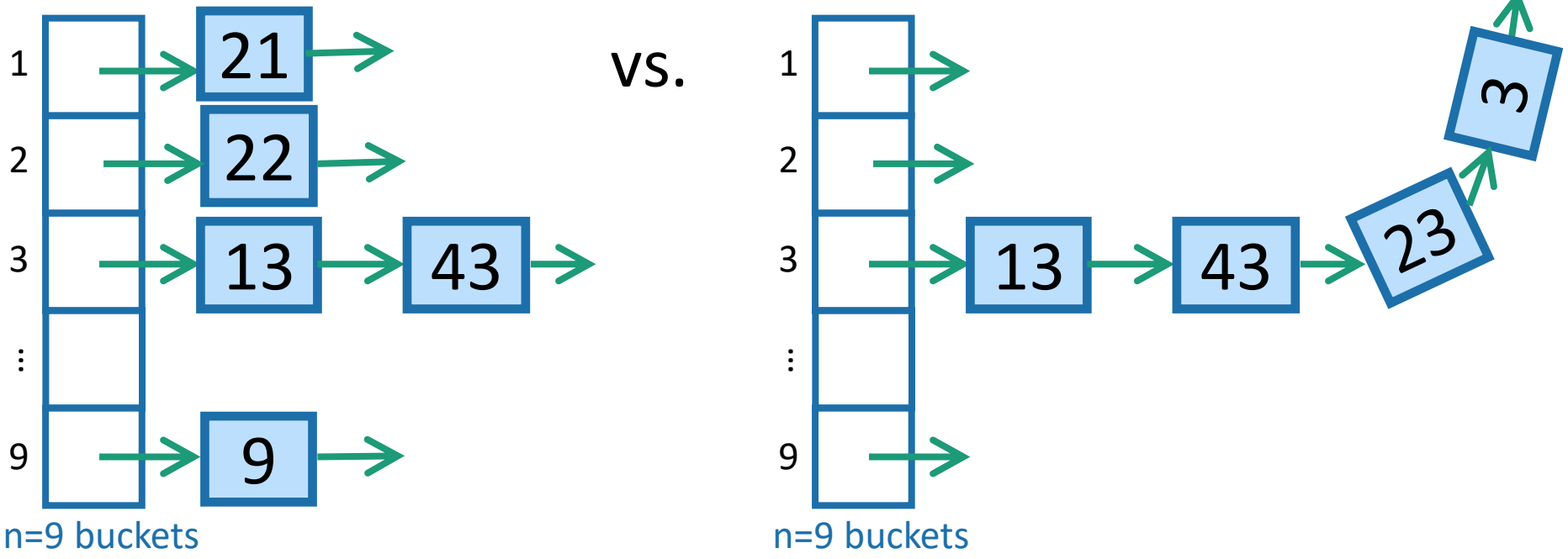
Search for 43 and remove it.

For demonstration purposes only!
This is a terrible hash function! Don't use this!



What we want from a hash table

1. We want there to be not many buckets (say, n).
   – This means we don't use too much space
2. We want the items to be pretty spread-out in the buckets.
   – This means it will be fast to SEARCH/INSERT/DELETE

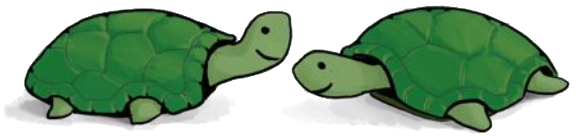


Worst-case analysis

- Goal: Design a function $h: U \to \{1, \ldots, n\}$ so that:
  - No matter what n items of U a bad guy chooses, the buckets will be balanced.
  - Here, balanced means O(1) entries per bucket.

- If we had this, then we'd achieve our dream of O(1) INSERT/DELETE/SEARCH

Can you come up with such a function?



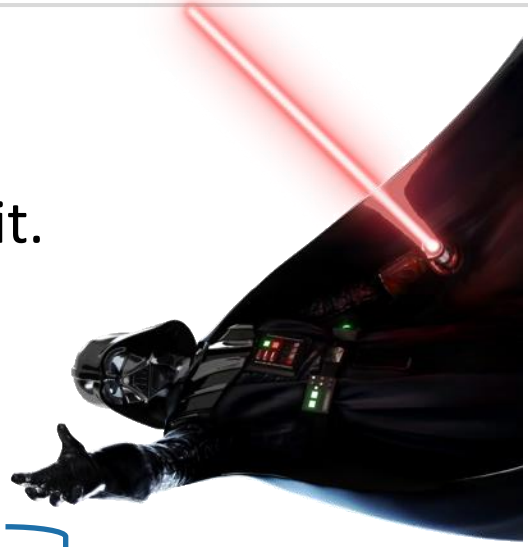
Think-Share Terrapins

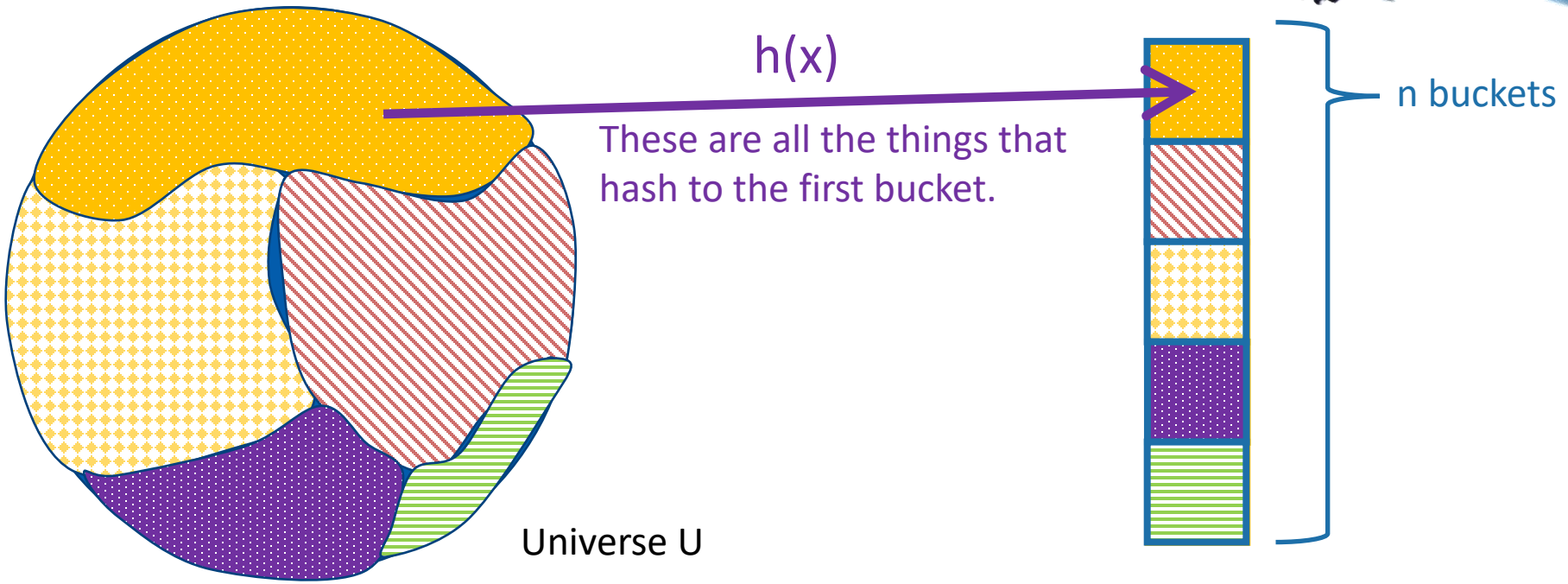
This is impossible!

No deterministic hash function can defeat worst-case input!

We really can't beat the bad guy here.

- The universe U has M items
- They get hashed into n buckets
- At least one bucket has at least M/n items hashed to it.
- M is waayyyy bigger than n, so M/n is bigger than n.
- **Bad guy chooses n of the items that landed in this very full bucket.**





Solution:

Randomness

The game

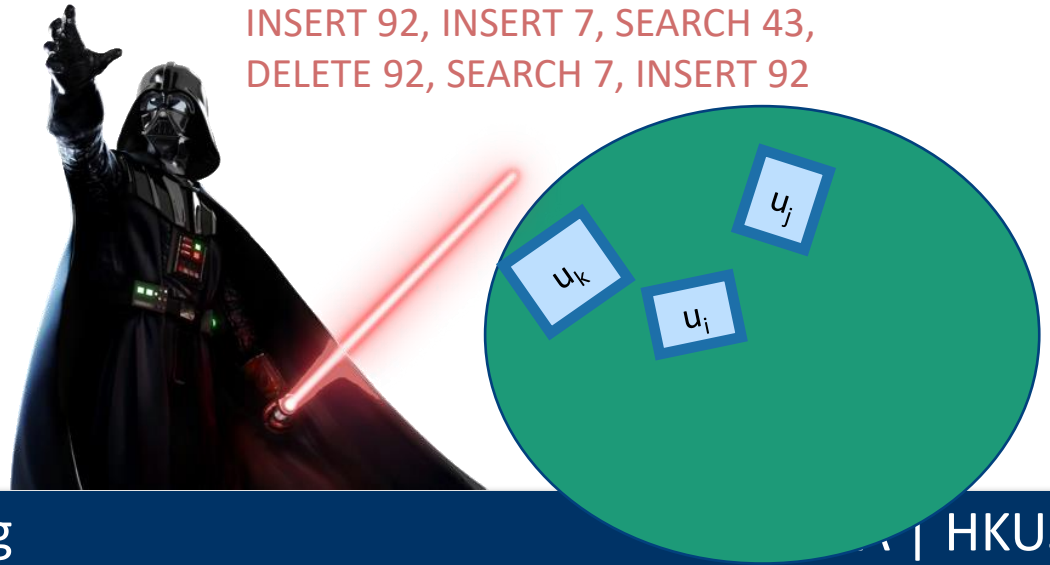
What does **random** mean here? Uniformly random?

Plucky the pedantic penguin

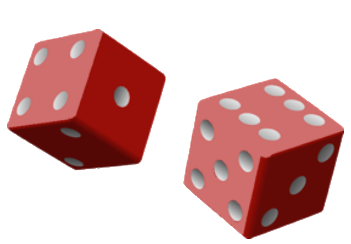
1. An adversary chooses any n items $u_1, u_2, \ldots, u_n \in U$, and any sequence of INSERT/DELETE/SEARCH operations on those items.

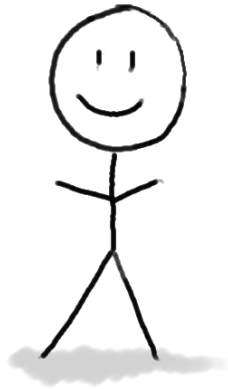
13	22	43	92	7

INSERT 13, INSERT 22, INSERT 43,
INSERT 92, INSERT 7, SEARCH 43,
DELETE 92, SEARCH 7, INSERT 92

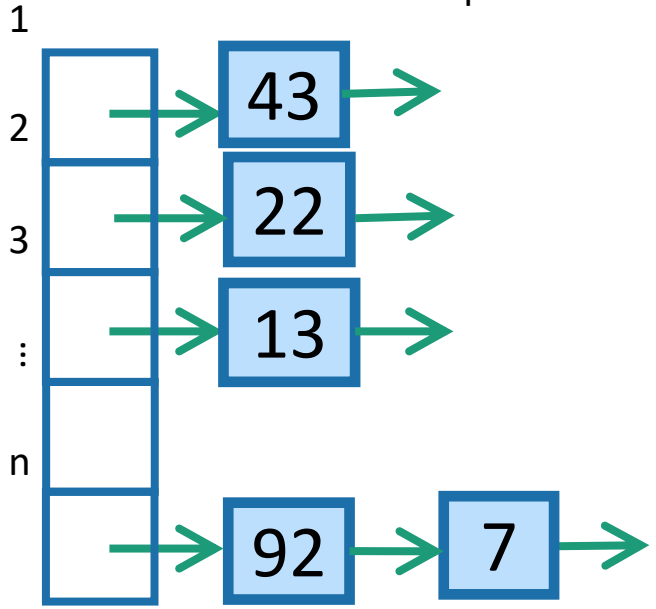


2. You, the algorithm, choose a **random** hash function $h: U \to \{1, \ldots, n\}$.



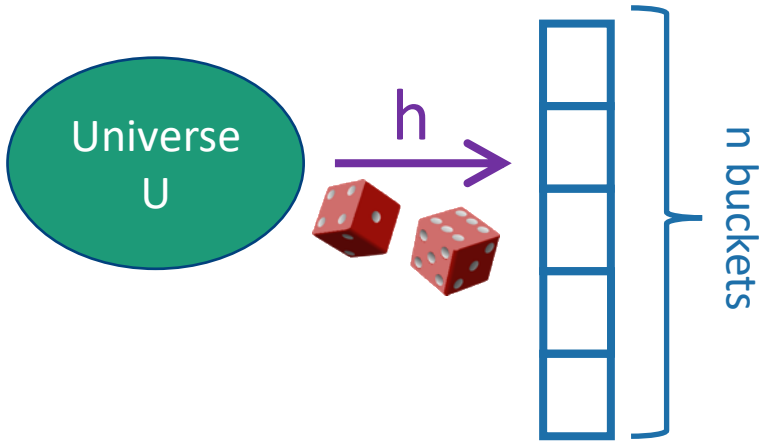


3. **HASH IT OUT** #hashpuns



Example of a random hash function

- Say that $h: U \to \{1, \ldots, n\}$ is a uniformly random function.
  - That means that **h(1)** is a **uniformly random** number between 1 and n.
  - **h(2)** is also a **uniformly random** number between 1 and n, independent of h(1).
  - **h(3)** is also a **uniformly random** number between 1 and n, independent of h(1), h(2).
  - …
  - **h(M)** is also a **uniformly random** number between 1 and n, independent of h(1), h(2), …, h(M-1).



Randomness helps

Intuitively: The bad guy can't foil a hash function that he doesn't yet know.

Lucky the Lackadaisical Lemur

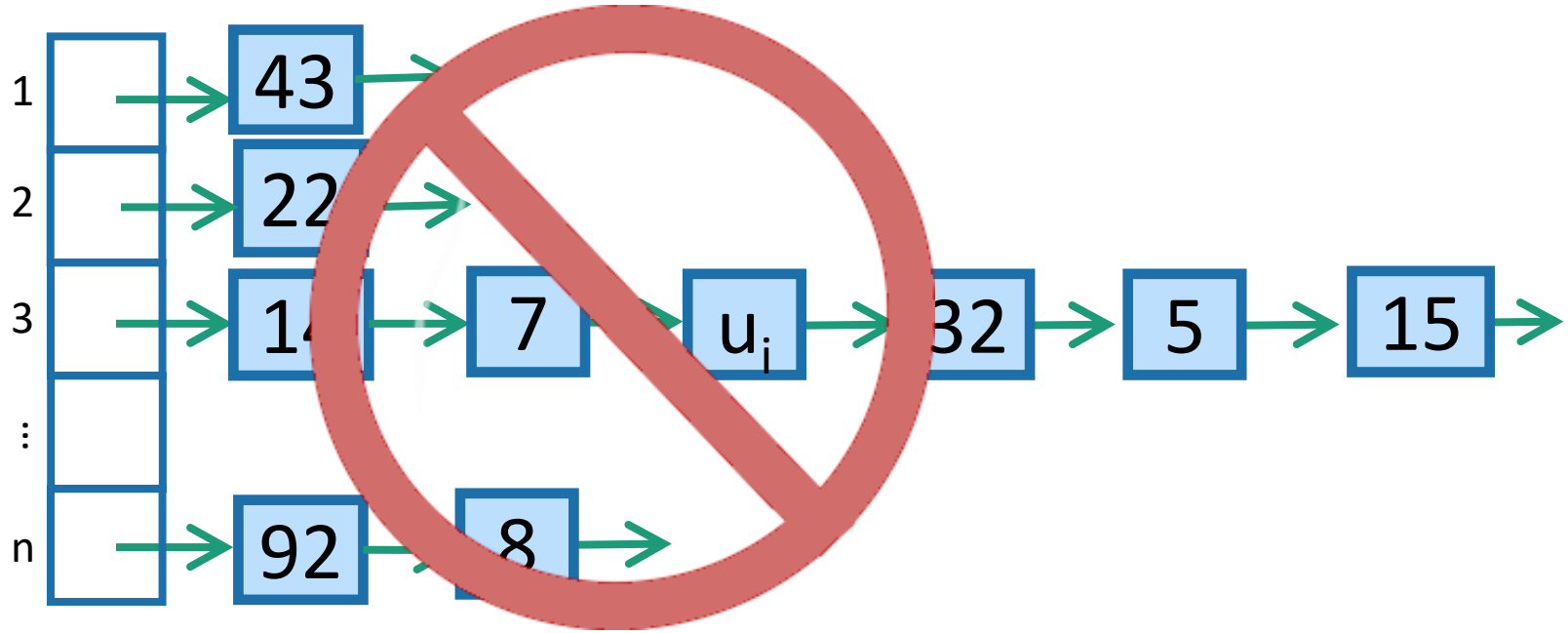
Plucky the Pedantic Penguin

Why not? What if there's some strategy that foils a random function with high probability?

We'll need to do some analysis…

What do we want?

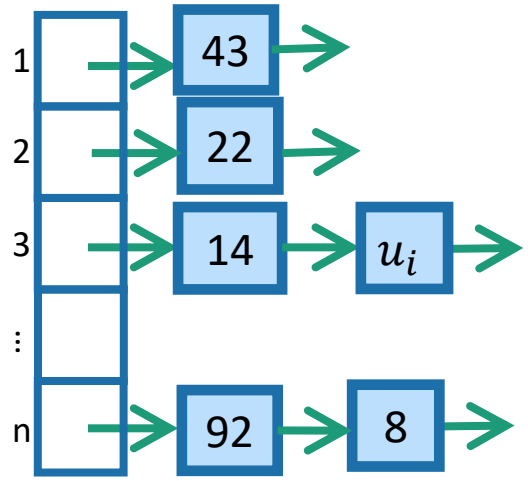
It's **bad** if lots of items land in u_i's bucket.
So we want **not that**.



More precisely

- We want:
  - For all ways a bad guy could choose $u_1, u_2, \ldots, u_n$, to put into the hash table, and for all $i \in \{1, \ldots, n\}$,
    E[number of items in u_i's bucket] ≤ 2.
- If that were the case:
  - For each INSERT/DELETE/SEARCH operation involving u_i,
    E[time of operation] = O(1)

We could replace "2" here with any constant; it would still be good. But "2" will be convenient.



Note that the expected size of u_i's linked list is not the same as the expected {maximum size of linked lists}. What is the latter?

This is what we wanted at the beginning of lecture!

So we want:

- For all i=1, …, n,

  E[number of items in u_i's bucket] ≤ 2.

Aside

- For all i=1, …, n,

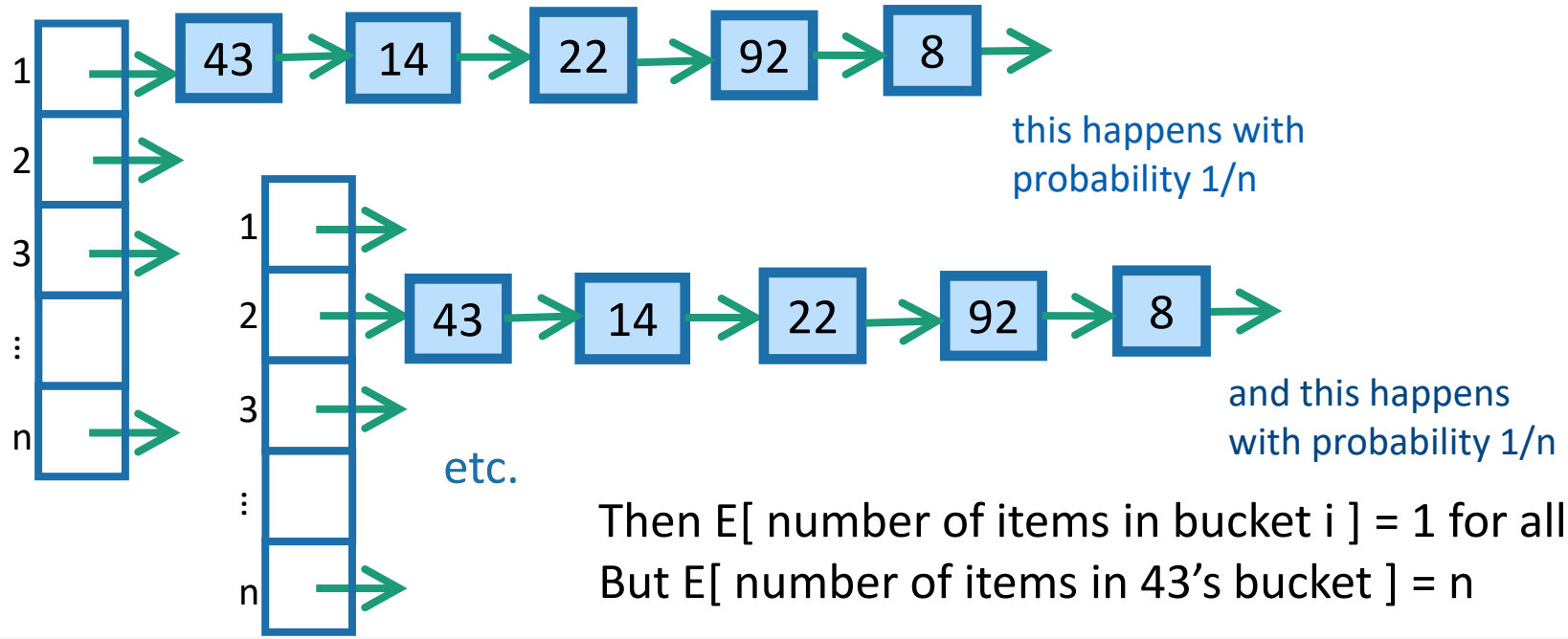
$$E[\text{ number of items in } u_i\text{'s bucket }] \leq 2.$$

VS

- For all i=1,…,n:

$$E[\text{ number of items in bucket i }] \leq 2$$

Suppose that:



Then E[number of items in bucket i] = 1 for all i.
But E[number of items in 43's bucket] = n

This distinction came up on your pre-lecture exercise!

香港科技大学(广州)
THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY (GUANGZHOU)

- Solution to pre-lecture exercise:
  - E[number of items in bucket 1] = n/6
  - E[number of items that land in the same bucket as item 1] = n

So we want:

- For all i=1, …, n,

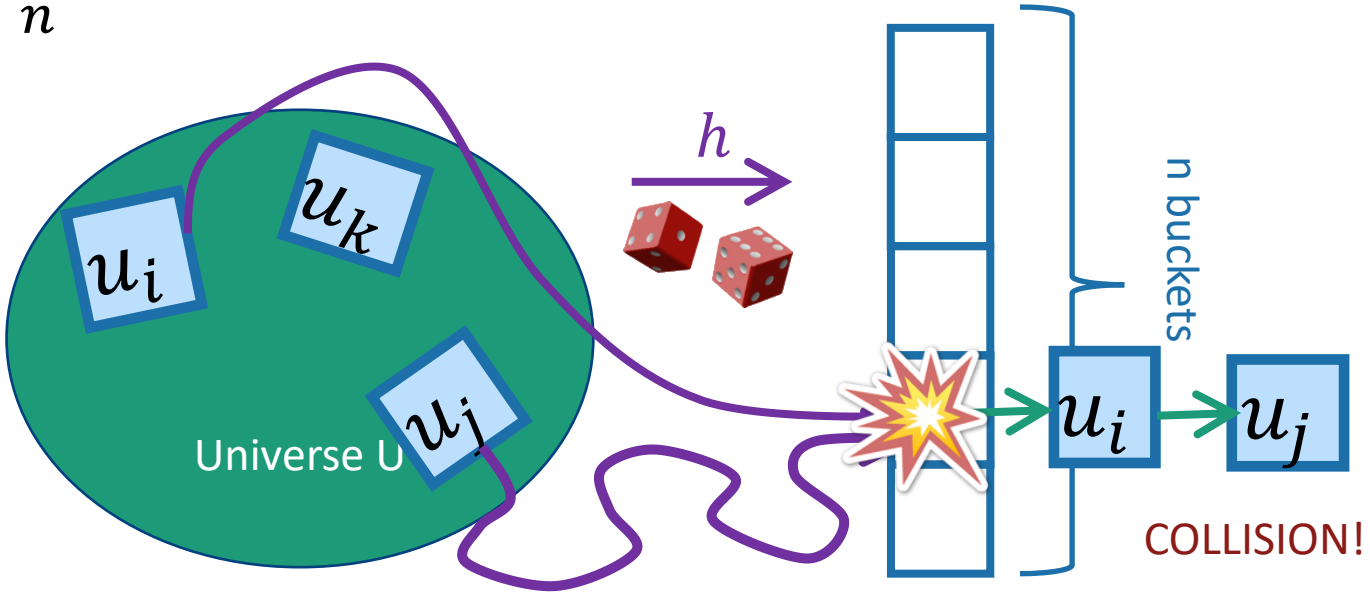
  E[number of items in u_i's bucket] ≤ 2.

Expected number of items in u_i's bucket?

h is uniformly random

- $E[\ ] = \sum_{j=1}^{n} P\{h(u_i) = h(u_j)\}$
- $= 1 + \sum_{j \neq i} P\{h(u_i) = h(u_j)\}$
- $= 1 + \sum_{j \neq i} 1/n$
- $= 1 + \frac{n-1}{n} \leq 2.$

That's what we wanted!



A uniformly random hash function leads to balanced buckets

- We just showed:
  - For all ways a bad guy could choose $u_1, u_2, \ldots, u_n$, to put into the hash table, and for all $i \in \{1, \ldots, n\}$,

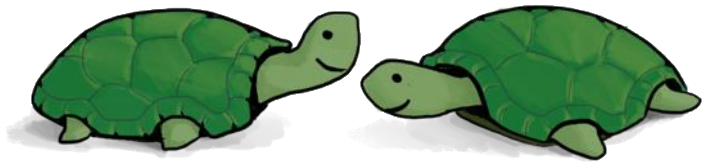
    E[number of items in u_i's bucket] ≤ 2.
- Which implies:
  - No matter what sequence of operations and items the bad guy chooses,

    E[time of INSERT/DELETE/SEARCH] = O(1)
- So, our solution is:

  Pick a uniformly random hash function?

What's wrong with this plan?

- Hint: How would you implement (and store) a uniformly random function $h: U \to \{1, \ldots, n\}$?



Think-Share Terrapins

- If h is a uniformly random function:
  - That means that **h(1)** is a **uniformly random** number between 1 and n.
  - **h(2)** is also a **uniformly random** number between 1 and n, independent of h(1).
  - **h(3)** is also a **uniformly random** number between 1 and n, independent of h(1), h(2).
  - …
  - **h(n)** is also a **uniformly random** number between 1 and n, independent of h(1), h(2), …, h(n-1).

A uniformly random hash function is not a good idea.

- In order to store/evaluate a uniformly random hash function, we'd use a lookup table:

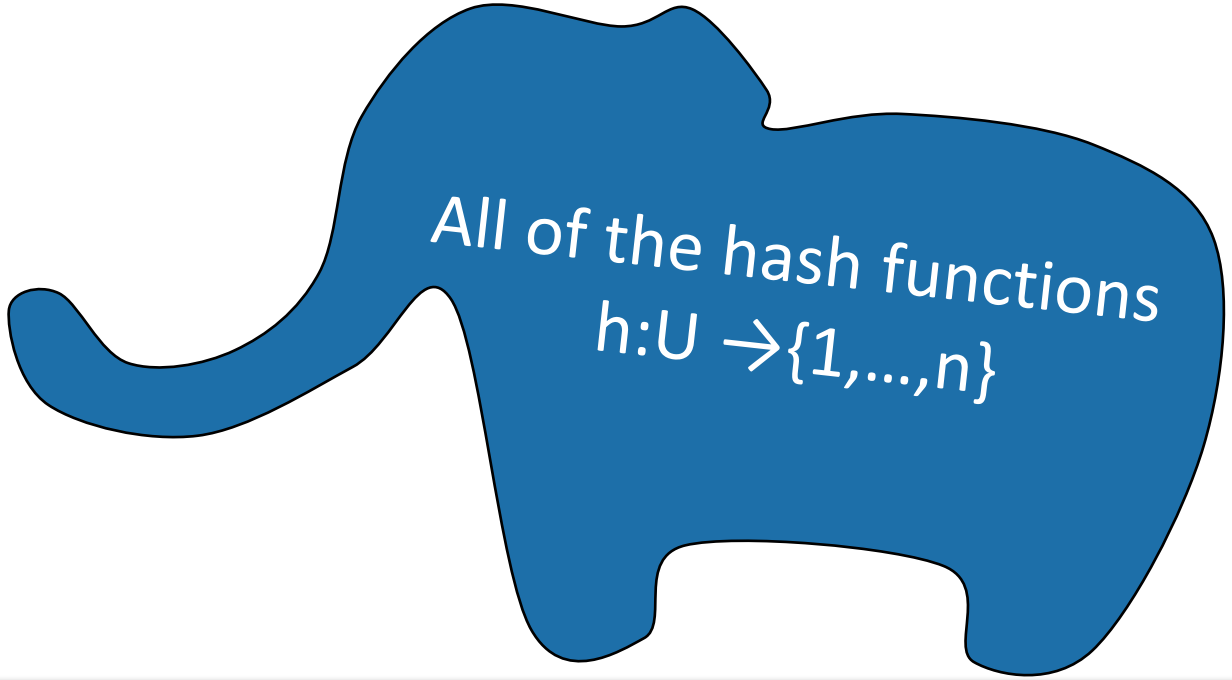
All of the M things in the universe

x	h(x)
AAAAAA	1
AAAAAB	5
AAAAAC	3
AAAAAD	3
…	
ZZZZZY	7
ZZZZZZ	3

- Each value of h(x) takes log(n) bits to store.
- Storing M such values requires Mlog(n) bits.
- In contrast, direct addressing (initializing a bucket for every item in the universe) requires only M bits.

Another way to say this

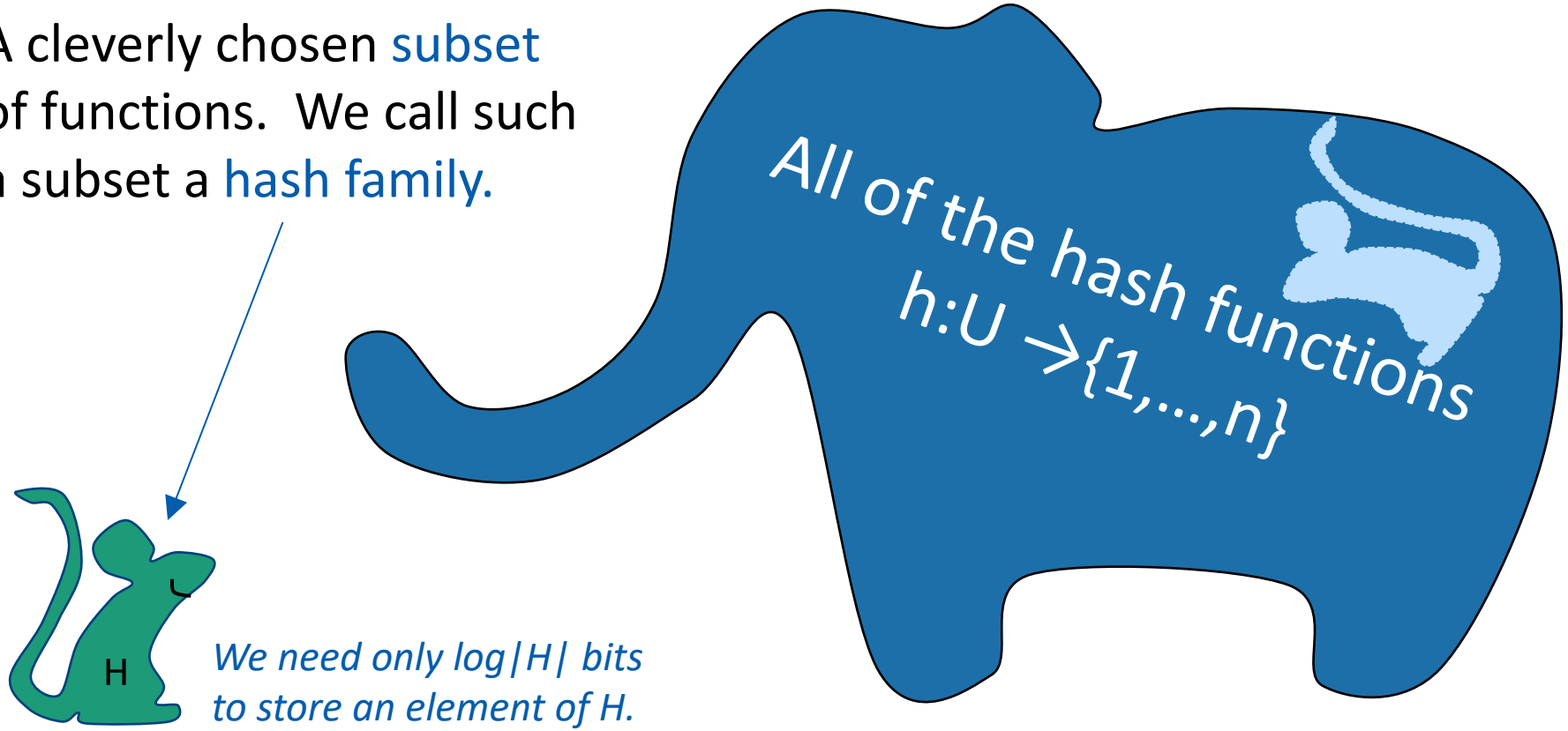
- There are lots of hash functions.
- There are n^M of them.
- Writing down a random one of them takes $\log(n^M)$ bits, which is $M\log(n)$.



Solution

- Pick from a smaller set of functions.

A cleverly chosen subset of functions. We call such a subset a hash family.



We need only log|H| bits to store an element of H.

Outline

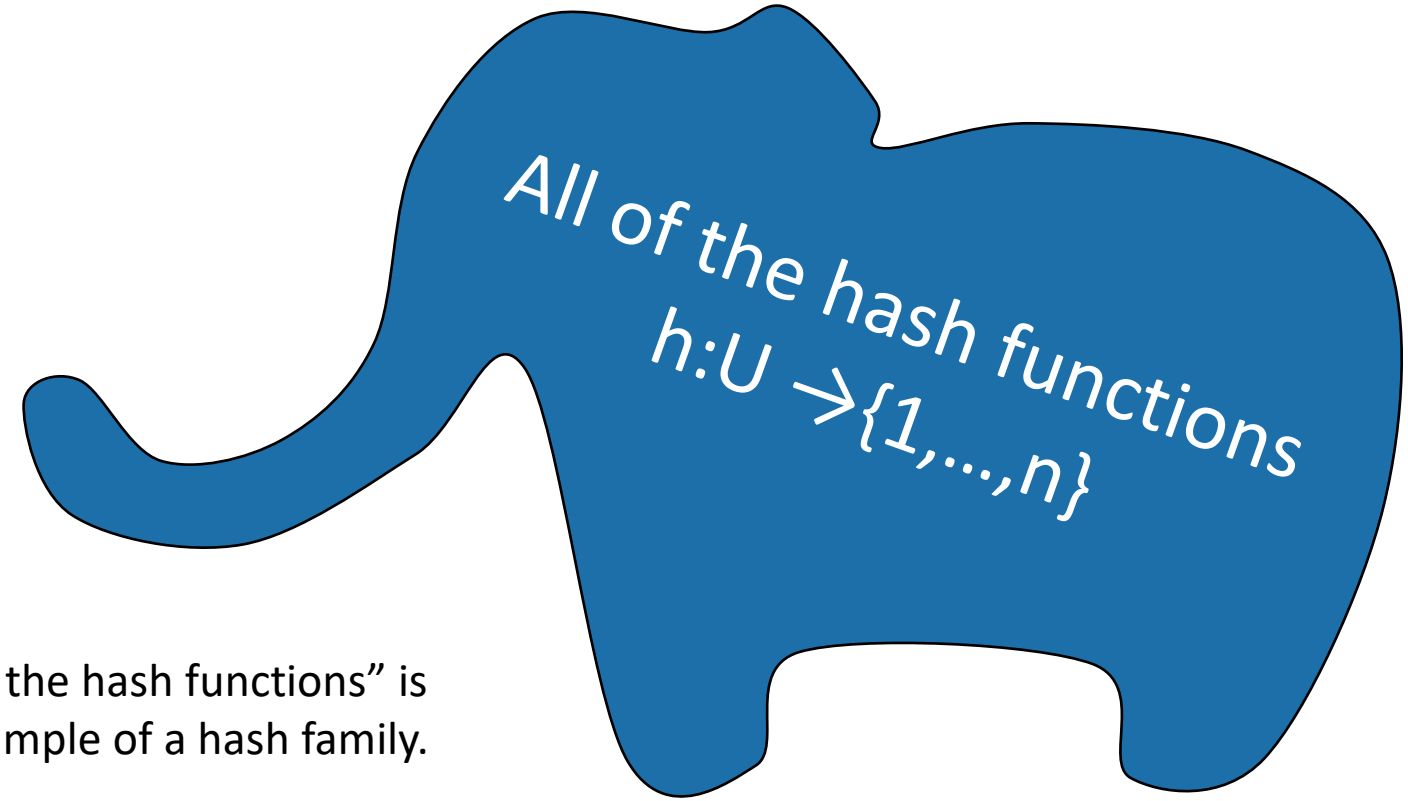
- **Hash tables** are another sort of data structure that allows fast INSERT/DELETE/SEARCH.
  - like self-balancing binary trees
  - The difference is we can get better performance in expectation by using randomness.

- **Hash families** are the magic behind hash tables.

- **Universal hash families** are even more magic.

Hash families

- A hash family is a collection of hash functions.



"All of the hash functions" is an example of a hash family.

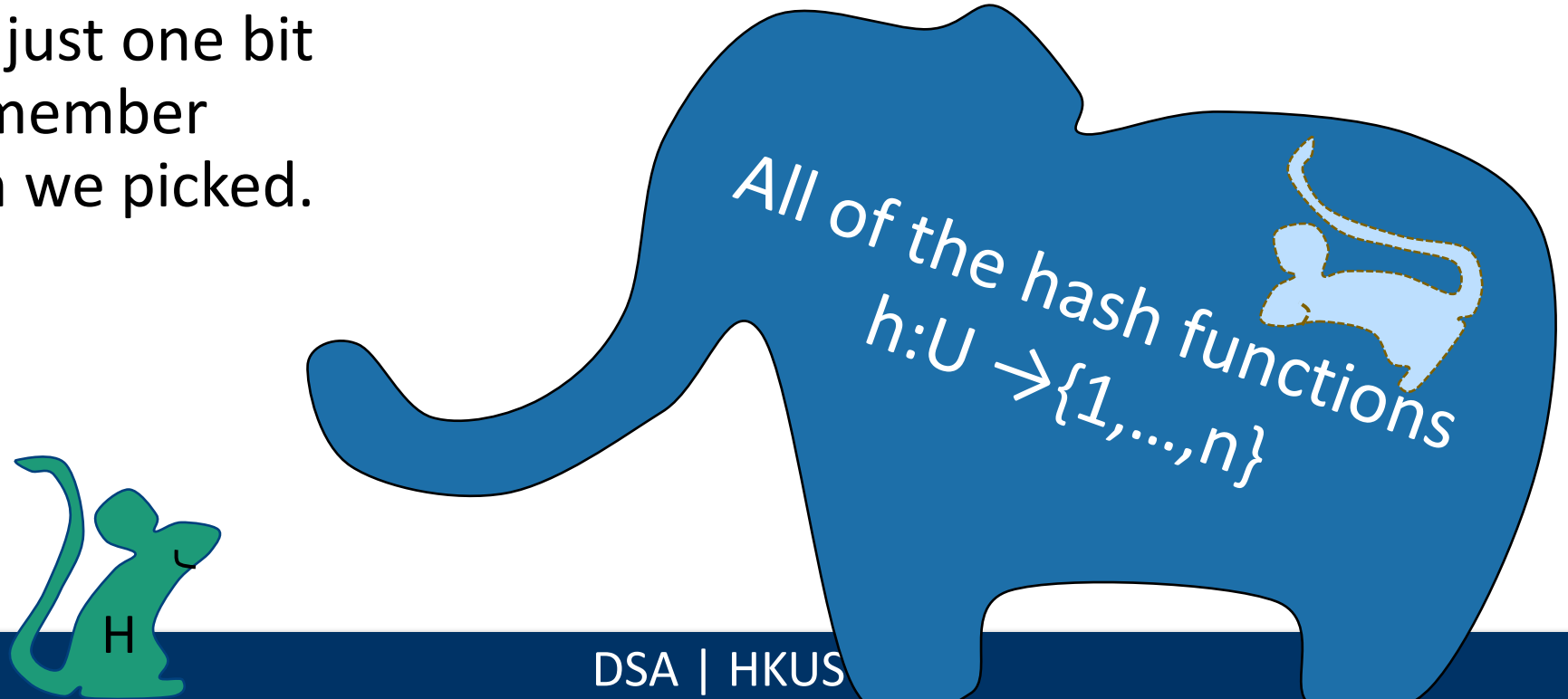
Example: a smaller hash family

香港科技大学(广州)
THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY (GUANGZHOU)

This is still a terrible idea!
Don't use this example!
For pedagogical purposes only!

- H = { function which returns the least sig. digit,
  function which returns the most sig. digit }
- Pick h in H at random.

- Store just one bit to remember which we picked.



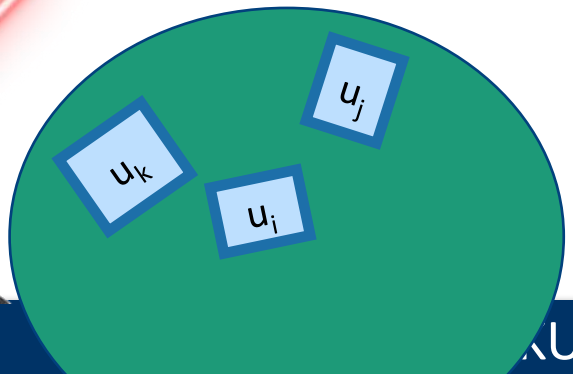
The game

h_0 = Most_significant_digit
h_1 = Least_significant_digit
H = {h_0, h_1}

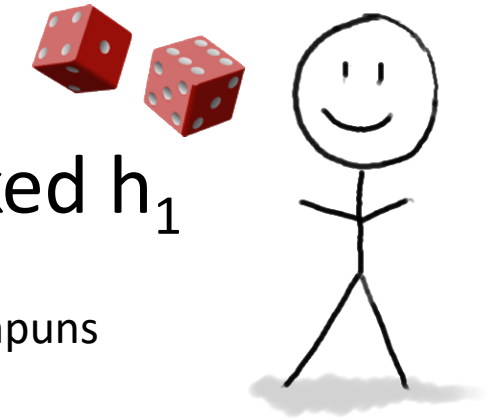
1. An adversary (who knows H) chooses any n items $u_1, u_2, \ldots, u_n \in U$, and any sequence of INSERT/DELETE/SEARCH operations on those items.

19	22	42	92	0

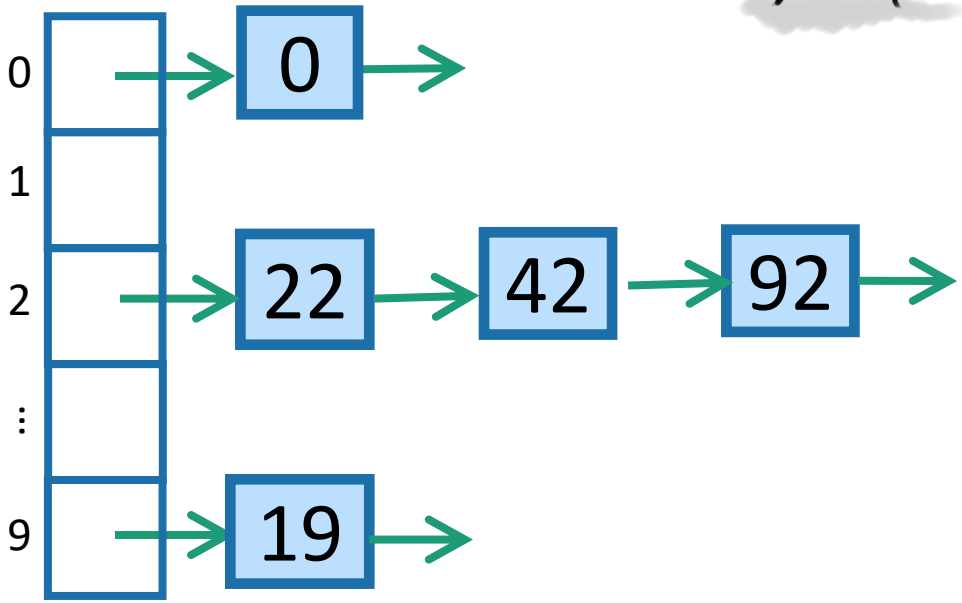
INSERT 19, INSERT 22, INSERT 42,
INSERT 92, INSERT 0, SEARCH 42,
DELETE 92, SEARCH 0, INSERT 92



2. You, the algorithm, chooses a **random** hash function $h: U \to \{0, \ldots, 9\}$. Choose it randomly from H.



3. **HASH IT OUT** #hashpuns



This is not a very good hash family

- H = { function which returns least sig. digit,

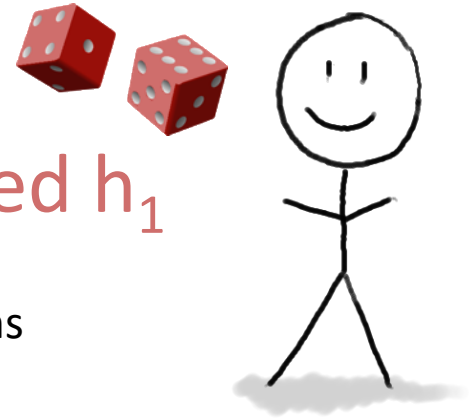
  function which returns most sig. digit }
- On the previous slide, the adversary could have been a lot more adversarial…

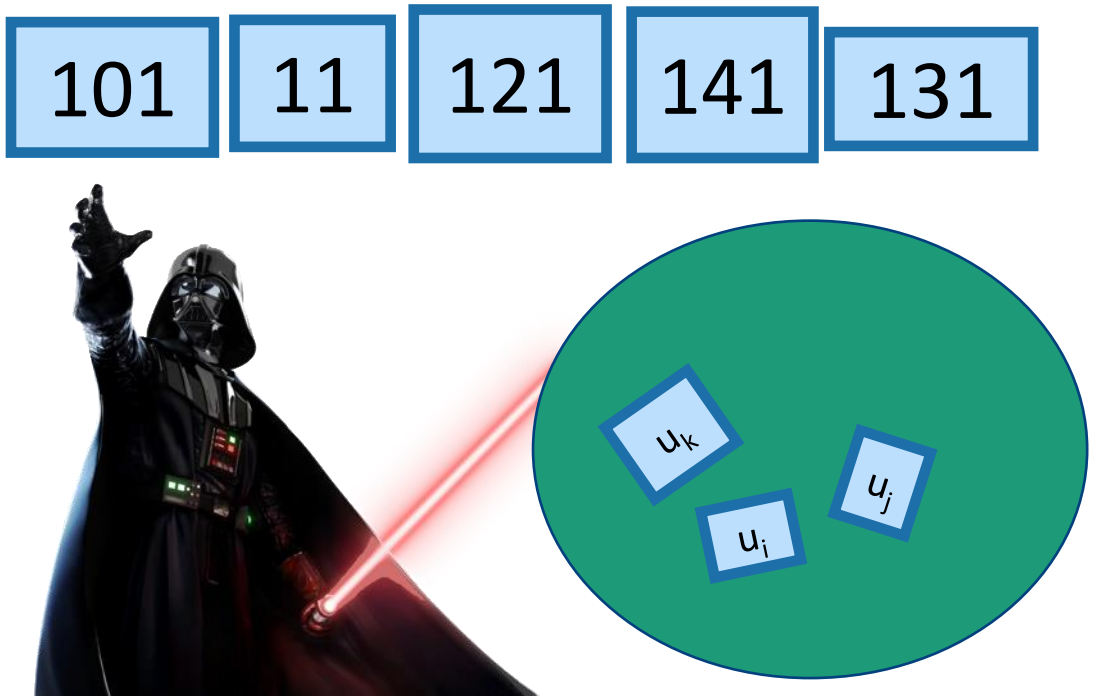
The game

h_0 = Most_significant_digit
h_1 = Least_significant_digit
H = {h_0, h_1}

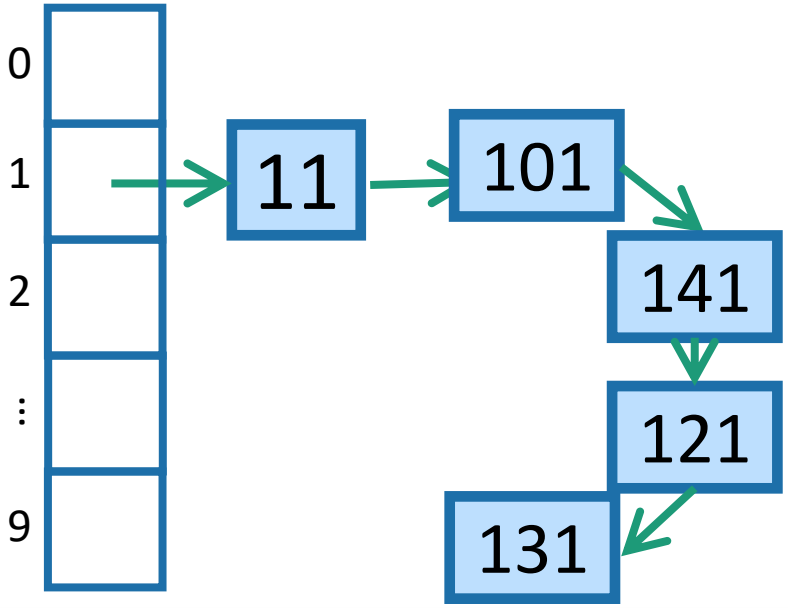
1. An adversary (who knows H) chooses any n items $u_1, u_2, \ldots, u_n \in U$, and any sequence of INSERT/DELETE/SEARCH operations on those items.

2. You, the algorithm, chooses a **random** hash function $h: U \to \{0, \ldots, 9\}$. Choose it randomly from H.



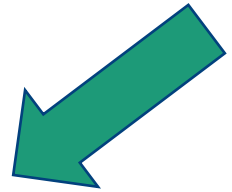


3. **HASH IT OUT** #hashpuns



Outline

- **Hash tables** are another sort of data structure that allows fast INSERT/DELETE/SEARCH.
  - like self-balancing binary trees
  - The difference is we can get better performance in expectation by using randomness.
- **Hash families** are the magic behind hash tables.
- **Universal hash families** are even more magic.



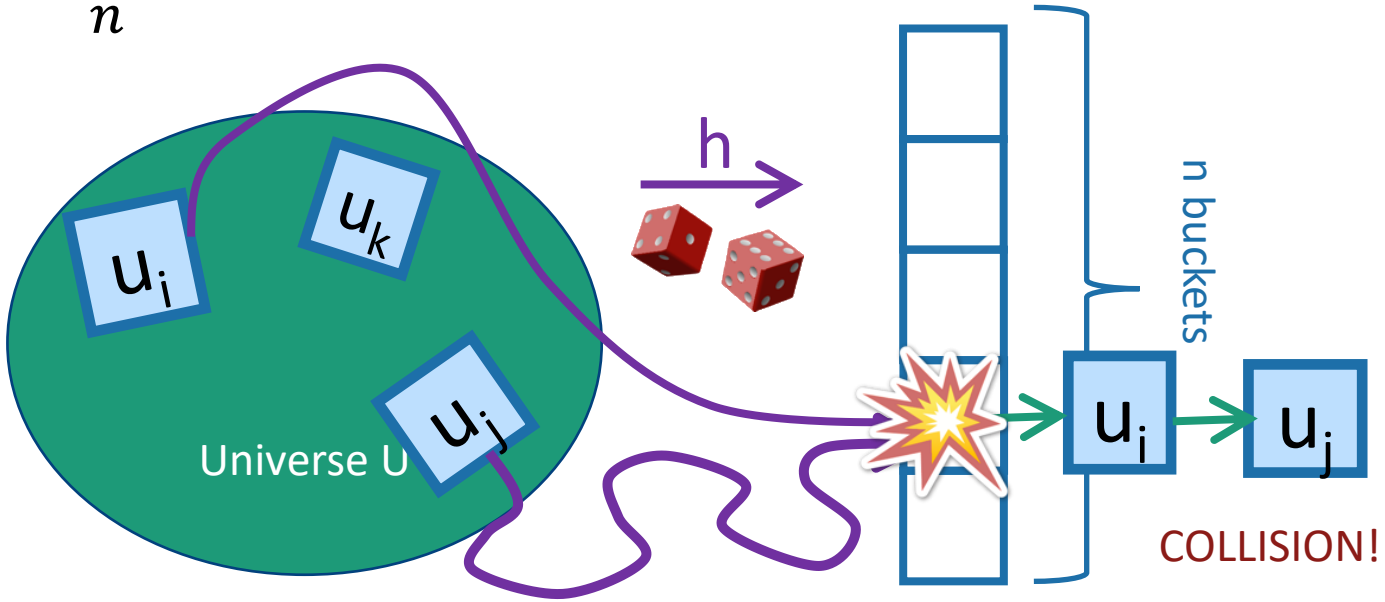
How to pick the hash family?

- Definitely not like in that example.
- Let’s go back to that computation from earlier….

Expected number of items in u_i's bucket?

- $E[\ ] = \sum_{j=1}^{n} P\{ h(u_i) = h(u_j)\}$
- $= 1 + \sum_{j \neq i} P\{ h(u_i) = h(u_j)\}$
- $= 1 + \sum_{j \neq i} 1/n$
- $= 1 + \frac{n-1}{n} \leq 2.$

All that we needed was that this is 1/n



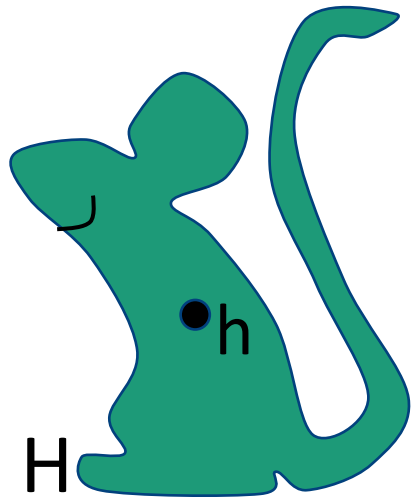
Strategy

- Pick a small hash family H, so that when I choose h randomly from H,

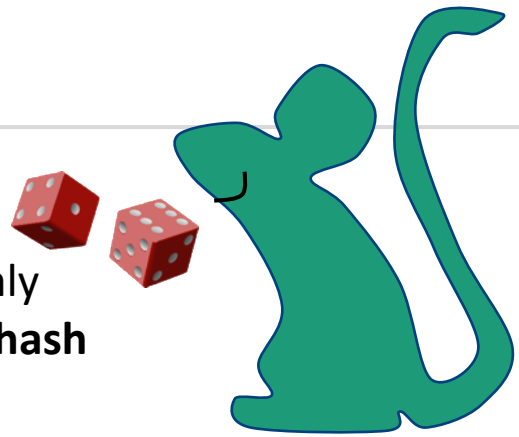
$$\text{for all } u_i, u_j \in U \quad \text{with } u_i \neq u_j,$$
$$P_{h \in H}\{ h(u_i) = h(u_j)\} \leq \frac{1}{n}$$

In English: fix any two elements of U. The probability that they collide under a random h in H is small.

- A hash family H that satisfies this is called a **universal hash family.**

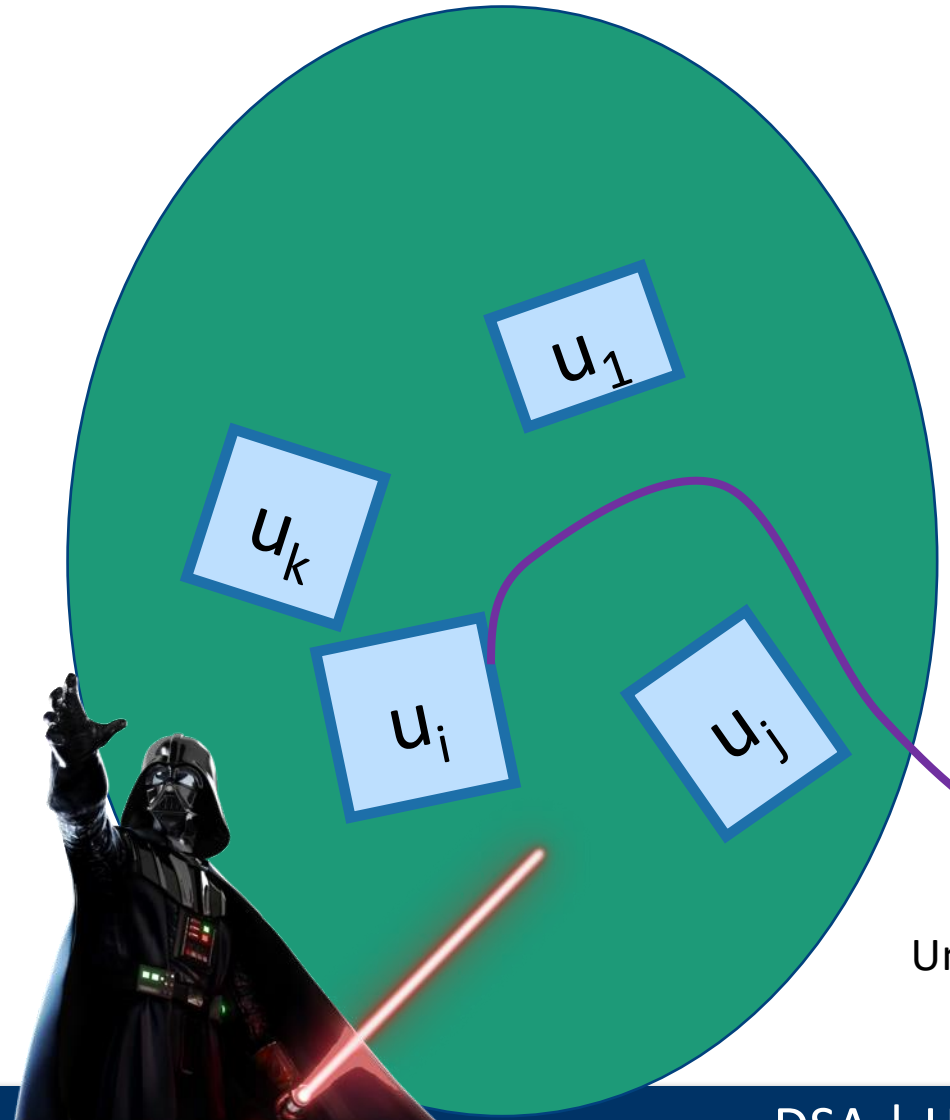


So the whole scheme will be



Choose h randomly from a **universal hash family** H

We can store h using log|H| bits.



Probably these buckets will be pretty balanced.

Universal hash family

- H is a ***universal hash family*** if, when h is chosen uniformly at random from H,

$$\text{for all } u_i, u_j \in U \quad \text{with } u_i \neq u_j,$$

$$P_{h \in H}\{ h(u_i) = h(u_j)\} \leq \frac{1}{n}$$

Example

- Pick a small hash family H, so that when I choose h randomly from H,

$$\text{for all } u_i, u_j \in U \quad \text{with } u_i \neq u_j,$$
$$P_{h \in H}\{ h(u_i) = h(u_j)\} \leq \frac{1}{n}$$

- H = the set of all functions $h: U \to \{1, \ldots, n\}$
  - We saw this earlier – it corresponds to picking a uniformly random hash function.
  - Unfortunately, this H is really really large.

Non-example

- Pick a small hash family H, so that when I choose h randomly from H,

$$\text{for all } u_i, u_j \in U \quad \text{with } u_i \neq u_j,$$
$$P_{h \in H}\{ h(u_i) = h(u_j)\} \leq \frac{1}{n}$$

- h_0 = Most_significant_digit
- h_1 = Least_significant_digit
- H = {h_0, h_1}

NOT a universal hash family:

$$P_{h \in H}\{h(101) = h(111)\} = 1 > \frac{1}{10}$$

A small universal hash family??

- Here's one:
  - Pick a prime $p \geq M$.
  - Define

$$f_{a,b}(x) = ax + b \quad mod\ p$$

$$h_{a,b}(x) = f_{a,b}(x) \quad mod\ n$$

  - Define:

$$H = \{\, h_{a,b}(x) \ : \ a \in \{1, \dots, p-1\}, b \in \{0, \dots, p-1\} \,\}$$

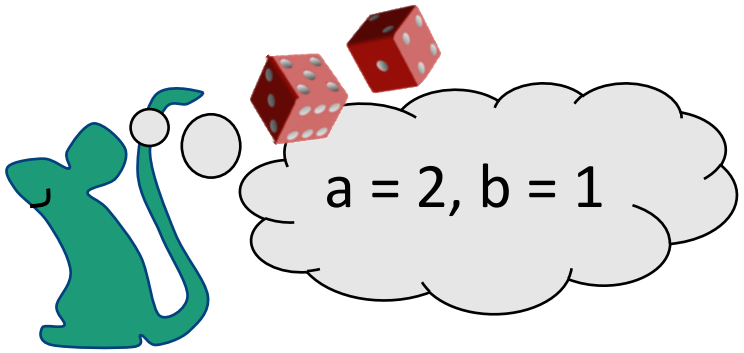
How do you pick the prime number p that's not too larger than M?

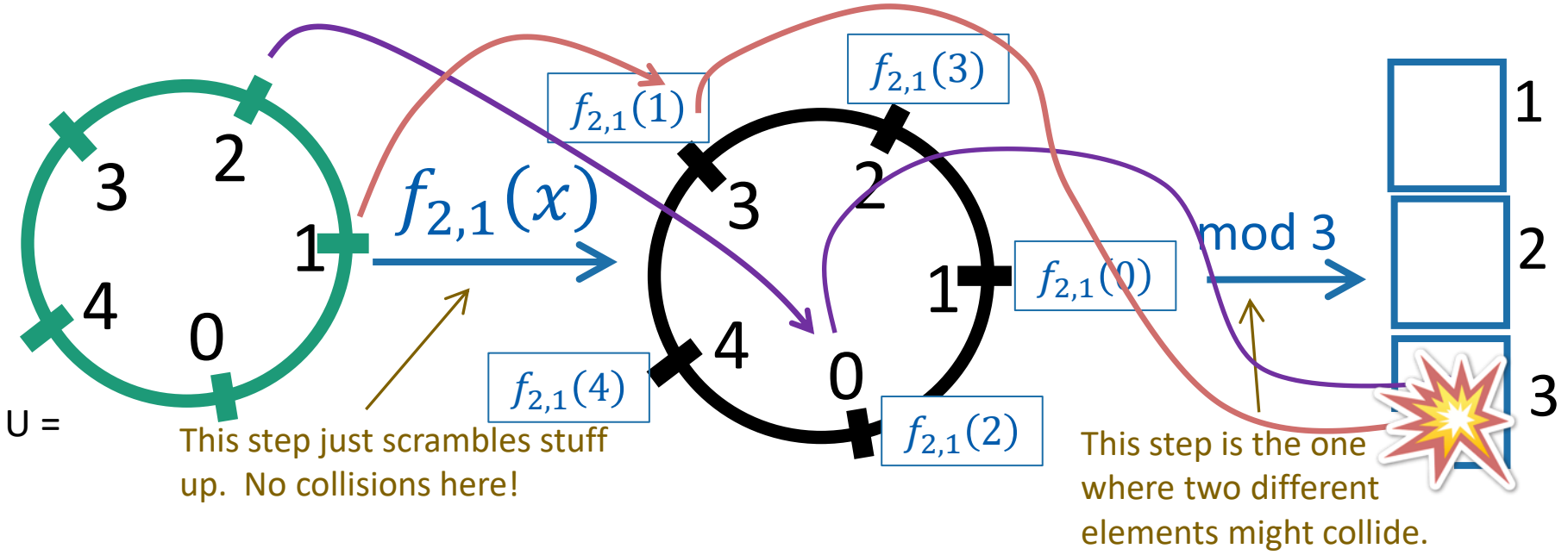
- Claim:

  H is a universal hash family.

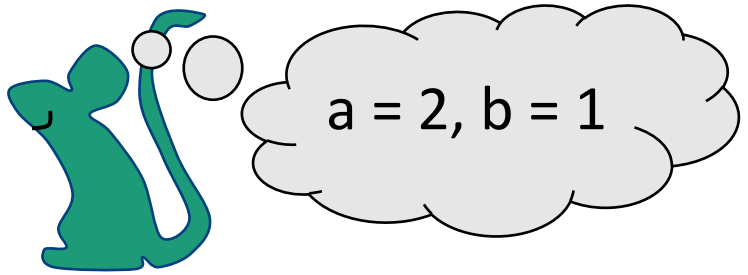
Say what?

- Example: M = p = 5, n = 3
- To draw h from H:
  - Pick a random a in {1,…,4}, b in {0,…,4}
- As per the definition:
  - $f_{2,1}(x) = 2x + 1 \quad mod\ 5$
  - $h_{2,1}(x) = f_{2,1}(x) \quad mod\ 3$





h takes O(log M) bits to store



- Just need to store two numbers:
  - a is in {1,…,p-1}
  - b is in {0,…,p-1}
  - So about 2log(p) bits
  - By our choice of p (close to M), that's O(log(M)) bits.
- Also, given a and b, h is fast to evaluate!
  - It takes time O(1) to compute h(x).
- Compare: direct addressing was M bits!
  - Twitter example: 2log(M) =2*280 log(128) = **3920** vs M = $\mathbf{128^{280}}$

Why does this work?

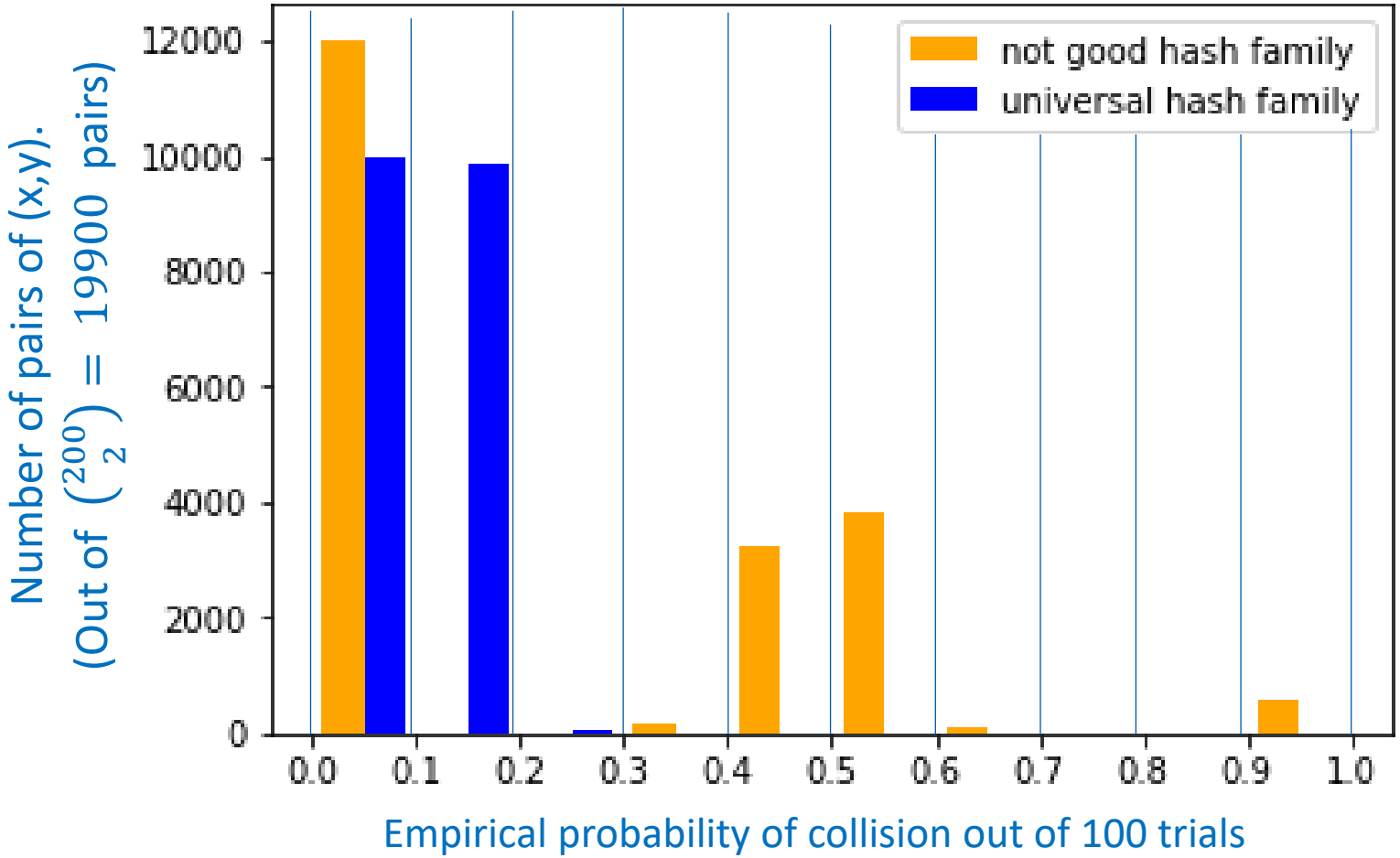
- This is actually a little complicated.
  - Find the proof if you are curious.
  - You are NOT RESPONSIBLE for the proof in this class.
  - But you should know that a universal hash family of size $O(M^2)$ exists.

Try to prove that this is a universal hash family!

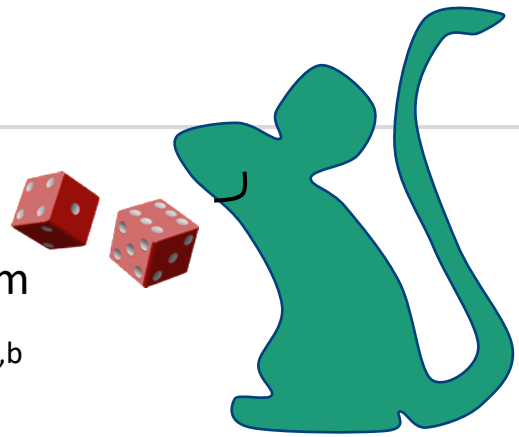
But let's check that it does work

- Check out the Python notebook

M=200, n=10

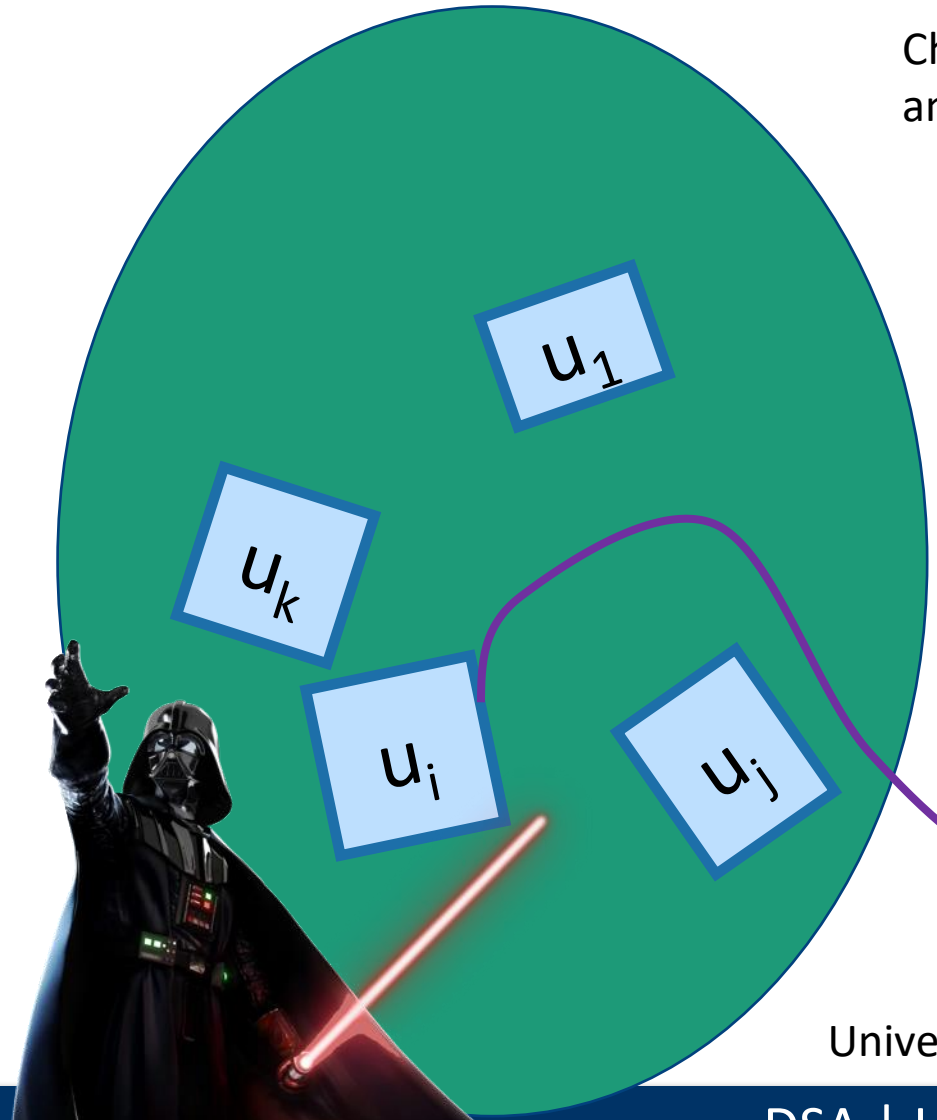


So the whole scheme will be

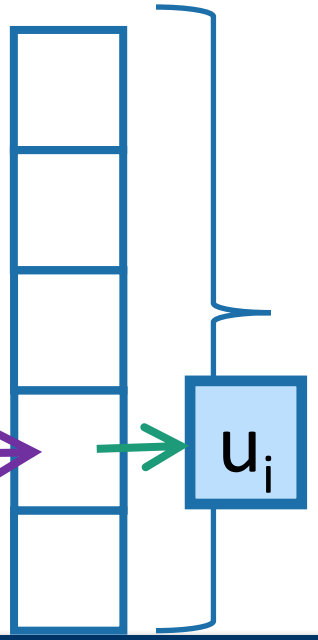


Choose a and b at random and form the function $h_{a,b}$

We can store h in space O(log(M)) since we just need to store a and b.



$h_{a,b}$



Probably these buckets will be pretty balanced.

Universe U

Outline

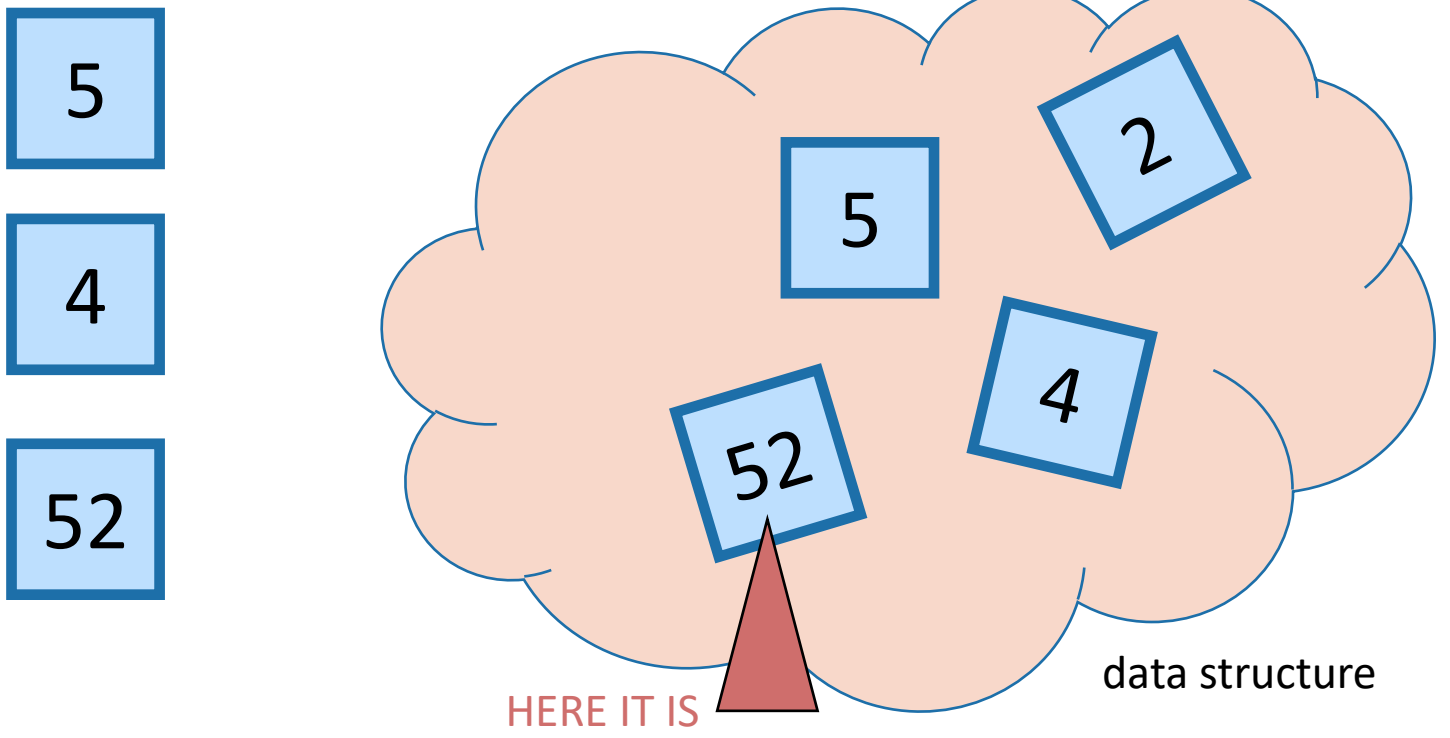
- **Hash tables** are another sort of data structure that allows fast INSERT/DELETE/SEARCH.
  - like self-balancing binary trees
  - The difference is we can get better performance in expectation by using randomness.
- **Hash families** are the magic behind hash tables.
- **Universal hash families** are even more magic.

Recap 

Want O(1) INSERT/DELETE/SEARCH

- We are interested in putting nodes with keys into a data structure that supports fast INSERT/DELETE/SEARCH.

- INSERT
- DELETE
- SEARCH

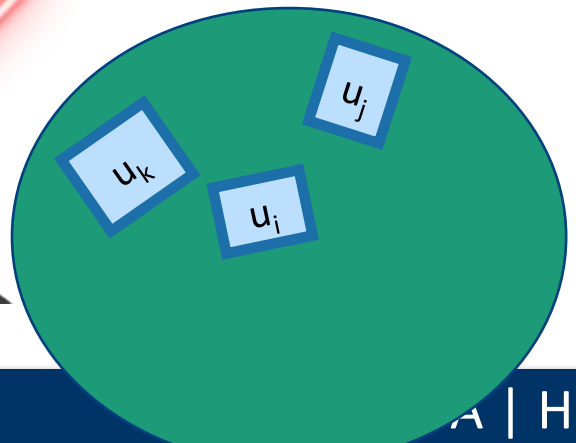


We studied this game

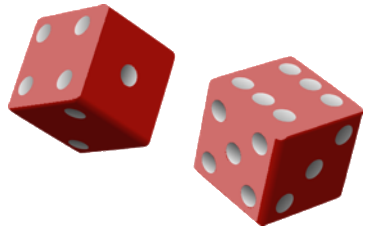
1. An adversary chooses any n items $u_1, u_2, \ldots, u_n \in U$, and any sequence of L INSERT/DELETE/SEARCH operations on those items.

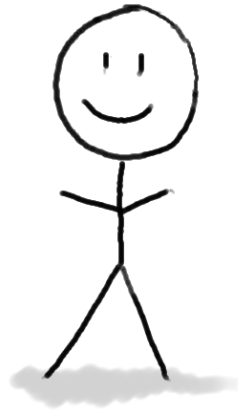
13	22	43	92	7

INSERT 13, INSERT 22, INSERT 43,
INSERT 92, INSERT 7, SEARCH 43,
DELETE 92, SEARCH 7, INSERT 92

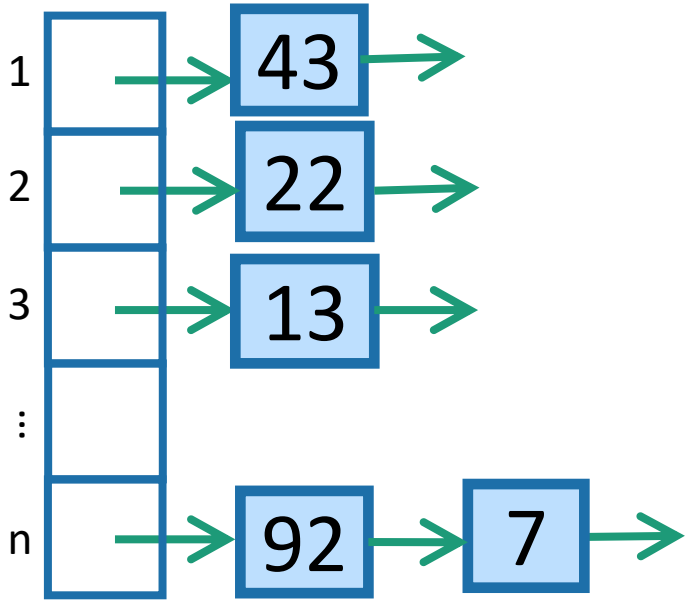


2. You, the algorithm, choose a **random** hash function $h: U \to \{1, \ldots, n\}$.





3. **HASH IT OUT**



Uniformly random h was good

- If we choose h uniformly at random,

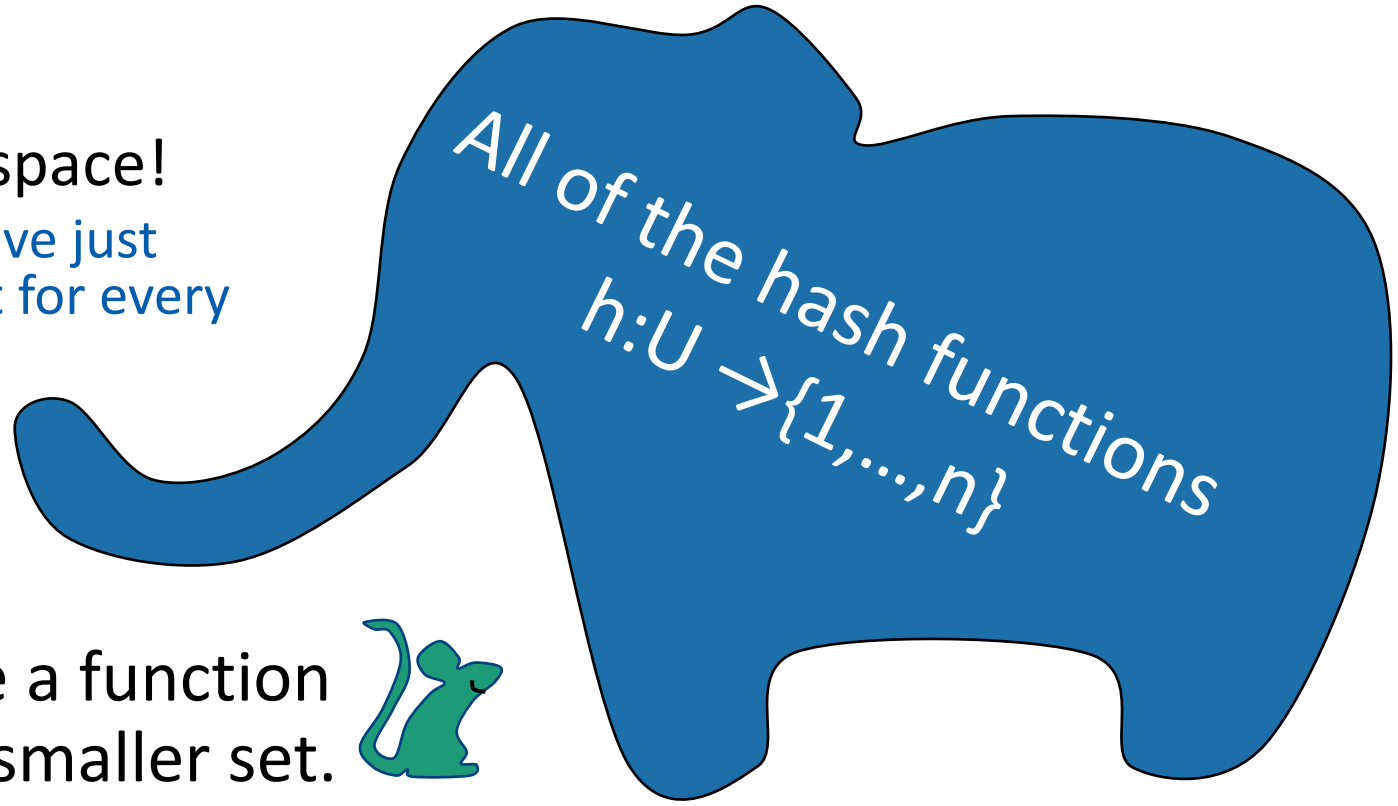
$$\text{for all } u_i, u_j \in U \quad \text{with } u_i \neq u_j,$$

$$P_{h \in H}\{ h(u_i) = h(u_j)\} \leq \frac{1}{n}$$

- That was enough to ensure that all INSERT/DELETE/SEARCH operations took O(1) time in expectation, even on adversarial inputs.

Uniformly random h was bad

- If we actually want to implement this, we have to store the hash function h.
  - That takes a lot of space!
    - We may as well have just initialized a bucket for every single item in U.
  - Instead, we chose a function randomly from a smaller set.



Universal Hash Families

H is a universal hash family if:

- If we choose h uniformly at random in H,

$$\text{for all } u_i, u_j \in U \quad \text{with } u_i \neq u_j,$$

$$P_{h \in H}\{ h(u_i) = h(u_j)\} \leq \frac{1}{n}$$

This was all we needed to make sure that the buckets were balanced in expectation!

- We gave an example of a really small universal hash family, of size $O(M^2)$
- That means we need only O(log M) bits to store it.

Conclusion:

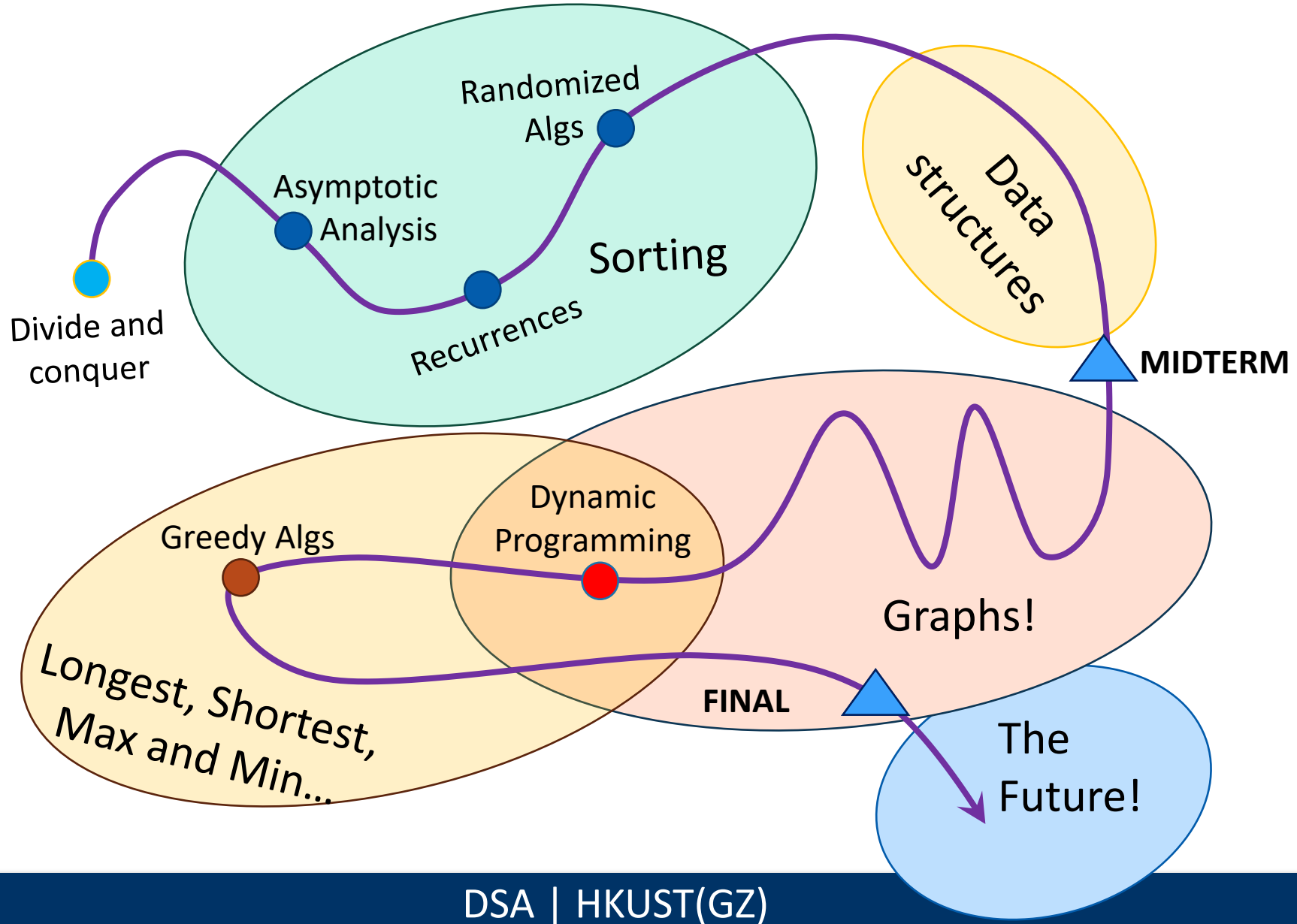
Hashing a universe of size M into n buckets, where at most n of the items in M ever show up.

- We can build a hash table that supports INSERT/DELETE/SEARCH in O(1) expected time
- Requires O(n log(M)) bits of space.
  - O(n) buckets
  - O(n) items with log(M) bits per item
  - O(log(M)) to store the hash function

Sorting lower bounds and O(n)-time sorting

Roadmap

More detailed schedule on the website!



Sorting

- We've seen a few O(n log(n))-time algorithms.
  - MERGESORT has worst-case running time O(n log(n))
  - QUICKSORT has expected running time O(n log(n))

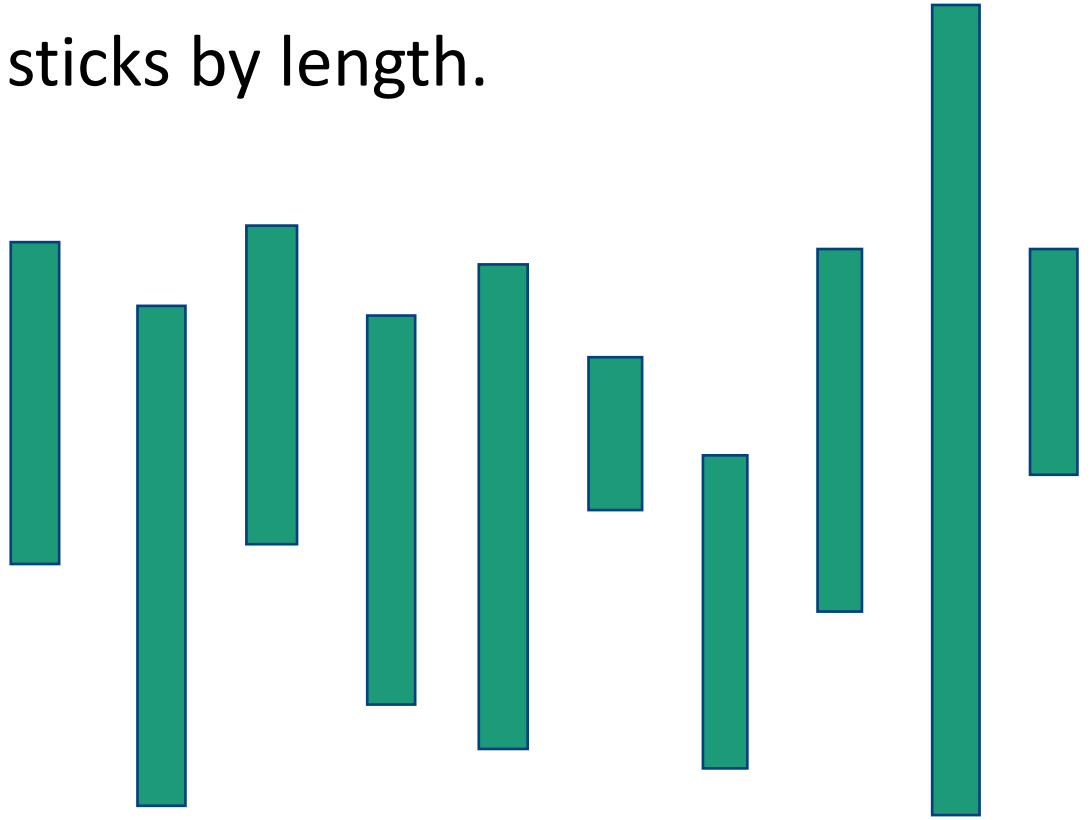
Can we do better?

Depends on who you ask…

An O(1)-time algorithm for sorting:

StickSort

- Problem: sort these n sticks by length.



- Now they are sorted this way.
- Algorithm:
  - Drop them on a table.

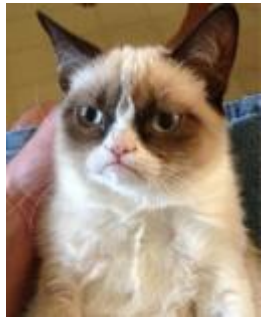
That may have been unsatisfying

- But StickSort does raise some important questions:
  - What is our model of computation?
    - Input: array
    - Output: sorted array
    - Operations allowed: comparisons

    -vs-

    - Input: sticks
    - Output: sorted sticks in vertical order
    - Operations allowed: dropping on tables
  - What are reasonable models of computation?

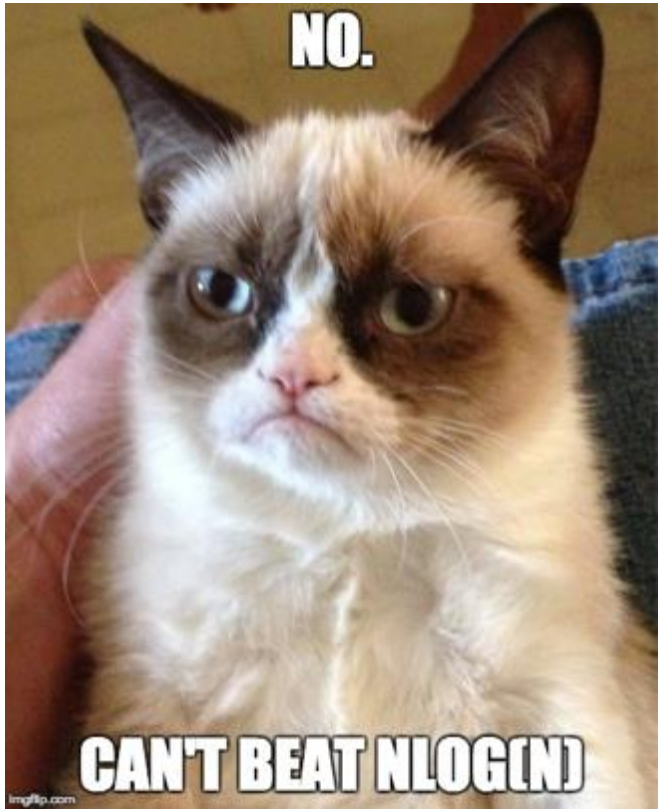
Today: two models



- Comparison-based sorting model
  - This includes MergeSort, QuickSort, InsertionSort
  - We'll see that any algorithm in this model must take at least Ω(n log(n)) steps.

- Another model (more reasonable than the stick model…)
  - CountingSort and RadixSort
  - Both run in time O(n)

Comparison-based sorting



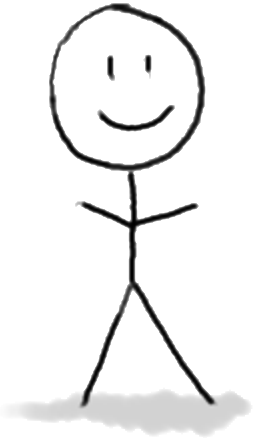
Comparison-based sorting algorithms

- You want to sort an array of items.
- You can’t access the items’ values directly: you can only compare two items and find out which is bigger or smaller.

Comparison-based sorting algorithms

Want to sort these items.
There's some ordering on them, but we don't know what it is.

Is  bigger than  ?



Algorithm

YES

The algorithm's job is to output a correctly sorted list of all the objects.

 is shorthand for

"the first thing in the input list"

There is a genie who knows what the right order is.

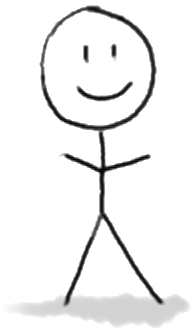
The genie can answer YES/NO questions of the form:
is [this] bigger than [that]?

All the sorting algorithms we have seen work like this.

eg, QuickSort:

Pivot!

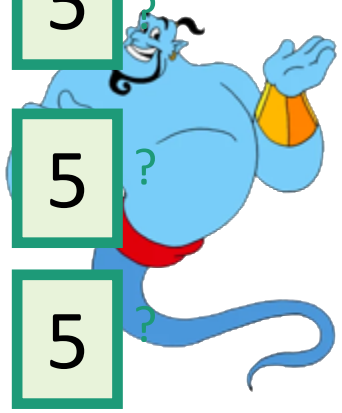
7	6	3	5	1	4	2



Is 7 bigger than 5 ? YES

Is 6 bigger than 5 ? YES

Is 3 bigger than 5 ? NO



5

etc.

Lower bound of Ω(n log(n)).

- Theorem:
  - Any deterministic comparison-based sorting algorithm must take Ω(n log(n)) steps.
  - Any randomized comparison-based sorting algorithm must take Ω(n log(n)) steps in expectation.

This covers all the sorting algorithms we know!!!

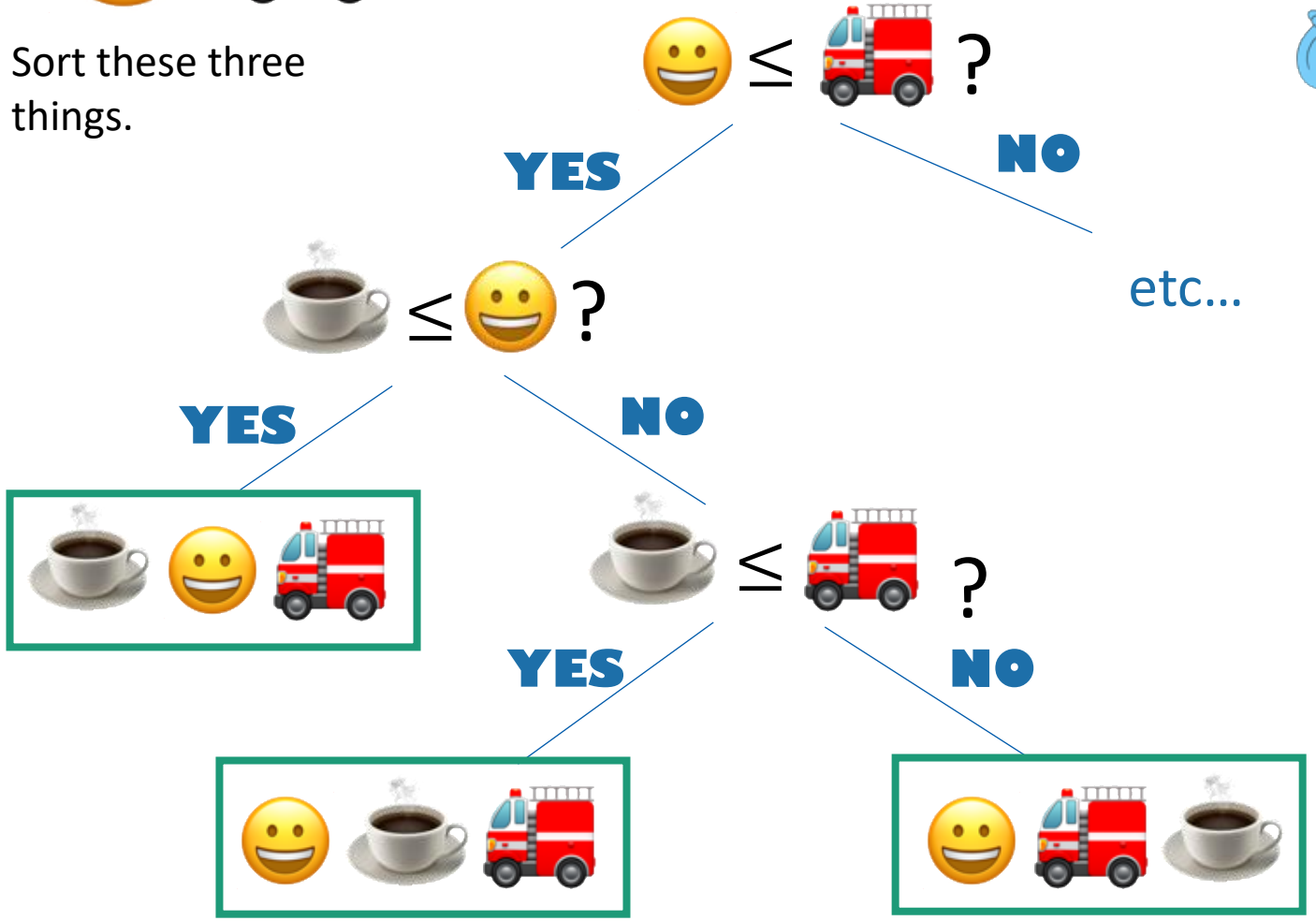
- How might we prove this?

1. Consider all comparison-based algorithms, one-by-one, and analyze them.
2. Don't do that.

Instead, argue that all comparison-based sorting algorithms give rise to a **decision tree**.
Then analyze decision trees.

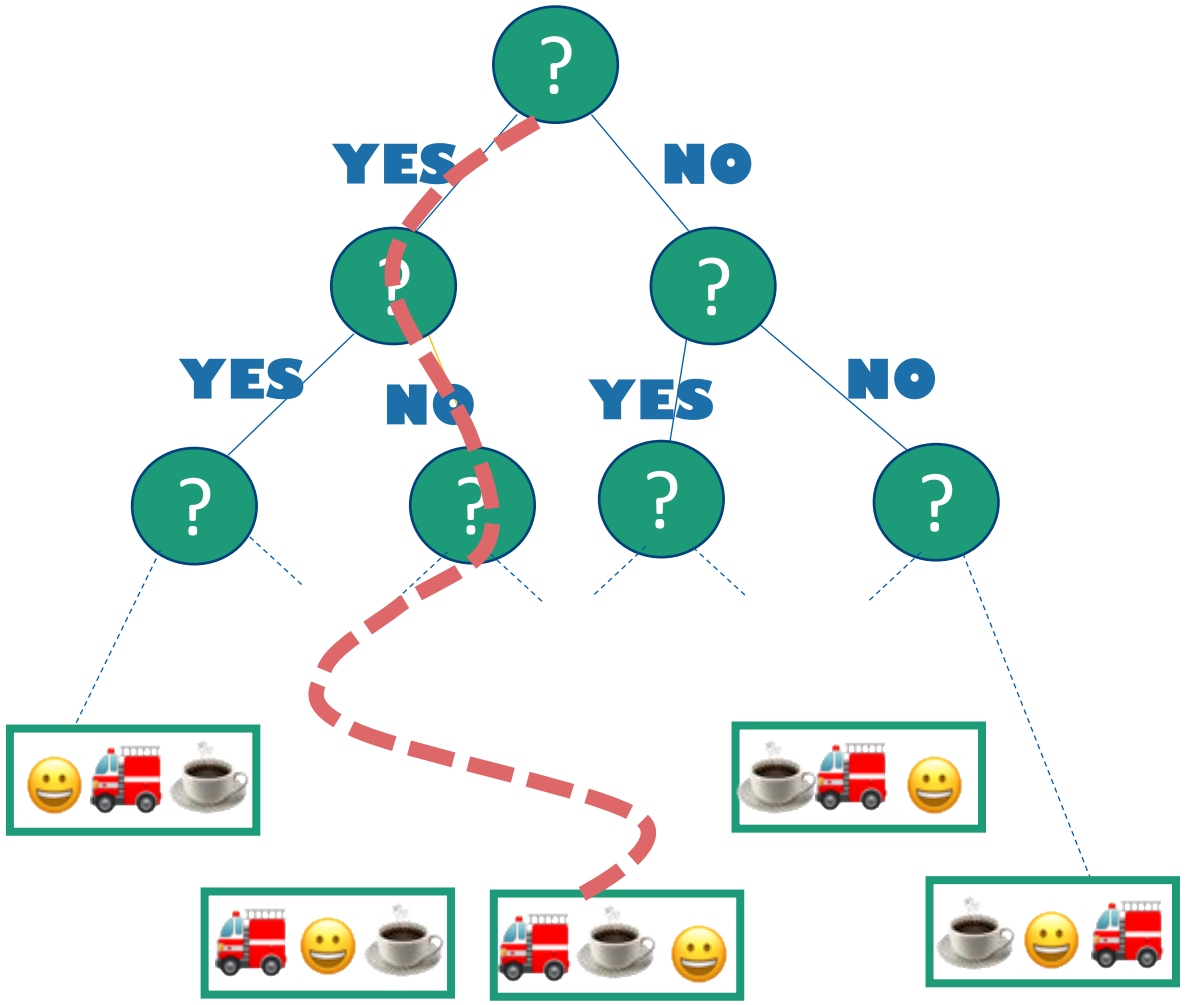
Decision trees

Sort these three things.

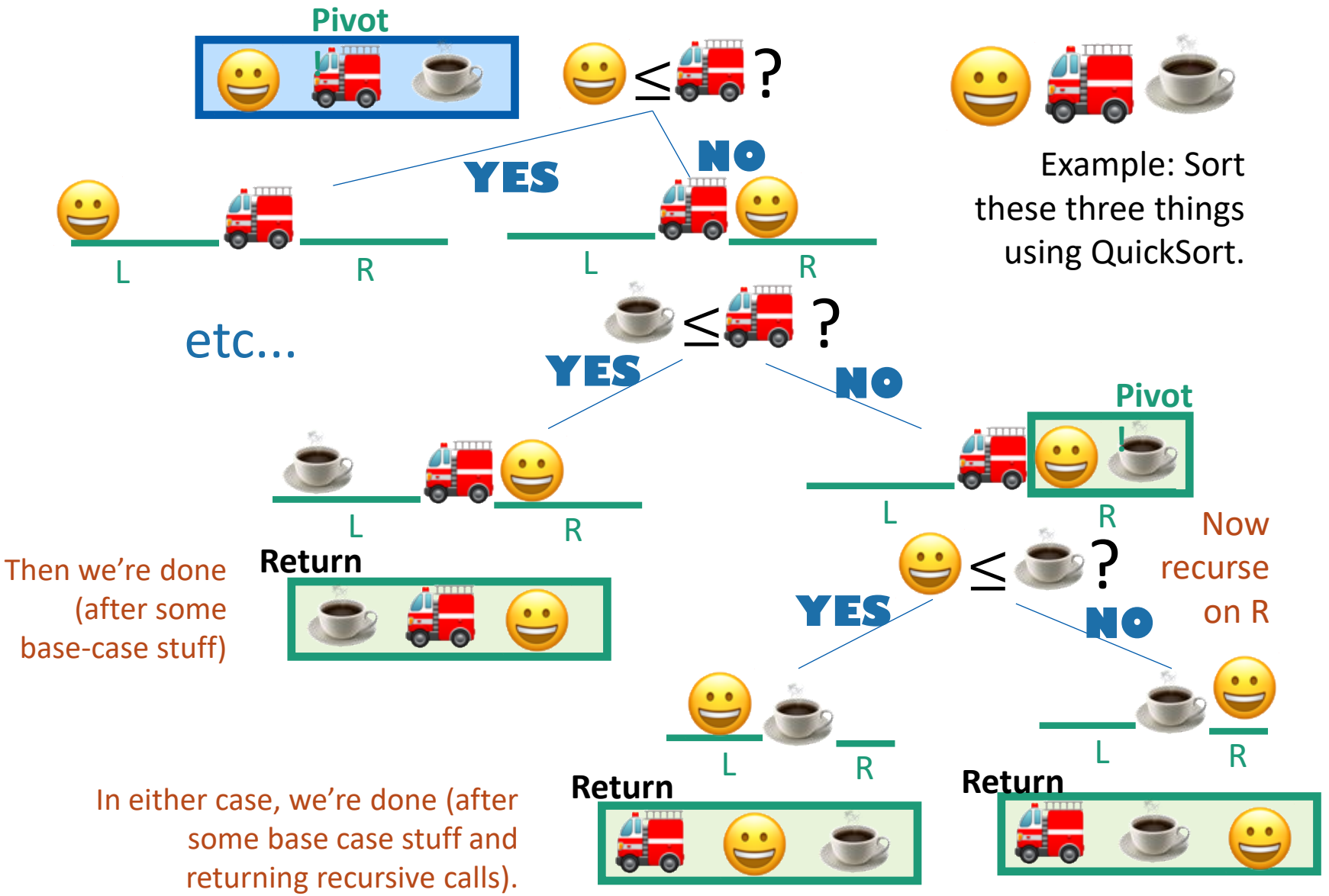


Decision trees

- Internal nodes correspond to yes/no questions.
- Each internal node has two children, one for "yes" and one for "no."
- Leaf nodes correspond to outputs.
  - In this case, all possible orderings of the items.
- Running an algorithm on a particular input corresponds to a particular path through the tree.



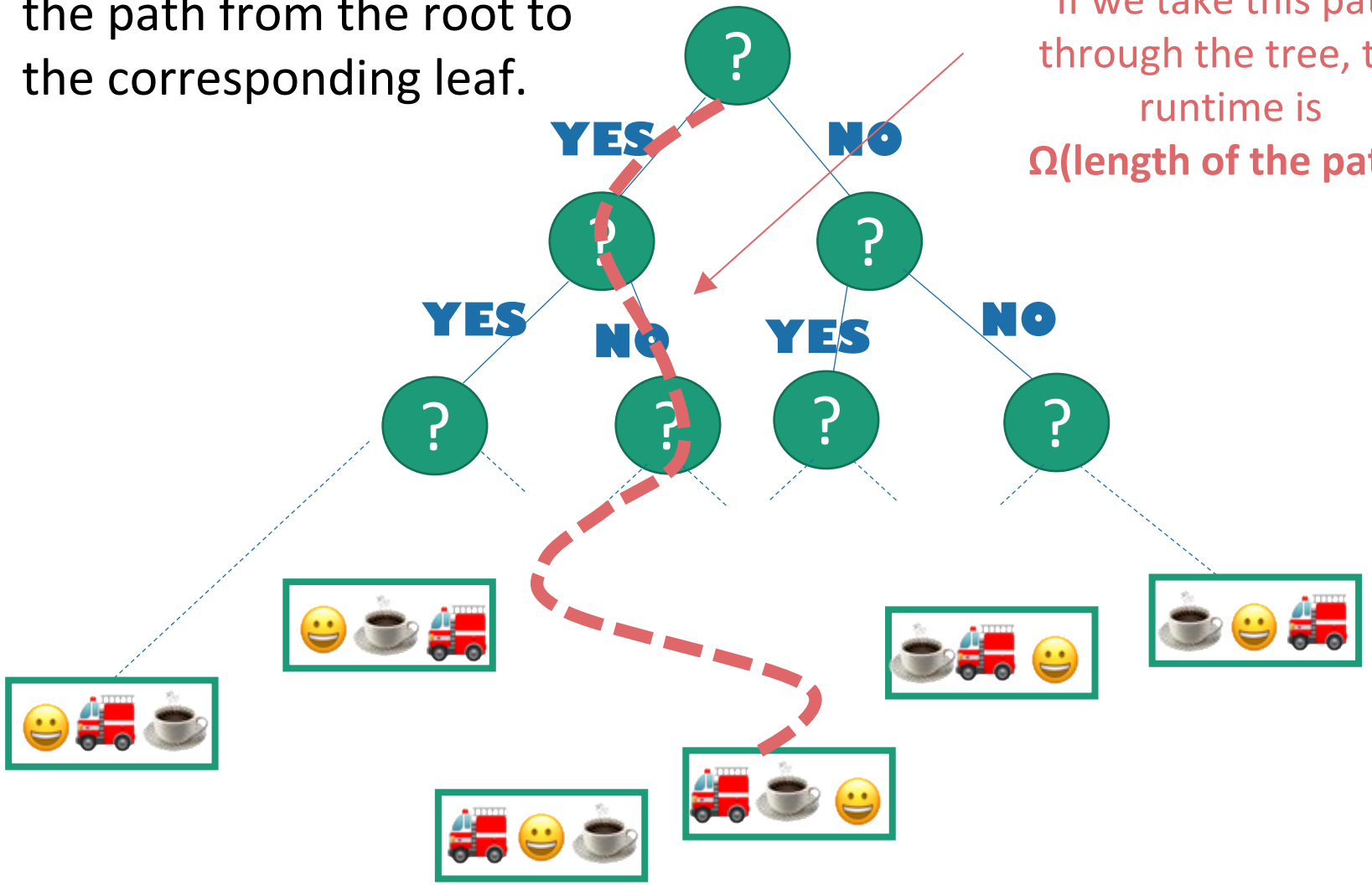
Comparison-based algorithms look like decision trees.



Example: Sort these three things using QuickSort.

Q: What's the runtime on a particular input?

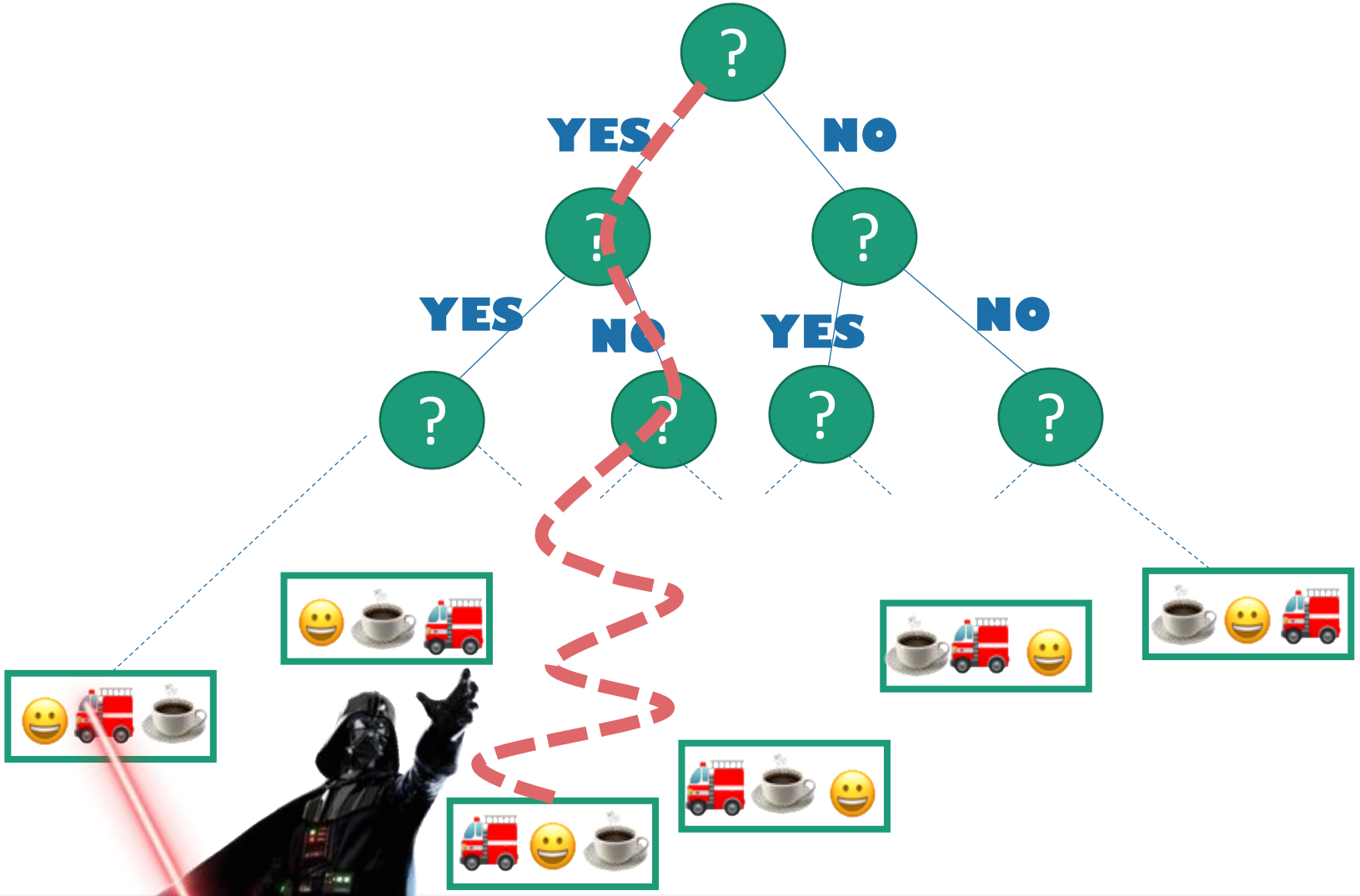
A: At least the length of the path from the root to the corresponding leaf.



If we take this path through the tree, the runtime is **Ω(length of the path).**

Q: What's the worst-case runtime?

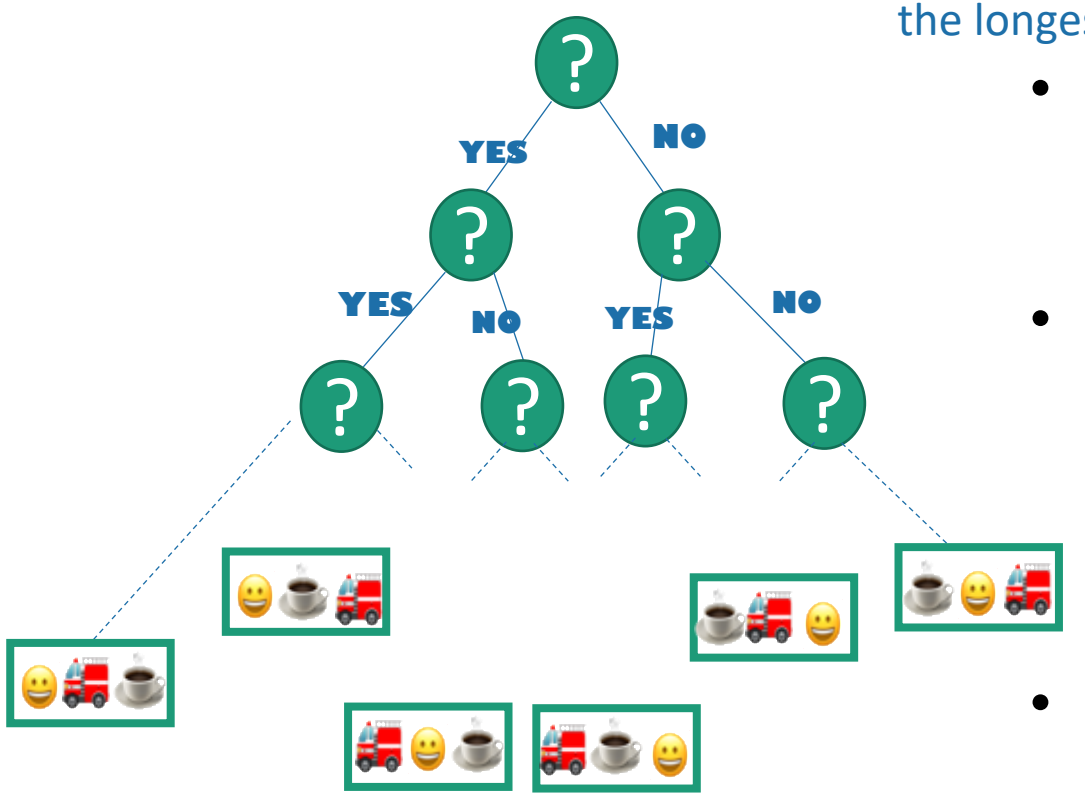
A: At least Ω(length of the longest path).



How long is the longest path?

being sloppy about floors and ceilings!

香港科技大学(广州)
THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY (GUANGZHOU)



We want a statement: in all such trees, the longest path is at least _____

- This is a binary tree with at least n! leaves.
- The shallowest tree with n! leaves is the completely balanced one, which has depth log(n!).
- So in all such trees, the longest path is at least log(n!).

- n! is about $(n/e)^n$ (Stirling's approx.*).
- log(n!) is about n log(n/e) = Ω(n log(n)).

Conclusion: the longest path has length at least Ω(n log(n)).

*Stirling's approximation is a bit more complicated than this, but this is good enough for the asymptotic result we want.

Lower bound of Ω(n log(n)).

- Theorem:
  - Any deterministic comparison-based sorting algorithm must take Ω(n log(n)) steps.

- Proof recap:
  - Any deterministic comparison-based algorithm can be represented as a decision tree with n! leaves.
  - The worst-case running time is at least the depth of the decision tree.
  - All decision trees with n! leaves have depth Ω(n log(n)).
  - So any comparison-based sorting algorithm must have worst-case running time at least Ω(n log(n)).

Aside: What about randomized algorithms?

- For example, QuickSort?
- Theorem:
  - Any randomized comparison-based sorting algorithm must take Ω(n log(n)) steps in expectation.

- Proof:
  - (same ideas as deterministic case)
  - (you are not responsible for this proof in this class)

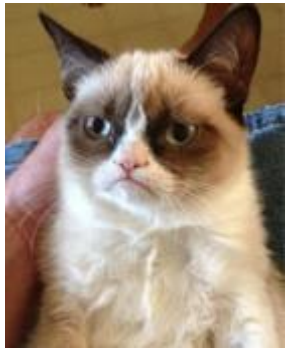
\end{Aside}

Try to prove this yourself!

Ollie the over-achieving ostrich

So that's bad news

- Theorem:
  - Any deterministic comparison-based sorting algorithm must take Ω(n log(n)) steps.



- Theorem:
  - Any randomized comparison-based sorting algorithm must take Ω(n log(n)) steps in expectation.

On the bright side, MergeSort is optimal!

- This is one of the cool things about lower bounds like this: we know when we can declare victory!

But what about StickSort?

- StickSort can't be implemented as a comparison-based sorting algorithm. So these lower bounds don't apply.
- But StickSort was kind of silly.

Can we do better?

- Is there another model of computation that's less silly than the StickSort model, in which we can sort faster than nlog(n)?

Especially if I have to spend time cutting all those sticks to be the right size!

Beyond comparison-based sorting algorithms

Another model of computation

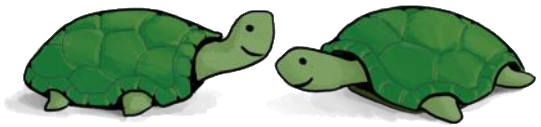
- The items you are sorting have meaningful values.

9	6	3	5	2	1	2

instead of

Pre-lecture exercise

- How long does it take to sort n people by their month of birth?



Share your answers

1 (Jan)

1 (Jan)

4 (Apr)

5 (May)

Another model of computation

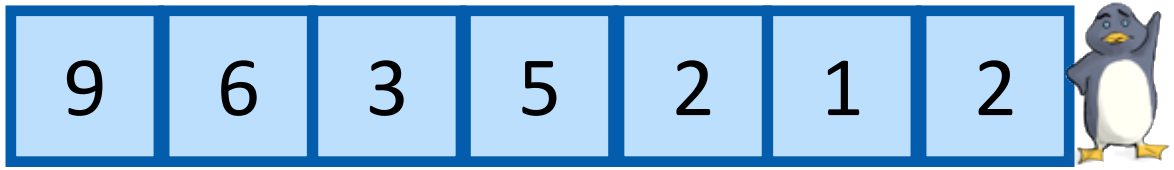
- The items you are sorting have meaningful values.

9	6	3	5	2	1	2

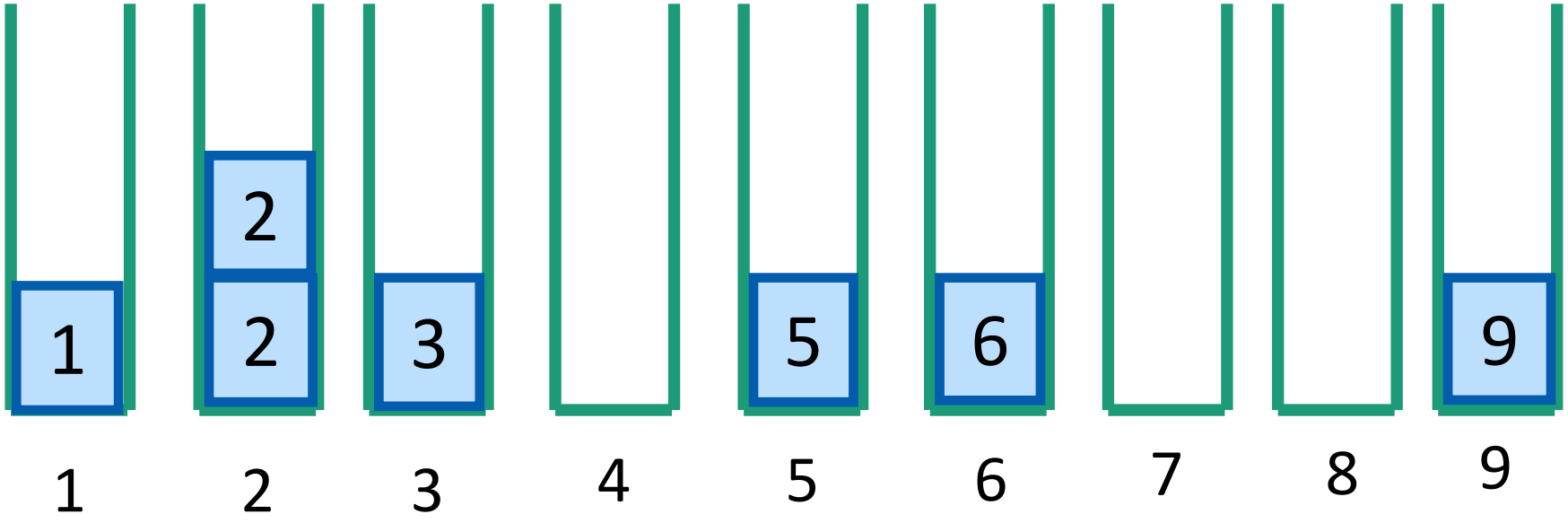
instead of

Why might this help?

CountingSort:



Implement the buckets as linked lists. They are first-in, first-out. This will be useful later.



Concatenate the buckets!

SORTED!

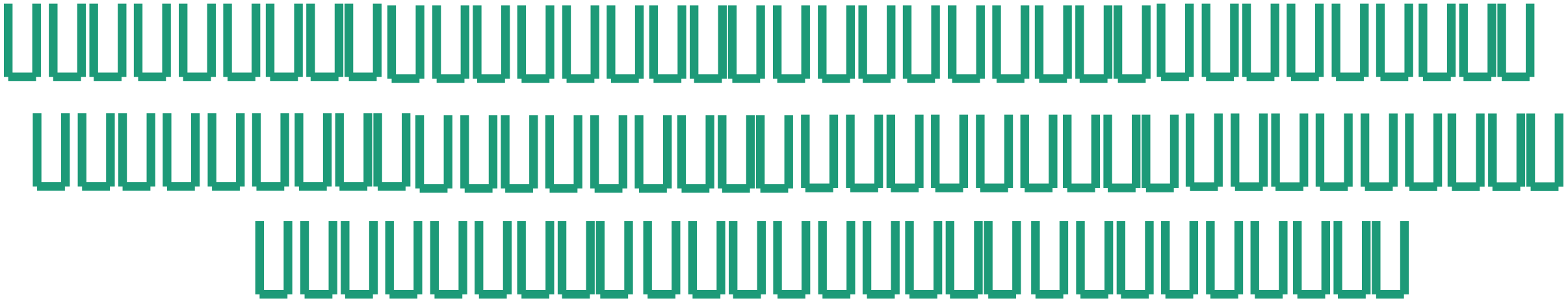
In time O(n).

Assumptions

- Need to be able to know what bucket to put something in.
  - We assume we can evaluate the items directly, not just by comparison
- Need to know what values might show up ahead of time.

2	12345	13	2^{1000}	50	100000000	1

- Need to assume there are not too many such values.

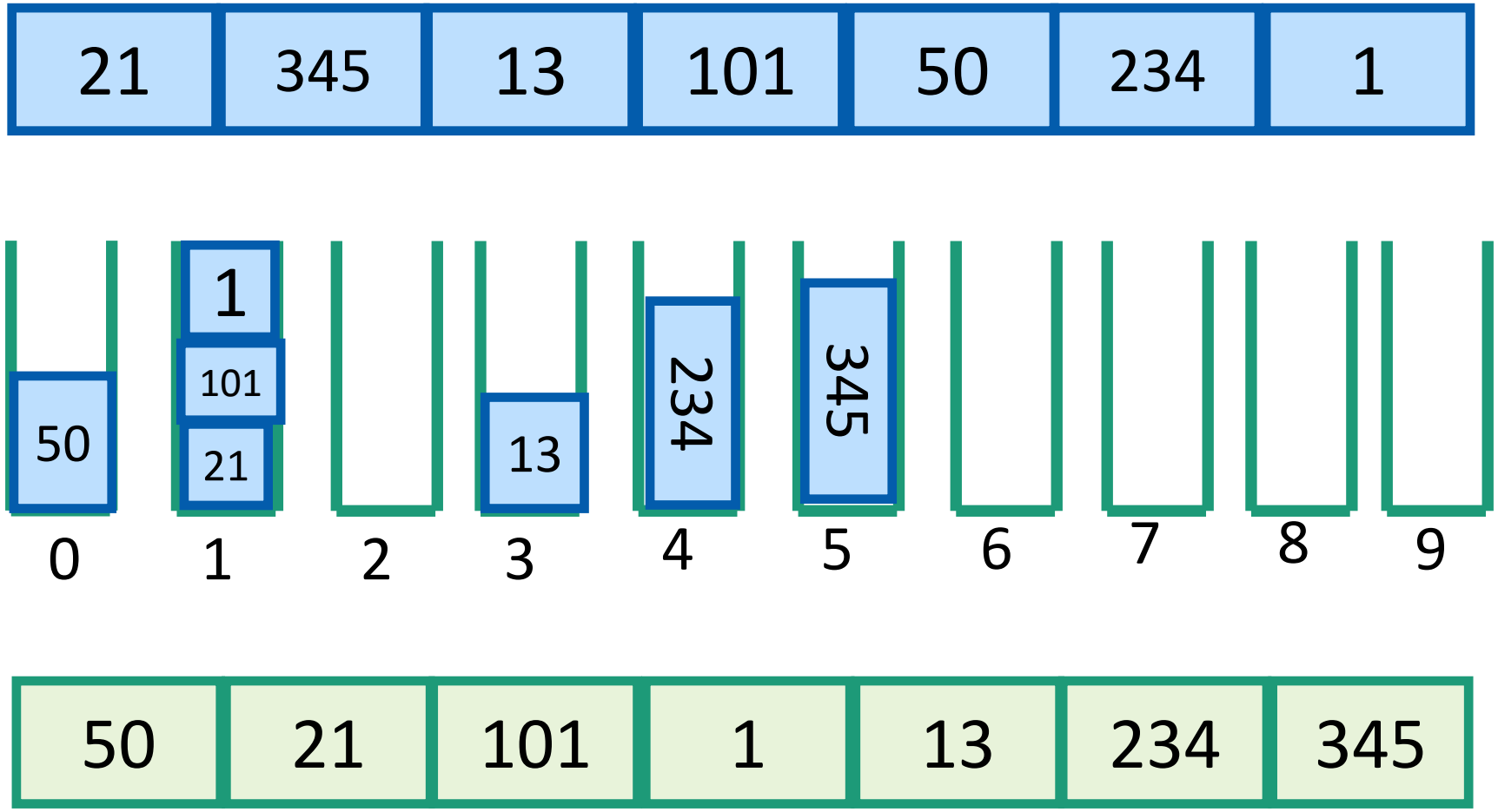


RadixSort

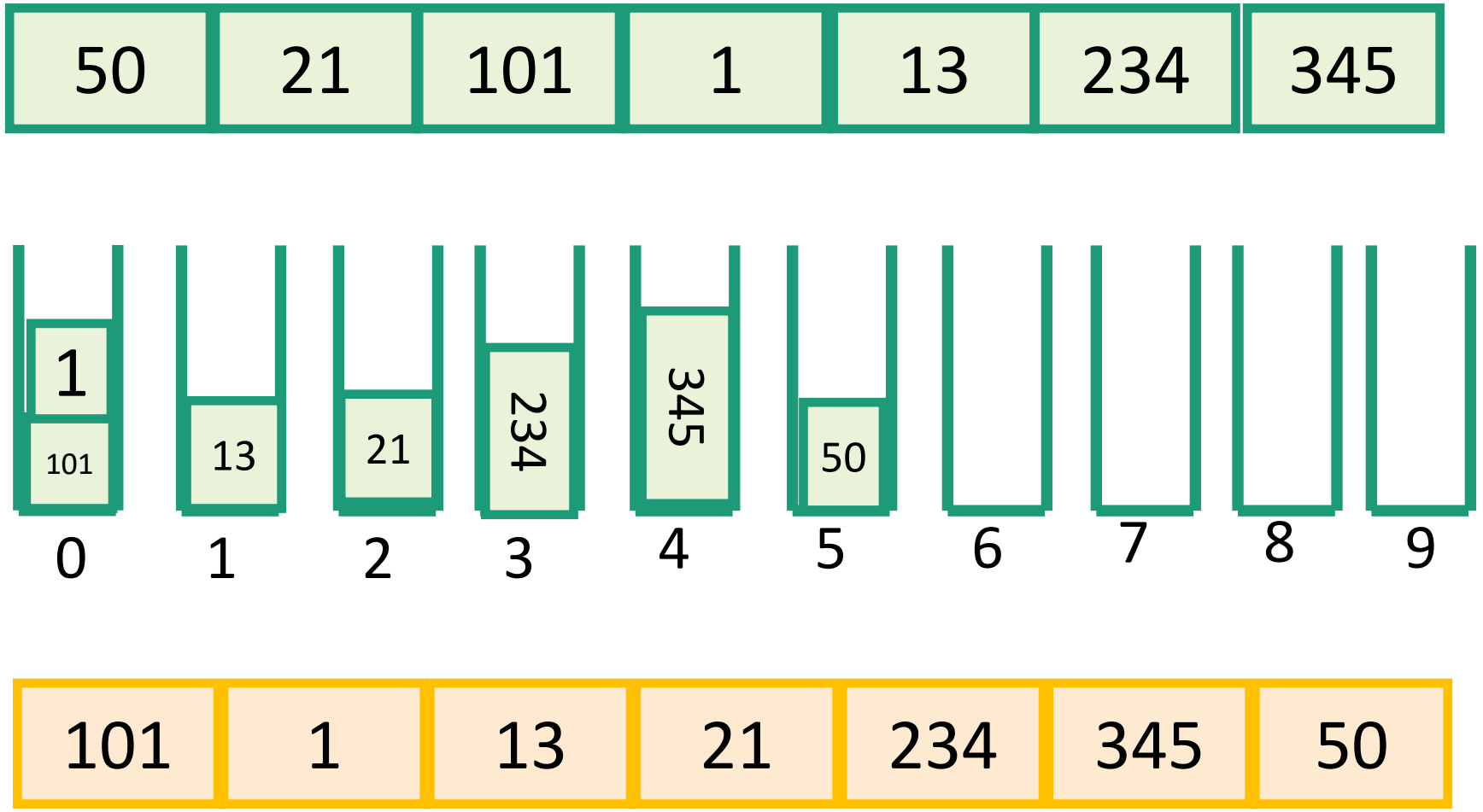
- For sorting integers up to size M
  - or more generally for lexicographically sorting strings
- Can use less space than CountingSort

- Idea: CountingSort on the least-significant digit first, then the next least-significant, and so on.

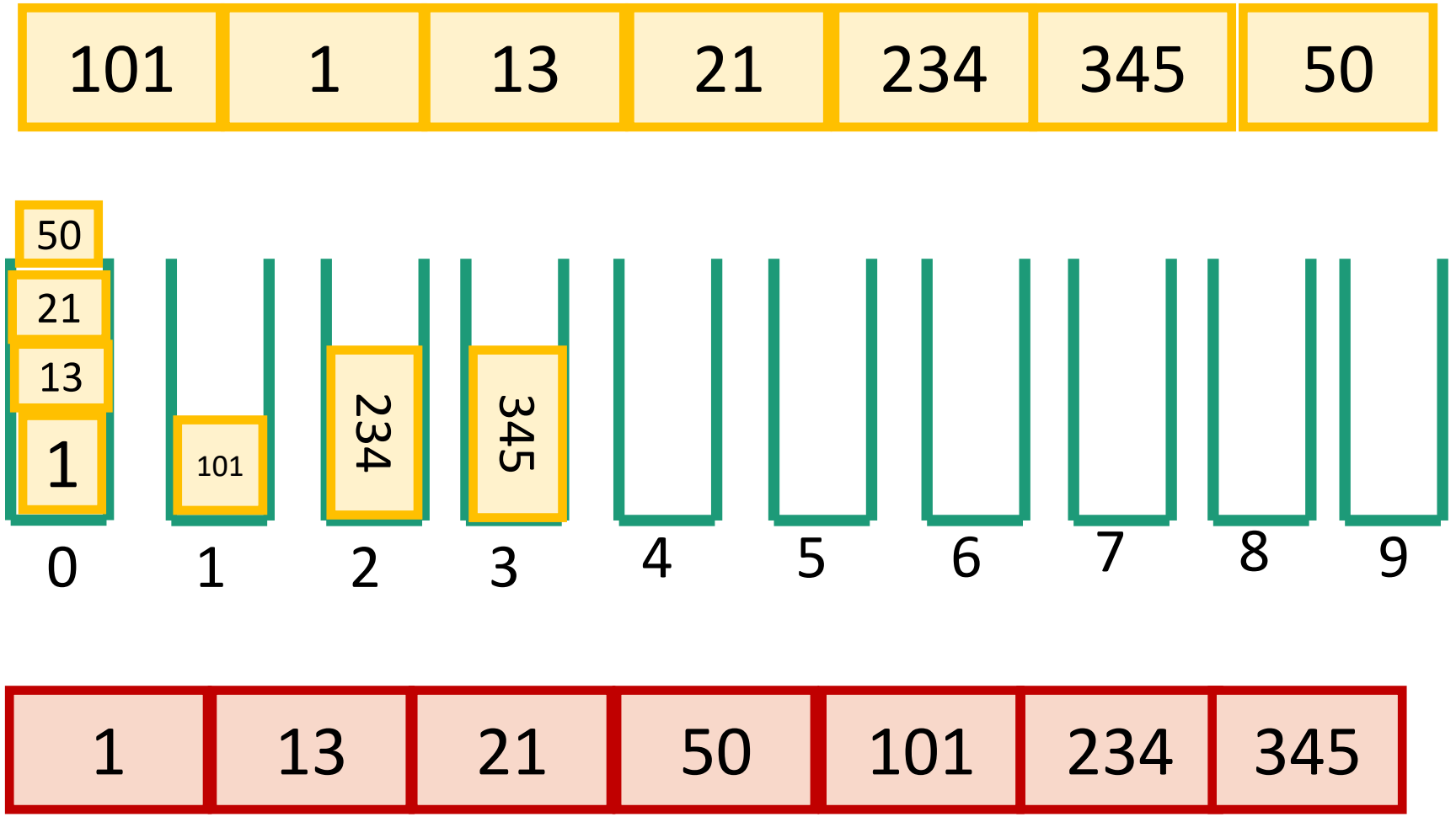
Step 1: CountingSort on least significant digit



Step 2: CountingSort on the 2nd least sig. digit



Step 3: CountingSort on the 3rd least sig. digit



It worked!!

Why does this work?

Original array:

21	345	13	101	50	234	1

Next array is sorted by the first digit.

50	21	101	1	13	234	345

Next array is sorted by the first two digits.

101	01	13	21	234	345	50

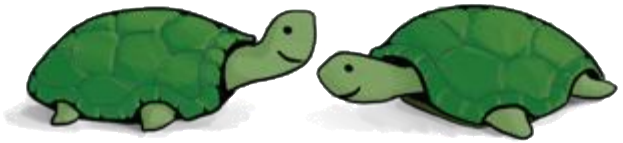
Next array is sorted by all three digits.

001	013	021	050	101	234	345

Sorted array

To prove this is correct…

- What is the inductive hypothesis?



Think-Share Terrapins

Original array:

21	345	13	101	50	234	1

Next array is sorted by the first digit.

50	21	101	1	13	234	345

Next array is sorted by the first two digits.

101	01	13	21	234	345	50

Next array is sorted by all three digits.

001	013	021	050	101	234	345

Sorted array

RadixSort is correct

- Inductive hypothesis:
  - After the k'th iteration, the array is sorted by the first k least-significant digits.
- Base case:
  - "Sorted by 0 least-significant digits" means not yet sorted, so the IH holds for k=0.
- Inductive step:
  - TO DO
- Conclusion:
  - The inductive hypothesis holds for all k, so after the last iteration, the array is sorted by all the digits. Hence, it's sorted!

Inductive step

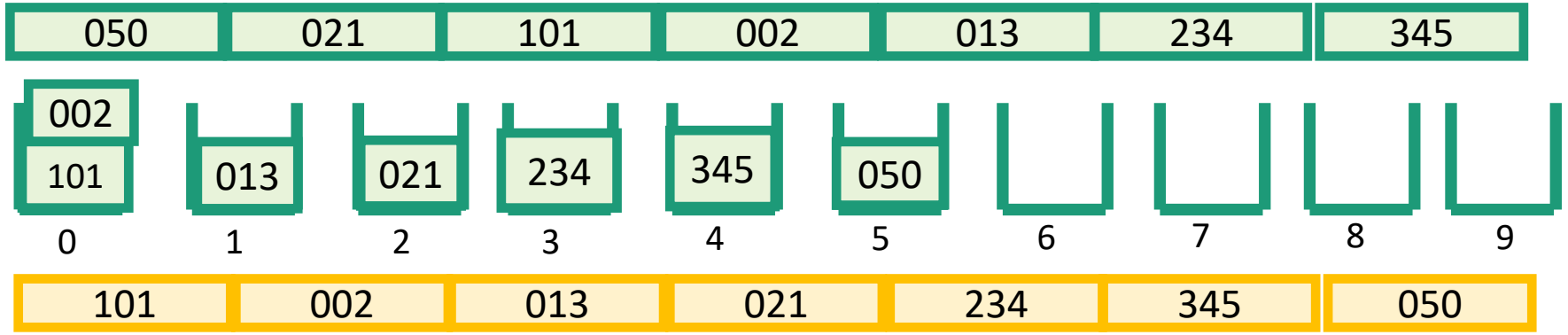
Inductive hypothesis:

After the k'th iteration, the array is sorted by the first k least-significant digits.

- Need to show: if IH holds for k=i-1, then it holds for k=i.
  - Suppose that after the i-1'st iteration, the array is sorted by the first i-1 least-significant digits.
  - Need to show that after the i'th iteration, the array is sorted by the first i least-significant digits.

EXAMPLE: i=2

IH: this array is sorted by first digit.



Want to show: this array is sorted by 1st and 2nd digits.

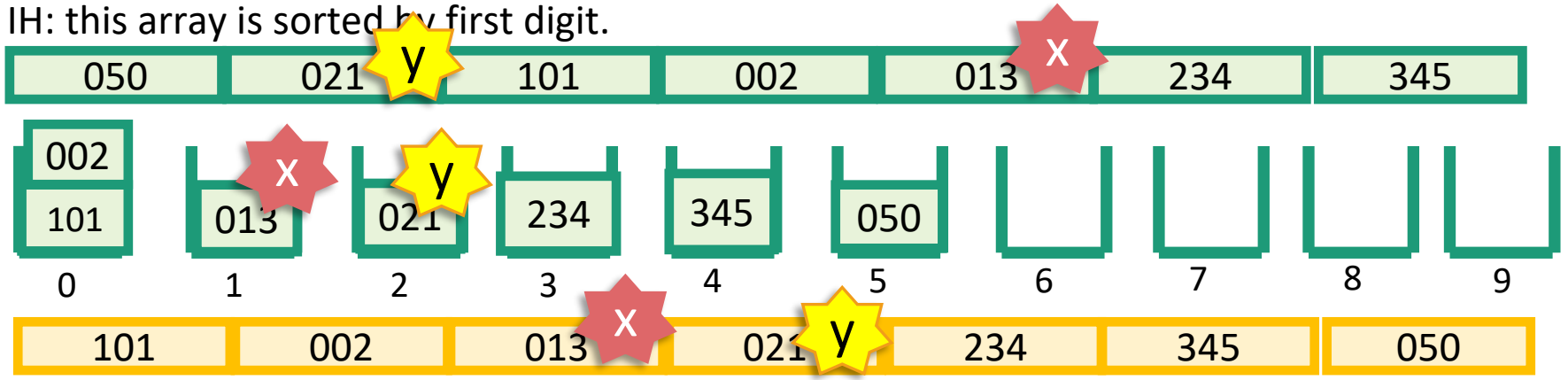
Proof sketch...proof on next (skipped) slide

Want to show: after the i'th iteration, the array is sorted by the first i least-significant digits.

- Let $x=[x_d x_{d-1} \ldots x_2 x_1]$ and $y=[y_d y_{d-1} \ldots y_2 y_1]$ be any x,y.
- Suppose $[x_i x_{i-1} \ldots x_2 x_1] < [y_i y_{i-1} \ldots y_2 y_1]$.
- Want to show that x appears before y at end of i'th iteration.
- CASE 1: $x_i < y_i$
  - x is in an earlier bucket than y. ✓

Aka, we want to show that for any x and y so that x belongs before y, we put x before y.

EXAMPLE: i=2



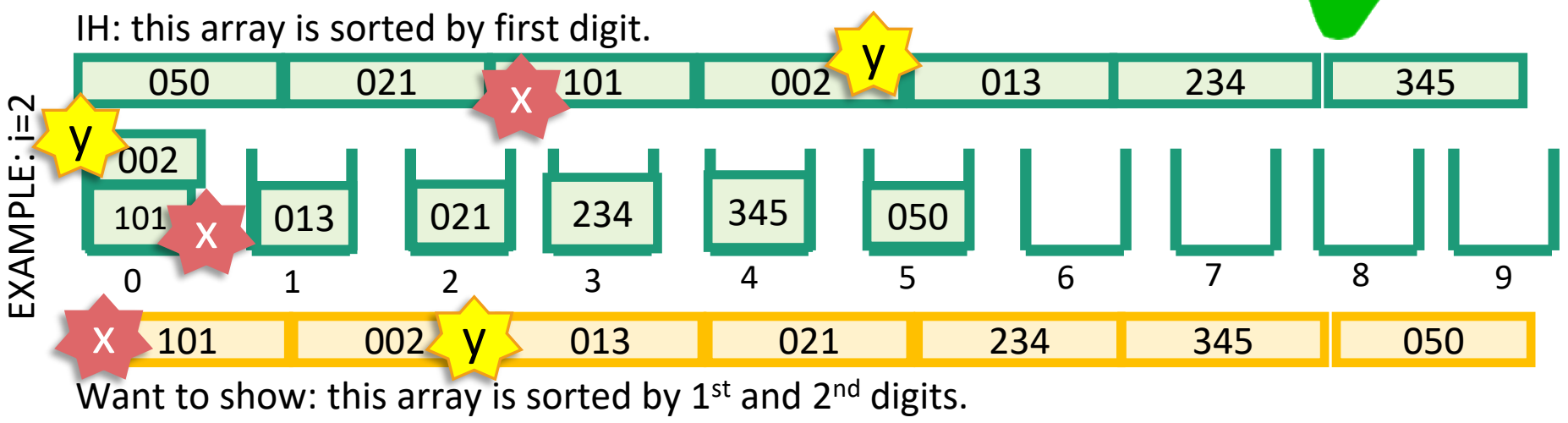
Want to show: this array is sorted by 1st and 2nd digits.

Proof sketch...proof on next (skipped) slide

- Let $x=[x_d x_{d-1} \ldots x_2 x_1]$ and $y=[y_d y_{d-1} \ldots y_2 y_1]$ be any x,y.
- Suppose $[x_i x_{i-1} \ldots x_2 x_1] < [y_i y_{i-1} \ldots y_2 y_1]$.
- Want to show that x appears before y at end of i'th iteration.
- CASE 1: $x_i < y_i$
  - x is in an earlier bucket than y.
- CASE 2: $x_i = y_i$
  - $[x_{i-1} \ldots x_2 x_1] < [y_{i-1} \ldots y_2 y_1]$,
  - x and y in same bucket, but x was put in the bucket first.

Want to show: after the i'th iteration, the array is sorted by the first i least-significant digits.

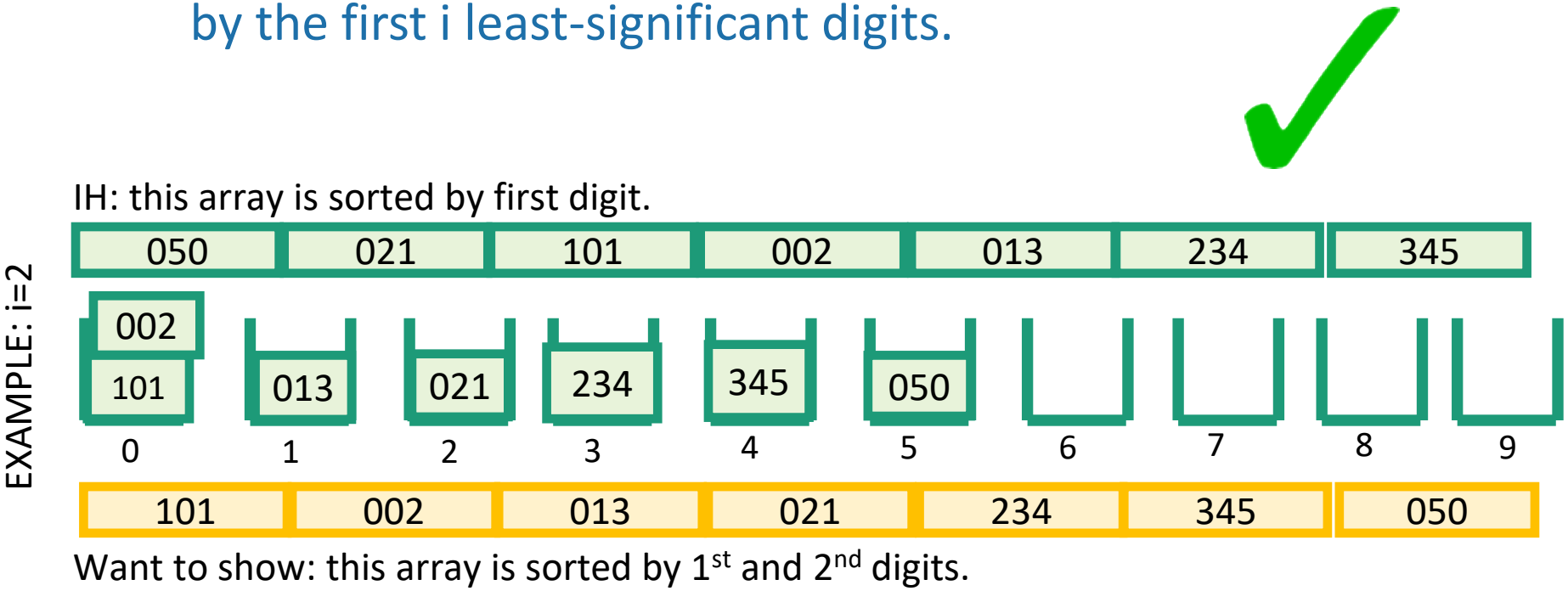
Aka, we want to show that for any x and y so that x belongs before y, we put x before y.



Inductive step

Inductive hypothesis:
After the k'th iteration, the array is sorted by the first k least-significant digits.

- Need to show: if IH holds for k=i-1, then it holds for k=i.
  - Suppose that after the i-1'st iteration, the array is sorted by the first i-1 least-significant digits.
  - Need to show that after the i'th iteration, the array is sorted by the first i least-significant digits.



RadixSort is correct

- Inductive hypothesis:
  - After the k'th iteration, the array is sorted by the first k least-significant digits.
- Base case:
  - "Sorted by 0 least-significant digits" means not sorted, so the IH holds for k=0.
- Inductive step:
  - TO DO

- Conclusion:
  - The inductive hypothesis holds for all k, so after the last iteration, the array is sorted by all the digits. Hence, it's sorted!

What is the running time?

for RadixSorting
numbers base-10.

- Suppose we are sorting n d-digit numbers (in base 10).

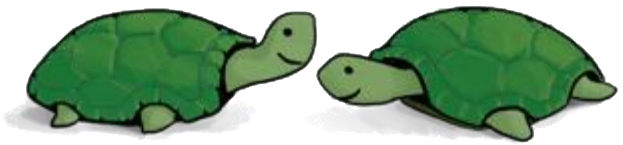
e.g., n=7, d=3:

021	345	013	101	050	234	001

1. How many iterations are there?

2. How long does each iteration take?

3. What is the total running time?



Think-Share Terrapins

What is the running time?

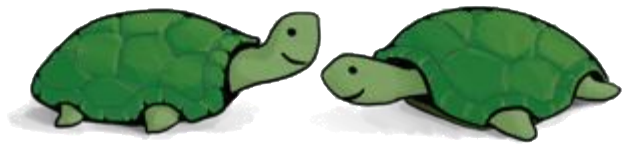
for RadixSorting numbers base-10.

- Suppose we are sorting n d-digit numbers (in base 10).

e.g., n=7, d=3:

021	345	013	101	050	234	001

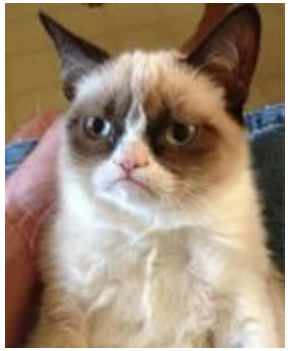
1. How many iterations are there?
   – d iterations
2. How long does each iteration take?
   – Time to initialize 10 buckets, plus time to put n numbers in 10 buckets. O(n).
3. What is the total running time?
   – O(nd)



Think-Share Terrapins

This doesn't seem so great

- To sort n integers, each of which is in {1,2,…,n}…
- d = $\lfloor \log_{10}(n) \rfloor + 1$
  - For example:
    - n = 1234
    - $\lfloor \log_{10}(1234) \rfloor + 1 = 4$
  - More explanation on next (skipped) slide.
- Time = $O(nd) = O(n \log(n))$.
  - Same as MergeSort!



Can we do better?

- RadixSort base 10 doesn't seem to be such a good idea…
- But what if we change the base? (Let's say base r)
- We will see there's a trade-off:
  - Bigger r means more buckets
  - Bigger r means fewer digits

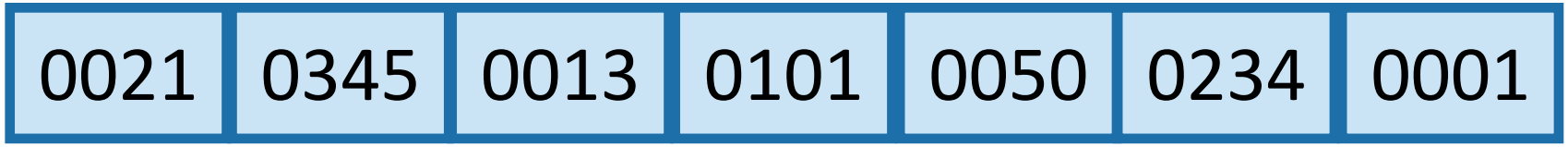
Example: base 100

Original array:

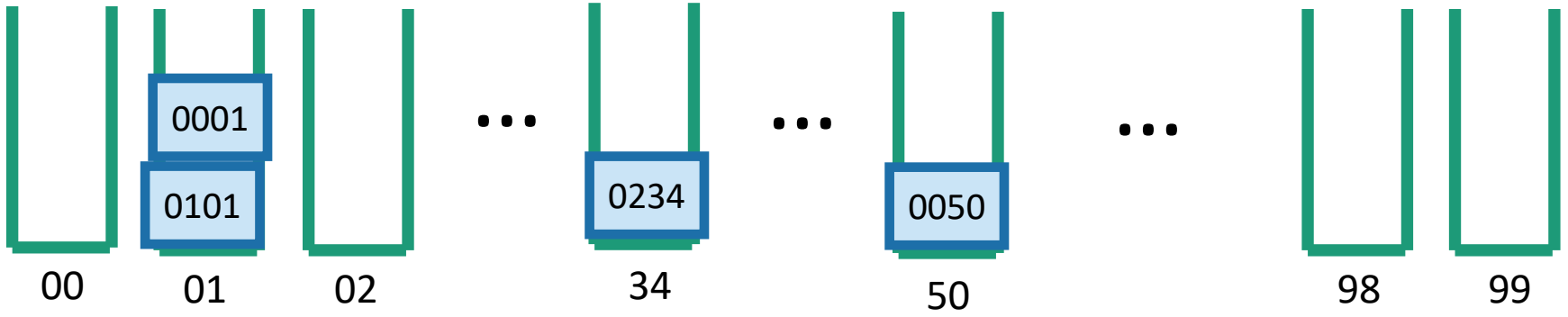
21	345	13	101	50	234	1

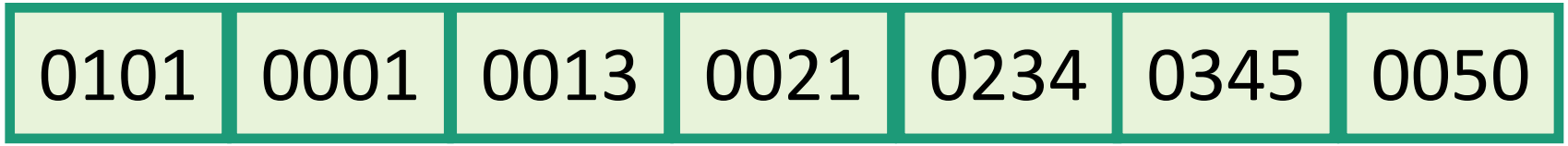
Example: base 100

Original array:



100 buckets:

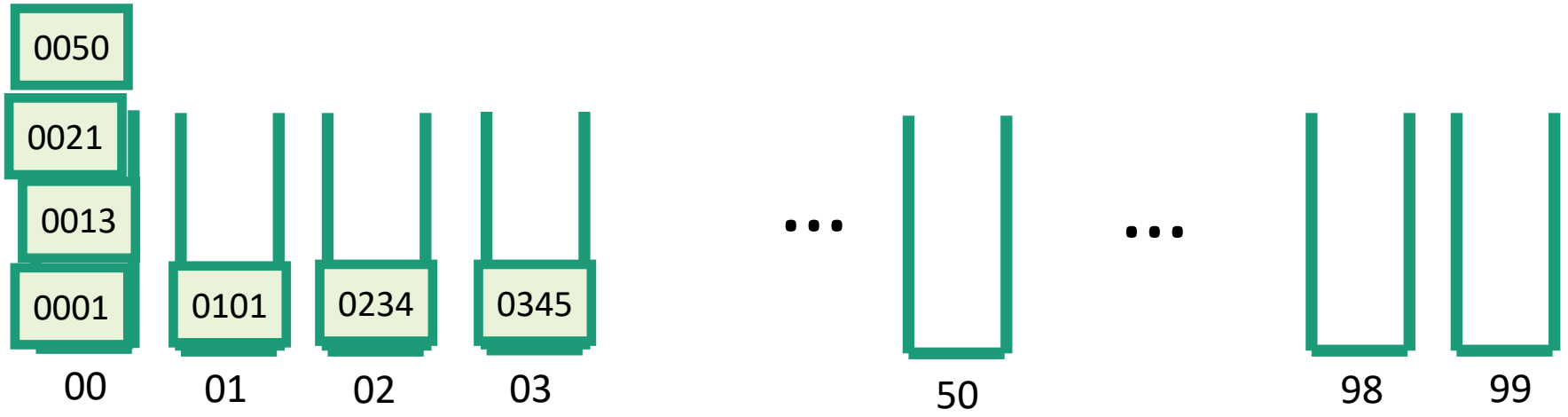




Example: base 100

0101	0001	0013	0021	0234	0345	0050

100 buckets:



0001	0013	0021	0050	0101	0234	0345

Sorted!

Example: base 100

Original array

0021	0345	0013	0101	0050	0234	0001

0101	0001	0013	0021	0234	0345	0050

0001	0013	0021	0050	0101	0234	0345

Sorted array

Base 100:

- d=2, so only 2 iterations.
- 100 buckets

vs.

Base 10:

- d=3, so 3 iterations.
- 10 buckets

Bigger base means more buckets but fewer iterations.

General running time of RadixSort

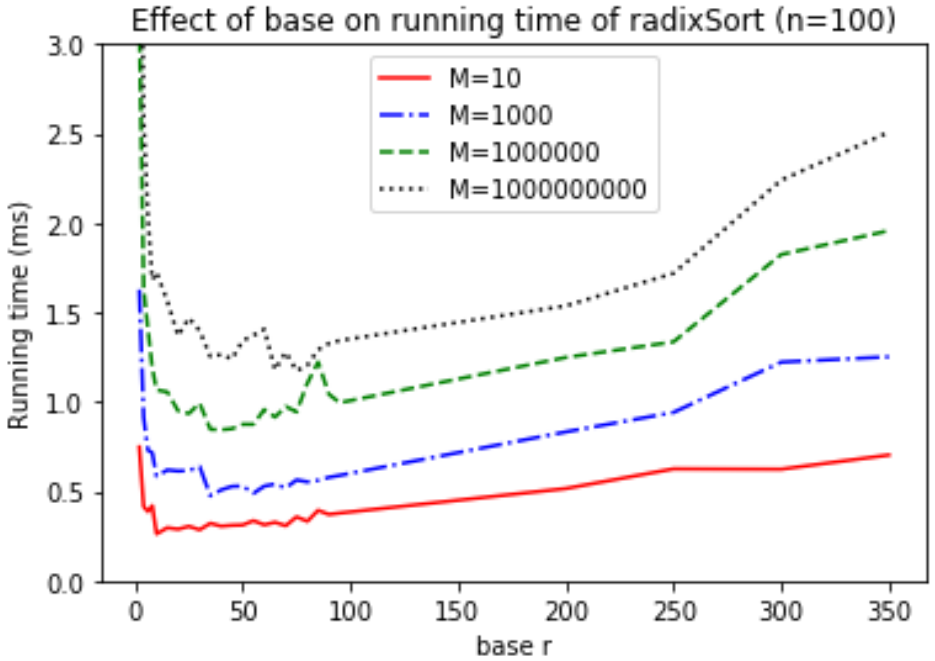
- Say we want to sort:
  - n integers,
  - maximum size M,
  - in base r.
- Number of iterations of RadixSort:
  - Same as number of digits, base r, of an integer x of max size M.
  - That is $\mathrm{d} = \lfloor \log_r(M) \rfloor + 1$
- Time per iteration:
  - Initialize r buckets, put n items into them
  - $O(n + r)$ total time.
- Total time:
  - $O\big(d \cdot (n + r)\big) = O\big((\lfloor \log_r(M) \rfloor + 1) \cdot (n + r)\big)$

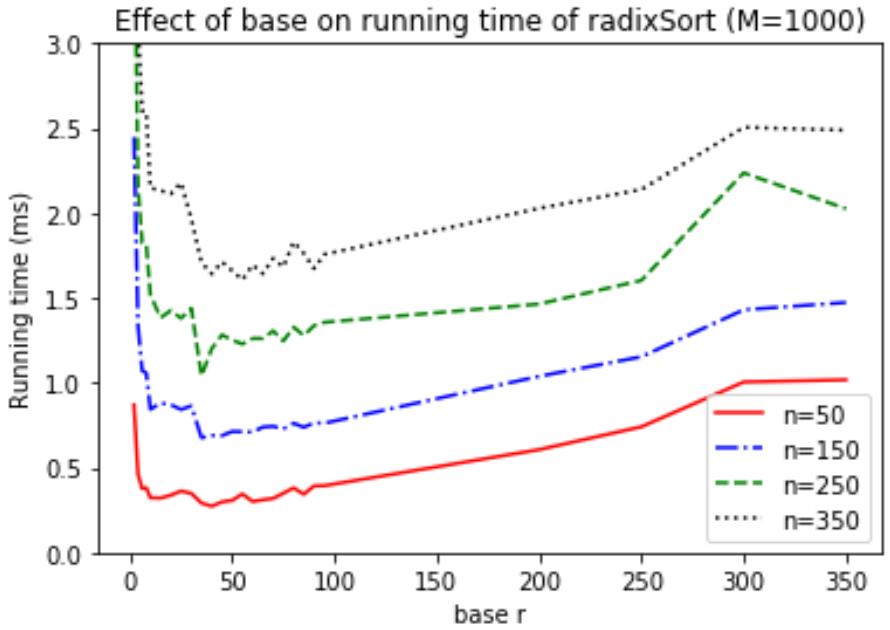
Convince yourself that this is the right formula for d.

Trade-offs

Running time: $O\big((\lfloor \log_r(M) \rfloor + 1) \cdot (n + r)\big)$

- Given n, M, how should we choose r?
- Looks like there's some sweet spot:





IPython Notebook

A reasonable choice: r=n

- Running time:

$$O\left(\left(\lfloor \log_r(M) \rfloor + 1\right) \cdot (n + r)\right)$$

Intuition: balance n and r here.

- Choose n=r:

$$O\left(n \cdot \left(\lfloor \log_n(M) \rfloor + 1\right)\right)$$

Choosing r = n is pretty good. What choice of r optimizes the asymptotic running time? What if I also care about space?

Ollie the over-achieving ostrich

Running time of RadixSort with r=n

- To sort n integers of size at most M, time is

$$O\left(n \cdot \left(\lfloor \log_n(M) \rfloor + 1 \right)\right)$$

- So the running time (in terms of n) depends on how big M is in terms of n:
  - If $M \leq n^c$ for some constant c, then this is O(n).
  - If $M = 2^n$, then this is $O\left(\frac{n^2}{\log(n)}\right)$
- The number of buckets needed is r=n.

What have we learned?

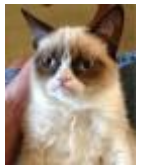
You can put any constant here instead of 100.

- RadixSort can sort n integers of size at most n^{100} in time O(n), and needs enough space to store O(n) integers.
- If your integers have size much much bigger than n (like 2^n), maybe you shouldn't use RadixSort.
- It matters how we pick the base.

Recap

- How difficult sorting is depends on the model of computation.
- How reasonable a model of computation is is up for debate.

- Comparison-based sorting model
  - This includes MergeSort, QuickSort, InsertionSort
  - Any algorithm in this model must use at least Ω(n log(n)) operations. ☹
  - But it can handle arbitrary comparable objects. ☺
- If we are sorting small integers (or other reasonable data):
  - CountingSort and RadixSort
  - Both run in time O(n) ☺
  - Might take more space and/or be slower if integers get too big ☹



The End