

APS1070

Foundations of Data Analytics and
Machine Learning

Winter 2021

Week 2:

- *Python and Math Checklists*
- *Asymptotic Complexity*
- *Analysis of Sorting Algorithms*
- *Dictionary Abstract Data Type*



Slide Attribution

These slides contain materials from various sources. Special thanks to the following authors:

- Scott Sanner
- Ali Hadi Zadeh
- Jason Riordon
- Mark C. Wilson

Last Time

- Course Overview
- End-to-End Machine Learning as the overall theme for APS1070
 - Programming
 - Mathematics
 - Machine Learning Theory

Agenda

➤ Today's focus is on **Algorithm Complexity**

- Python Checklist
 - Mathematics Checklist
 - Asymptotic Complexity
 - Analysis of Sorting Algorithms
 - Dictionary Abstract Data Type
 - Hash Tables
- } Review

Basic Python Checklist

Checklist: Python Basics

☐ Data Types

- ☐ Single: int, float, bool
- ☐ Multiple: str, list, set, tuple, dict
- ☐ index [], slice [::], mutability

☐ Conditionals

- ☐ if, elif, else

☐ Functions

- ☐ def, return, recursion, default vals

☐ Loops

- ☐ for, while, range
- ☐ list comprehension

☐ Operations

- ☐ arithmetic: +, *, -, /, //, %, **
- ☐ boolean: not, and, or
- ☐ relational: ==, !=, >, <, >=, <=

☐ Display

- ☐ print, end, sep

☐ Files

- ☐ open, close, with
- ☐ read, write
- ☐ CSV

☐ Object-Oriented Programming (OOP)

- ☐ class, methods, attributes
- ☐ __init__, __str__, polymorphism

Python Review

- Tutorial 0 – Python Basics
- Coursera - University of Toronto MOOCs
 - Learn to Program: The Fundamentals
(<https://www.coursera.org/learn/learn-to-program>)
 - Learn to Program: Crafting Quality Code
(<https://www.coursera.org/learn/program-code>)
- APS1070 Piazza Discussion Board
(<https://piazza.com/class/kvslwcx12il4sn>)

Example: Object-Oriented Programming

class Point:

```
def __init__(self, x=0, y=0):
```

```
    self.x = x
```

```
    self.y = y
```

```
def __str__(self):
```

```
    return '(' + str(self.x) + ',' + str(self.y) + ')'
```

```
def add(self, r):
```

```
    self.x = self.x + r.x
```

```
    self.y = self.y + r.y
```

```
def distance_from_origin(self):
```

```
    return (self.x ** 2 + self.y ** 2) ** 0.5
```

```
def halfway(self, target):
```

```
    mx = (self.x + target.x)*0.5
```

```
    my = (self.y + target.y)*0.5
```

```
    return Point(mx,my)
```

class Square:

```
def __init__(self, x1, y1, x2, y2):
```

```
    self.lower_left = Point(x1, y1)
```

```
    self.upper_right = Point(x2, y2)
```

```
def __str__(self):
```

```
    s1 = str(self.lower_left)
```

```
    s2 = str(self.upper_right)
```

```
    return 'square: ' + s1 + s2
```

```
def area(self):
```

```
    return (self.upper_right.x - self.lower_left.x) * \
           (self.upper_right.y - self.lower_left.y)
```

```
>>> p = Point(1,2)
```

```
>>> print(p)
```

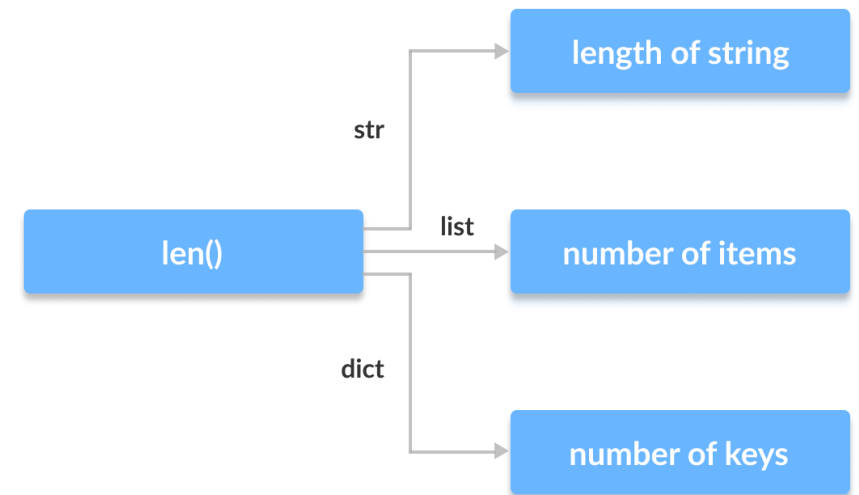
```
>>> s = Square(0,0,2,2)
```

```
>>> print(s)
```


Python Polymorphism

- Polymorphism gives you the ability to represent objects of different types using a single interface.
- How does Python know to add **int + int** or concatenate **str + str**?
- In Python `__name__` represents reserved methods associated with a class (or object). Some common reserved methods include:

- `__str__`
- `__add__`
- `__gtr__`
- ...



Example: Polymorphism

Here we see **polymorphism** implemented using the **method area()**. This method works on **objects** of the **types**— **Rectangle** and **Square**. And it operates **differently** on **objects** of **different classes**.

```
class Rectangle:
```

```
    def __init__(self, length, breadth):
```

```
        self.l = length
```

```
        self.b = breadth
```

```
    def area(self):
```

```
        return self.l * self.b
```

```
class Square:
```

```
    def __init__(self, side):
```

```
        self.s = side
```

```
    def area(self):
```

```
        return self.s ** 2
```

```
rec = Rectangle(10, 20)
```

```
squ = Square(10)
```

```
>>> rec = Rectangle(10, 20)
```

```
>>> squ = Square(10)
```

```
>>> print("Area of rectangle is: ", rec.area())
```

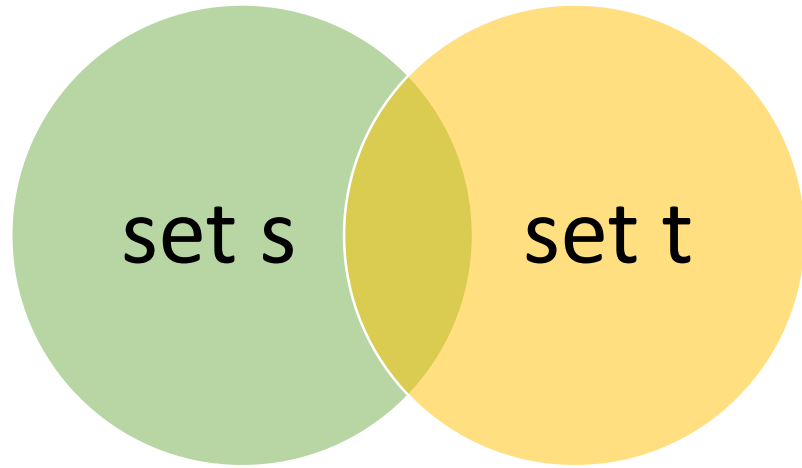
```
>>> print("Area of square is: ", squ.area())
```

Mathematics Checklist

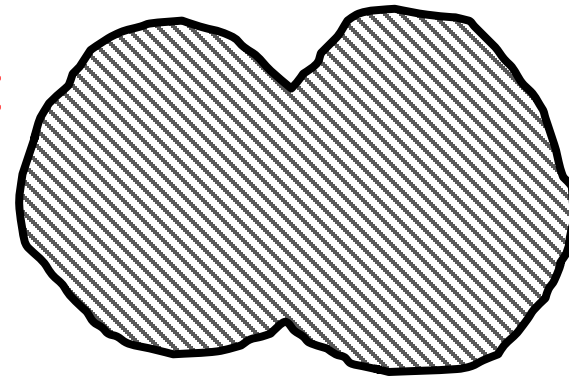
Sets

- A set is an unordered collection of objects (called elements).
- Important sets:
 - $\mathbb{N} = \{0, 1, 2, \dots\}$, the set of natural numbers.
 - \mathbb{R} , the set of real numbers.
 - $\emptyset = \{\}$, the empty set having no elements.
- Notation:
 - $X = \{2, 3, 5, 7, 11\}$ or $X = \{x \in \mathbb{N} : x \text{ is prime and } x < 12\}$.
- Operations:
 - $A \cap B = \{x : x \in A \text{ and } x \in B\}$
 - $A \cup B = \{x : x \in A \text{ or } x \in B\}$
 - $|A|$ is the number of elements of A .

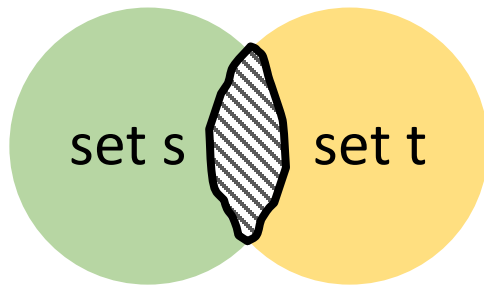
Set Theory



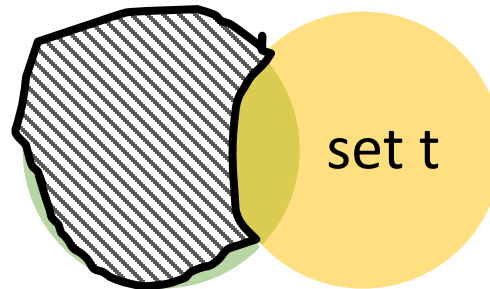
$s \cup t$



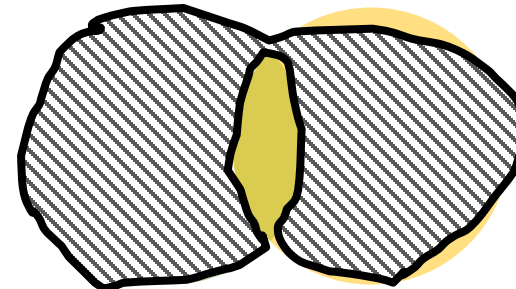
$s \cap t$



$s \setminus t$



$s \Delta t$



Set Operations in Python

- Just as in set theory we can perform common mathematical operations on sets.

Operation	Equiv.	Description
<code>len(s)</code>		number of elements in set s
<code>x in s</code>		test x for membership in s
<code>x not in s</code>		test x for non-membership in s
<code>s.issubset(t)</code>	$s \leq t$	test whether every element in s is in t
<code>s.issuperset(t)</code>	$s \geq t$	test whether every element in t is in s
<code>s.union(t)</code>	$s \mid t$	new set with elements from both s and t
<code>s.intersection(t)</code>	$s \& t$	new set with elements common to s and t
<code>s.difference(t)</code>	$s - t$	new set with elements in s but not in t
<code>s.symmetric_difference(t)</code>	$s \wedge t$	new set with elements in either s or t but not both
<code>s.copy()</code>		new set with a copy of s

Functions

- A **function** is a mapping f from a set X (the **domain**) to a set Y (the **codomain**) such that every $x \in X$ maps to a unique $y \in Y$.
- Important functions from \mathbb{R} to \mathbb{R} :
 - Power functions $f(x) = x$, $f(x) = x^2$, $f(x) = x^3$, etc.
 - Exponential functions $f(x) = 2^x$, $f(x) = (1.5)^x$, etc.
 - Logarithm (inverse of exponential) has a different domain.
- **Ceiling** rounds up to nearest integer, e.g. $\lceil 3.7 \rceil = 4$.
- **Floor** rounds down, e.g. $\lfloor 3.7 \rfloor = 3 = \lfloor 3 \rfloor$.

Basic Properties of Important Functions

- For $a > 1$ the exponential $f(x) = a^x$ is increasing and positive and satisfies $a^{x+y} = a^x a^y$ for all $x, y \in \mathbb{R}$.
- For $a > 1$ the logarithm \log_a is increasing and satisfies $\log_a(xy) = \log_a x + \log_a y$ for all $x, y > 0$.
- We write $\ln = \log_e$ and $\lg = \log_2$. Note that $\log_a x = \log_a b \log_b x$.
- Derivatives: $\frac{d}{dx} e^x = e^x, \frac{d}{dx} \ln x = \frac{1}{x}$

Sums

- A **sequence** is a function $f : \mathbb{N} \rightarrow \mathbb{R}$
- Notation: f_0, f_1, f_2, \dots where $f_i = f(i)$.
- Sum: $f_m + f_{m+1} + \dots + f_n = \sum_{i=m}^n f_i$
- Important sums:

$$\sum_{i=1}^n i = \frac{n(n+1)}{2}$$
$$\sum_{i=m}^n a^i = \frac{a^{n+1} - a^m}{a - 1}$$

Asymptotic Complexity

What is an Algorithm?

- **Algorithm:** An algorithm is a sequence of clearly stated rules that specify a step-by-step method for solving a given problem.
- The rules should be unambiguous and sufficiently detailed that they can be carried out without creativity.
- Examples of algorithms: a (sufficiently detailed) cake recipe, primary school method for multiplication of decimal integers; quicksort.
- Algorithms predate electronic computers by thousands of years (example: Euclid's greatest common divisor algorithm)
- A **program** is a sequence of computer instructions implementing the algorithm

Q: Why Should we analyze algorithms?

Why Should we analyze algorithms?

- Experience shows that enormously more performance gains can be achieved by optimizing the algorithm than by optimizing other factors such as:
 - processor
 - language
 - compiler
 - human programmer
- The analysis process often results in us discovering simpler algorithms.
- Many algorithms have parameters that must be set before implementation. Analysis allows us to set the optimal values.

Example: Fibonacci Sequence

This sequence is recursively defined by

$$F(n) = \begin{cases} n & \text{if } n = 0 \text{ or } n = 1; \\ F(n-1) + F(n-2) & \text{if } n \geq 2. \end{cases}$$

Example: Fibonacci Sequence

This immediately suggests a recursive algorithm.

Algorithm 1 Slow method for computing Fibonacci numbers

```
1: function SLOWFIB(integer  $n$ )  
2:   if  $n < 0$  then return 0  
3:   else if  $n = 0$  then return 0  
4:   else if  $n = 1$  then return 1  
5:   else return SLOWFIB( $n - 1$ ) + SLOWFIB( $n - 2$ )
```

The algorithm slowfib is obviously correct, but does a lot of repeated computation. With a small (fixed) amount of extra space, we can do better, by working from the bottom up instead of from the top down.

Example: Fibonacci Sequence

Algorithm 2 Fast method for computing Fibonacci numbers

```
1: function FASTFIB(integer  $n$ )
2:   if  $n < 0$  then return 0
3:   else if  $n = 0$  then return 0
4:   else if  $n = 1$  then return 1
5:   else
6:      $a \leftarrow 1$                 ▷ stores  $F(i)$  at bottom of loop
7:      $b \leftarrow 0$              ▷ stores  $F(i - 1)$  at bottom of loop
8:     for  $i \leftarrow 2$  to  $n$  do
9:        $t \leftarrow a$ 
10:       $a \leftarrow a + b$ 
11:       $b \leftarrow t$ 
12:   return  $a$ 
```

Even a bad implementation in a slow interpreted language on an ancient machine of fastfib will beat the best implementation of slowfib, once n becomes big enough.

Fibonacci Sample Code

Q: Implement fastfib and slowfib in Python. Which is faster for which values of n ? What is the maximum value of n for which each gives a result in a reasonable time?

How Efficient is your Code?

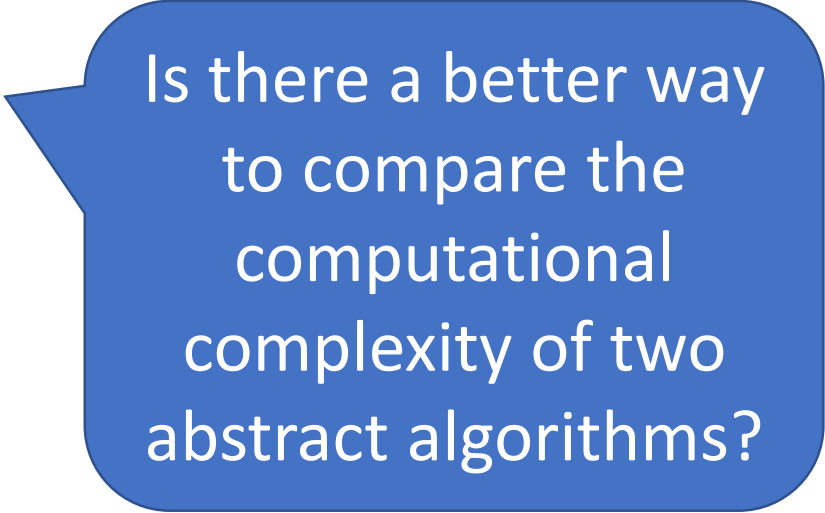
Evaluating an implementation?

Use empirical timing

How Efficient is your Code?

There are many contributors to computational complexity:

- Hardware: processor(s), memory, cache, etc.
- OS, version of Python, libraries, drivers
- Programs running in the background
- Implementation dependent
- Choice of input
- Which inputs to test



Is there a better way
to compare the
computational
complexity of two
abstract algorithms?

Algorithm Analysis

When we're considering algorithm computational complexity, we're interested with what happens as the **size of the input** to the algorithm grows:

- **Time:** How much longer does it run?
- **Space:** How much memory does it use?

Q: How can we answer these questions?

How do we Analyze Algorithms?

Evaluating an implementation?

Use empirical timing

Evaluating an algorithm?

Use asymptotic analysis

The goal is NOT to come up with an exact time, but an ESTIMATE...

Assumptions in Asymptotic Alg. Analysis

Basic (**elementary**) operations take **constant time**

- Arithmetic
- Assignment
- Access one array index
- Comparing two simple values ($\text{is } x < 3$)

Other operations are summations or products

- Consecutive statements are **summed**
- Loops are (cost of loop body) **X** (number of loops)

Running Time

The **total running time** $T(\text{input})$ of algorithm A on the input is the **number of elementary operations** used when input is fed into A.

Running time		Input size			
<i>Function</i>	<i>Notation</i>	10	100	1000	10^7
Constant	1	1	1	1	1
Logarithmic	$\log n$	1	2	3	7
Linear	n	1	10	100	10^6
“Linearithmic”	$n \log n$	1	20	300	7×10^6
Quadratic	n^2	1	100	10000	10^{12}
Cubic	n^3	1	1000	10^6	10^{18}
Exponential	2^n	1	10^{27}	10^{298}	$10^{3010296}$

Note: there are about 3×10^{18} nanoseconds in a century.

Q: Analyzing Algorithms

- Algorithm A takes n^2 elementary operations to sort a file of n lines, while Algorithm B takes $50 n \log n$. Which algorithm is better when $n = 10$?
- when $n = 10^6$? How do we decide which algorithm to use?

Q: Analyzing Algorithms

Algorithm Swapping two elements in an array

Require: $0 \leq i \leq j \leq n - 1$

function SWAP(array $a[0..n - 1]$, integer i , integer j)

$t \leftarrow a[i]$

$a[i] \leftarrow a[j]$

$a[j] \leftarrow t$

return a

► Running time?

Q: Analyzing Algorithms

Algorithm Finding the maximum in an array

```
function FINDMAX(array  $a[0..n-1]$ )
```

 $k \leftarrow 0$

- ▷ location of maximum so far

for $j \leftarrow 1$ **to** $n - 1$ **do**

if $a[k] < a[j]$ **then**

$$k = j$$
return k

► Running time?

Q: Analyzing Algorithms

Algorithm	Example: exponential change of variable in loop
------------------	---

```
 $i \leftarrow 1$   
while  $i \leq n$  do  
     $i \leftarrow 2 * i$   
    print  $i$ 
```

► Running time?

Fibonacci

Q: What do we know about the running time $T(n)$ of `slowfib` for term n of the Fibonacci sequence?

Analyzing Code

➤ What are the run-times for the following code?

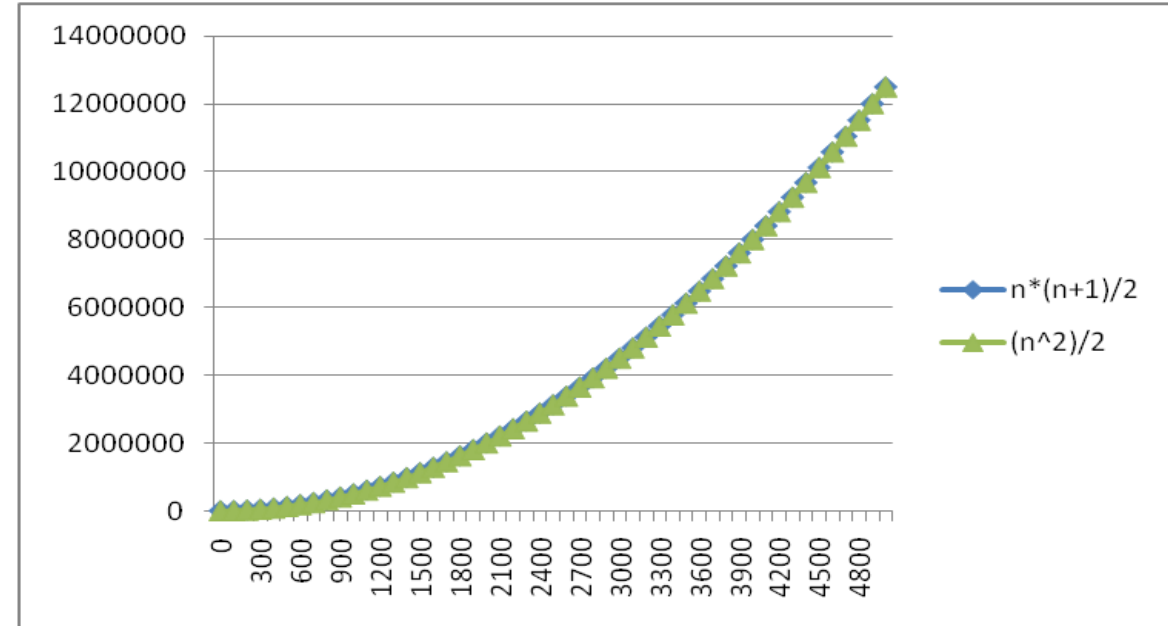
1. `for i in range(n):`
 `x = x + 1`

2. `for i in range(n):`
 `for j in range(n):`
 `x = x + 1`

3. `for i in range(n):`
 `for j in range(i+1):`
 `x = x + 1`

No Need to be so Exact

- **Constants do not matter**
 - Consider $6N^2$ and $20N^2$
 - When $N \gg 20$, the N^2 is what is driving the function's increase
- Lower-order terms are also less important
 - $n*(n+1)/2$ vs. just $n^2/2$
 - The linear term is inconsequential



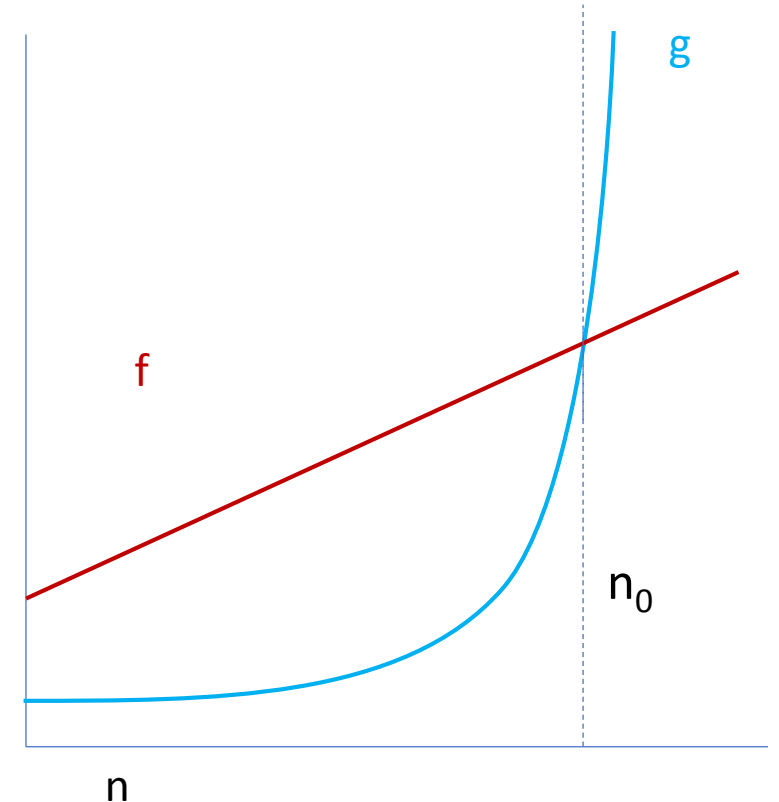
We need a better notation for performance that focuses on dominant terms only

Big-Oh Notation (Formally)

- Given two functions $f(n)$ & $g(n)$ for input n , we say $f(n)$ is in $O(g(n))$ iff there exist positive **constants c and n_0** such that

$$f(n) \leq c g(n) \text{ for all } n \geq n_0$$

- Eventually $g(n)$ is always an upper bound on $f(n)$ ignoring constants



The Gist of Big-Oh

➤ Consider only the most significant (dominant) term in the operation count and remove constant multipliers:

1. $5n + 3 \rightarrow O(n)$
2. $7n + 0.5n^2 + 2000 \rightarrow O(n^2)$
3. $300n + 12 + n \log n \rightarrow O(n \log n)$
4. $7n + 0.5n^2 + 2000 + 2^n \rightarrow O(2^n)$

More Examples

Q: For the following examples, identify if it is **true or false**?

1. $4+3n$ is $O(n)$
2. $n+2 \log n$ is $O(\log n)$
3. $\log n+2$ is $O(1)$
4. n^{50} is $O(1.1^n)$

Big Oh: Common Categories

From fastest to slowest

- $O(1)$ **constant**
- $O(\log n)$ **logarithmic**
- $O(n)$ **linear**
- $O(n \log n)$ "n log n" or **"linearithmic"**
- $O(n^2)$ **quadratic**
- $O(n^3)$ **cubic**
- $O(n^k)$ **polynomial** (where k is constant)
- $O(k^n)$ **exponential** (where constant $k > 1$)

Complexity Analysis of Code

```
for i in range(N):  
    sum += i
```

```
for i in range(N):  
    for j in range(N):  
        sum += i + j
```

```
for i in range(N):  
    for j in range(N):  
        for k in range(N):  
            sum += i + j + k;
```

Worst-Case Analysis

- In general, we are interested in three types of performance
 - Best-case / Fastest
 - Average-case
 - Worst-case / Slowest
- When determining worst-case, we tend to be pessimistic
 - If there is a **conditional**, count the branch that will run the slowest
 - This will give a loose bound on how slow the algorithm may run

Pros and Cons of Worst and Average Analysis?

- Worst-case bounds are valid for all instances: this is important for mission-critical applications.
- Worst-case bounds are often easier to derive mathematically.
- Worst-case bounds often hugely exceed typical running time and have little predictive or comparative value.
- Average-case running time is often more realistic. Quicksort is a classic example.
- Average-case analysis requires a good understanding of the probability distribution of the inputs.
- Average-case analysis is often more practically useful, provided the algorithm will be run on “random” data.

10 Minute Break

Analysis of Sorting Algorithms

Searching

- Given we have an array, how do we find a particular value?

32	17	87	79	95	76	24	1	32
----	----	----	----	----	----	----	---	----

- Go through indices until
 - Value is found → **true**
 - Reach the end → **false**

Q: What is the complexity in big O notation?

Searching a sorted list

- We can sort any array:

1	17	23	24	32	76	79	87	95
---	----	----	----	----	----	----	----	----

- Repeatedly split search space in half
- Logarithmic $O(\log n)$: repeatedly split search space in half
 - How many times k can we split before we reach singleton? $\frac{1}{2}^k n = 1$
 - $\frac{1}{2}^k = 1/n \rightarrow 2^k = n \rightarrow k = \log n$ operations until we terminate!

Naïve sorting: Bubble Sort

- We want to sort the following array of n elements:

32	77	87	79	95	76	24	1	31
----	----	----	----	----	----	----	---	----

- Repeatedly loop to find next smallest element:
 - for $i = 0..n-1$
 - Find smallest element index j from indices $i..length-1$ then $swap(i,j)$

- First two iterations:

1	77	87	79	95	76	24	32	31
1	24	87	79	95	76	77	32	31

- Complexity is $O(n^2)$: why?
 - Have a double nested loop. Can we do better?

We can do better: start by merging sorted lists

➤ Complexity of merging two sorted lists of length n ?

List A:

1	17	23	43
---	----	----	----

List B:

2	19	23	31
---	----	----	----

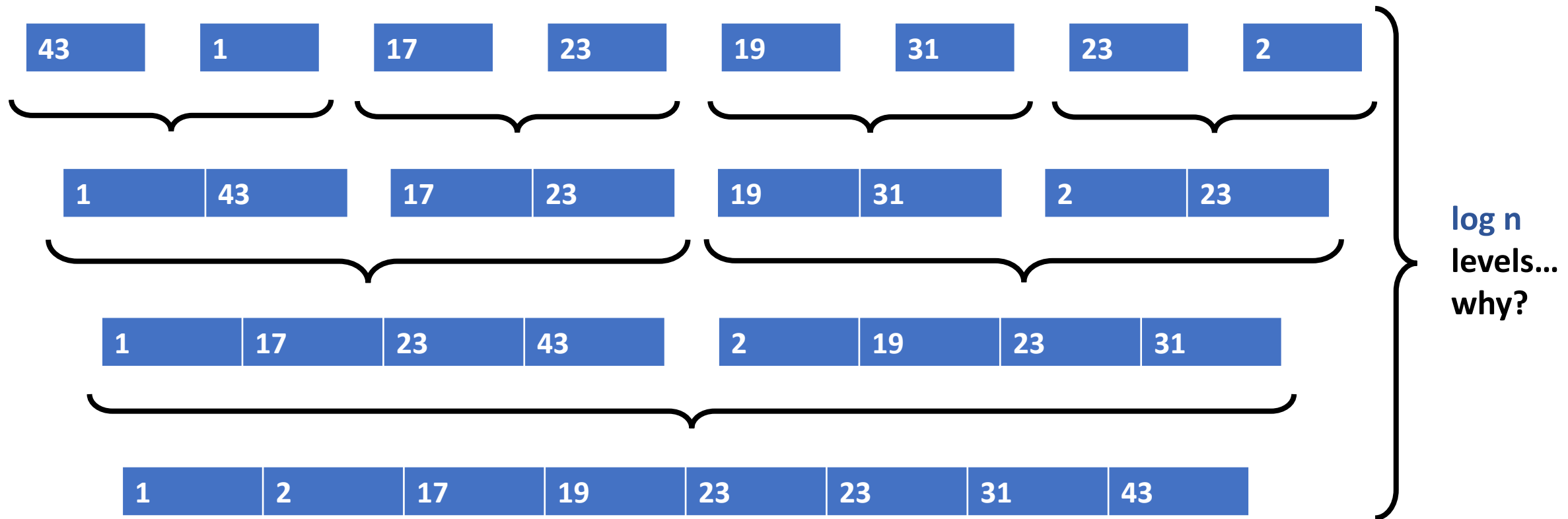
Merger:

1	2	17	19	23	23	31	43
---	---	----	----	----	----	----	----

➤ Linear $O(n)$: just maintain indices of next element in each list

Merge Sort

- Subdivide until singletons, then repeatedly merge up to full array size n



- Overall complexity? $O(n \log n)$: n operations at each level, $\log n$ levels

Dictionary Abstract Data Type

Dictionary

- An abstract data type that supports operations to **insert**, **find**, and **delete** an element with given **search key**.
- Used for databases. Other names are **table ADT** and **associative array**.
- There are many ways in which this could be implemented:
 - unsorted list;
 - sorted list;
 - binary search tree;
 - hash table.

Unsorted List

- Inserting an element is constant time $O(1)$.
- The only way to find an element is to check each element.
- This takes time in $O(n)$ for any reasonable implementation.
- Also, deletion is $O(n)$ because first we should find it and then other elements must be pushed to the left (array).

Sorted List

- In a sorted list, inserting an element is harder because you have to find the right place and therefore it is $O(n)$.
- In a sorted list, finding an element is easier because the order allows us to perform a binary search which is $O(\log n)$
- Again, deletion is $O(n)$ because other elements must be pushed to the left (array).

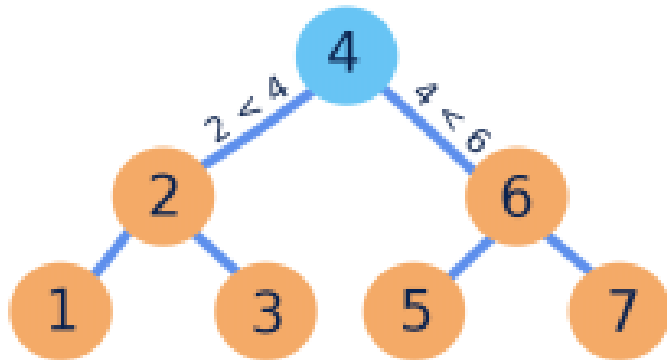
Efficiency of various implementations

Table: Average case running time (asymptotic order)

Data structure	Insert	Delete	Find
Unsorted list (array)	1	n	n
Sorted list (array)	n	n	$\log n$

Efficiency of various implementations

Binary Search Tree



Source: [Image from Tamara Nelson-Fromm, UIUC](#)

Table: Average case running time (asymptotic order)

Data structure	Insert	Delete	Find
Unsorted list (array)	1	n	n
Sorted list (array)	n	n	$\log n$
Binary search tree	$\log n$	$\log n$	$\log n$

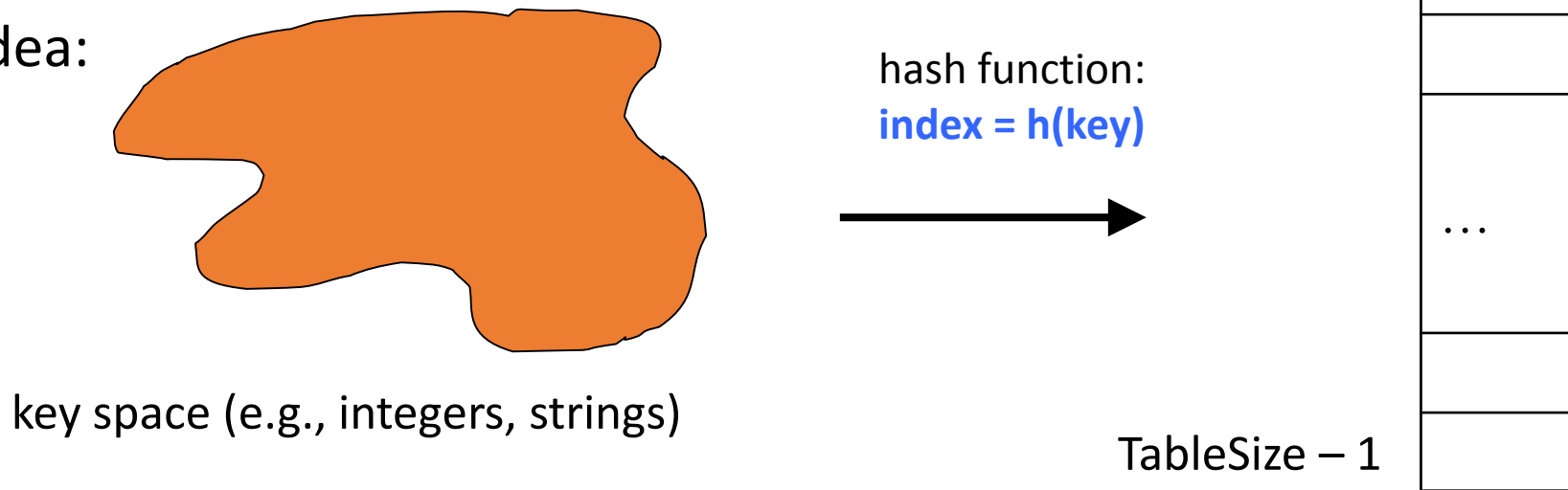
Q: Can we do better???

Hash Functions and Tables

- Order irrelevant, aim for constant-time (i.e., $O(1)$) **find**, **insert**, and **delete**
 - “On average” under some often-reasonable *assumptions*

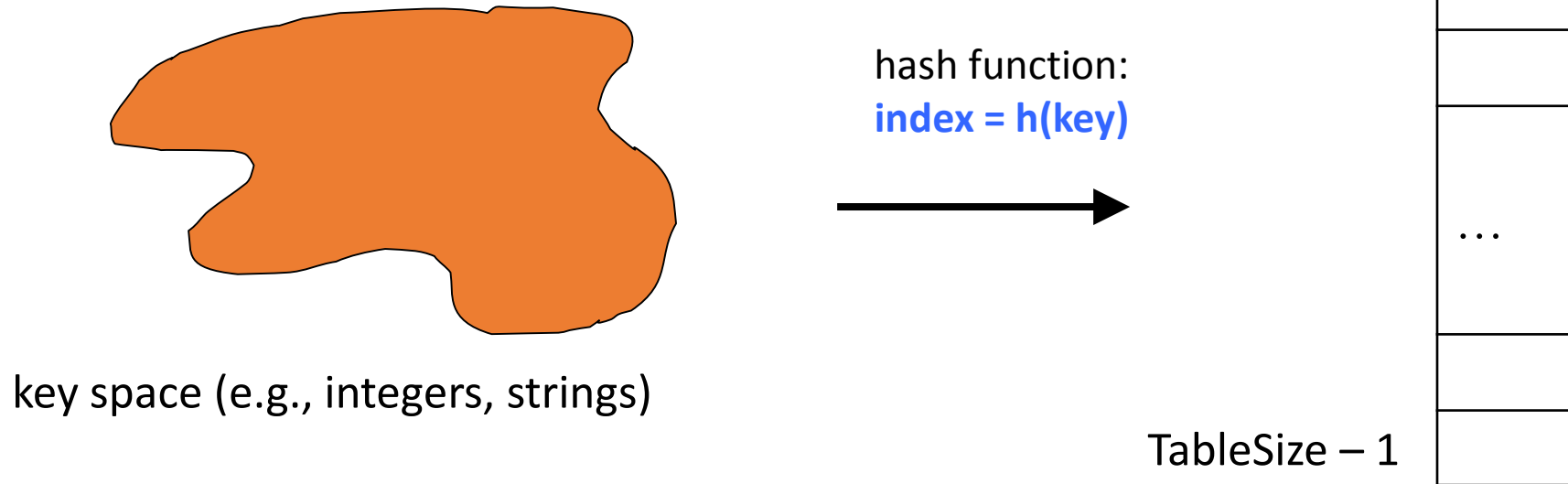
- A hash table is an array of some fixed size
 - Maps **keys**->**indices** which contain values

- Basic idea:



Hash Functions

- An ideal hash function:
- Fast to compute
- **Design so that two keys rarely hash to the same index**
 - Must happen if elements stored exceeds table size
 - Can lead to *collisions*, more on this later



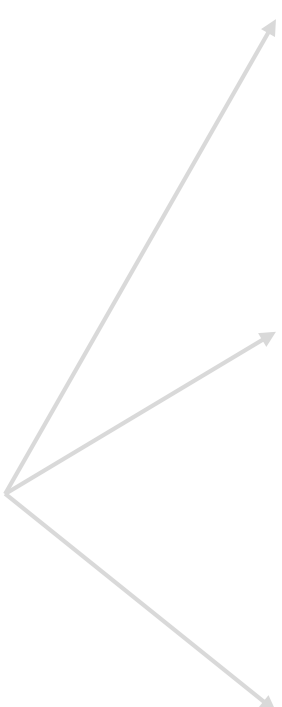
Example: Hash Strings

Key space $K = s_0s_1s_2\dots s_{k-1}$

where s_i are chars: $s_i \in [0, 256]$

Some choices:

Which ones best avoid collisions?


$$h(K) = (s_0) \% \text{TableSize}$$

$$h(K) = \left(\sum_{i=0}^{k-1} s_i \right) \% \text{TableSize}$$

$$h(K) = \left(\sum_{i=0}^{k-1} s_i \cdot 37^i \right) \% \text{TableSize}$$

Collision Resolution

- **Collision**: When two keys map to the same location in the hash table
- We try to avoid it, but cannot if number-of-keys exceeds table size
- So hash tables should support **collision resolution**
 - Ideas?

Collision Resolution via Open Addressing

- Uses no extra space – every element is stored in the hash table
- If a key k hashes to a value $h(k)$ that is already occupied, we probe (look for an empty space).
 - The most common probing method is **linear probing**, which moves left one index at a time, wrapping around, if necessary, until it finds an empty address

Example: Linear Probing

Table 3.3: Open addressing with linear probing (OALP).

Data [key,value]	Hash: key/10	Table address	Comments
[20,A]	2	2	
[15,B]	1	1	
[45,C]	4	4	
[87,D]	8	8	
[39,E]	3	3	
[31,F]	3	0	try 3, 2, 1, 0
[24,G]	2	9	try 2, 1, 0, 9

Open Addressing: Double Hashing

- Another method is **double hashing**.
- Use a second hash $\Delta(k) = t$ to find a place t and if it is occupied again move to the left by a fixed step size t , wrapping around, if necessary, until we find an empty address.

Example: Double Hashing

if the hash function is given by $\Delta(k) = (h(k) + k) \bmod 10$.

Table 3.4: Open addressing with double hashing (OADH).

Data [key,value]	Hash: key/10	Table address	Comments
[20,A]	2	2	using $\Delta(31) = 4$ using $\Delta(24) = 6$
[15,B]	1	1	
[45,C]	4	4	
[87,D]	8	8	
[39,E]	3	3	
[31,F]	3	9	
[24,G]	2	6	

Collision Resolution via Chaining

- Chaining uses an “overflow” list for each element in the hash table.
- Elements that hash to the same slot are placed in a list.
- A drawback is the additional space overhead. Also, the distribution of sizes of lists turns out to be very uneven.

Separate Chaining

0	/
1	/
2	/
3	/
4	/
5	/
6	/
7	/
8	/
9	/

Chaining:

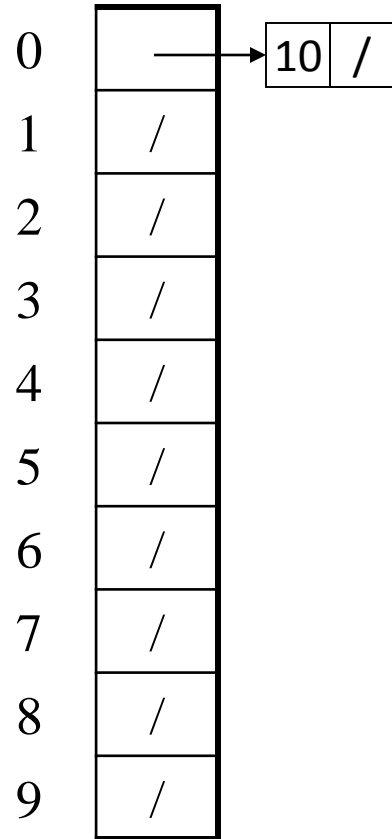
All keys that map to the same table location are kept in a list (a.k.a. a “chain” or “bucket”)

As easy as it sounds

Example:

insert 10, 22, 107, 12, 42
with mod hashing
and **TableSize** = 10

Separate Chaining



Chaining:

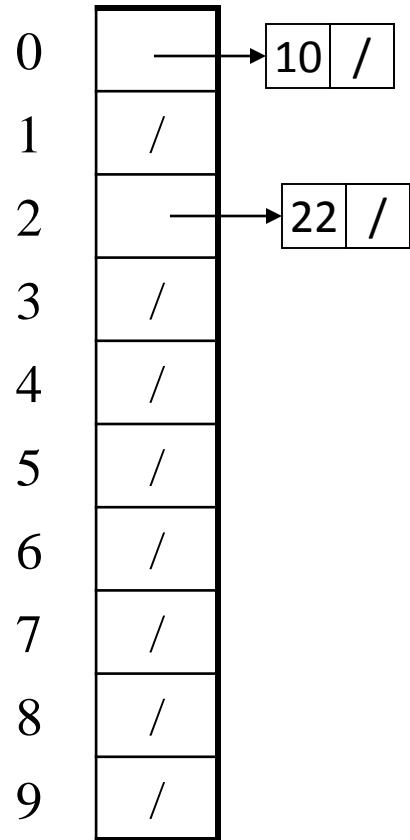
All keys that map to the same table location are kept in a list (a.k.a. a “chain” or “bucket”)

As easy as it sounds

Example:

insert 10, 22, 107, 12, 42
with mod hashing
and **TableSize** = 10

Separate Chaining



Chaining:

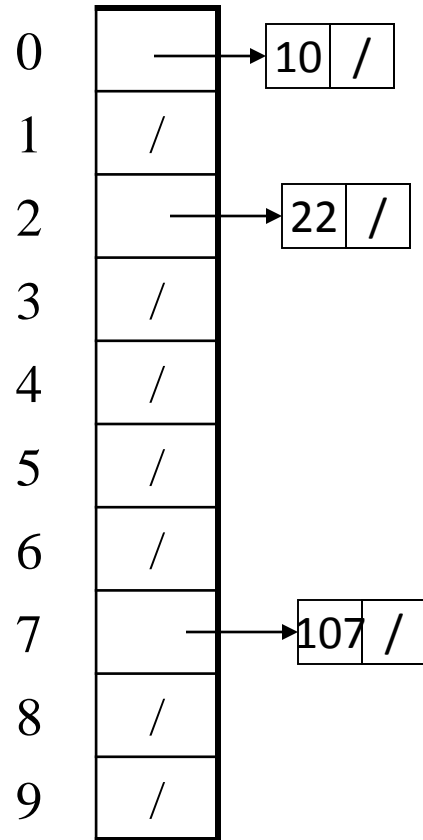
All keys that map to the same table location are kept in a list (a.k.a. a “chain” or “bucket”)

As easy as it sounds

Example:

insert 10, 22, 107, 12, 42
with mod hashing
and **TableSize** = 10

Separate Chaining



Chaining:

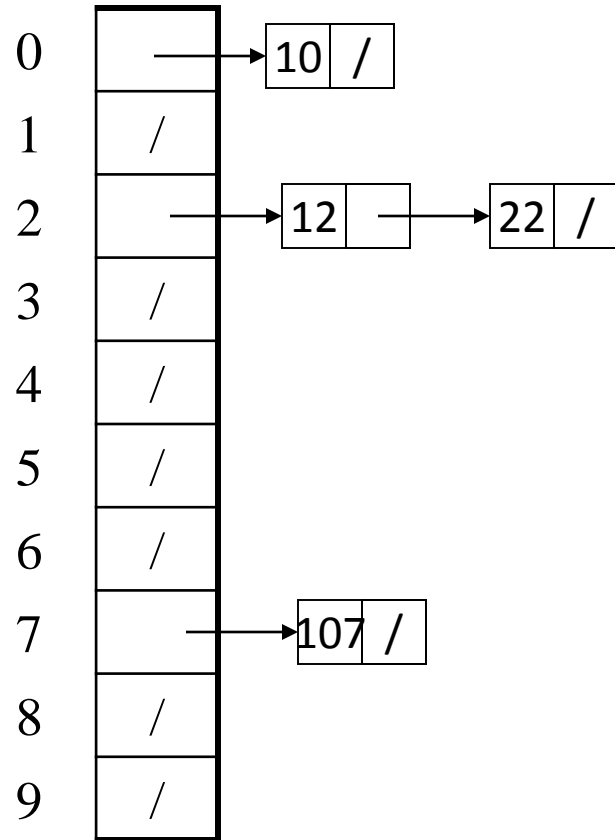
All keys that map to the same table location are kept in a list (a.k.a. a “chain” or “bucket”)

As easy as it sounds

Example:

insert 10, 22, 107, 12, 42
with mod hashing
and **TableSize** = 10

Separate Chaining



Chaining:

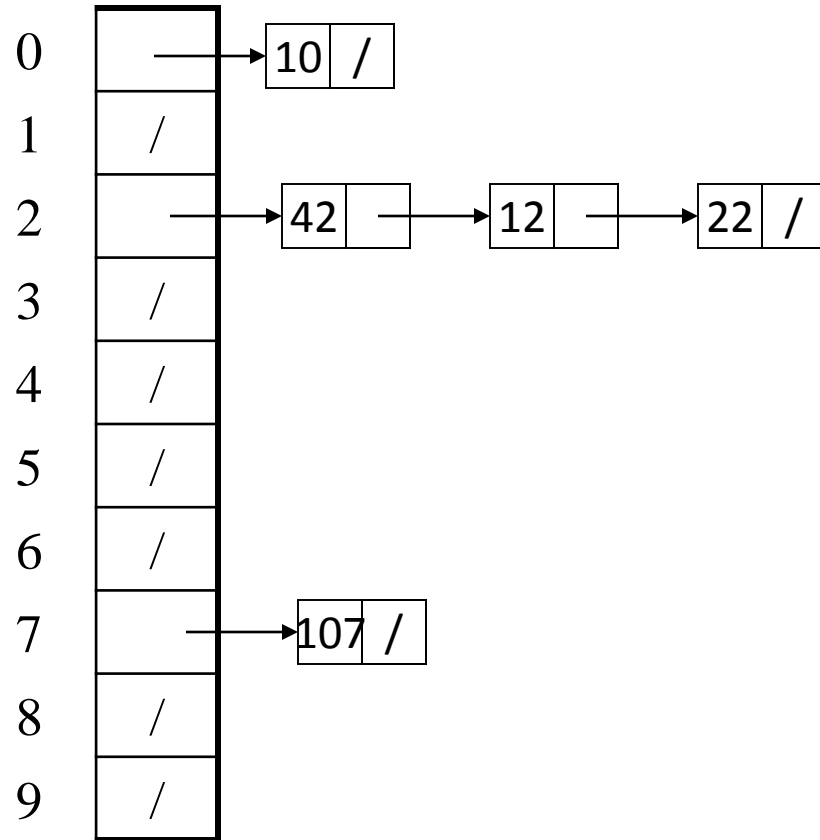
All keys that map to the same table location are kept in a list (a.k.a. a “chain” or “bucket”)

As easy as it sounds

Example:

insert 10, 22, 107, 12, 42
with mod hashing
and **TableSize** = 10

Separate Chaining



Chaining:

All keys that map to the same table location are kept in a list (a.k.a. a “chain” or “bucket”)

As easy as it sounds

Example:

insert 10, 22, 107, 12, 42
with mod hashing
and **TableSize** = 10

Analysis of Hashing

- Q: When hashing n keys into a table with m slots, how often do you think collisions occur when n is much smaller than m ?

Analysis of Hashing

- We count the cost (running time) by number of key comparisons.
- We often use the simple uniform hashing model. That is, each of the n keys is equally likely to hash into any of the m slots. So we can consider a “balls in bins” model.
- If n is much smaller than m , collisions will be few and most slots will be empty. If n is much larger than m , collisions will be many and no slots will be empty. The most interesting behaviour is when m and n are of comparable size.
- **Define the load factor to be $\lambda := n/m$**

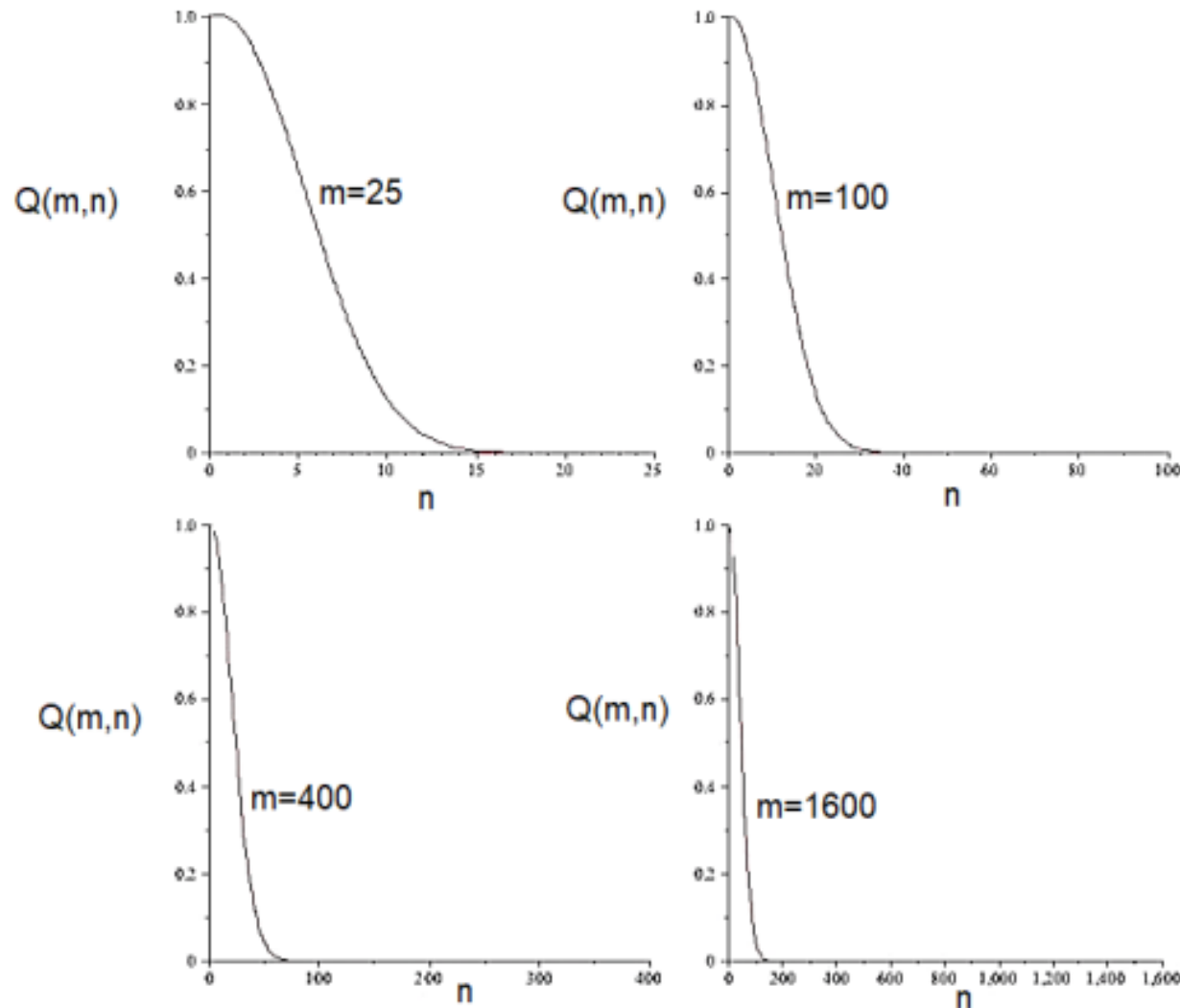
How often do collisions occur?

- The probability of no collisions when n balls are thrown into m bins uniformly at random is $Q(m, n)$.
- Note that when the load factor is very small $\lambda \rightarrow 0$, collisions are unlikely (for example $Q(m, 0) = 1$ and $Q(m, 1) = 1$).
- At the other extreme case of $\lambda > 1$, collisions are absolutely certain (i.e. $Q(m, n) = 0$).

How often do collisions occur?

- Birthday Paradox: If there are 23 or more people in a room, the chance is greater than 50% that two or more of them have the same birthday.
- When a table with 365 slots is only $23/365 = 6.3\%$ full, the next attempt to insert an element is slightly more likely to result in a collision than not.

How often do collisions occur?



Actually, the point where collision become almost certain scales with \sqrt{m} . This means that for example, in a table with 1600 slots, we almost certainly get a collision after filling 128 elements (table being 92% empty).

Efficiency of Various Implementation

Table: Average case running time (asymptotic order)

Data structure	Insert	Delete	Find
Unsorted list (array)	1	n	n
Sorted list (array)	n	n	$\log n$
Binary search tree	$\log n$	$\log n$	$\log n$
Hash table (chaining)	λ	λ	λ

Table: Worst case running time (asymptotic order)

Data structure	Insert	Delete	Find
Unsorted list (array)	1	n	n
Sorted list (array)	n	n	$\log n$
Binary search tree	n	n	n
Hash table (chaining)	n	n	n

Hashing in Practice (as of Dec 2018)

- Java Collections Framework uses chaining to implement HashMap, resizing when $\lambda > 0.75$, and table size a power of 2.
- C++ uses chaining to implement unordered map, resizing when $\lambda > 1$, and prime table size.
- C# uses chaining, resizing when $\lambda > 1$, and prime table size.
- Python uses open addressing, resizing when $\lambda > 0.66$, and table size a power of 2.

Custom Hash Function

- Worst-case scenario, too many collisions can take us from $O(1)$ to $O(n)$
- Depending on your algorithm or data, you may need to use a different hash function or design your own
- Custom designed hash functions are very common

(polymorphism)
__hash__ → used to apply a hash
function: **index = h(key)**

Next Time

- Reading assignment 2 Due - Jan. 24 at 21:00
 - Pages 257-264 (page numbers from the pdf file) Section 8.1 in Chapter 8 of [“Mathematics for Machine Learning” by Marc P. Deisenroth et al., 2020](#)
 - Complete a quiz on Quercus and submit it by the deadline
- Week 2 Lab on Thursday Jan 20: Tutorial 1
 - Basic Data Science
- Week 3 Lecture on Tuesday Jan 25 – Foundations of Learning
- Project 1 Due - Feb. 4 at 23:00