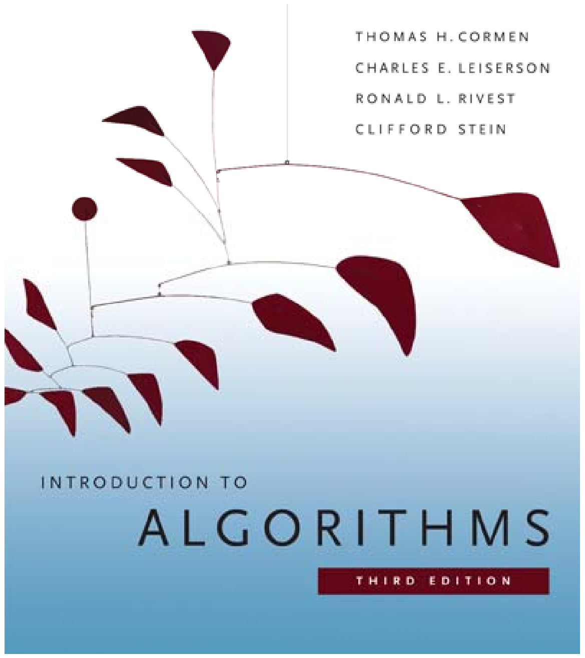
6.006- *Introduction to Algorithms*


THOMAS H. CORMEN
CHARLES E. LEISERSON
RONALD L. RIVEST
CLIFFORD STEIN

INTRODUCTION TO
ALGORITHMS
THIRD EDITION

Lecture 5

Prof. Manolis Kellis

Unit #2 – Genomes, Hashing, and Dictionaries

Unit	Pset	Week	Date	Lecture (Tuesdays and Thursdays)		Recitation (Wed and Fri)	
Intro	PS1	1	Tue Feb 01	1	Introduction and Document Distance	1	Python and Asymptotic Complexity
Binary	Out: 2/1		Thu Feb 03	2	Peak Finding Problem	2	Peak Finding correctness & analysis
Search	Due: Mon 2/14	2	Tue Feb 08	3	Scheduling and Binary Search Trees	3	Binary Search Tree Operations
Trees	HW lab: Sun 2/13		Thu Feb 10	4	Balanced Binary Search Trees	4	Rotations and AVL tree deletions
Hashing	PS2 Out: 2/15	3	Tue Feb 15	5	Hashing I : Chaining, Hash Functions	5	Hash recipes, collisions, Python dicts
	Due: Mon 2/28		Thu Feb 17	6	Hashing II : Table Doubling, Rolling Hash	6	Probability review, Pattern matching
	HW lab:Sun 2/27	4	Tue Feb 22	-	President's Day - Monday Schedule - No Class	-	No recitation
			Thu Feb 24	7	Hashing III : Open Addressing	7	Universal Hashing, Perfect Hashing
Sorting	PS3. Out: 3/1	5	Tue Mar 01	8	Sorting I : Insertion & Merge Sort, Master Theorem	8	Proof of Master Theorem, Examples
	Due: Mon 3/7		Thu Mar 03	9	Sorting II : Heaps	9	Heap Operations
	HW lab: Sun 3/6	6	Tue Mar 08	10	Sorting III: Lower Bounds, Counting Sort, Radix Sort	10	Models of computation
			Wed Mar 09	Q1	Quiz 1 in class at 7:30pm. Covers L1-R10. Review Session on Tue 3/8 at 7:30pm.		
Graphs	PS4. Out: 3/10		Thu Mar 10	11	Searching I: Graph Representation, Depth-1st Search	11	Strongly connected components
and	Due: Fri 3/18	7	Tue Mar 15	12	Searching II: Breadth-1st Search, Topological Sort	12	Rubik's Cube Solving
Search	HW lab:W 3/16		Thu Mar 17	13	Searching III: Games, Network properties, Motifs	13	Subgraph isomorphism
Shortest	PS5	8	Tue Mar 29	14	Shortest Paths I: Introduction, Bellman-Ford	14	Relaxation algorithms
Paths	Out: 3/29		Thu Mar 31	15	Shortest Paths II: Bellman-Ford, DAGs	15	Shortest Path applications
	Due: Mon 4/11	9	Tue Apr 05	16	Shortest Paths III: Dijkstra	16	Speeding up Dijkstra's algorithm
	HW lab:Sun 4/10		Thu Apr 07	17	Graph applications, Genome Assembly	17	Euler Tours
Dynamic	PS6	10	Tue Apr 12	18	DP I: Memoization, Fibonacci, Crazy Eights	18	Limits of dynamic programming
Program	Out: Tue 4/12		Wed Apr 13	Q2	Quiz 2 in class at 7:30pm. Covers L11-R17. Review Session on Tue 4/13 at 7:30pm.		
ming	Due: Fri 4/29		Thu Apr 14	19	DP II: Shortest Paths, Genome sequence alignment	19	Edit Distance, LCS, cost functions
	HW lab:W 4/27	11	Tue Apr 19	-	Patriot's Day - Monday and Tuesday Off	-	No recitation
			Thu Apr 21	20	DP III: Text Justification, Knapsack	20	Saving Princess Peach
		12	Tue Apr 26	21	DP IV: Piano Fingering, Vertex Cover, Structured DP	21	Phylogeny
Numbers	PS7 out Thu4/28		Thu Apr 28	22	Numerics I - Computing on large numbers	22	Models of computation return!
Pictures	Due: Fri 5/6	13	Tue May 3	23	Numerics II - Iterative algorithms, Newton's method	23	Computing the nth digit of π
(NP)	HW lab: Wed 5/4		Thu May 5	24	Geometry: Line sweep, Convex Hull	24	Closest pair
		14	Tue May 10	25	Complexity classes, and reductions	25	Undecidability of Life
Beyond			Thu May 12	26	Research Directions (15 mins each) + related classes		
		15	Finals week	Q3	Final exam is cumulative L1-L26. Emphasis on L18-L26. Review Session on Fri 5/13 at 3pm		

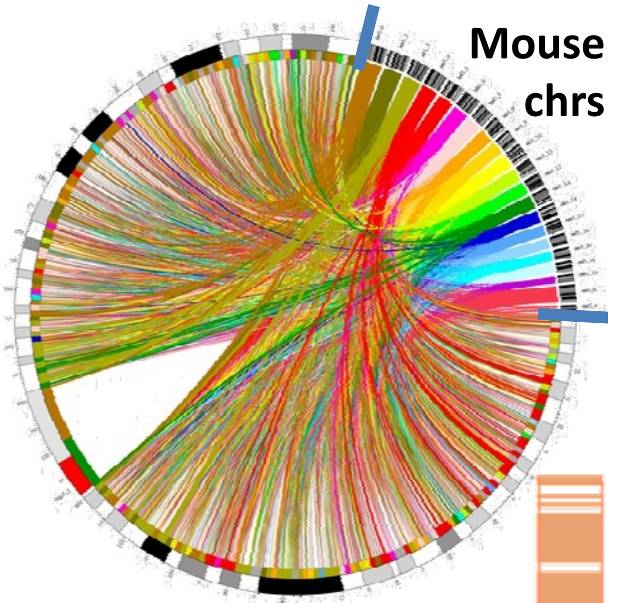
(hashing out…) Our plan ahead

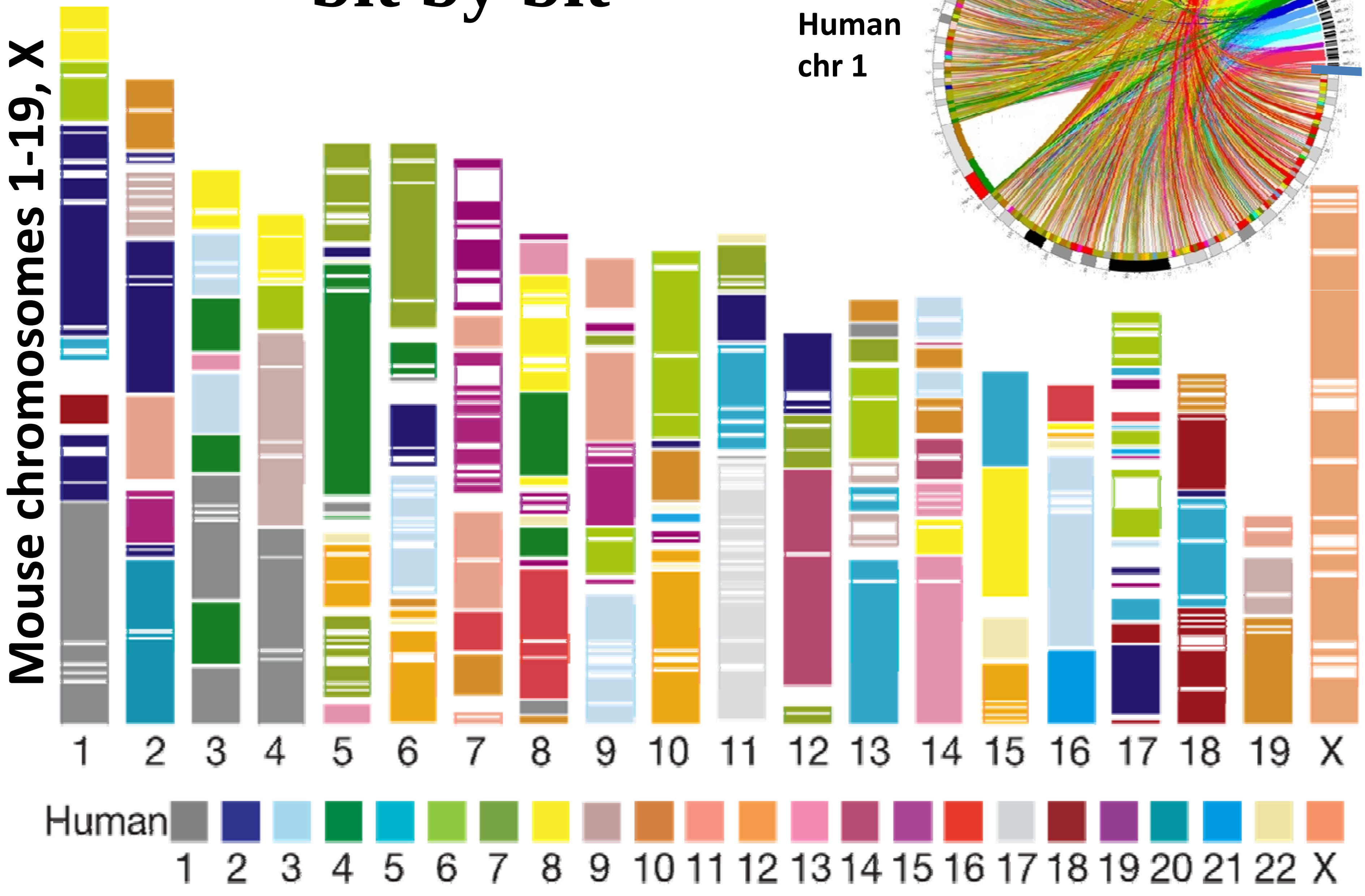
- **Today: Genomes, Dictionaries, and Hashing**
  - Intro, basic operations, collisions and chaining
  - Simple uniform hashing assumption
  - Hash functions, python implementation
- **Thursday: Speeding up hash tables**
  - Faster comparison: Signatures
  - Faster hashing: Rolling Hash
- **Next week: Space issues**
  - Dynamic resizing and amortized analysis
  - Open addressing, deletions, and probing

Our plan for today: Hashing I

- **Today: Genomes, Dictionaries, and Hashing**
  - ➢Matching genome segments
  - ❑Introduction to dictionaries
  - ❑Hash function: definition
  - ❑Resolving collisions with chaining
  - ❑Simple uniform hashing assumption
  - ❑Hash functions in practice: mod / mult
  - ❑Python implementation
- **Thursday: Speeding up hash tables**
- **Next week: Space issues**

Comparing two genomes bit by bit





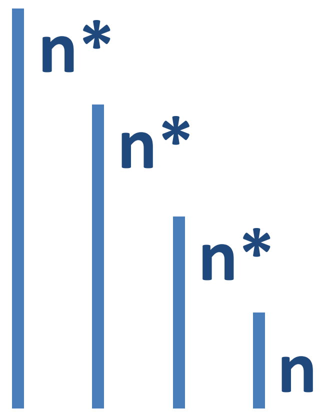
DNA matching: All about strings

- How to find ‘corresponding’ pieces of DNA
- Given two DNA sequences
  - Strings over 4-letter alphabet
- Find longest sub**string** that appears in both
  - Algorithm vs. Arithmetic
  - Algorithm vs. Arithmetic
  - L19: Sub**sequence** - much harder (e.g. Algorithm)
- Other applications:
  - Plagiarism detection
  - Word autocorrect
  - Jeopardy!

Naïve Algorithm

- Say strings S and T of length n
- For L = n downto 1

```
➢for all length L substrings X1 of S
    ➢for all length L substrings X2 of T
        ➢if X1=X2, return L
```



- Runtime analysis
  - n candidate lengths
  - n strings of that length in X1
  - n strings of that length in X2
  - L time to compare the strings
  - Total runtime: $\Omega(n^4)$

Improvement 1: Binary Search on L

- Start with L=n/2
- for all length L substrings X1 of S
- for all length L substrings X2 of T
- if X1=X2, success, try larger L
  if failed, try smaller L

- Runtime analysis

  $\Omega(n^4) \rightarrow \Omega(n^3 \log n)$

Improvement 2: Python Dictionaries

- For every possible length L=n,…,1
  - Insert all length L substrings of S into a dictionary
  - For each length L substring of T, check if it exists in dictionary

- Possible lengths for outer loop: n
- For each length:
  - at most n substrings of S inserted into dictionary, each insertion takes time O(1) * L (L is paid because we have to read string to insert it)
  - at most n substrings of T checked for existence inside dictionary, each check takes time O(1) * L
  - Overall time spent to deal with a particular length L is O(Ln)
- Hence overall $O(n^3)$
- With binary search on length, total is $O(n^2 \log n)$
- “Rolling hash” dictionaries improve to O(n log n) (next time)

Our plan for today: Hashing I

- **Today: Genomes, Dictionaries, and Hashing**
  - ☑Matching genome segments
  - ➢Introduction to dictionaries
  - ❑Hash function: definition
  - ❑Resolving collisions with chaining
  - ❑Simple uniform hashing assumption
  - ❑Hash functions in practice: mod / mult
  - ❑Python implementation
- **Thursday: Speeding up hash tables**
- **Next week: Space issues**

Dictionaries: Formal Definition

- It is a **set** containing **items**; each item has a **key**
- what keys and items are is quite flexible
- Supported Operations:
  - **Insert(*key, item*):** add *item* to set, indexed by *key*
  - **Delete(*key*):** delete item indexed by *key*
  - **Search(*key*):** return the item corresponding to the given *key*, if such an item exists
  - **Random_key():** return a random key in dictionary
- **Assumption**: every item has its own key (or that inserting new item clobbers old
- **Application** (and origin of name): Dictionaries
  - *Key* is word in English, *item* is word in French

Dictionaries are everywhere

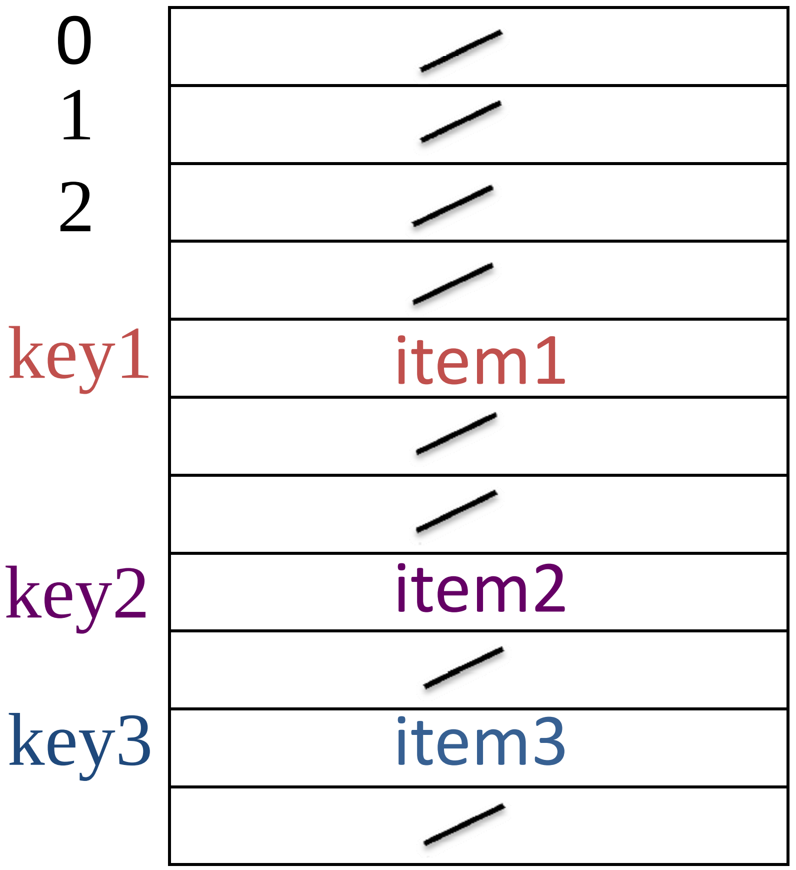
- Spelling correction
  - *Key* is misspelled word, *item* is correct spelling
- Python Interpreter
  - Executing program, see a variable name (*key*)
  - Need to look up its current assignment (*item*)
- Web server
  - Thousands of network connections open
  - When a packet arrives, must give to right process
  - *Key* is source IP address of packet, *item* is handler

Implementation

- use BSTs!
  - can keep keys in a BST, keeping a pointer from each key to its value
  - O(log n) time per operation
- Often not fast enough for these applications!
- Can we beat BSTs?

if only we could do all operations in O(1)…

Dictionaries: Attempt #1

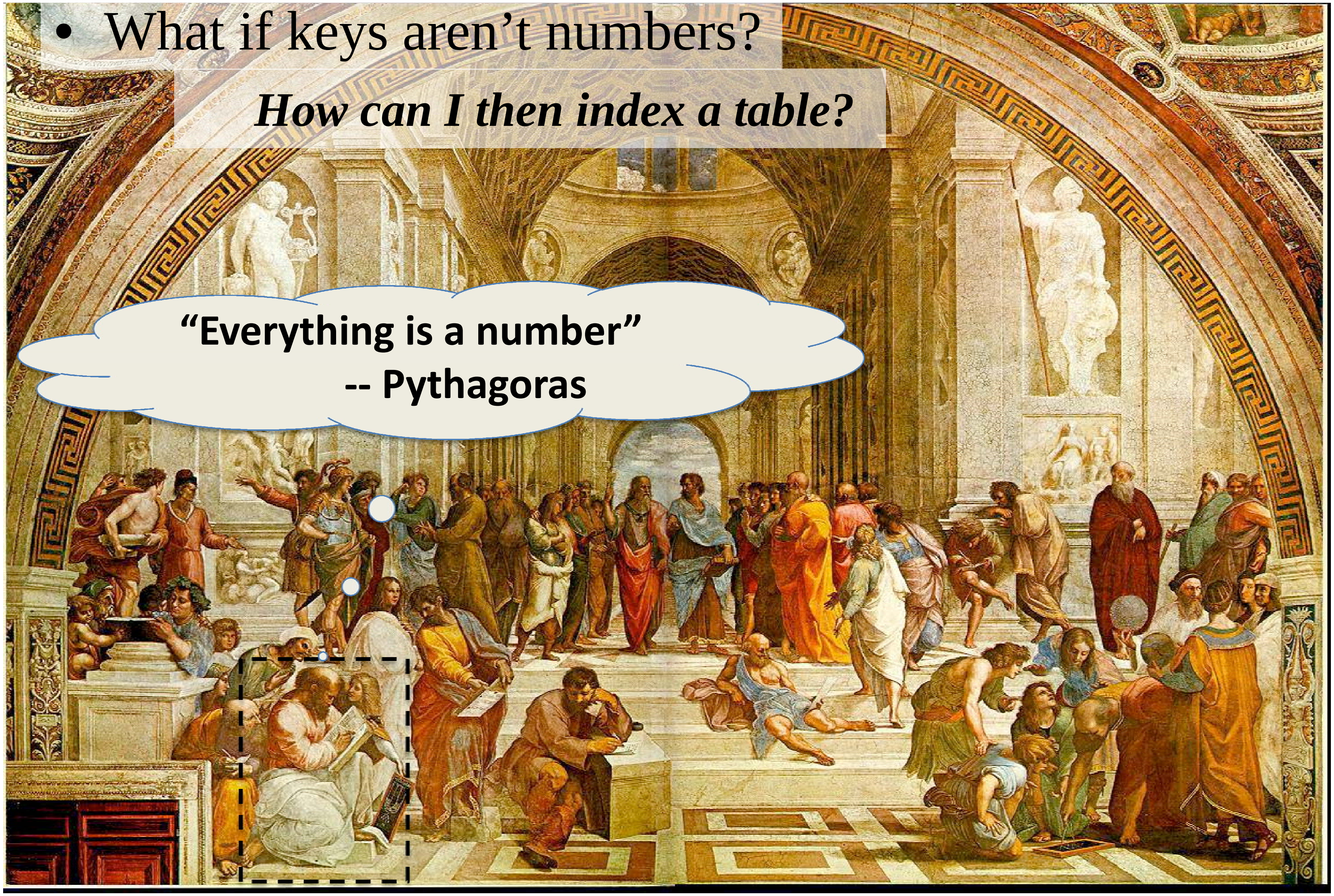


- Forget about BSTs..
- Use table, indexed by keys!

Problems…

- What if keys aren't numbers?

How can I then index a table?



Interpreting words as numbers

- What if keys aren't numbers?
  - Anything in the computer is a sequence of bits
  - So we can pretend it's a number
- Example: English words
  - 26 letters in alphabet

    $\Rightarrow$ can represent each with 5 bits
  - Antidisestablishmentarianism has 28 letters
  - 28*5 = 140 bits
  - So, store in array of size 2^{140} ….oops
- Isn't this too much space for 100,000 words?

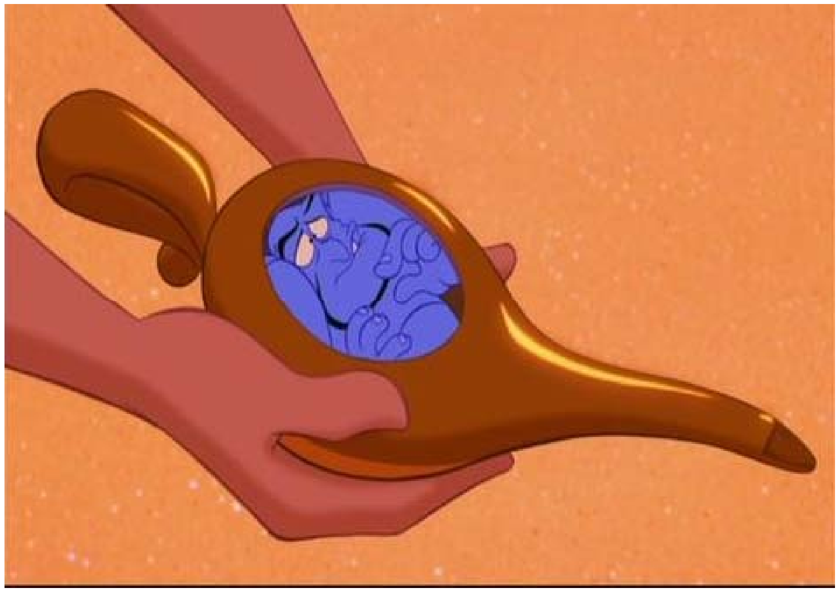
Our plan for today: Hashing I

- **Today: Genomes, Dictionaries, and Hashing**
  - ☑ Matching genome segments
  - ☑ Introduction to dictionaries
  - ➢ Hash function: definition
  - ❑ Resolving collisions with chaining
  - ❑ Simple uniform hashing assumption
  - ❑ Hash functions in practice: mod / mult
  - ❑ Python implementation
- **Thursday: Speeding up hash tables**
- **Next week: Space issues**

Hash Functions

- Exploit sparsity
  - Huge universe U of possible keys
  - But only n keys actually present
  - Want to store in table (array) of size m~n
- Define hash function $h:U \rightarrow \{1..m\}$
  - Filter key k through h( ) to find table position
  - Table entries are called buckets
- Time to insert/find key is
  - Time to compute h (generally length of key)
  - Plus one time step to look in array

The ‘magic’ of hash functions



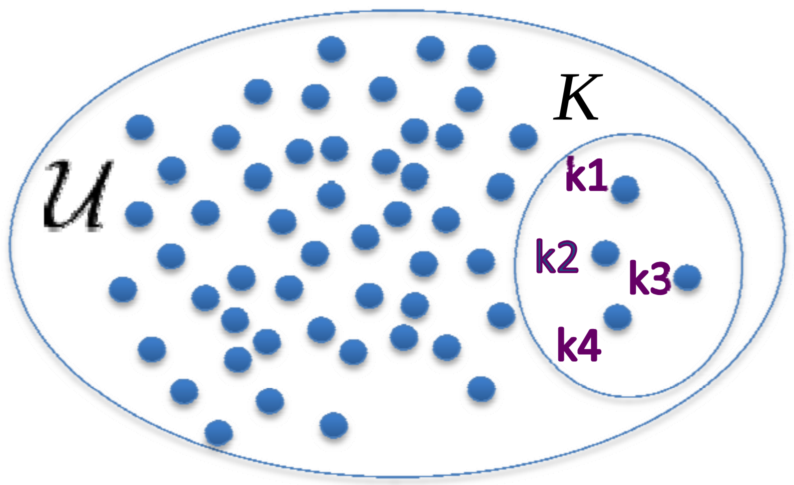
PHENOMENAL COSMIC POWERS!!

→

itty bitty living space

With apologies to Disney

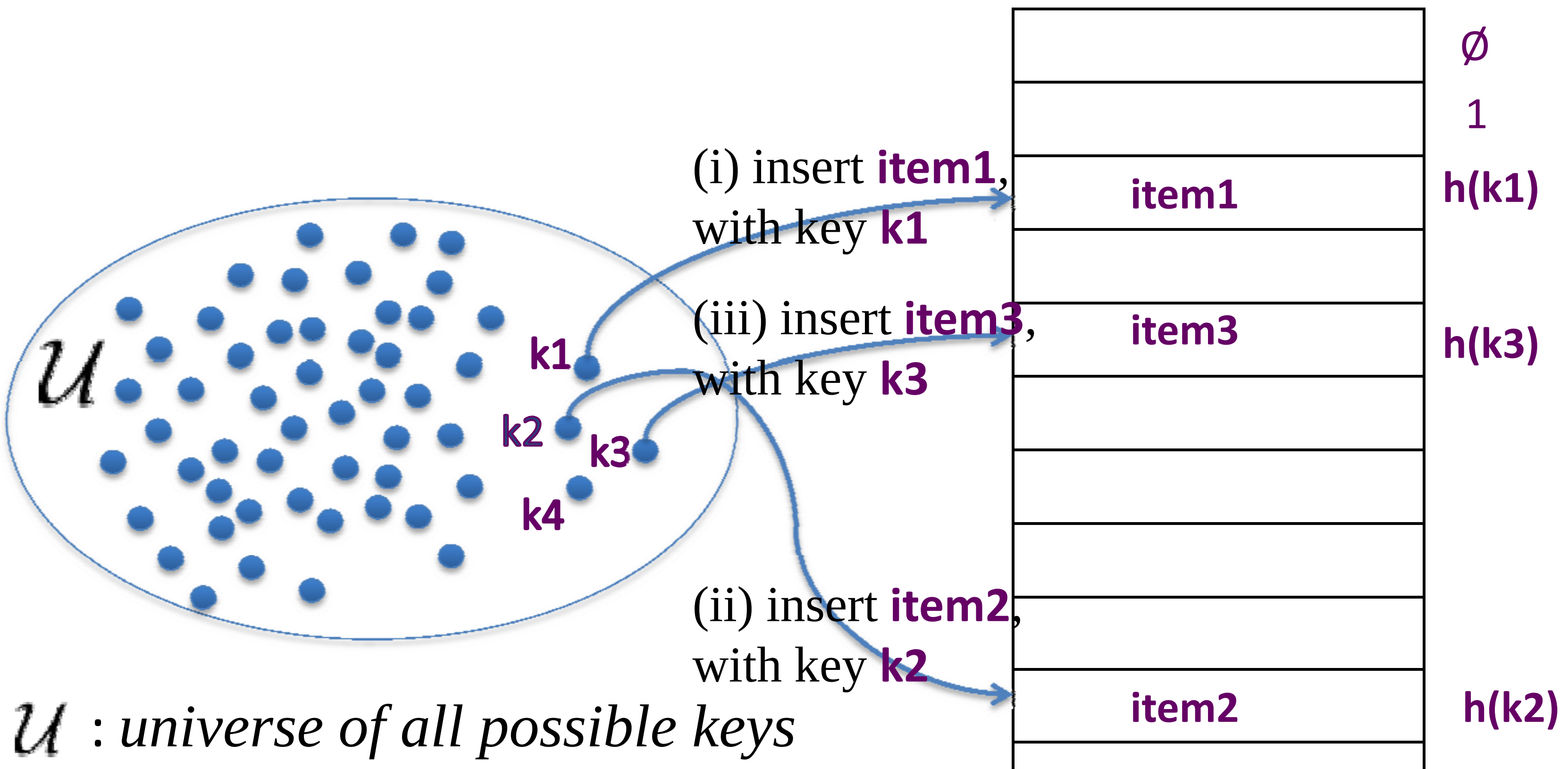
Hashing exploits sparsity of space



$\mathcal{U}$: *universe of all possible keys; huge set*

K : *actual keys; small set but not known in advance*

All keys map to small space…

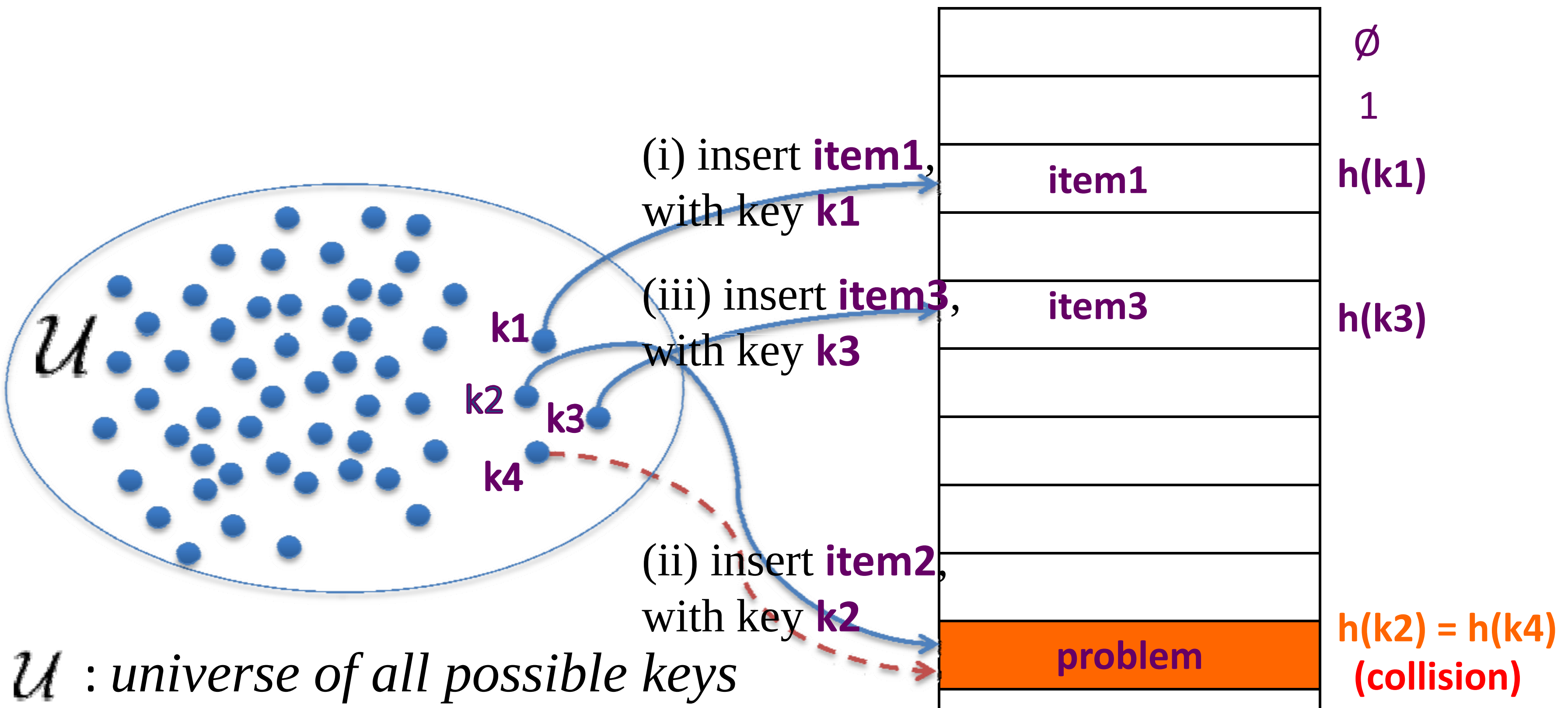


$\mathcal{U}$
k1
k2
k3
k4
(i) insert **item1**, with key **k1**
(iii) insert **item3**, with key **k3**
(ii) insert **item2**, with key **k2**
item1
item3
item2
∅
1
h(k1)
h(k3)
h(k2)
m-1

$\mathcal{U}$: *universe of all possible keys*

(iv) suppose we now try to inset **item4**, with key **k4** and h(k4)=h(k2)…

… leading to collisions



𝒰
k1
k2
k3
k4
(i) insert **item1**, with key **k1**
(iii) insert **item3**, with key **k3**
(ii) insert **item2**, with key **k2**
item1
item3
problem
∅
1
h(k1)
h(k3)
h(k2) = h(k4)
(collision)
m-1

𝒰 : *universe of all possible keys*

(iv) suppose we now try to inset **item4**, with key **k4** and h(k4)=h(k2)…

Our plan for today: Hashing I

- **Today: Genomes, Dictionaries, and Hashing**
  - ☑ Matching genome segments
  - ☑ Introduction to dictionaries
  - ☑ Hash function: definition
  - ➢ Resolving collisions with chaining
  - ❑ Simple uniform hashing assumption
  - ❑ Hash functions in practice: mod / mult
  - ❑ Python implementation
- **Thursday: Speeding up hash tables**
- **Next week: Space issues**

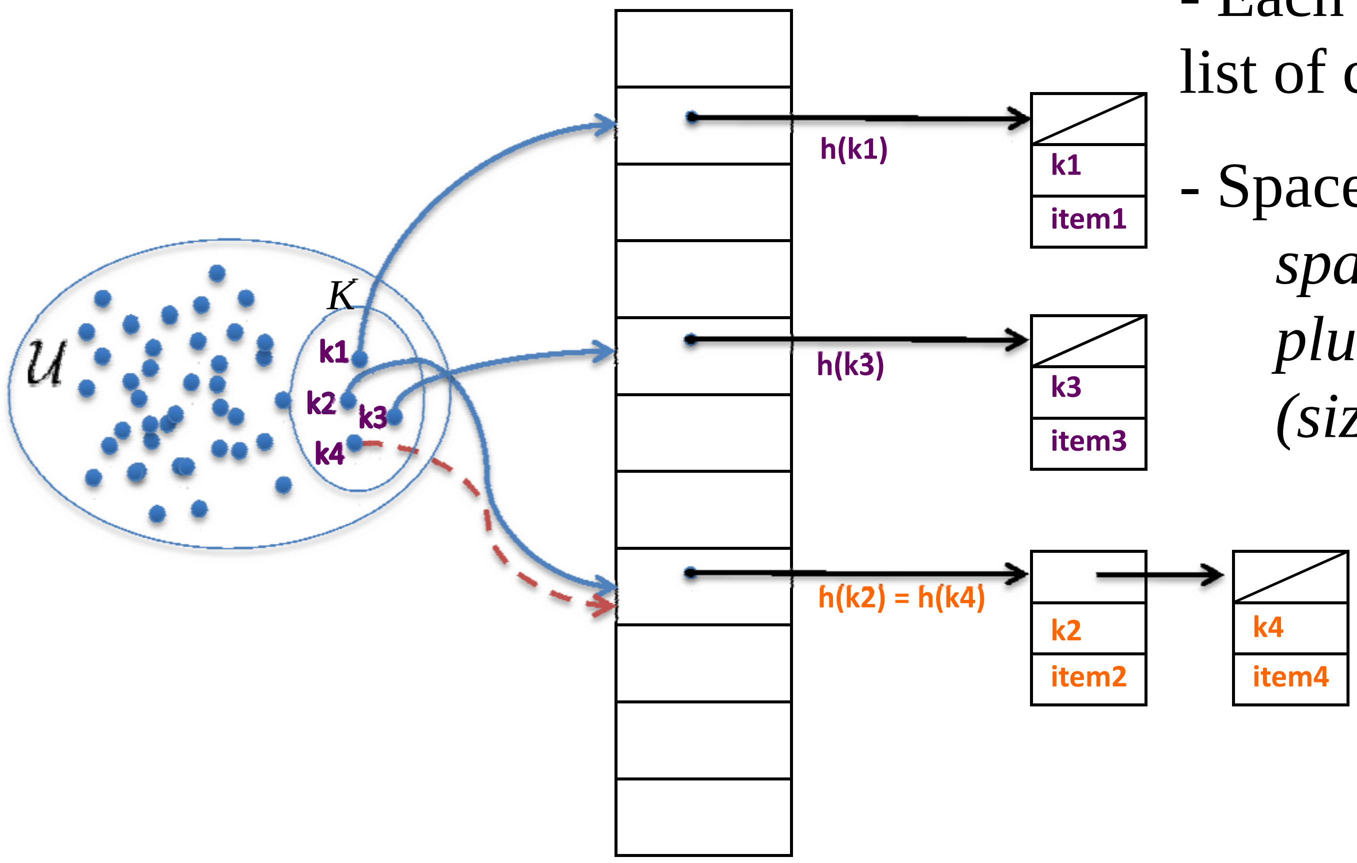
Collisions

- What went/can go wrong?
  - Distinct keys x and y
  - But h(x) = h(y)
  - Called a collision
- This is unavoidable: if table smaller than range, some keys must collide…
  - Pigeonhole principle
- What do you put in the bucket?

Coping with collisions

- **Idea1**: Change to a new “uncoliding” hash function and re-hash all elements in the table
  - Hard to find, and can take a long time if m=O(n)
- **Idea2**: Chaining
  - Linked list of hashed items for each bucket (today)
- **Idea3**: Open addressing
  - Find a different, empty bucket for y (next lecture)
- **Idea4**: Perfect hashing (not covered in 6.006)
  - Create a 2nd-level hash table of size k^2 for each k-element bin, and try several 2nd-level hash functions until no collisions are found (see 6.046)

Chaining



- Each bucket, linked list of contained items

- Space used is
  space of table
  plus one unit per item
  (size of key and item)

𝒰 : *universe of all possible keys*

K : *actual keys, not known in advance*

Problem Solved?

- To find key, must scan whole list in key’s bucket
- Length L list costs L key comparisons
- If all keys hash to same bucket, lookup cost $\Theta(n)$

Solution: optimism

Our plan for today: Hashing I

- **Today: Genomes, Dictionaries, and Hashing**
  - ☑Matching genome segments
  - ☑Introduction to dictionaries
  - ☑Hash function: definition
  - ☑Resolving collisions with chaining
  - ➢Simple uniform hashing assumption
  - ❑Hash functions in practice: mod / mult
  - ❑Python implementation
- **Thursday: Speeding up hash tables**
- **Next week: Space issues**

Simple uniform hashing assumption

- Definition:
  - Each key $k \in K$ of keys is equally likely to be hashed to any slot of table T, independent of where other keys are hashed.

Let n be the number of keys in the table, and let m be the number of slots.

Define the ***load factor*** of T to be

$$\alpha = n/m$$

$$= \text{average number of keys per slot.}$$

Chaining Analysis under SUHA

Average case analysis:

- n items in table of m buckets
- Average number of items/bucket is $\alpha = n/m$
- So expected time to find some key x is $(1+\alpha)$
- $O(1)$ if $\alpha = O(1)$, i.e. $m = \Omega(n)$

apply hash function and access slot

search the list

Summary (rehash)

- Matching big genomes is a hard problem
  - And you will tackle it in your problem set!
- Dictionaries are pervasive
- Hash tables implement them efficiently
  - Under an optimistic assumption of random keys
  - Can be “made true” by heuristic hash functions
- Key idea for beating BSTs: Indexing
  - Sacrificed operations: previous, successor
- Chaining strategy for collision resolution
- Next two lectures: speed & space improvements

Unit #2: Genomes, Hashing, Dictionaries

- **Today: Genomes, Dictionaries, and Hashing**
  - ☑ Intro, basic operations ☑ collisions and chaining
  - ☑ Simple uniform hashing assumption
  - ☑ Hash functions, ☑ Python implementation
- **Thursday: Speeding up hash tables**
  - ➢ Faster comparison: Signatures
  - ➢ Faster hashing: Rolling Hash
- **Next week: Space issues**
  - ➢ Dynamic resizing and amortized analysis
  - ➢ Open addressing, deletions, and probing