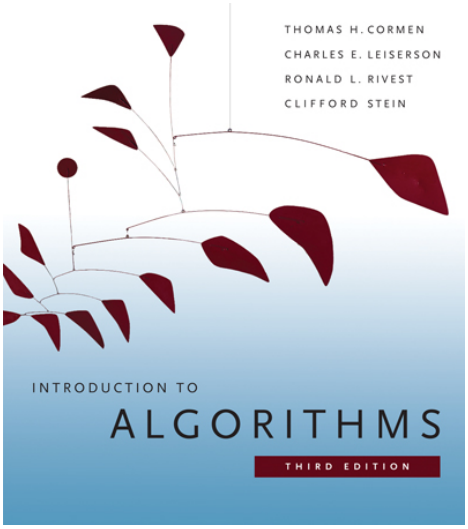
6.006- *Introduction to Algorithms*



Lecture 6

Prof. Manolis Kellis

CLRS: Chapter 17 and 32.2.

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 Search Trees	Out: 2/1		Thu Feb 03	2	Peak Finding Problem	2	Peak Finding correctness & analysis
	Due: Mon 2/14	2	Tue Feb 08	3	Scheduling and Binary Search Trees	3	Binary Search Tree Operations
	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 and Search	PS4. Out: 3/10		Thu Mar 10	11	Searching I: Graph Representation, Depth-1st Search	11	Strongly connected components
	Due: Fri 3/18	7	Tue Mar 15	12	Searching II: Breadth-1st Search, Topological Sort	12	Rubik's Cube Solving
	HW lab:W 3/16		Thu Mar 17	13	Searching III: Games, Network properties, Motifs	13	Subgraph isomorphism
Shortest Paths	PS5	8	Tue Mar 29	14	Shortest Paths I: Introduction, Bellman-Ford	14	Relaxation algorithms
	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 Programming	PS6	10	Tue Apr 12	18	DP I: Memoization, Fibonacci, Crazy Eights	18	Limits of dynamic programming
	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.		
	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 Pictures (NP)	PS7 out Thu4/28		Thu Apr 28	22	Numerics I - Computing on large numbers	22	Models of computation return!
	Due: Fri 5/6	13	Tue May 3	23	Numerics II - Iterative algorithms, Newton's method	23	Computing the nth digit of π
	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		

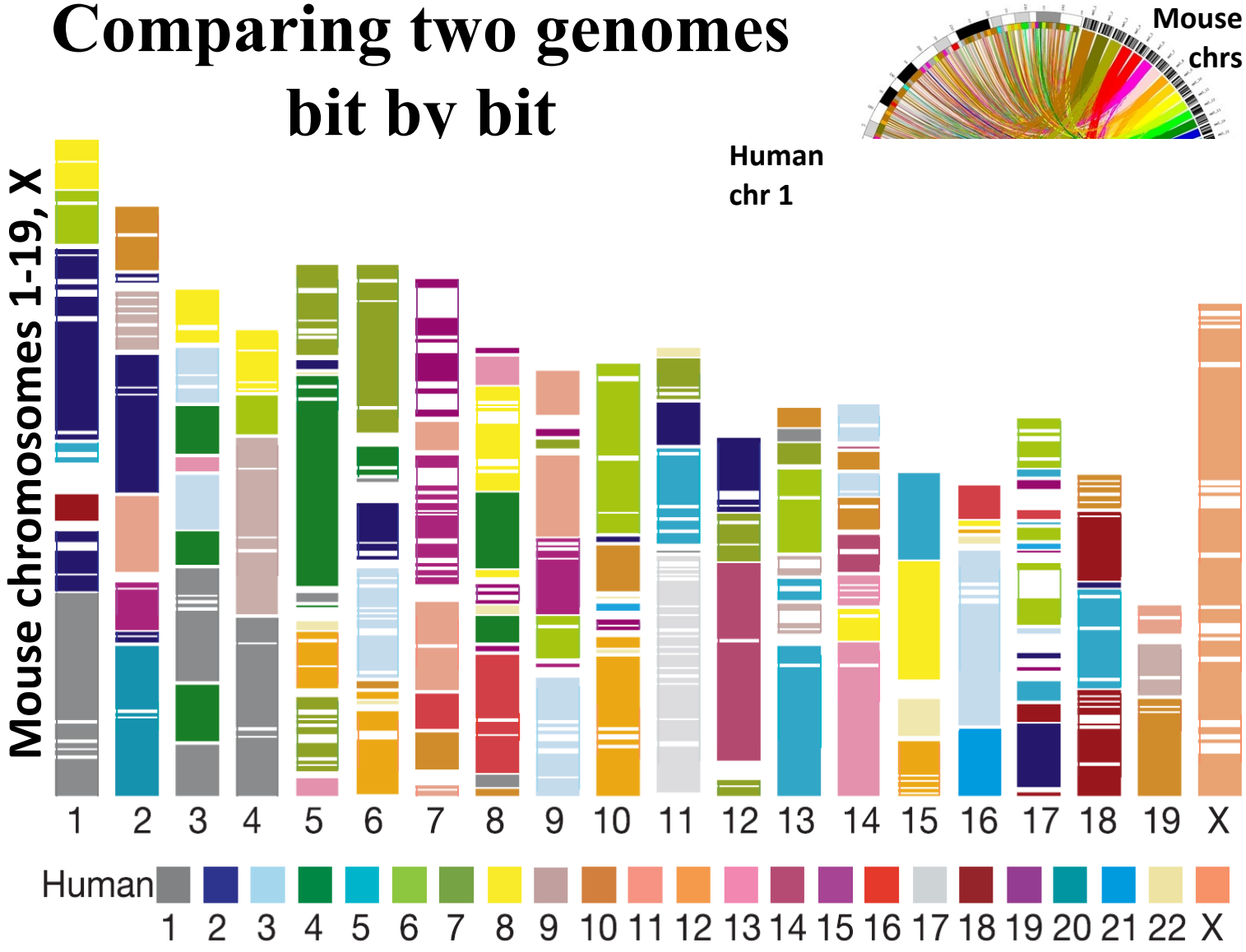
Unit #2: Hashing

- **Last time: Genomes, Dictionaries, Hashing**
  - Intro, basic operations, collisions and chaining
  - Simple uniform hashing assumption
- **Today: Faster hashing, hash functions**
  - Hash functions, python implementation
  - 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 II

- **Today: Hashing in practice,**
  - ➢Review: Genomes, Dictionaries, Hashing
  - ❑Review: Hash functions, collisions, chaining
  - ❑Hash functions in practice: div, mult, Python
  - ❑Speeding up hashing: $O(n^4) \rightarrow O(n^2 \lg n) \rightarrow O(n)$
    - ❑Faster hashing: rolling hash
    - ❑Fewer string comparisons: signatures
  - ❑Summary, conclusion
- **Next week: Space issues**

Comparing two genomes bit by bit



Naïve Algorithm

- Say strings S and T of length n
- For L = n downto 1 **n***
  - ➢ for all length L substrings X1 of S **n***
    - ➢ for all length L substrings X2 of T **n***
      - ➢ if X1=X2, return L **n**
- 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: $\Omega(n^4)$

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

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

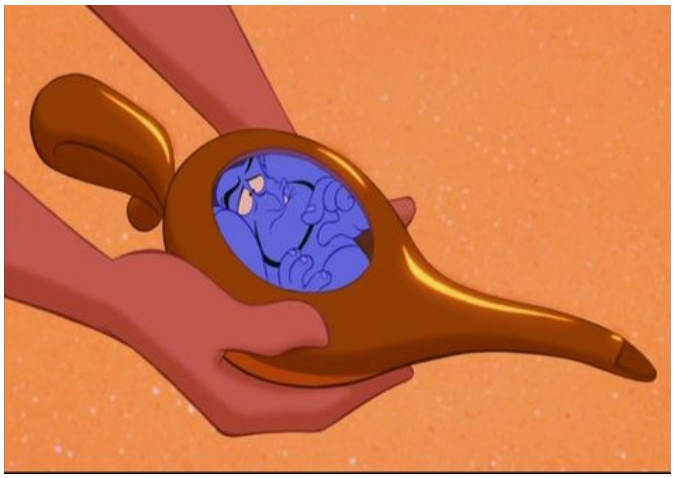
    $\Rightarrow$ can represent each with 5 bits
  - Antidisestablishmentarianism has 28 letters
  - 28*5 = 140 bits
  - So, store in array of size 2^{140} ….oops
- **Why hashing**: 2^{140} is too much space when in fact only 100,000 words actually appear

Our plan for today: Hashing II

- **Today: Hashing in practice,**
  - ☑ Review: Genomes, Dictionaries, Hashing
  - ➢ Review: Hash functions, collisions, chaining
  - ❑ Hash functions in practice: div, mult, Python
  - ❑ Speeding up hashing: $O(n^4) \rightarrow O(n^2 \lg n) \rightarrow O(n)$
    - ❑ Faster hashing: rolling hash
    - ❑ Fewer string comparisons: signatures
  - ❑ Summary, conclusion
- **Next week: Space issues**

The ‘magic’ of hash functions

PHENOMENAL COSMIC POWERS!!



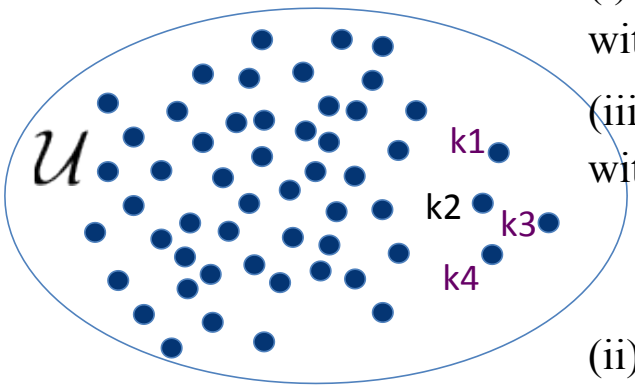
itty bitty living space

With apologies to Disney

Huge universe of possible keys…

… all map to small table m

h:U {1..m}



(i) insert **item1**, with key **k1**

(iii) insert **item3**, with key **k3**

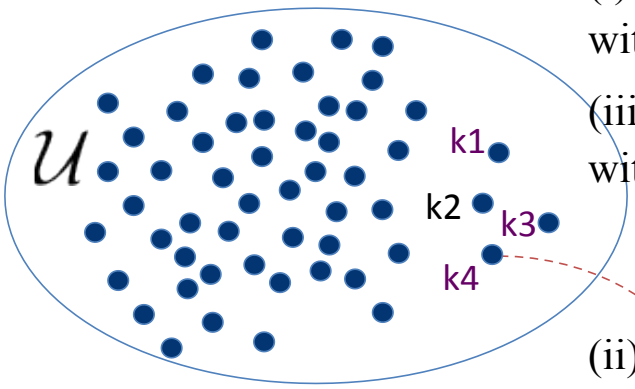
(ii) insert **item2**, with key **k2**

Ø		
1		
	item1	**h(k1)**
	item3	**h(k3)**
	item2	**h(k2)**
m-1		

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

Huge universe of possible keys … all map to small table m

… leading to collisions



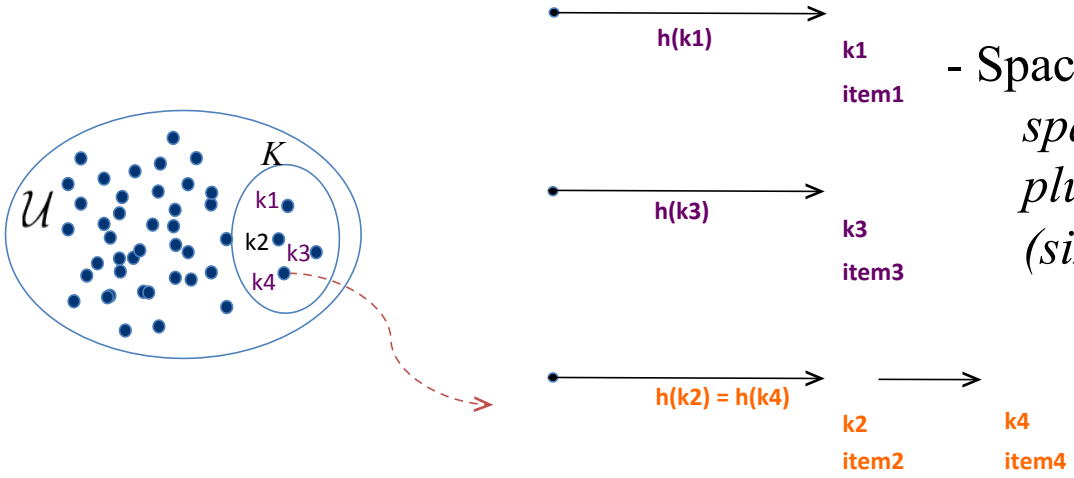
$\mathcal{U}$: *universe of all possible keys*

Coping with collisions: Chaining

$\mathcal{U}$: *universe of all possible keys*
K : *actual keys, not known in advance*

- Each bucket, linked list of contained items

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



How long will 1 lookup take?
- Worst case: O(n) **Average Case: O(1)?** **SUHA**

E.g. During look-up of **k4**:
1. **Hash** key **k4**, obtaining address **h(k4)**, traverse linked list
2. **Traverse** linked list, **compare k4** to **k2**: mismatch → continue
3. **Compare k4** to **k4**: match → return **item4**

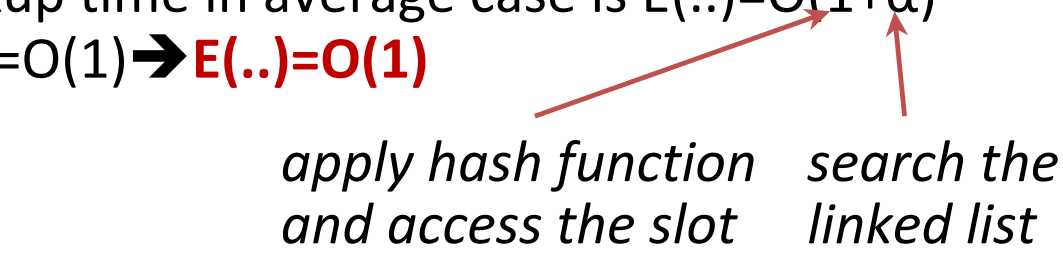
Simple uniform hashing assumption

- 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 $\alpha = n/m$

= average number of keys per slot.

Expected lookup time in average case is $E(..)=O(1+\alpha)$
If $m=\Omega(n) \rightarrow \alpha=O(1) \rightarrow$ **$E(..)=O(1)$**



Our plan for today: Hashing II

- **Today: Hashing in practice,**
  - ☑ Review: Genomes, Dictionaries, Hashing
  - ☑ Review: Hash functions, collisions, chaining
  - ➢ Hash functions in practice: div, mult, Python
  - ❑ Speeding up hashing: $O(n^4) \rightarrow O(n^2 \lg n) \rightarrow O(n)$
    - ❑ Faster hashing: rolling hash
    - ❑ Fewer string comparisons: signatures
  - ❑ Summary, conclusion
- **Next week: Space issues**

Problem: Reality is not always nice

- Keys are often very nonrandom
  - Regularity (evenly spaced sequence of keys)
  - All sorts of mysterious patterns
- Solution: pick a hash function whose values "look" random
  - Though a given key must always have same hash
- Similar to pseudorandom generators
  - Answer based on key & random seed for whole table
- For any function, some set of keys that is bad
  - but hopefully not your set
- In practice, this is all about good ***heuristics***!

Division Hash Function

- h(k) = k mod m
- k_1 and k_2 collide when $k_1 = k_2 \pmod m$
  - Unlikely if keys are random
- Problem: if m is a power of 2, the hash doesn't even depend on all the bits of k

If $k = 1011000111011010_2$ and $r = 6$, then $h(k) = 011010_2$.

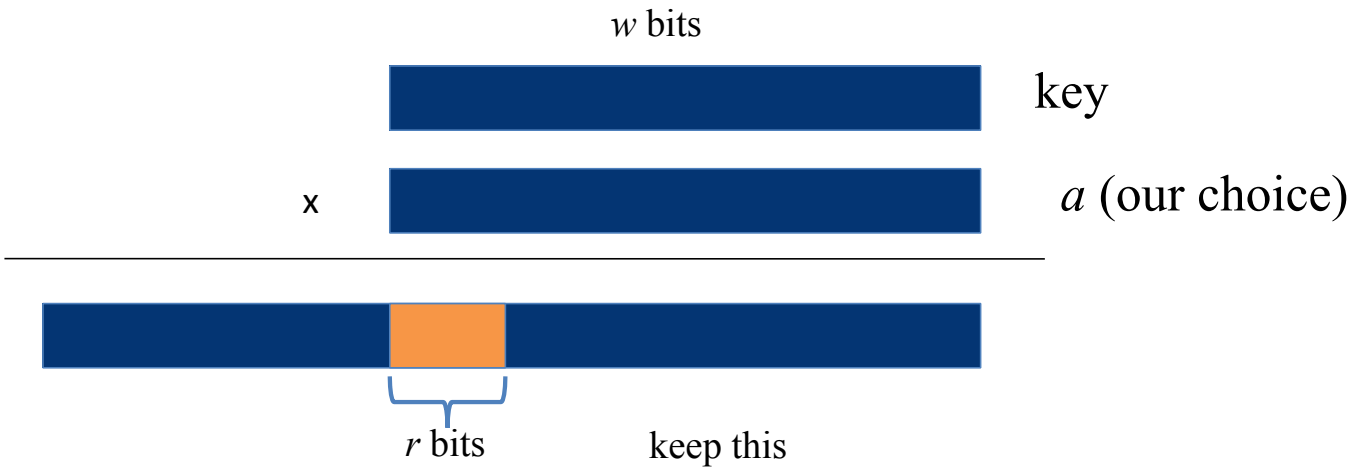
$h(k)$

Problems with division method

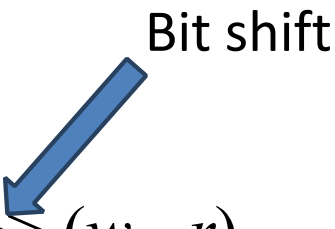
- Regularity
  - Suppose keys are x, 2x, 3x, 4x,….
  - Suppose x and chosen m have common divisor d
  - Then only use 1/d fraction of table
    - x series cycles back, leaving d-1 out of d entries blank
    - E.g, m power of 2 and all keys are even, only use half
- So make m a prime number
  - But finding a prime number is hard
  - And now you have to divide (slow)

Multiplication Hash Function

- Suppose we're aiming for table size 2^r
- and keys are w bits long, where $w>r$ is the machine word
- Multiply k with some a (fixed for the hash function)
- then keep certain bits of the result as follows



Multiplication Hash Function



- The formula:

$$h(k) = [(a * k) \bmod 2^w] >> (w - r)$$

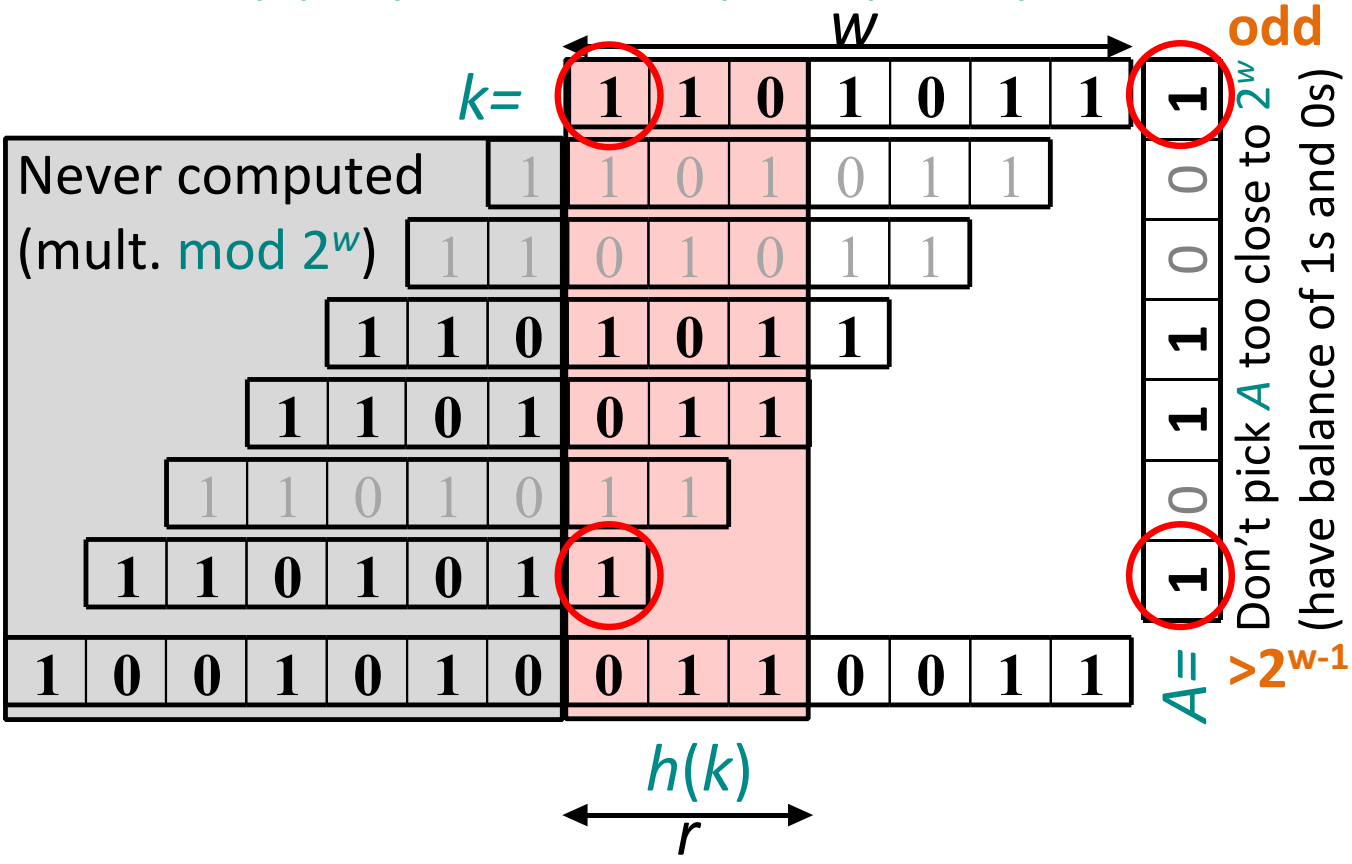
  - Multiply by a
  - When overflow machine word, wrap
  - Take high r bits of resulting machine word
  - (Assumes table size smaller than machine word)

Benefit: Multiplying and bit shifts faster than division

Good practice: Make a an odd integer (why?) $> 2^{w-1}$

Multiplication method intuition (1)

$h(k) = (A \cdot k \bmod 2^w) \text{ rsh } (w - r)$

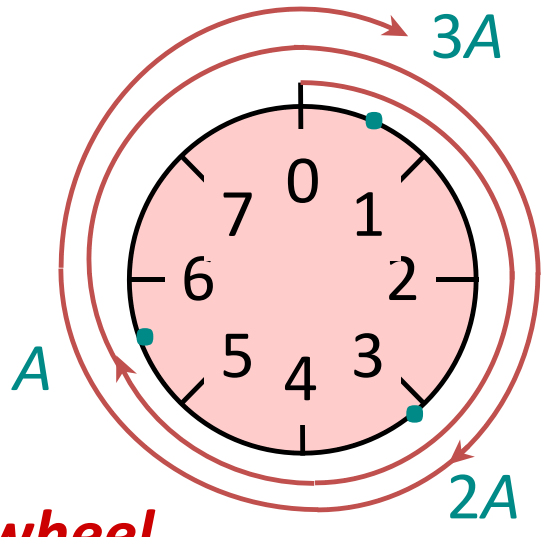


Multiplication method intuition (2)

$h(k) = (A \cdot k \bmod 2^w) \text{ rsh } (w - r)$

Suppose that $m = 8 = 2^3$ and that our computer has $w = 7$-bit words:

$$
\begin{array}{rr}
 & 1\,0\,1\,1\,0\,0\,1 = A \\
\times & 1\,1\,0\,1\,0\,1\,1 = k \\
\hline
 & 1\,0\,0\,1\,0\,1\,0\,0\,1\,1\,0\,0\,1\,1 \\
 & \quad h(k)
\end{array}
$$



Modular wheel

Don't pick A too close to 2^w
(explore more of the wheel space)

Python Implementation

- Python objects have a hash method
  - Number, string, tuple, any object implementing __hash__
- Maps object to (arbitrarily large) integer
  - So really, should be called prehash
- Take mod m to put in a size-m hash table
- Peculiar details
  - Integers map to themselves
  - Strings that differ by one letter don't collide
  - "better" than random for common sequences
    - 1,2,3,4,5 or var_a, var_b, var_c, var_d

Our plan for today: Hashing II

- **Today: Hashing in practice,**
  - ☑ Review: Genomes, Dictionaries, Hashing
  - ☑ Review: Hash functions, collisions, chaining
  - ☑ Hash functions in practice: div, mult, Python
  - ➢ Speeding up hashing: $O(n^4) \rightarrow O(n^2 \lg n) \rightarrow O(n)$
    - ❑ Faster hashing: rolling hash
    - ❑ Fewer string comparisons: signatures
  - ❑ Summary, conclusion
- **Next week: Space issues**

Recall: Longest Common Substring

- Strings S,T of length n, want to find longest common substring
- Algorithms from last time:

$$O(n^4) \rightarrow O(n^3 \log n) \rightarrow O(n^2 \log n)$$

  - Naïve → Binary Search on L → Hash table for T
- Winner algorithm used a hash table of size n:

~~Binary search on maximum match length L;~~ to check if a length works:

- Insert all length-L substrings of S in hash table
- For each length-L substring x of T
  - Look in bucket h(x) to see if x is in S

Analysis?

Runtime Analysis: bottlenecks

- Binary search cost: O(log n) length values L tested
- For each length value L, here are the costly operations:
  - Inserting all L-length substrings of S: n-L hashes
    - Each hash takes L time, so total work $\Theta((n-L)L)=\Theta(n^2)$
  - Hashing all L-length substrings of T: n-L hashes
    - another $\Theta(n^2)$
  - Time for comparing substrings of T to substrings of S:
    - How many comparisons?
    - Under SUHA, each substring of T is compared to an expected O(1) of substrings of S found in its bucket
    - Each comparison takes O(L)
    - Hence, time for all comparisons: $\Theta(nL)=\Theta(n^2)$
- So $\Theta(n^2)$ work for each length
- Hence $\Theta(n^2 \log n)$ including binary search

Making comparisons even faster?

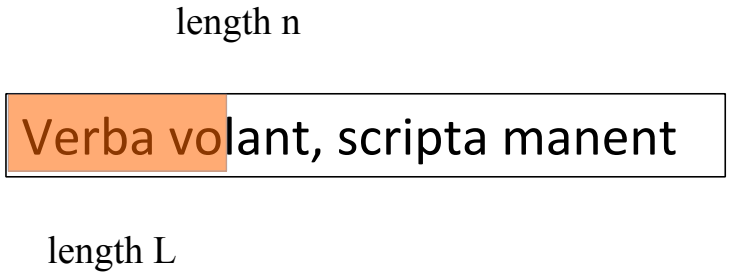
- Amdahl's law: if one part of the code takes 20% of the time, then no matter how much you improve it, you only get 20% speedup
- Corollary: must improve all asymptotically worst parts to change asymptotic runtime
- In our case
  - Must compute sequence of n hashes faster: $n^2 \rightarrow n$
  - Must reduce cost of comparing in bucket: $n^2 \rightarrow n$

Our plan for today: Hashing II

- **Today: Hashing in practice,**
  - ☑ Review: Genomes, Dictionaries, Hashing
  - ☑ Review: Hash functions, collisions, chaining
  - ☑ Hash functions in practice: div, mult, Python
  - ➢ Speeding up hashing: $O(n^4) \rightarrow O(n^2 \lg n) \rightarrow O(n)$
    - ➢ Faster hashing: rolling hash
    - ❑ Fewer string comparisons: signatures
  - ❑ Summary, conclusion
- **Next week: Space issues**

Rolling Hash

- We make a sequence of n substring hashes
  - Substring lengths L
  - Total time $O(nL) = O(n^2)$
- Can we do better?
  - For our particular application, yes!



Rolling Hash Idea

- e.g. hash all 3-substrings of "there"
- Recall division hash: x mod m
- Recall string to number:
  - First substring "the" = $t\cdot(26)^2 + h\cdot(26) + e$
- If we have "the", can we compute "her" as f("the")?

$$\text{"her"} = h\cdot(26)^2 + e\cdot(26) + r$$

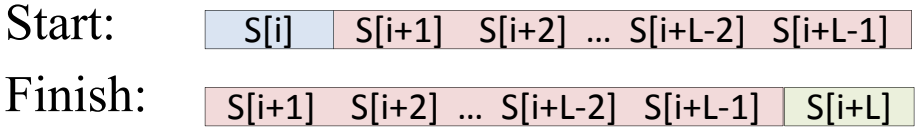
$$= 26 \cdot \left(h\cdot(26) + e \right) + r$$

$$= 26 \cdot \left(t\cdot(26)^2 + h\cdot(26) + e - t\cdot(26)^2 \right) + r$$

$$= 26 \cdot \left(\text{"the"} - t\cdot(26)^2 \right) + r$$

- i.e. subtract first letter's contribution to number, shift, and add last letter

General rule for $s[i+1..]) = f(s[i..])$



1. **Subtract 1st**
2. **Shift**
3. **Add last**

- Strings = base-b numbers
- Start: Current substring S[i … i+L-1]

$S[i]\cdot b^{L-1} + S[i+1]\cdot b^{L-2} + S[i+2]\cdot b^{L-3} \ldots + S[i+L-1]$

1 $- S[i]\cdot b^{L-1}$ *subtract higher-order term*

$S[i+1]\cdot b^{L-2} + S[i+2]\cdot b^{L-3} \ldots + S[i+L-1]$

2 $* b$ *shift to the left*

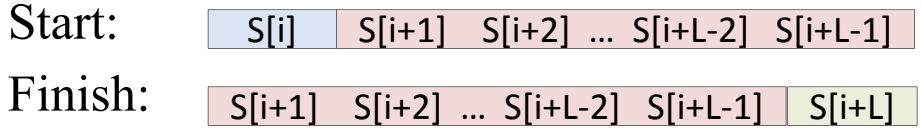
$S[i+1]\cdot b^{L-1} + S[i+2]\cdot b^{L-2} \ldots + S[i+L-1]\cdot b$

3 $+S[i+L]$ *add lower-order term*

$S[i+1]\cdot b^{L-1} + S[i+2]\cdot b^{L-2} \ldots + S[i+L-1]\cdot b + S[i+L]$

- Finish: Next substring S[i+1 … i+L]

Ensuring O(1) cut/shift/paste operations



1. **Subtract 1st**
2. **Shift**
3. **Add last**

- So: **S[i+1 … i+L] = b S[i … i+L-1]- b^L S[i] + S[i+L]**
- where

  $S[i \ldots i+L-1] = S[i] \cdot b^{L-1} + S[i+1] \cdot b^{L-2} + \ldots + S[i+L-1]$ (*)
- **But** S[i … i+L-1] may be a huge number (so huge that we may not even be able to store in the computer, e.g. L=50, b=26)
- **Solution** only keep its ***division hash***: S[…] mod m
- This can be computed without computing S[…], using mod magic!
- Recall: **(ab) mod m = (a mod m) (b mod m) (mod m)**
  (a+b) mod m = ((a mod m) + (b mod m)) (mod m)

- With a clever parenthesization of (*): O(L) to hash string!

Applying mod magic in hash space

Start: | S[i] | S[i+1] | S[i+2] | … | S[i+L-2] | S[i+L-1] |

Finish: | S[i+1] | S[i+2] | … | S[i+L-2] | S[i+L-1] | S[i+L] |

1. **Subtract 1st**
2. **Shift**
3. **Add last**

- Recall: $S[i+1 \ldots i+L] = b\, S[i \ldots i+L-1] - b^L\, S[i] + S[i+L]$
- But can we also compute:
  hash$(S[i+1 \ldots i+L])$ from ***hash***$(S[i \ldots i+L-1])$?
- Still mod magic to the rescue!
- $h(S[i+1 \ldots i+L]) = h(b\, S[i \ldots i+L-1]) - h(b^L \cdot S[i]) + h(S[i+L])$

Can compute b^L mod m in O(L) by recursive multiplication mod m

Total time: O(L) time for the first hash
+O(L) to compute b^L mod m
+L*O(1) for each additional hash

Computing n-L hashes costs O(n)

Our plan for today: Hashing II

- **Today: Hashing in practice,**
  - ☑ Review: Genomes, Dictionaries, Hashing
  - ☑ Review: Hash functions, collisions, chaining
  - ☑ Hash functions in practice: div, mult, Python
  - ➢ Speeding up hashing: $O(n^4) \rightarrow O(n^2 \lg n) \rightarrow O(n)$
    - ❑ Faster hashing: rolling hash
    - ☑ ➢ Fewer string comparisons: signatures
  - ❑ Summary, conclusion
- **Next week: Space issues**

Faster Comparison

- **First Idea:** when we find a match for some length, we can stop and go to the next value of length in our binary search.
- **But,** the real problem is "false positives"
  - Strings in same bucket that don't match, but we waste time on
- **Analysis:**
  - n substrings to size-n table: average load **1**
  - SUHA: for every substring x of T, there is **1** other string in x's bucket (in expectation)
  - Comparison work: **L** per string (in expectation)
  - So total work for all strings of T: **$nL = Q(n^2)$**

Solution: Bigger table!

- What size?
- Table size **$m = n^2$**
  - n substrings to size-m table: average load **1/n**
  - SUHA: for every substring x of T, there is **1/n** other strings in x's bucket (in expectation)
  - Comparison work: **L/n** per string (in expectation)
  - So total work for all strings of T: **$n(L/n) = L = O(n)$**

- Downside?
  - Bigger table
  - (n^2 isn't realistic for large n)

Signatures

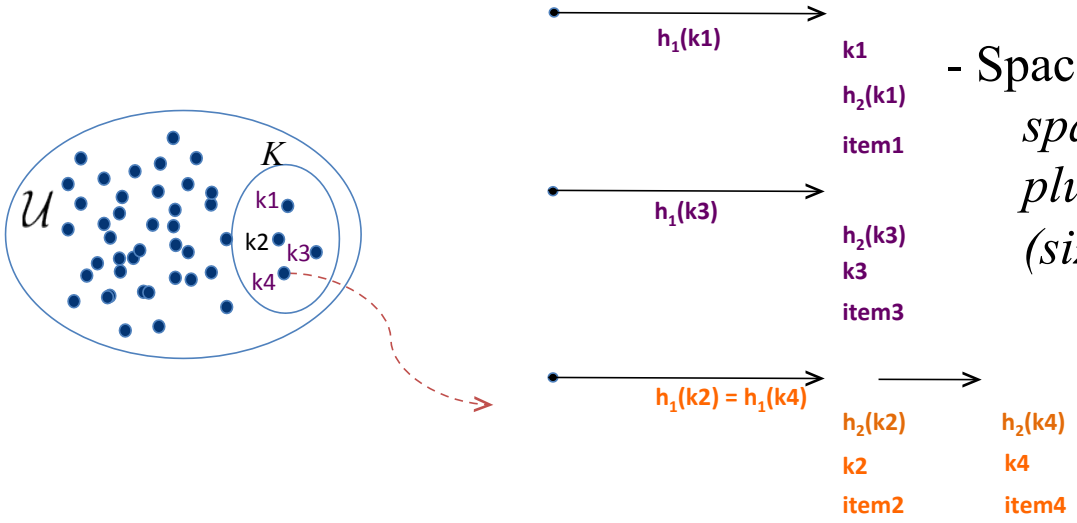
- Note n^2 table isn't needed for fast lookup
  - Size n enough for that
  - n^2 is to reduce cost of false positive compares
- So don't bother making the n^2 table
  - Just compute for each string another hash value in the larger range $1..n^2$
  - Called a signature
  - If two signatures differ, strings differ
  - Pr[same sig for two different strings] = $1/n^2$
    - (simple uniform hashing)

Chaining with signatures

$\mathcal{U}$: *universe of all possible keys*
K : *actual keys, not known in advance*

- Each bucket, linked list of contained items

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



How long will 1 lookup take?

- Worst case: O(n) **Average Case: O(1)?** **SUHA**

E.g. During look-up of **k4**:
1. **Hash** key **k4** with h_1 and h_2 obtaining address and signature
2. **Traverse** linked list, **compare signatures**: mismatch → cont.
3. **Compare k4** to **k4**: match → return **item4**

Application

- Hash substrings to size n table
- But store a signature with each substring
  - Using a second hash function to $[1..n^2]$
- Check each T-string against its bucket
  - First check signature, if match then compare strings
  - Signature is a small number, so comparing them is O(1)

strictly speaking O(logn); but if $n^2<2^{32}$ the signature fits inside a word of the computer; in this case, the comparison takes O(1)

Application

- Runtime Analysis:
  - for each T-string:
    O(bucket size)=O(1) work to compare signatures;
  - so overall O(n) time in signature comparisons
  - Time spent in string comparisons?
    L x (Expected Total Number of False-Signature Collisions)
    - n out of the n^2 values in $[1..n^2]$ are used by S-strings
    - so probability of a T-string signature-colliding with some S-string: n/n^2
    - hence total expected number of collisions 1

    so total time spent in String Comparisons is L

fine print: time needed to compute signatures using rolling hash O(n) for each signature, hence total time O(2n)=O(n). (again, idea to optimize bottlenecks, by paying small incremental costs in non-bottleneck spots)

Our plan for today: Hashing II

- **Today: Hashing in practice,**
  - ☑ Review: Genomes, Dictionaries, Hashing
  - ☑ Review: Hash functions, collisions, chaining
  - ☑ Hash functions in practice: div, mult, Python
  - ☑ Speeding up hashing: $O(n^4) \rightarrow O(n^2 \lg n) \rightarrow O(n)$
    - ☑ Faster hashing: rolling hash
    - ☑ Fewer string comparisons: signatures
  - ➢ Summary, conclusion
- **Next week: Space issues.** Dynamic resizing, open addressing, probe sequences, deletions

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

Summary

- Reduced compare cost to O(n)/length
  - By using a big hash table
  - Or signatures in a small table
- Reduced hash computation to O(n)/length
  - Rolling hash function
- Total cost of phases: O(n log n)

- Not the end: suffix tree achieves O(n)

Hashing and Genomes are coming back

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 Search Trees	Out: 2/1		Thu Feb 03	2	Peak Finding Problem	2	Peak Finding correctness & analysis
	Due: Mon 2/14	2	Tue Feb 08	3	Scheduling and Binary Search Trees	3	Binary Search Tree Operations
	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 and Search	PS4. Out: 3/10		Thu Mar 10	11	Searching I: Graph Representation, Depth-1st Search	11	Strongly connected components
	Due: Fri 3/18	7	Tue Mar 15	12	Searching II: Breadth-1st Search, Topological Sort	12	Rubik's Cube Solving
	HW lab:W 3/16		Thu Mar 17	13	Searching III: Games, Network properties, Motifs	13	Subgraph isomorphism
Shortest Paths	PS5	8	Tue Mar 29	14	Shortest Paths I: Introduction, Bellman-Ford	14	Relaxation algorithms
	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 Programming	PS6	10	Tue Apr 12	18	DP I: Memoization, Fibonacci, Crazy Eights	18	Limits of dynamic programming
	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.		
	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 Pictures (NP)	PS7 out Thu4/28		Thu Apr 28	22	Numerics I - Computing on large numbers	22	Models of computation return!
	Due: Fri 5/6	13	Tue May 3	23	Numerics II - Iterative algorithms, Newton's method	23	Computing the nth digit of π
	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		