CSE 100: TREE SEARCH AND HASH TABLES (1)

Announcements

- Midterm 1 is on Wednesday
  - Check seating arrangements
- HW2 would be released soon

Tree Search: Thought questions

- Which search algorithm will take longer for predict-completions (return the N most frequent completions)?

A. DFS
B. BFS
C. They will take the same

- If you ignore frequencies, which search algorithm will find words in increasing order of length in a MWT?

A. DFS
B. BFS
C. both
D. neither

- Which search algorithm will find words in increasing order of frequency in a MWT?

A. DFS
B. BFS
C. both
D. neither

Finding data fast

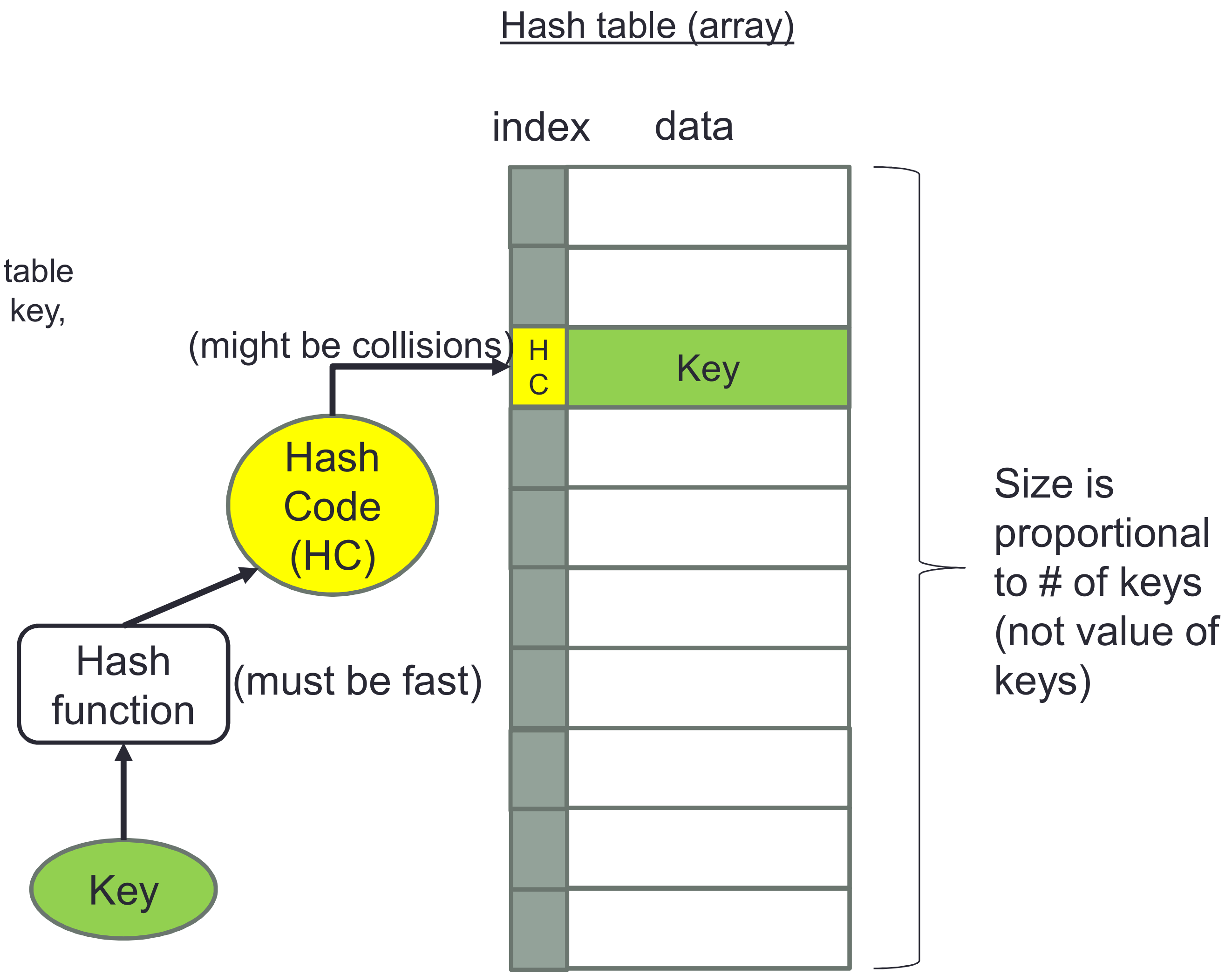
Imagine that you want to store integers between 0 and 1,000,000. You want to be able to find out whether an element is present in your set. You know you can do this with a BST with average running time of O(logN). But you decide to use an array with 1,000,000 Boolean values to store your data to try to make this faster. An entry will be true if the item is in your structure, and false otherwise.

What is the (Big-O) running time to "insert" an item into this proposed structure?

A. O(N)
B. O(logN)
C. O(1)

Hashing

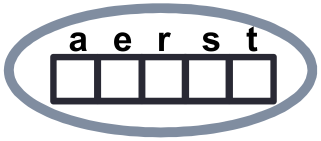
- Let’s modify our array-based look up table
- Need a hash-function h(x): takes in a key, returns an index in the array
- gold standard: random hash function



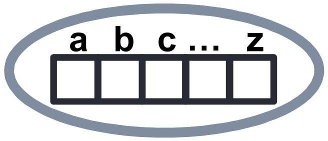
“Hashing” and MWTs

- The idea of hashing was at the heart of the MWT data structure we looked at. We mapped chars to ints to get their position in the array.

The hash function was:
H(‘a’) = 0
H(‘e’) = 1
H(‘r’) = 2
H(‘s’) = 3
H(‘t’) = 4



What is the hash function you would use for your PA if you implement a MWT with a vector? (ignore spaces for now)



H(c) = ________________________

This hash function has no collisions!

Goals for today

- Explain the idea behind and advantages of hashing and hash tables
- Calculate collision probabilities in hash tables

The birthday collision "paradox"

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Jan																															
Feb																															
Mar																															
Apr																															
May																															
Jun																															
Jul																															
Aug																															
Sep																															
Oct																															
Nov																															
Dec																															

Probability of Collisions

- Suppose you have a hash table that can hold 100 elements. It currently stores 9 elements (in 9 different locations in the hash table). What is the probability that your next insert will cause a collision (assuming a totally random hash function)?

A. 0.09

B. 0.10

C. 0.37

D. 0.90

E. 1.00

Probability of Collisions

- Suppose you have a hash table that can hold 100 elements. It currently stores 30 elements (each in one of 30 possible different locations in the hash table). What is the probability that your next two inserts will cause at least one collision (assuming a totally random hash function)? (Choose the closest match)

A. .09

B. .30

[✓ C. .52]

D. .74

E. .90

Probability of Collisions

- If you have a hash table with M slots and N keys to insert in it, then the probability of at least 1 collision is:

$$P_{N.M}(collision) = 1 - P_{N.M}(no\ collision)$$

$$= 1 - \prod_{i=1}^{N} P_{N.M}(ith\ key\ no\ collision)$$

Hashtable collisions and the "birthday paradox"

- Suppose there are 365 slots in the hash table: M=365
- What is the probability that there will be a collision when inserting N keys?
  - For N = 10, $\text{prob}_{N,M}$(collision) = 12%
  - For N = 20, $\text{prob}_{N,M}$(collision) = 41%
  - For N = 30, $\text{prob}_{N,M}$(collision) = 71%
  - For N = 40, $\text{prob}_{N,M}$(collision) = 89%
  - For N = 50, $\text{prob}_{N,M}$(collision) = 97%
  - For N = 60, $\text{prob}_{N,M}$(collision) = 99+%

- So, among 60 randomly selected people, it is almost certain that at least one pair of them have the same birthday
- In general: collisions are likely to happen, unless the hash table is quite sparsely filled
- So, if you want to use hashing, can't use perfect hashing because you don't know the keys in advance, and don't want to waste huge amounts of storage space, you have to have a strategy for dealing with collisions

PA2: Hash functions for strings

(Worksheet, problem 1)

- This hash function adds up the integer values of the chars in the string (*then need to take the result mod the size of the table*):

```
int hash(std::string const & key) {
  int hashVal = 0, len=key.length();
  for(int i=0; i<len; i++) {
   hashVal += key[i];
 }
 return hashVal;
}
```

- What is wrong with this hash function **for storing words of 8-characters**?

A. Nothing

B. It is too complex (takes too long) to compute

C. It will lead to collisions between words with similar endings

[✓ D. It will not distribute keys well in a large table]

E. It will never distribute keys well in any table

Collision resolution strategies

- Unless we are doing "perfect hashing" we have to have a collision resolution strategy, to deal with collisions in the table.
- The strategy has to permit find, insert, and delete operations that work correctly!
- Collision resolution strategies you need to know are:
  - Separate chaining (from your reading)
  - Linear probing (from your reading)
  - Double hashing (from your reading)

Resolving Collisions: Separate Chaining

- using the hash function H(K) = K mod M, insert these integer keys:

701 (1), 145 (5), 218 (1), 12 (5), 750 (1)

in this table:

index:	0	1	2	3	4	5	6

Is there an upper bound to the number of elements that we can insert into a hash table of size M with separate chaining?

A. Yes, because of space constraints in the array
B. No, but inserting too many elements can affect the run time of insert
C. No, but inserting too many elements can affect the run time of find
D. Both B and C

What is the worst-case running time for insert and find for separate chaining?

Linear probing: inserting a key (Worksheet, problem 2 again)

- When inserting a key K in a table of size M, with hash function H(K)

1. Set indx = H(K)
2. If table location indx already contains the key, no need to insert it. Done!
3. Else if table location indx is empty, insert key there. Done!
4. Else collision. Set indx = (indx + 1) mod M.
5. If indx == H(K), table is full! (Throw an exception, or enlarge table.) Else go to 2.

M = 7, H(K) = K mod M

insert these keys 701 (1), 145 (5), 218 (1), 12 (5), 750 (1) in this table,

using linear probing:

index:	0	1	2	3	4	5	6

Linear probing: searching for a key and deleting keys

- Using linear probing, can you delete an element by removing it completely from the table? Why or why not?
  - A. Yes, it's fine to remove an element by removing it completely from the table.
  - B. No, you cannot remove an element by removing it completely from the table.

			16	10	23		
index:	0	1	2	3	4	5	6

Analysis of open-addressing (e.g. linear probing) hashing

- What is the *worst-case* time to find a single element in a hash table with N elements in it?

A. O(1)
B. O(log(N))
C. O(N)
D. O(Nlog(N))
E. O(N^2)

Resolving Collisions: Double hashing

- A sequence of possible positions to insert an element are produced using two hash functions
- $h_1(x)$: to determine the position to insert in the array, $h_2(x)$: the offset from that position

701 (1,4), 145 (5,5), 218 (1,2), 12 (5,3), 750 (1,5)

in this table:

		701				145	
index:	0	1	2	3	4	5	6

Hashtable worksheet (continued—do on your own)

A hash table of size 10 uses open addressing with hash function h(k) = k mod 10 and linear probing for collision resolution. After inserting 6 values into an empty hash table, the table looks like this:

0	
1	
2	42
3	23
4	34
5	52
6	46
7	33
8	
9	

Which of the following gives a possible order in which the key values could have been inserted into the table?

A. 46, 42, 34, 52, 23, 33
B. 34, 42, 23, 52, 33, 46
C. 46, 34, 42, 23, 52, 33
D. 42, 46, 33, 23, 34, 52

0	
1	
2	42
3	23
4	34
5	52
6	46
7	33
8	
9	

Considering the same values and hash tables above, which of the following statements are true? (A=true, B=false)

1. 42 must have been inserted before 52
2. 34 must have been inserted before 52
3. 46 must have been inserted after 42
4. 33 must have been inserted after 46
5. 34 must have been inserted after 23

Now write down all of the other order constraints that must hold that you can think of.

0	
1	
2	42
3	23
4	34
5	52
6	46
7	33
8	
9	

What is the probability of the next key that is inserted in the hash table from the previous problem going into each of the open spaces? Assume the value is chosen uniformly randomly (no value is more likely to be chosen than any other).

0	
1	
2	42
3	23
4	34
5	52
6	46
7	33
8	
9	

What is the probability that at least one of the next two keys inserted will cause a collision?

Consider a hash table of size 100. For what data will the hash function h(k) = k mod 100 definitely be a bad choice (select all that apply)?

1. If your keys are always greater than 100
2. If your keys are always multiples of 5
3. If your keys are always less than 50
4. If your keys are always multiples of 7

A. Definitely a bad choice: Will always distribute keys unevenly
B. Not necessarily a bad choice: Might distribute keys evenly

Resolving Collisions: Double hashing

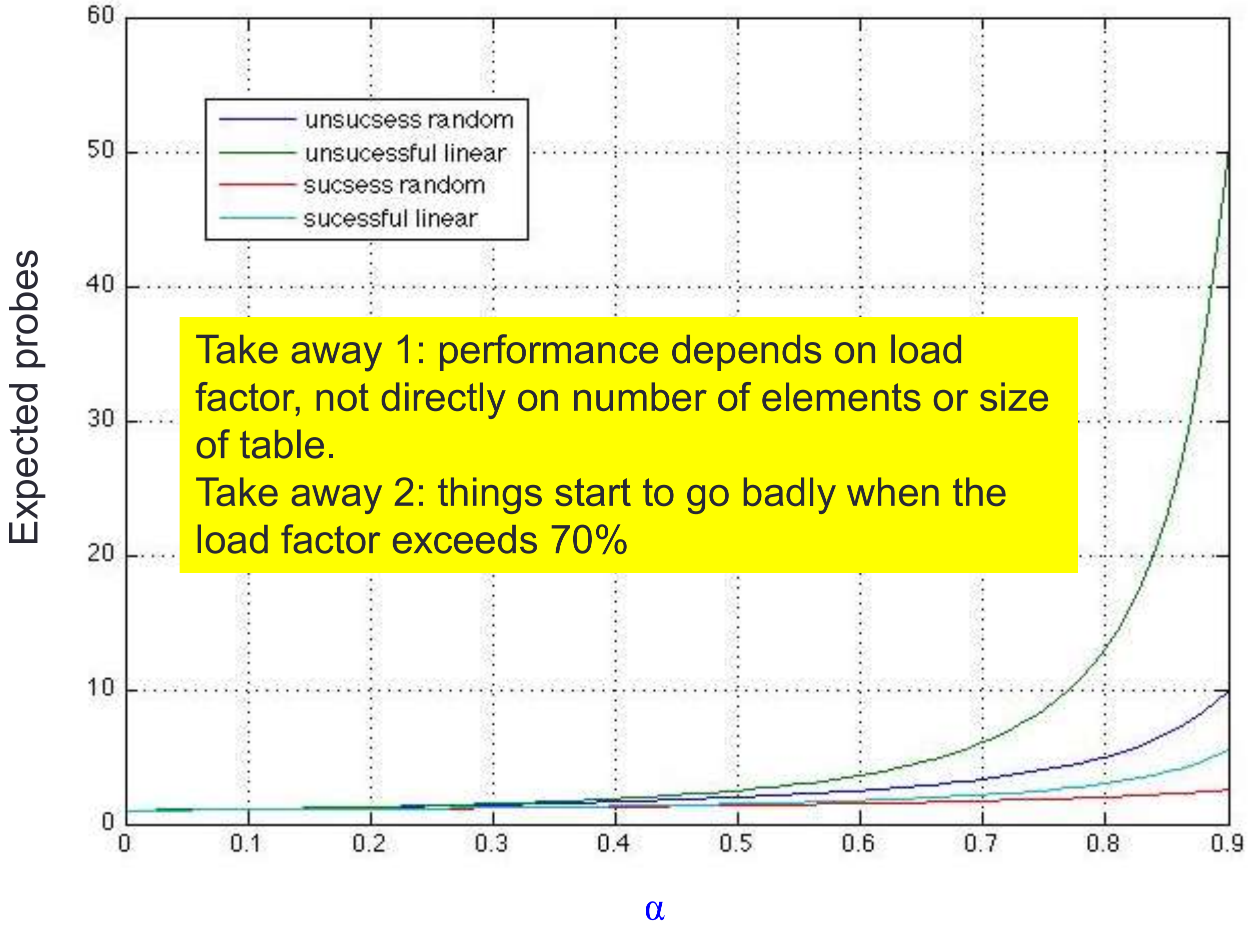
- A sequence of possible positions to insert an element are produced using two hash functions
- $h_1(x)$: to determine the position to insert in the array, $h_2(x)$: the offset from that position

701 (1,4), 145 (5,5), 218 (1,2), 12 (5,3), 750 (1,5)

in this table:

		701				145	
index:	0	1	2	3	4	5	6

Dependence of average performance on load



Take away 1: performance depends on load factor, not directly on number of elements or size of table.

Take away 2: things start to go badly when the load factor exceeds 70%

Average case costs with separate chaining

What you need to know:

- Separate chaining performance also depends only on load factor, and *not the number of elements* or size of table
- In practice it performs extremely well, even with relatively high loads

How big should a hash table be?

- The size of a hash table, M, is the number of buckets in the hash table. Let's say you want to store up to 100 keys in your hash table. Which of the following is the best choice of hash table size?

A. 100

B. 101

C. 151

D. 200

Cuckoo Hashing: General Idea & Demo

- Use two (unrelated!) hash functions to hash into two different tables
- If you have a collision, the *new* item gets to stay and then other is kicked out (like Cuckoo bird chicks!)

http://www.lkozma.net/cuckoo_hashing_visualization/

Cuckoo Hashing: Running time

- What is the worst case for successful find in a hash table that uses Cuckoo hashing?

A. O(1)

B. O(logN)

C. O(N)

D. More than O(N)

Is the bound the same for unsuccessful find?

Applications of Hashing - Bloom Filters

0	1	0	1	0	0	0	1	0	1	0	1

Provides one-sided guarantees:

- If an item is in the set and you query for membership, it will be correct.
- If an item is not in the set and you query for membership, it will usually be correct but could be wrong.

How does it work? Hashing.

Applications of Hashing - Bloom Filters

0	1	0	1	0	0	0	1	0	1	0	1

What if you want to do spell check, but have VERY limited space?
What if you want to transmit a set of addresses visited with very limited space?

In each case, there’s a key 1-sided assumption.

Bloom Filters

0	1	0	1	0	0	0	1	0	1	0	1

How does it work?

- Insert based on multiple hashes – mark all as 1.
- Search based on multiple hashes – if all are 1, found, else, not found.

Bloom Filter Example

Example:
H1(k) = k mod 7
H2(k) = k*3 % 7

Add the following Members in the set: 5, 11

Find another element which would be considered part of the set which isn’t.

0	0	0	0	0	0	0

Applications of Hashing – The Heavy Hitters Problem

1	2	1	1	1	2	4	1	4	2	4	1

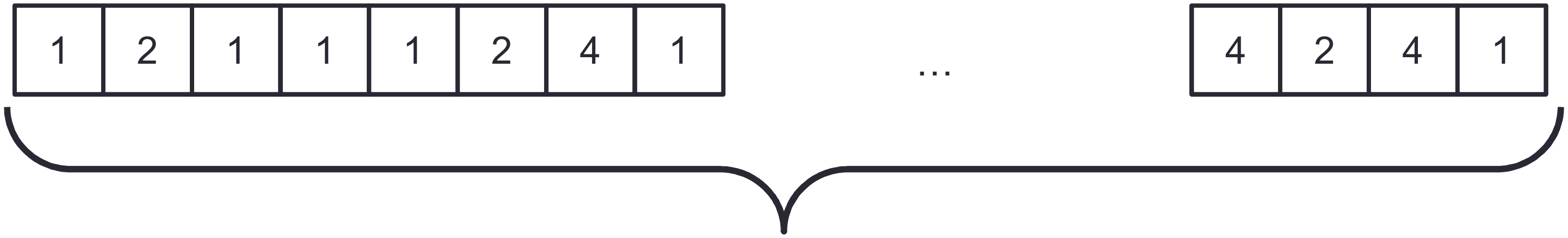
Given a set of elements of size *n,* and some (much) smaller value k
determine which elements occur at least n/k times.

In the example above for:
n=12
k=3

Which elements should be returned?

A. {1}
B. {1, 4}
C. {1, 4, 2}
D. {}

The Heavy Hitters Problem



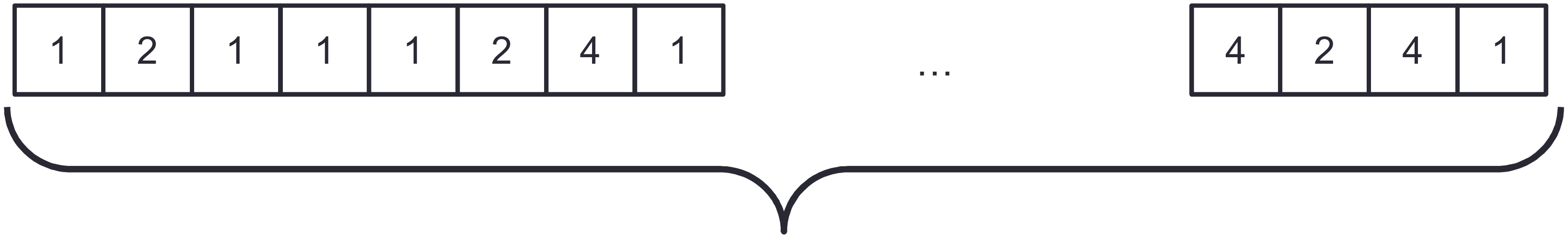
1 2 1 1 1 2 4 1 … 4 2 4 1

n is generally VERY LARGE (millions or billions)
You can't store it in memory, or even perhaps on disk!

Given a set of elements of size *n,* and some (much) smaller value k determine which elements occur at least n/k times.

What are the real-world applications of this problem?

The Heavy Hitters Problem: A sad fact



1	2	1	1	1	2	4	1	…	4	2	4	1

n is generally VERY LARGE (millions or billions)
You can't store it in memory, or even perhaps on disk!

Given a set of elements of size *n*, and some (much) smaller value k determine which elements occur at least n/k times.

There is no single pass algorithm that can solve this problem in a sublinear amount of auxiliary space!

Building Toward Count-Min Sketches: Sacrificing Exact Solutions for Large Savings!

Given a set of elements of size *n,* and some (much) smaller value k determine which elements occur at least n/k times.

Example:
n=12
k=2

1	35	10	1	1	10	1	35	1	35	16	1
1	3	2	1	1	2	1	3	1	3	0	1

Idea: Use a small Hash Map (of size b << n) to store counts

Example: b = 4, H(k) = k mod 4

1	6	2	3

What's the problem here?

Building Toward Count-Min Sketches: Sacrificing Exact Solutions for Large Savings!

Given a set of elements of size *n,* and some (much) smaller value k determine which elements occur at least n/k times.

	1	35	10	17	9	14	1	35	17	35	16	9
H(k) = k mod 4:	1	3	2	1	1	2	1	3	1	3	0	1

Example:
n=12
k=2

Idea: Use a small Hash Map (of size b << n) to store counts

Example: b = 4, H(k) = k mod 4

1	6	2	3

Any ideas on how to improve this?

Count-Min Sketch:
Sacrificing Exact Solutions for Large Savings!

Given a set of elements of size *n,* and some (much) smaller value k determine which elements occur at least n/k times.

1	35	10	17	9	14	1	35	17	35	16	9

Example:
n=12
k=2

Idea: Use ℓ small Hash Maps (of size b << n) to store counts

Example: ℓ=3, b = 4

$H_0(k) = (k\%5)\ \%\ 4$				
$H_1(k) = (k\%7)\ \%\ 4$				
$H_2(k) = (k\%13)\ \%\ 4$				
	0	1	2	3

Count in each hash function, advance slides to get to the example

Count-Min Sketch:
Sacrificing Exact Solutions for Large Savings!

Idea: Use ℓ small Hash Maps (of size b much less than n) to store counts

Given a set of elements of size *n*, and some (much) smaller value k determine which elements occur at least n/k times.

Example:
n=12
k=2

	1	35	10	17	9	14	1	35	17	35	16	9
$H_0(k) = (k\%5) \% 4$												
$H_1(k) = (k\%7) \% 4$												
$H_2(k) = (k\%13) \% 4$												

Have to compute hash for each number…

Example: ℓ=3, b = 4

$H_0(k) = (k\%5) \% 4$				
$H_1(k) = (k\%7) \% 4$				
$H_2(k) = (k\%13) \% 4$				
	0	1	2	3

Count-Min Sketch:

Idea: Use ℓ small Hash Maps (of size b much less than n) to store counts

Sacrificing Exact Solutions for Large Savings!

Given a set of elements of size *n*, and some (much) smaller value k determine which elements occur at least n/k times.

Example:
n=12
k=2

	1	35	10	17	9	14	1	35	17	35	16	9
$H_0(k)$ = (k%5) % 4	1	0	0	2	0	0	1	0	2	0	1	0
$H_1(k)$ = (k%7) % 4	1	0	3	3	2	0	1	0	3	0	2	2
$H_2(k)$ = (k%13) % 4	1	1	2	0	1	1	1	1	0	1	3	1

Start tallying and stop ~1/2 done

Example: ℓ=3, b = 4

	0	1	2	3
$H_0(k)$ = (k%5) % 4				
$H_1(k)$ = (k%7) % 4				
$H_2(k)$ = (k%13) % 4				

Count-Min Sketch: Sacrificing Exact Solutions for Large Savings!

Idea: Use ℓ small Hash Maps (of size b much less than n) to store counts

Given a set of elements of size *n,* and some (much) smaller value k determine which elements occur at least n/k times.

Example:
n=12
k=2

	1	35	10	17	9	14	1	35	17	35	16	9
$H_0(k) = (k\%5)\ \%\ 4$	1	0	0	2	0	0	1	0	2	0	1	0
$H_1(k) = (k\%7)\ \%\ 4$	1	0	3	3	2	0	1	0	3	0	2	2
$H_2(k) = (k\%13)\ \%\ 4$	1	1	2	0	1	1	1	1	0	1	3	1

Example: ℓ=3, b = 4

$H_0(k) = (k\%5)\ \%\ 4$	7	3	2	0
$H_1(k) = (k\%7)\ \%\ 4$	4	2	3	3
$H_2(k) = (k\%13)\ \%\ 4$	2	8	1	1
	0	1	2	3

What is the estimate for 1?

A. 1
B. 2
C. 3
D. 4
E. Other

Count-Min Sketch: Sacrificing Exact Solutions for Large Savings!

Idea: Use ℓ small Hash Maps (of size b much less than n) to store counts

Given a set of elements of size *n,* and some (much) smaller value k determine which elements occur at least n/k times.

Example:
n=12
k=2

	1	35	10	17	9	14	1	35	17	35	16	9
$H_0(k) = (k\%5)\ \%\ 4$	1	0	0	2	0	0	1	0	2	0	1	0
$H_1(k) = (k\%7)\ \%\ 4$	1	0	3	3	2	0	1	0	3	0	2	2
$H_2(k) = (k\%13)\ \%\ 4$	1	1	2	0	1	1	1	1	0	1	3	1

Example: ℓ=3, b = 4

$H_0(k) = (k\%5)\ \%\ 4$	7	3	2	0
$H_1(k) = (k\%7)\ \%\ 4$	4	2	3	3
$H_2(k) = (k\%13)\ \%\ 4$	2	8	1	1
	0	1	2	3

Which of the following values are overestimated?

A. 10
B. 17
C. 9
D. 16
E. More than one of these

Which value is overestimated by the most?

Count-Min Sketch

- ℓ (number of hash functions) and b (size of hash tables) can be much smaller than n. Size of storage depends on k not n!
- Updating in the table is fast: O(1), assuming hash functions are fast
- There is a very high probability that the overestimate of an number will be "small"
- Count-Min Sketches can solve an approximation of the Heavy Hitters problem (think about how)