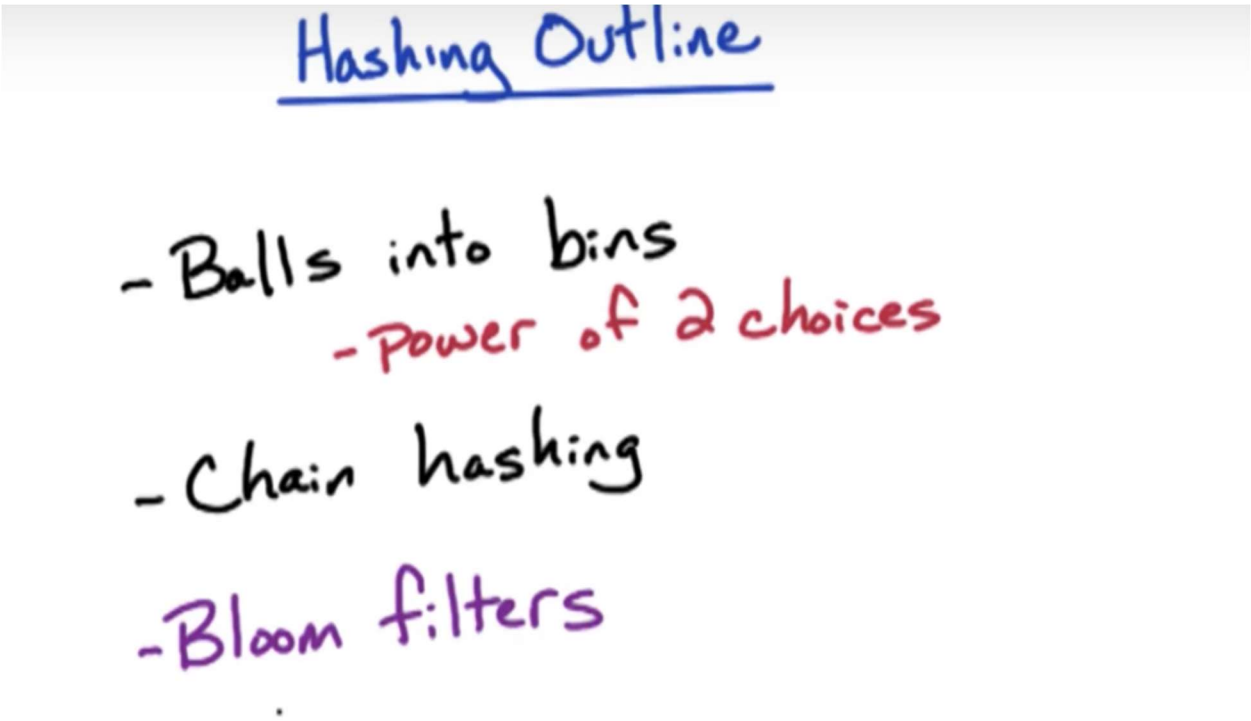
LESSON: RA3: Bloom Filters (Optional)
(Slide 1) Hashing Outline
***



In this lecture, we're going to talk about hashing and we're going to describe a popular technique called Bloom filters.

Before we get into hashing, we're going to talk about a toy problem - Balls into bins. We are going to throw balls randomly into a set of bins. We'll do some simple probabilistic analysis of this problem and this will give us some intuition about the design of hashing schemes.

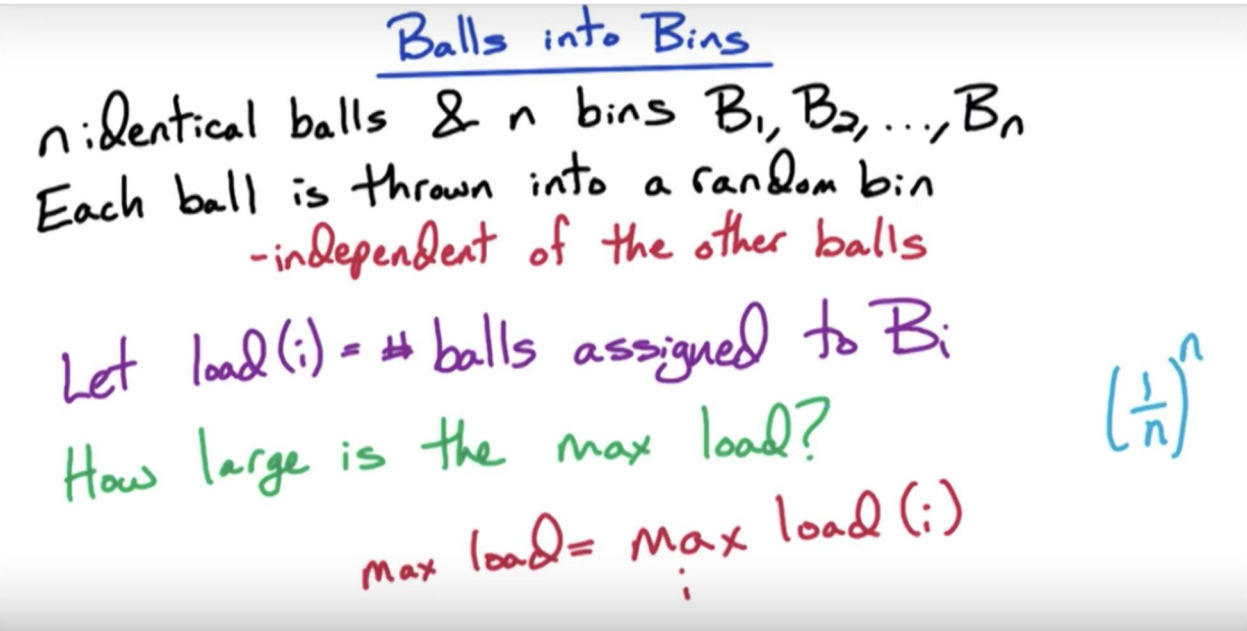
One of the neat ideas we're going to see with the Balls and bin's problems is the power of two choices. This is going to motivate some of our more sophisticated hashing schemes.

After we look at the toy problem, Balls into bins, then we'll look at the traditional hashing scheme: chain hashing. And then we'll look at our more sophisticated scheme: Bloom filters.

So let's dive into the Balls into bins.

***

(Slide 2) Balls Into Bins



For this toy problem, we have n balls, which are identical to each other, and we have n bins, which we'll label B1, B2, up through Bn. Each ball is thrown into a random bin, and this is done independent of the other bins.

Now, what we're going to do for this toy problem, is we're going to throw each ball into a random bin. The process for each ball is independent of what happened for the other balls. We want to look at the number of balls that are assigned to each bin. Therefore, we look at the random variable load(i), which is the number of balls assigned to bin i.

What we're interested in is the maximum load. This is the maximum number of balls in any particular bin. In other words, the max load is the max over i of load(i). How large can the max load get? - well, in the worst case, all of the balls might get assigned to the same bin. But, what's the chance of that? - It's quite small.

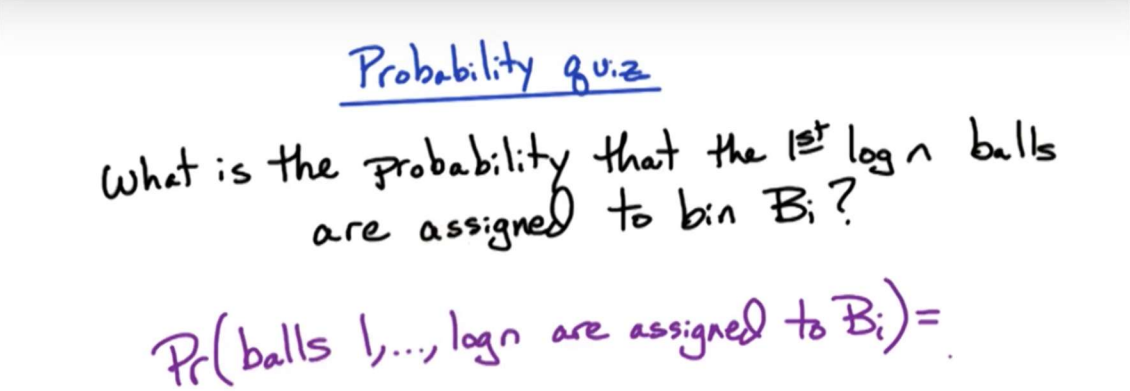
I mean, what's the chance of all n balls being assigned to bin B1, let's say. The probability of one particular ball being assigned to bin B1 is 1/n. What's the chance that all n balls get assigned to this particular bin? $(1/n)^n$. This is really tiny – so, this is very unlikely. In the worst case, the max load could be n because of such a scenario.

But, what is the typical scenario? How large is the max load typically? That's what we want to analyze now. We want to make a statement such as: "with high probability, the max load is,

some quantity, such as: square root n, log n, O(1). So, let's dive in and see what the max load is in a typical scenario.

***

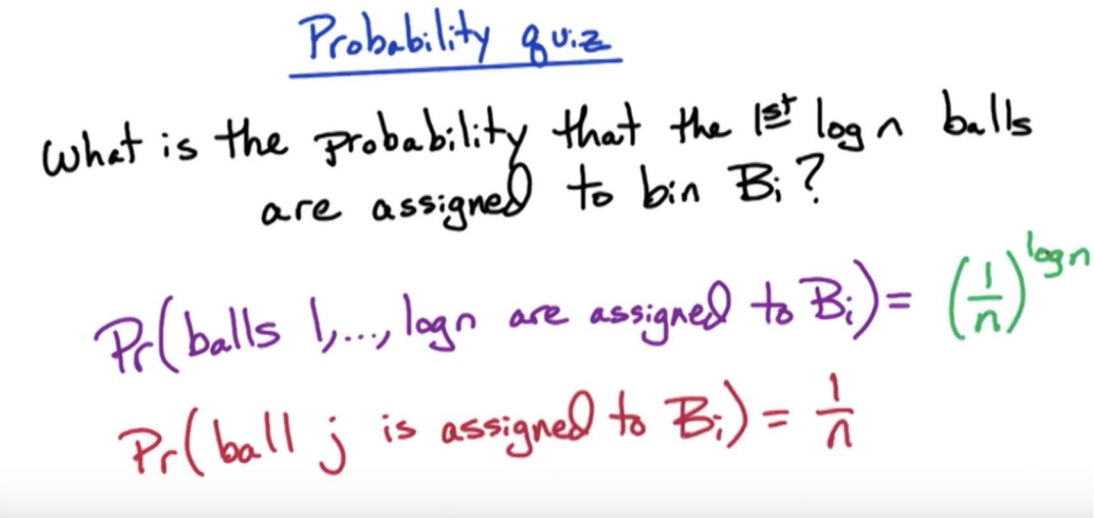
(1st Slide 3) Probability Quiz



Let's take a short probability quiz to give a quick refresher on some basic probability. Let's look at what is the probability that the first log n (log(n)) balls are assigned to bin Bi for a particular i. We want to look at the probability that the first log n balls are assigned to this particular Bi. Go ahead and write the quantity here.

***

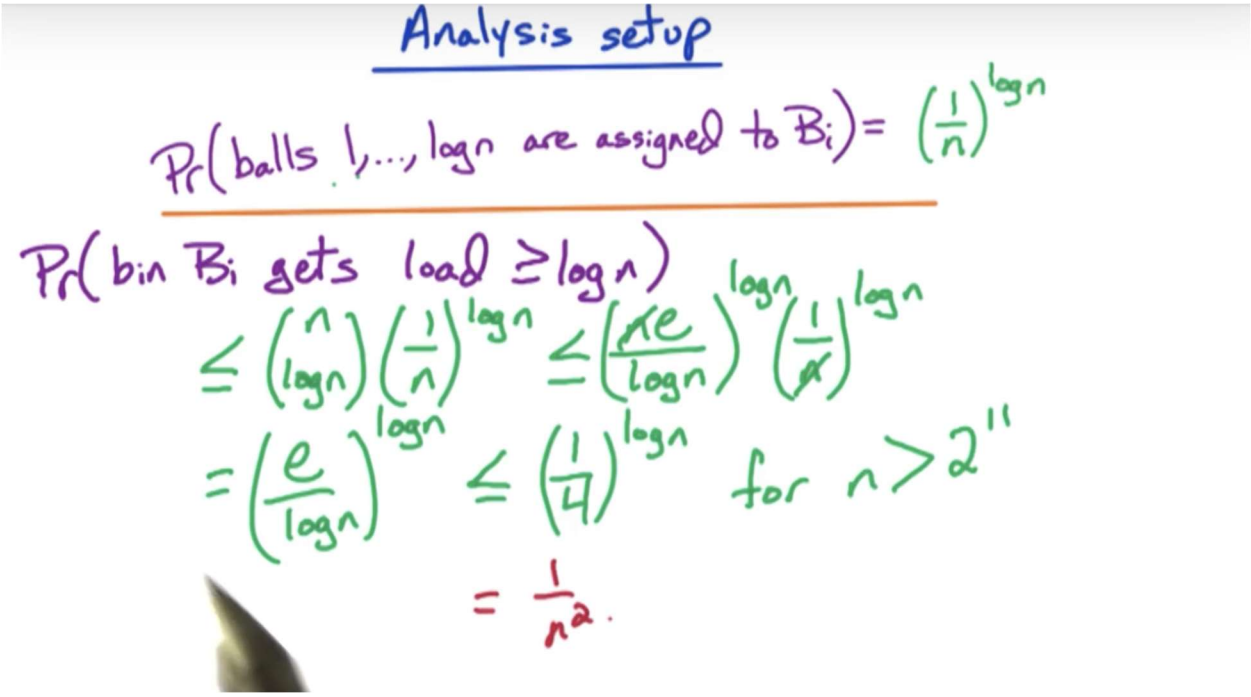
(2nd Slide 3) Probability Quiz (Answer)



Now, the solution is $(1/n)^{\log n}$. Why is that? Let's look at it more closely. Take a particular ball j - What is the probability that this ball is assigned to this particular bin Bi? Well, the ball is going to a randomly chosen bin, so the chance of going to any particular bin is 1/n. Therefore, the first ball has probability 1/n of being assigned to bin Bi. The second ball – same - has a probability 1/n of being assigned to bin Bi, and so on for all these log n balls. So the probability that all of them are assigned to bin Bi is the product of 1/n, for each - So, the total probability is $(1/n)^{\log n}$.

***

(Slide 4) Analysis Setup



Now let's dive into the analysis of the maximum load. What we've seen, so far, is that for a particular set of log n balls, the probability that these log n balls are assigned to a particular bin Bi is $(1/n)^{\log n}$. We're going to try to show that the max load is typically at most log n.

Now, what does typically mean? - we're going to have to detail exactly what we mean by that. But, in order to prove that the max load is at most log n, we want to show that the probability that a particular bin Bi gets load greater than log n - and we want to upperbound that probability and show that it's small.

It's unlikely to get a load at least log n. In order for bin Bi to get loaded at least log n, a particular set of log n balls have to get assigned to bin Bi (now, maybe more than log n balls get assigned to Bi -- but, we know that there's at least log n balls that are assigned to Bi). So, we're going to get an upper bound on this probability (Pr(bin Bi gets load >= log n)

First, we have to choose the particular set of log n balls that are going to be assigned to bin Bi.

- How many ways are there of choosing the log n balls? -- there's n choose log n : $\binom{n}{\log n}$.
- Now, for this particular set of log n balls, what's the chance that they are all assigned to Bin Bi? -- well, that's what we found before -- that's $(1/n)^{\log n}$.

Pr(bin Bi gets load >= log n) <= $\binom{n}{\log n}$ $(1/n)^{\log n}$

- Now, what happens for the other (n–log n) balls? Well, some of them might get assigned to this bin Bi as well, in which case we may be counting these events multiple times. So we're getting an upper bound on this probability.

  Notice, that if we added an extra term here which is $(1 - 1/n)^{n-\log n}$:

  Pr(bin Bi gets load >= log n) <= $\binom{n}{\log n}$ $(1/n)^{\log n}$ $(1 - 1/n)^{n-\log n}$

  This is saying that all of the other balls besides these log n balls that we chose are assigned to other bins.

  So, what is the probability that a particular ball is not assigned to bin Bi? That's (1 – 1/n) -- (1 – 1/n) balls that are not assigned to bin Bi.

  Then what is this bound-- $\binom{n}{\log n}$ $(1/n)^{\log n}$ $(1 - 1/n)^{n-\log n}$? -- this is actually equal to the probability that this bin Bi gets load exactly log n.

  But that's not what we want to bound. We want to bound the probability that the bin gets at least log n. So, we want to get an upper bound. We ignore where the other balls, the other (n – log n) balls are assigned. And, then, we get an upper bound on the probability because we allow these balls -- these (n – log n) balls to be assigned to any bin -- maybe Bi or maybe a different Bin.

Now let's try to get a handle on this term, $\binom{n}{\log n}$:

- Let's look at it more generally, $\binom{n}{k}$. Recall what is $\binom{n}{k}$ – – "n choose k":

  $$\binom{n}{k} = \frac{n!}{(n-k)!\,k!}$$

  It's n factorial over and minus K factorial times K factorial. If we expand this out we got n! on the numerator, but all the terms from n-k downwards cancel out with this n-k factorial in the denominator.

$$\binom{n}{k} = \frac{n!}{(n-k)!k!} = \frac{n\,(n-1)\cdots(n-k+1)}{(k)\,(k-1)\cdots(1)}$$

So we get n times n-1, down to n-k+1. The remaining terms again cancel with this (n-k)! on the denominator. And, then, also in the denominator what are we left with? We're left with k factorial which is k times k-1 times k-2, and, so on, down to 1. Let's try to get a handle on this quantity.

Notice the first term is n/k. The second term is similar, it is (n-1)/(k-1). If n is big, that's pretty similar to n/k, and so on. So we have n/k, (n-1)/(k-1), and so on down to (n-k+1)/1.

$$\binom{n}{k} = \frac{n!}{(n-k)!k!} = \frac{n\,(n-1)\cdots(n-k+1)}{(k)\,(k-1)\cdots(1)} = \frac{n}{k}\,\frac{(n-1)}{(k-1)}\,\frac{(n-2)}{(k-2)}\cdots\frac{(n-k+1)}{(1)} \approx (n/k)^k$$

So there's k quantities, k ratios there – so here are the k ratios. Each one, let's say, is approximately n/k. So this is approximately $(n/k)^k$:

Actually, this approximation is not too bad of a bound on this quantity. What one can show using Stirling's formula, is that $\binom{n}{k} <= (n\ e\ /\ k)^k$. So if we put an extra factor of e in the numerator, then we get a rigorous upper bound on $\binom{n}{k}$. And that's what we're aiming for. We're aiming for an upper bound on the load size of bin Bi.

- So we can upperbound $\binom{n}{\log n}$ by using this formula -- so, plugging in this bound for our case, we have k equals log n in our scenario. So we get the upper bound $(ne/\log n)^{\log n}$ … that's for the $\binom{n}{\log n}$.

and then we still have this term $(1/n)^{\log n}$:

$$\Pr(\text{bin Bi gets load} >= \log n) <= \binom{n}{\log n}\,(1/n)^{\log n} <= (ne/\log n)^{\log n}\ (1/n)^{\log n}$$

- Now, these terms are both raised to the power of log n, so we can cancel out this n with this n. So what are we left with? We're left with $(e\ /\ \log n)^{\log n}$:

$$\Pr(\text{bin Bi gets load} >= \log n) <= (ne/\log n)^{\log n}\ (1/n)^{\log n} = (e\ /\ \log n)^{\log n}$$

- Now notice the denominator is growing with n, whereas the numerator is a fixed constant. So as n grows this becomes smaller and smaller. We're going to look at this

asymptotically as a function of n so we can bound this inner term by any fixed cost and we want.

So let's bound it by the constant one quarter:

$$\Pr(\text{bin Bi gets load} >= \log n) <= (e / \log n)^{\log n} <= (1/4)^{\log n}$$

So, we're going to say (e / log n) is at most one fourth, and so we get this whole quantity is bounded by one fourth to the log n.

Now what is the bound that we used here on n? We used the fact that log n is bigger than four times e. When is that true? That's true when n is big enough. In particular, if n is bigger than 2^{11} then log n is bigger than four times e, so we can replace (e / log n) by 1/4th.

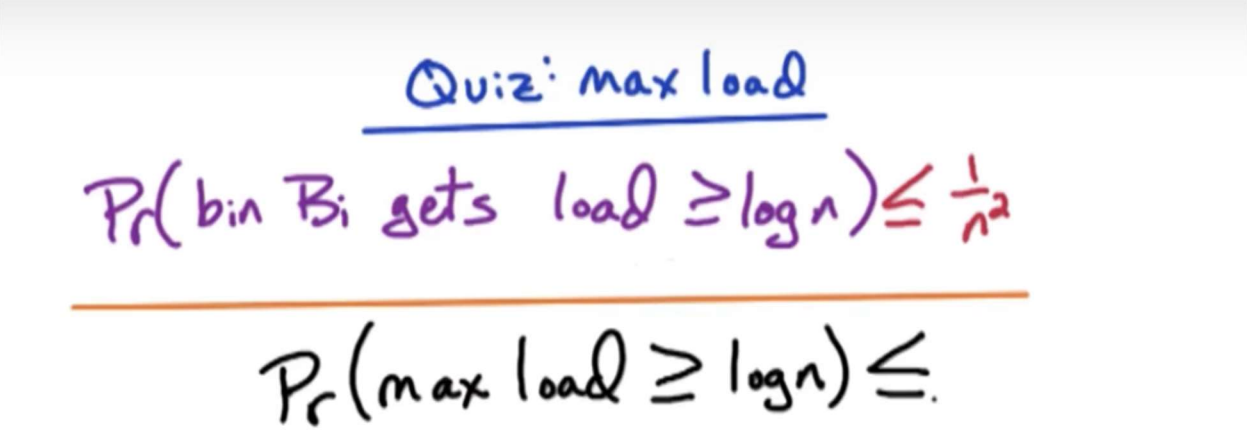
Now what is the nice thing about using one fourth here? Well, assuming that the log was base two, then this quantity $(1/4)^{\log n} = 1/n^2$.

So in summary, we've shown that the probability that bin Bi gets load at least log n is at most $(1/n^2)$ which is tiny as n grows:

$$\Pr(\text{bin Bi gets load} >= \log n) <= (1/n^2)$$

***

(1st Slide 5) Max Load Quiz



Now what we saw in the last slide was that the probability that a particular bin Bi gets load at least log n is at most $(1 / n^2)$. Now, can we use that to bound the max load? Can we upper bound the probability that the max load is at least log n? Why don't you go ahead and write this quantity here: Pr(max load >= log n) <= ?

If you have trouble doing it, don't worry, we'll go through it in a minute.

***

RA3: Bloom Filters: Max Load Quiz

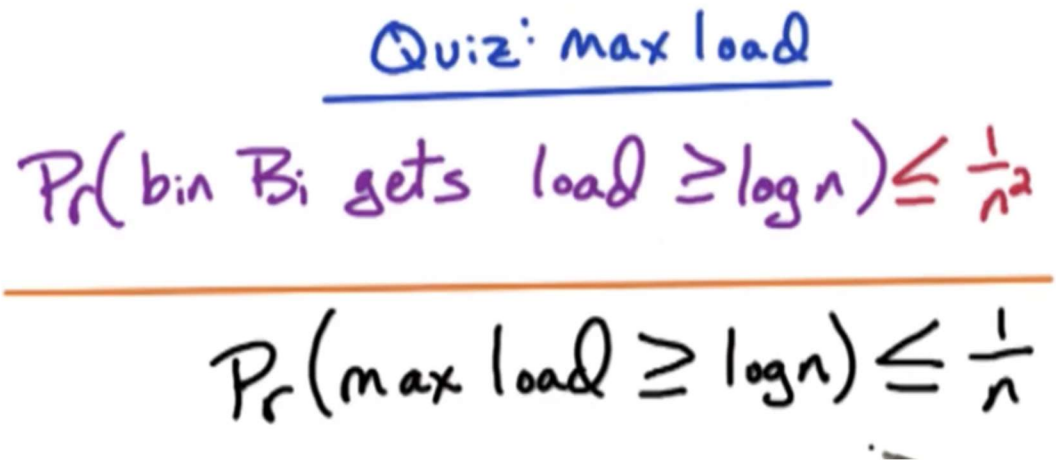
Question:
What is the probability that the maximum load is at least log n?

Pr(max load >= log n) <= ?
(Multiple choice – select one answer)

- 1/n
- $1/n^2$
- $2/n^2$
- 1/2

***

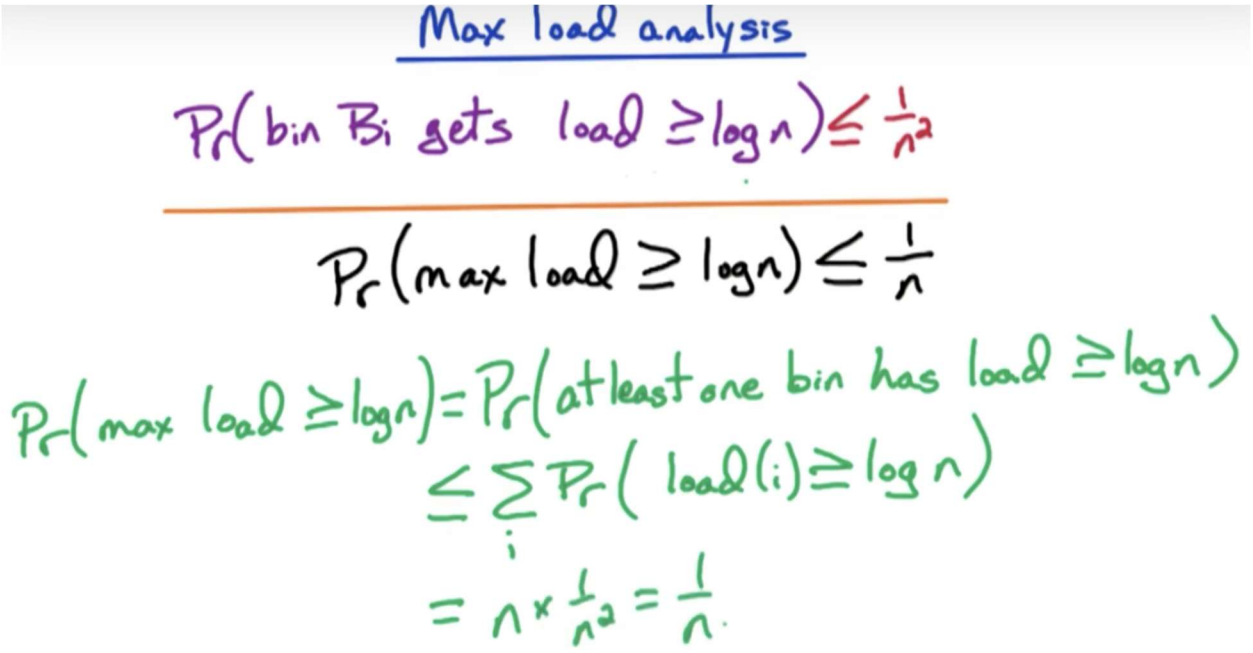
(2nd Slide 5) Max Load Quiz (Answer)

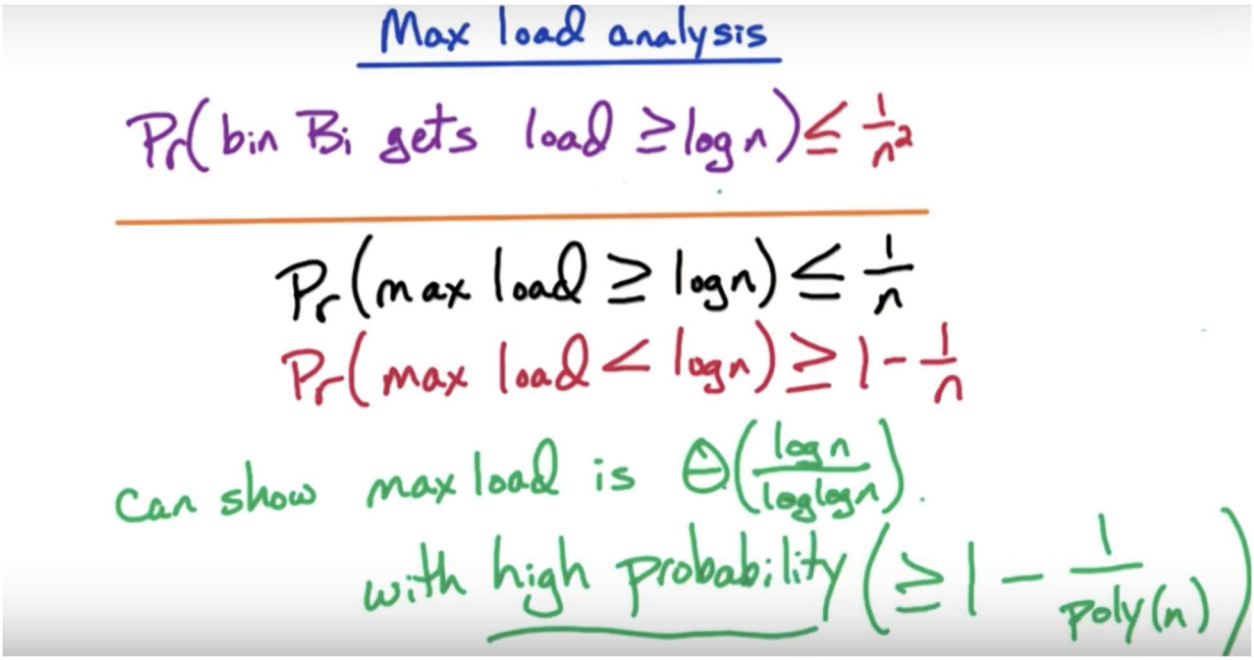


The solution is 1/n, the probability that the max load is at least log n, is at most 1/n. Let's go through it in more detail to see why this is the case.

***

(Slide 6) Max Load Analysis





Now we want to bound the probability that the max load is at least log n - that's the quantity here.

In order for the maximum load to be at least log n, at least one bin -- maybe several -- have to

have a load at least log n:

$$\Pr(\text{max load} \geq \log n) = \Pr(\text{at least one bin has load} \geq \log n)$$

In order to upperbound this probability, we can use our earlier analysis. We know the probability that a particular bin gets load >= log n is at most $1/n^2$.

So now, in order for at least one bin to gets load >= log n, let's look at all the choices for the bins. So we can sum over each bin, and then we can look at the probability that the load in bin Bi is at least log n, which is what we bounded right here. That's at most $1/n^2$. There are n choices for the bin, and for each particular bin the probability the load is at least log n is at most $1/n^2$, and this is 1/n:

$$\begin{aligned}\Pr(\text{max load} \geq \log n) &= \Pr(\text{at least one bin has load} \geq \log n) \\ &\leq \Sigma \Pr(\text{bin Bi gets load} \geq \log n) = n \times 1/n^2 = 1/n\end{aligned}$$

which proves the result that the max load is at least log n with probability at most 1/n.

Now, we want to look at the complementary event -- that the max load is strictly less than log n What is the probability of this? Well, this event is unlikely to happen. This ("max load >= log n") happens with a small probability, this ("max load < log n") is going to happen with a large probability. In particular, it happens with probability at least 1 – 1/n.

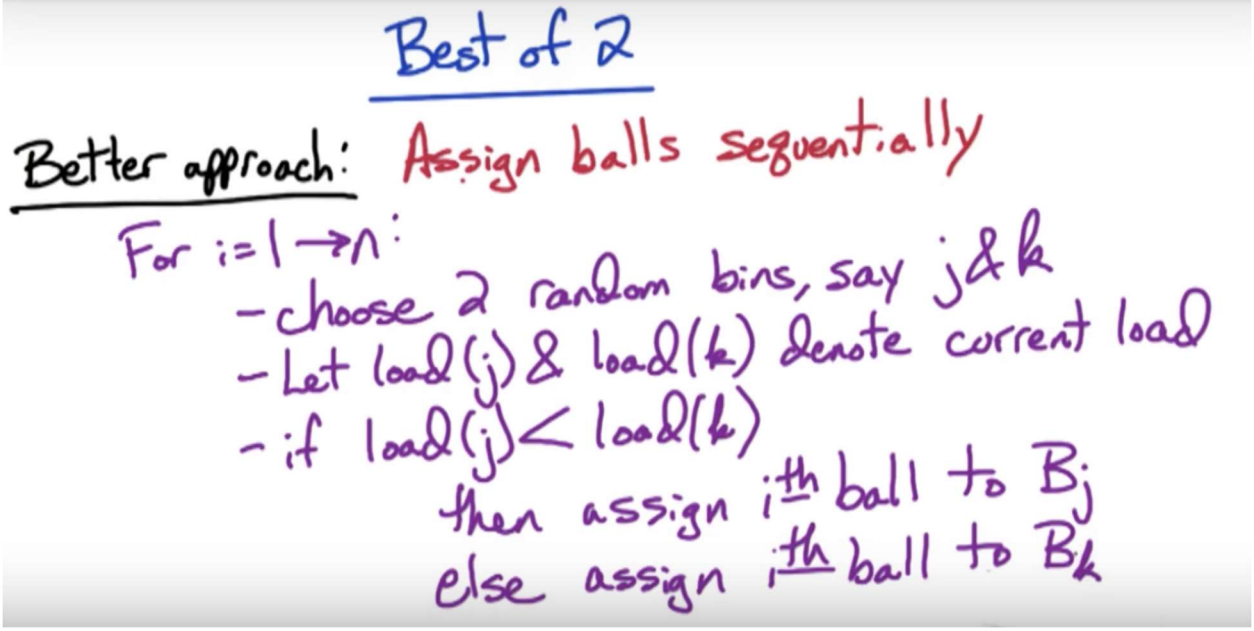
And in fact, one can do a similar analysis -- a little bit more carefully -- and you can show that the max load is $\Theta(\log n / \log \log n)$. So we can get an upper bound and a lower bound which are within constants of each other.

So, instead of an upper bound of log n, on the max load, we can get a slightly better bound of log n / log log n. And this error probability ("Pr(max load >= log n) <= 1/n") can be boosted from 1/n to $1/n^2$ or $1/n^{10}$, any polynomial, and then, we can get here by changing the constant up front.

And when this error probability is 1 over a polynomial in n, we say that this event happens with high probability. In particular, if it happens with high probability, then that means it happens with probability at least 1 – 1/poly(n), and we can make this polynomial as small as we want by increasing this constant ($\Theta(\log n / \log \log n)$) up front.

***

(Slide 7) Best of Two Scheme



Now, let's look at the following twist on the balls and bins problem. This is going to be a better approach, better in the sense that it's going to reduce the max load.

Now, in the previous balls and bins problem, each ball was going to a random bin and it was independent of the assignment for the other balls. Therefore, we could have assigned all the balls simultaneously, in parallel. They could have all been assigned to bins at the same time, or we could assign them one by one, sort of sequentially. So, we could have taken ball 1 and assigned it to a random bin, take ball 2, assign to a random bin, and so on, up to ball n. That's what we're going to do here. We're going to assign the balls sequentially, and we're going to look at a slight twist of the other scheme.

So, let's go through the balls from 1 through n. Index i will correspond to the current ball that we're considering.

In the old scheme, we choose one random bin say j, and we assigned the i ball to bin bj.
Now, in the new scheme what we're going to do is, we're going to choose two random bins, say j and k. Now, which bin are we going to assign the ith ball? Well, we're going to assign it to the better of these two bins. What exactly do we mean? We mean the one which has smaller load.

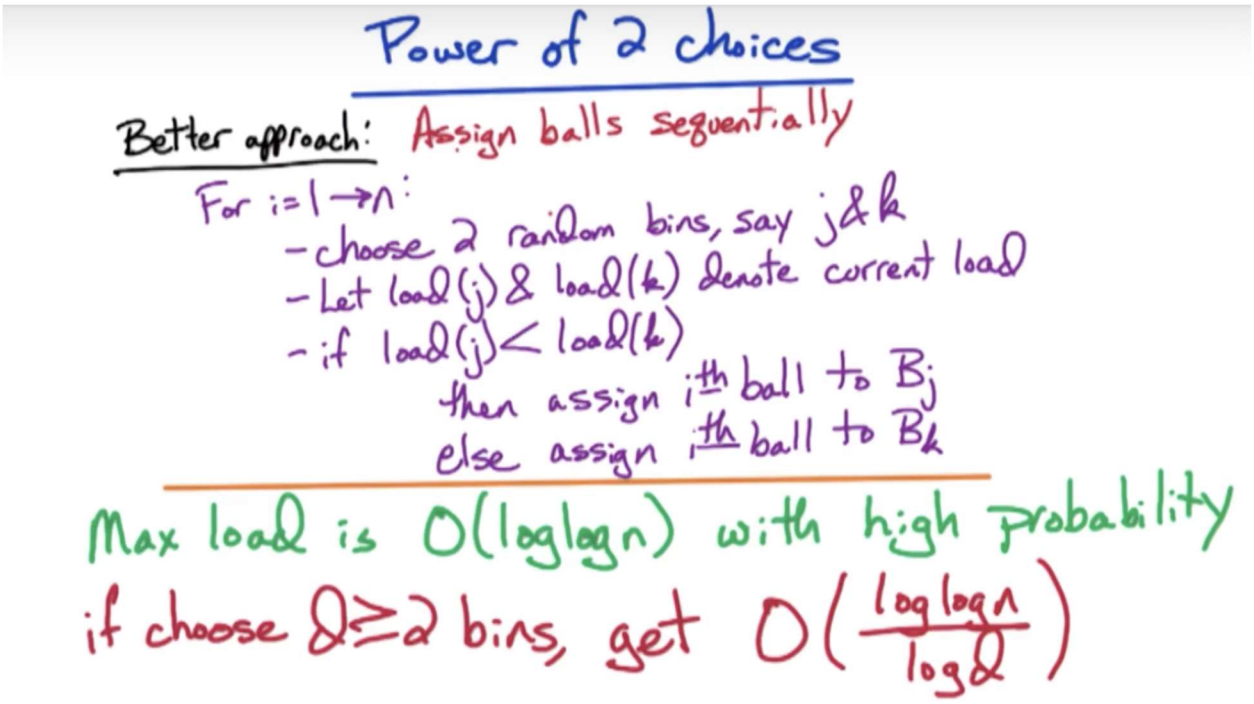
So, let load(j) and load(k) denote the current load in these particular bins Bj and Bk, and we're going to assign this ith ball to the least loaded of these two. So if the load of Bj is structurally smaller than the load of Bk, then we assign the ith ball to bin Bj. And in the other case, we

assign the ith ball to bin Bk.

So, the ith ball is going to the least loaded of these two bins.

***

(Slide 8) Power of Two Choices



This new best of two approach is a simple twist on the original approach. What are we doing now? Well, instead of choosing a random bin, we choose two random bins, and we assign the ith ball to the least loaded of these two random bins.

It turns out that with this scheme, the maximum load is O(log log n), with high probability. Recall that the previous approach had max load on the order of roughly log n and we've reduced it from log n to log log n just by, instead of choosing one random bin, we choose two random bins, and we send to the best of the two.

This is a substantial gain because log log n is quite small, even for very large n. So this is almost like O(1) - it's very close, it's a very small quantity.

After seeing this result, you might say, "Well, why choose two random bins. " Let's choose three random bins and maybe we'll get log log log n." Well, it turns out that the big gain is from one to two, and, after that, there's not much gain. In particular, if you choose d >= 2 bins, so instead of choosing two random bins you choose d random bins, and you assign the ith ball to the least loaded of all of these d bins, then the max load is going to be O(log log n / log d). So the improvement with d is very small.

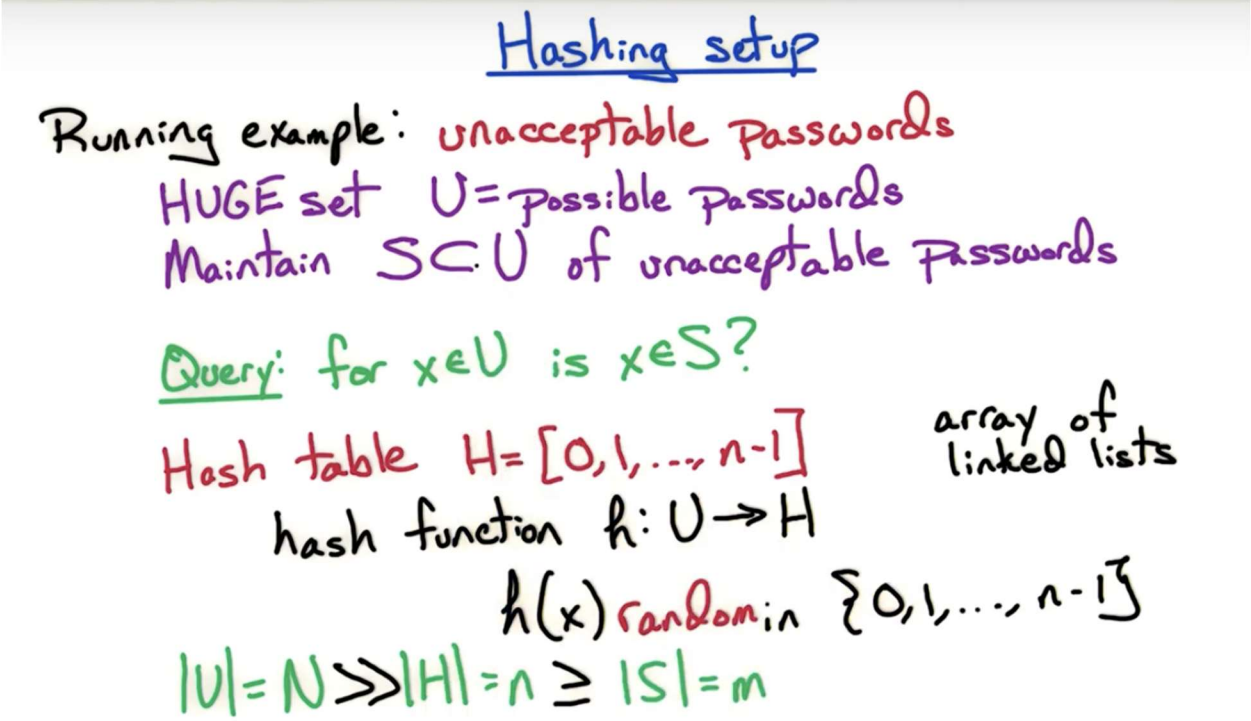
Now, keep this idea in mind for later, this idea of choosing the best of two choices. We're going

to use this intuition later to get better hashing schemes and then to drive the intuition behind the Bloom filters.

So finally, let's dive into hashing.

***

(Slide 9) Hashing Setup



Now, let's turn our attention back to hashing, which is our main focus. It will be useful to keep a running example in mind to motivate our various hashing schemes.

The example we'll use is unacceptable passwords. We want to maintain a database of unacceptable passwords. For example, these might be words that are in the dictionary. Now the setting is that a user will enter a proposed password and our system should quickly respond if that proposed password is acceptable or not. So, we need to quickly check whether the proposed password is in the database of unacceptable passwords or not.

Now let's formalize our setting a little bit more precisely:

- We have a huge set U which is the universe of possible passwords. Now, this is an enormous set. For example, if we simply look at passwords as strings or words of length A, then this set is of size 52^A, hence this set U is too large to maintain.
- Instead, we're going to maintain a subset of this universe, which we will denote as a subset S. S will contain the set of unacceptable passwords.
- The main operation our data structure needs to perform are queries. So, for an element X in our universe – So, X is a proposed password in this setting - is X in this subset S? So, is X an unacceptable password?

Now we want to build a data structure or hashing scheme, which answers these queries quickly.

Let's first look at how the traditional hashing scheme known as chain hashing works in this setting:

- Hash table: Now, in order to maintain this set S, we're going to use a hash table H of size n. In chain hashing, this table H is an array of linked lists.
- Hash function: We're going to use a hash function h which maps elements in U to elements in H. So each of the possible passwords is mapped to one of these n bins by h.
- Insertions: Now to insert an element into this subset S we simply find its hash value, then we go to that bin and then we add the element onto the linked list at that particular location.
- Queries: And then to do a query, we simply go to the hash value and we look at the linked list to check whether it's there or not.

For each element x of the universe U, h(x) maps to one of these n bins.

Now we're going to analyze random hash functions.

- So, we're going to assume that h(x) maps to a random bin. Moreover, we'll assume this choice, this random map, is independent of all other hashes. So where h(x) maps to is independent of where any other element of the universe maps to.
- So this h is a completely random hash function. Now if you think of this hash table as bins and you think of these elements in S as balls then what this hash function is doing is its assigning these balls into random bins. So, it's reminiscent of our balls into bins problem that we analyzed before.

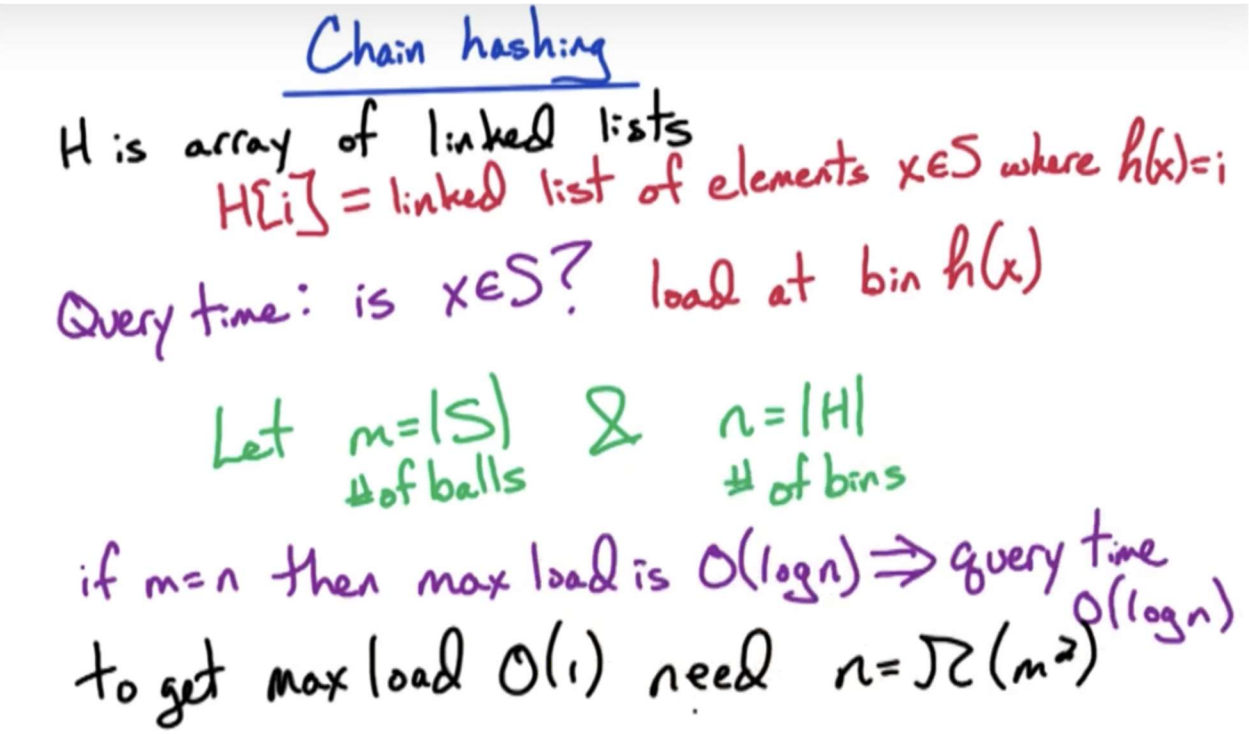
Now it will be useful to have a little bit of notation before we move on.

- This set U is huge, and we'll denote its size by N.
- The hash table -- we'll denote its size by n.
- N, as in RSA, will be exponential size in n,
- and we'll use m to denote the size of this database, S, that we're maintaining.

And once again, N is much bigger than n, and typically our hash table size is at least the size of the database we're maintaining. So, $n >= m$, and, our goal, of course, is to try to maintain this database as not much larger than m.

***

(Slide 10) Chain Hashing



Once again, in the traditional hashing scheme, chain hashing, H, our hash table, is an array of linked lists. H is an array of size n, and H[i], the ith element of H, is a linked list of elements in our subset S which map to i. So, in other words, H[i] is a linked list of those elements - So, those unacceptable passwords whose hash value is exactly i (i.e., h(x) = i).

Let's look at the query time. How long does it take us to answer a query of the form "Is X in a subset S (that we're maintaining)?". Now, in order to answer this query, what we have to do is look at the hash table at the index i, which is h(x), and then we have to go through that entire linked list and check whether x is in there in that linked list or not. So, the time it takes us is proportional to this size of this linked list.

What's the size of this linked list? It's the load at this bin. If we think of the elements of S as balls, these are getting assigned to bins, which are their hash values. The time it takes us to answer a query is proportional to the load size at the hash value h(x).

Let's introduce some notation:

- Let m be the size of our dictionary of unacceptable passwords (m = |S|),
- and let n, be the size of our hash table (n = |H|).

- So, in our balls in bins analogy, m is the number of balls that we're throwing in, and n is the number of bins.
- Now, if m = n, the number of balls is the same as the number of bins. This is the toy problem that we analyzed before, and what we saw is that the max load is O(log n), with high probability -- Of course, in the worst case, it might be O(n); n might be the max load; but that's an unlikely event -- With high probability, the max load is going to be O(log n), which means in the query time, in the worst case, it's going to be O(log n) with high probability.

Now, when n is huge, then O(log n) might be too slow for us. So how can we achieve faster query time? Well, one way is to try to increase the size of our hash table. In order to decrease this max load from O(log n) to O(1), so that the query time will be O(1) constant time queries, we're going to have to increase the size of the hash table from O(m) to $O(m^2)$.

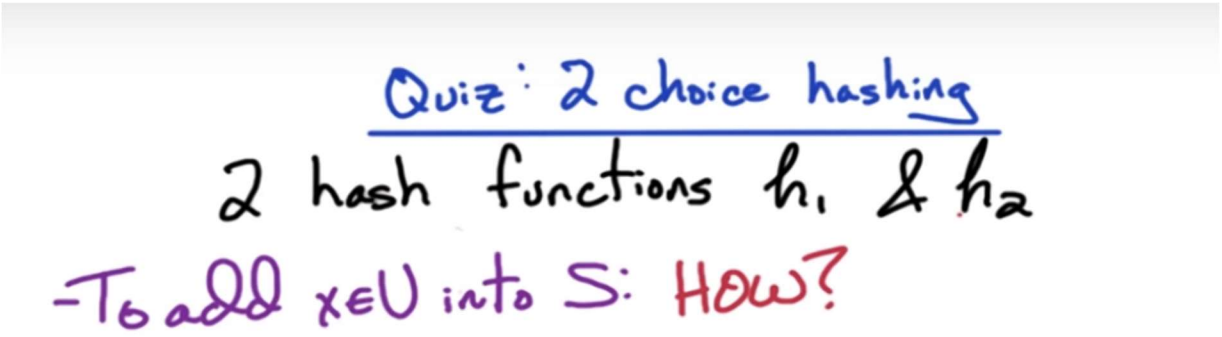
Now, that's quite a large price to pay. So, let's see if there are simpler ways to achieve reductions in the query time.

Now, our intuition for the following scheme comes from the balls and bins example from before. This is a simple scheme that we use right now. We're sending n balls into n bins. Each ball is going into a random bin. What do we use to improve that balls and bins scheme to improve the max load? Well, we use the two choice scheme, and what we saw is that the max load goes down to O(log log n), when we allow each ball to go to the least loaded of two random bins.

So, let's try and use a similar scheme now for hashing.

***

(1st Slide 11) Power of Two Hashing



What we're gonna do now is instead of using a single hash function we're going to choose a pair of hash functions h1 and h2. Each of these hash functions maps elements of U of our possible passwords into our hash table of size n.

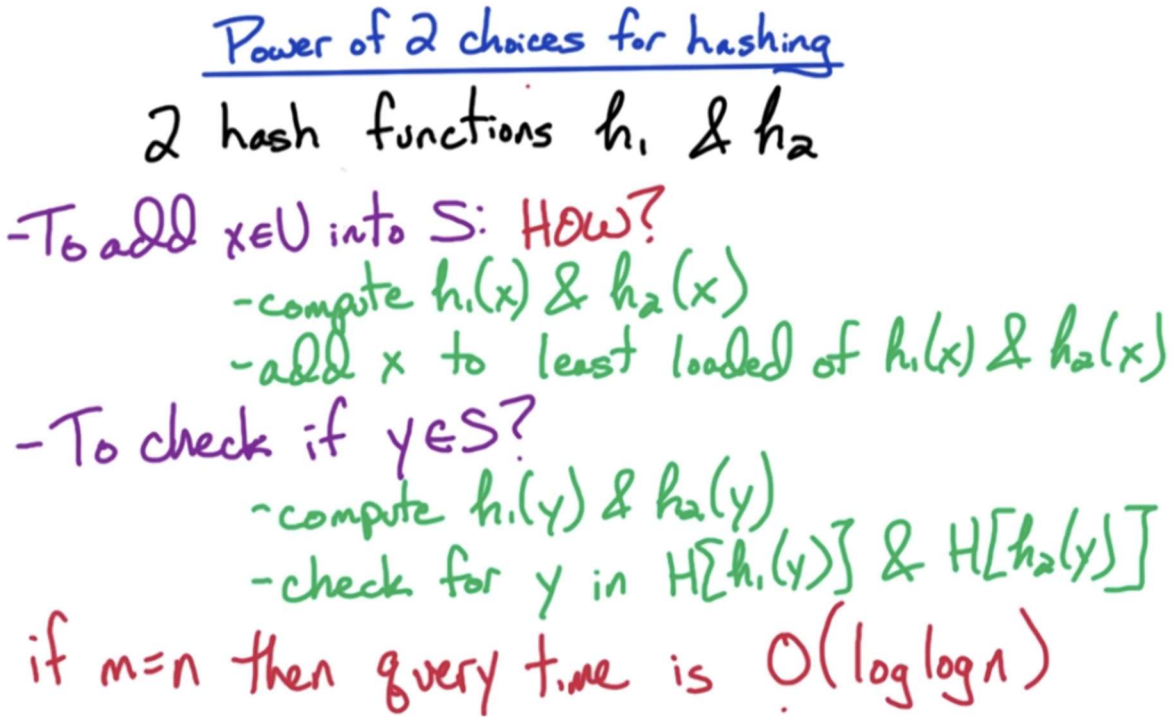
Now, we'll assume that these hash functions are random so each element acts in the universe of possible passwords maps to a random element of the hash table. h1(x) is random; h2(x) is random; and these are independent of each other and independent of the other hash values.

The first question is how do we insert an element, a possible password, into our dictionary of unacceptable passwords? In the traditional scheme, this is quite straightforward --what we do is we look at our hash function and we look at h(x) that tells us the hash value and now we look at the linked list at h(x) and we add element x onto that linked list.

But, now, we have two hash functions h1 and h2, so how do we do this insertion into our dictionary in this scenario, when we have two hash functions? -- This is a bit of an open-ended question; but, why don't you think about how you would insert an element into our dictionary using two hash functions.

***

(2nd Slide 11) Power of Two Hashing (Answer)



Let's go ahead and look at how we do this insertion.

- We want to insert this element x into our dictionary of unacceptable passwords: First thing we do is compute these two hash values so we compute h1(x) and h2(x).
- Think of our balls and bins analogy -- we have this ball x and what we've done is we've chosen two random bins h1(x) and h2(x). Which bin do we add the ball x into? --we added into the least loaded at all of these two bins. What is the load of the bin? --it's the size of the linked list. We can maintain the size of each of these linked lists so that we can quickly determine which of these two is least loaded and then we can add in x into that appropriate linked list and then we can increment the size of that linked list.
- So, this can all be done in O(1) time for an insertion.

Next question, is "how do we do a query?". How do we check whether an element y, a proposed password y, is in our dictionary of unacceptable passwords?

- We start off the same as an insertion: we compute the two hash values h1(y) and h2(y). These are the two possible locations for y. We have no way of determining which of these two locations it might be in, if at all, because we have no way of determining what the dictionary looked like at the time that we inserted y (if we did insert y).
- So what do we do? We check both bins. We check the linked list at h1(y) and we check the linked list at h2(y) and we look for y in both of these linked lists. So, we check the

linked list at h1(y) and we check the linked list at h2(y) and we look in both of these linked lists for the element y. If it's in either of these linked lists then we know that y is in the dictionary. If it's in neither of these linked lists, then we know that y was never inserted into the dictionary of unacceptable passwords.

- So how long does it take to do a query? --well, the query time now depends on the load at this location (h1(y)) and the load at this location (h2(y)). So, if we have an upper bound on the maximum load, then the query time is twice the maximum load.

Now if m = n -- so the size of our data dictionary of unacceptable passwords and the size of our hash table are the same – then, what we know from our balls and pinion analogy is that the query time -- the max load -- is gonna be O(log log n) in this scenario.

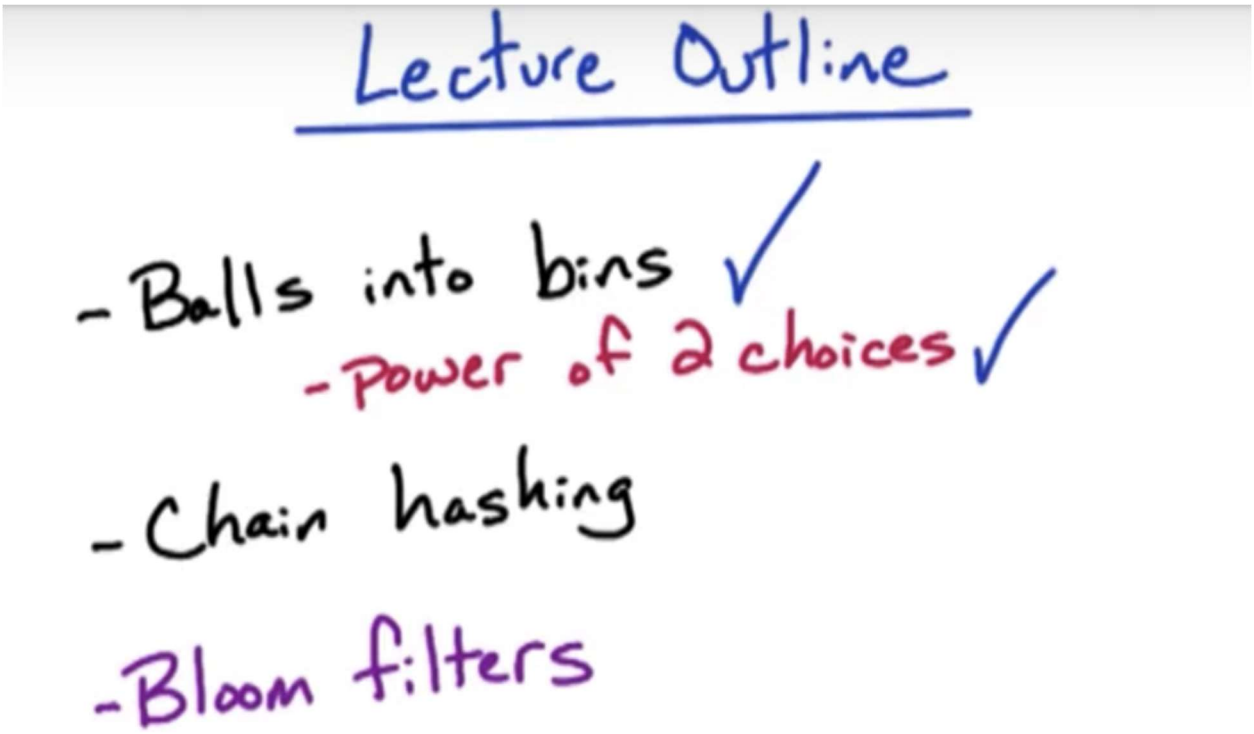
So just changing from one hash function to a pair of hash functions and using this scheme, then our query time goes down dramatically from O(log n) to O(log log n) and there's no extra cost in terms of the space; though, there is a question about how we maintain this hash function h especially if it's a truly random hash function.

In practice, we can't store a truly random hash function. Instead, we use pseudo-random hash functions. So, we use a hash function which we obtain from a library such as rand or drand48. But, for the purpose of our analysis, it's convenient to consider a truly random hash function so that we can do this nice analysis such as how we obtain the O(log log n) and max load for the m = n case --for the simple case of one hash function.

We skipped the analysis for the case of the balls and bins example where we did the two choices where we had each ball going to the best of two random bins. In that case, we claimed that the max load is O(log log n) -- that analysis is reasonable to do but it's much more complicated, so we skipped it in this lecture.

***

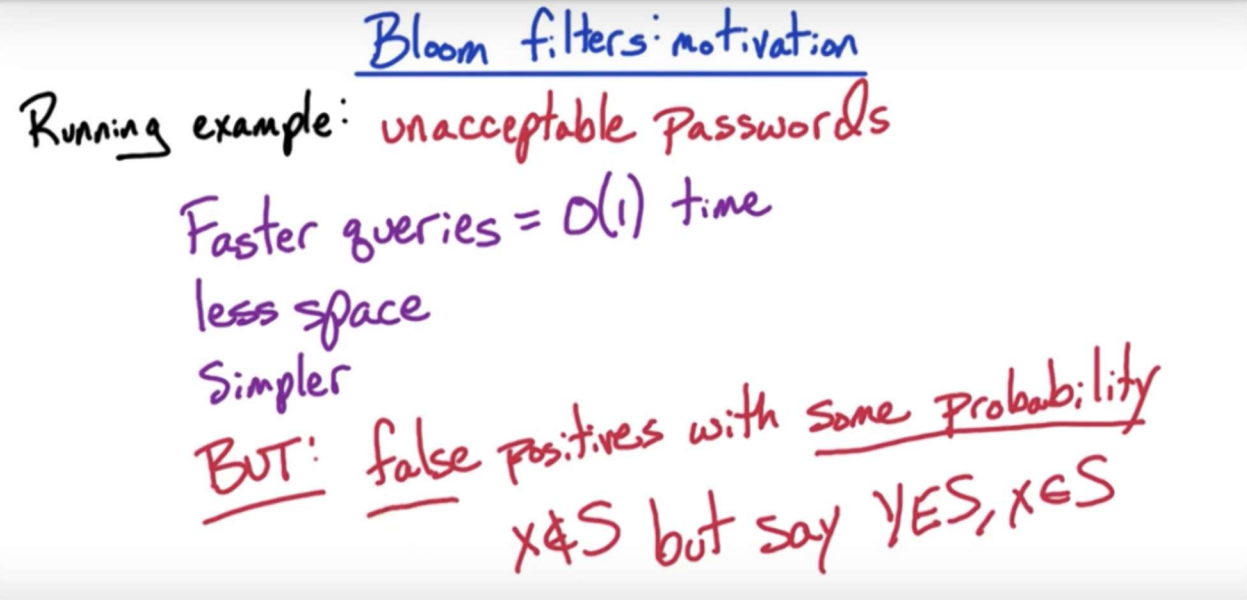
(Slide 12) Bloom Filters Lecture Outline



Now this completes our description of the traditional hashing approach -- the chain hashing. Now we can move on to the Bloom filters.

***

(Slide 13) Bloom Filters Motivation



Now we can finally describe Bloom filters. Let's keep in mind this running example from before the case of unacceptable passwords. We're going to describe a new data structure that has faster queries.

Recall, in the traditional hashing scheme that we previously described, the query time was O(log n) for the simple scheme or O(log log n) for the more advanced scheme which used the power of two choices that are ideal. Here we're going to achieve query time O(1). So constant query time, and this is guaranteed.

Recall, that the other query times were probabilistic statements. So, with high probability, the query time was O(log n) or O(log log n). In the worst case it was O(n). But, here, it's guaranteed to always be constant query time. This data structure will be very simple and it will use less space than before. There are no linked lists or anything like that. It will just be a simple binary array.

Now there are a lot of benefits. It's simpler, less space, faster queries. Now there must be some cost for this simplicity and this faster time. So, what is the cost, what is the tradeoff for this scheme? --well, this scheme is not always correct. Occasionally, there are false positives, and this happens with some probability that we'll analyze. We'll try to figure out what is this probability of false positives occurring.

What exactly do we mean by false positive? We have an element x which is not in our

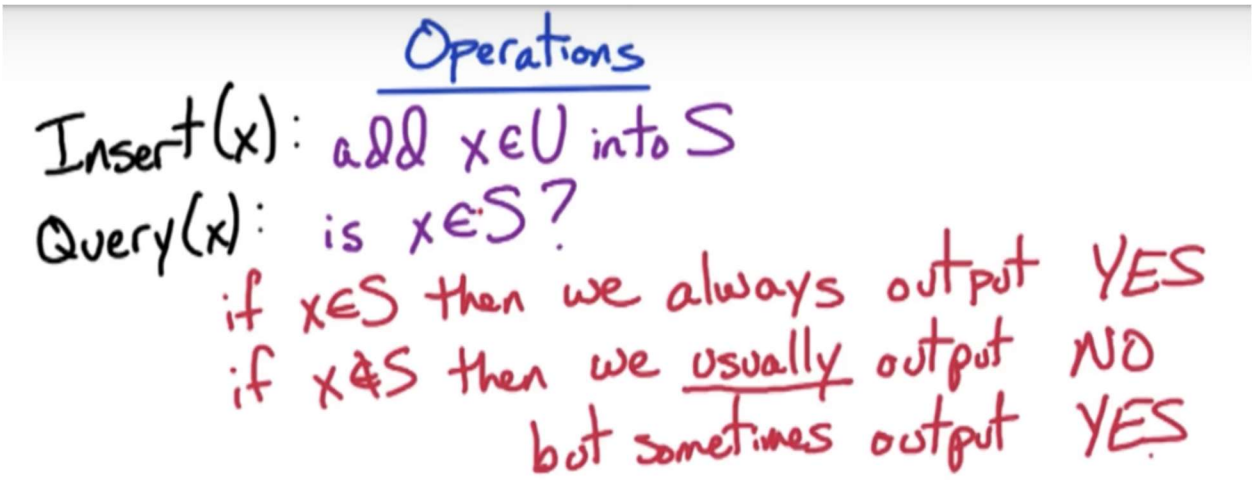
dictionary of unacceptable passwords. So this is an acceptable password, but our algorithm occasionally says, yes this X is in the dictionary. In this setting, false positives are acceptable. Why? Because we have an acceptable password but we say that the password is unacceptable. It's in our dictionaries. So we falsely say that the password is in our dictionary of unacceptable passwords. So somebody types in a password and we say, no that's not allowed. Ideally, it should have been allowed, but we said that, no is not allowed. So then the user has to enter a new password.

But, in exchange for these false positives, we have guaranteed query time. So, we answer the question of whether it was an acceptable password or not quickly. And, in this setting, false positives are reasonable. False negatives - that would have been a big cost - that would have been unacceptable in this setting. When we have an unacceptable password, we definitely want to say it's unacceptable. If we have an acceptable password, okay. If we occasionally say that it's an unacceptable password, that's okay. So in this setting, it's reasonable to have false positive with some small probability that we'll try to bound.

In other settings, it may be unacceptable to have false positives. In which case, Bloom filters might be a bad idea. So, this is not a universal scheme. You have to look at your setting and determine whether the price of having a simpler and faster scheme is worth the cost of having false positives -- Is it acceptable to have false positives with some small probability?

***

(Slide 14) Operations



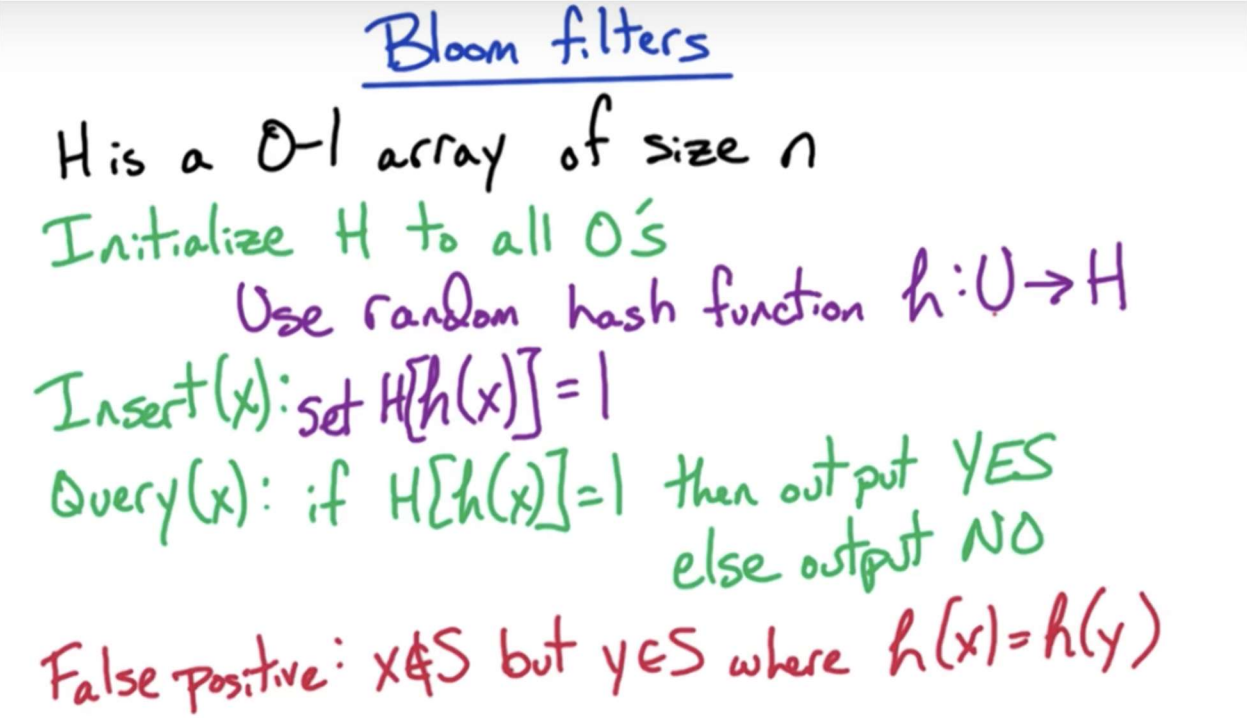
What are the basic operations that our data structure is going to support?

- First operation is insert x. So given a possible password x, we want to add this password into our dictionary of unacceptable passwords.
- The second operation is a query on x -- is this proposed password in our dictionary of unacceptable passwords? If this proposed password is in our dictionary of unacceptable passwords, then we're always going to output YES, so we're always correct in this case. The problem is that, when this proposed password is not in our dictionary, so it is an acceptable password, we usually output NO and we have to bound what we mean by usually. But occasionally, we're going to output YES. So when this proposed password is acceptable, occasionally we're going to have a false positive and say YES it is in the dictionary of unacceptable passwords, so this password is not allowed.

So we have false positives with some small rate and we have to bound that rate and see what it looks like.

***

(Slide 15) Bloom Filters



Finally, we can describe our Bloom filter data structure.

The basic data structure is simply a binary array a 0 - 1 array of size n. So, we have this binary array H. We don't have any linked lists hanging off it at all. It's just a binary array of size n -- that's the whole data structure.

We're going to start off by setting H to all zeros. So, all of the n bits are set to zero. As before, we're going to use a random hash function which maps elements of the universe of possible passwords into our hash table of size n.

How do we insert an element x of possible password into our dictionary x of unacceptable passwords? First off, we compute its hash value, then we set the bit in this array to one at that hash value. So we compute H[x] and we set H[x] = 1. Now it might already be 1, in which case we're not doing anything.

So, the bits only change from zeros to ones. We never change them back from ones to zeros. That's one of the limitations on this data structure -- There is no easy way to implement deletions because we never change bits from ones to zeros, we only change them from zeroes to ones.

Now how do we do a query? How do we check whether an element x is in our dictionary S? -- where we compute x's hash value (i.e., h(x)), and we check the array (i.e, H[h(x)]) ? The bit had that hash value and we see whether it's 1 or 0. If the bit at this hash value is 1 (if H[h(x)] = 1), then we output YES. We believe it is in the dictionary S. If it's 0, then we're guaranteed that it's not in the dictionary. Because if it's 0, that means we definitely did not insert it. If it's 1, then we think we might have inserted it but we're not sure. Somebody else might have been inserted at that hash value, and we have no way of checking whether it's x was inserted at the hash values or somebody else was inserted at the hash value. Because we're not maintaining a linked list at this point.

Let me repeat this point about how false positives can arise. We have some element x which we do a query on. It's not in our dictionary of unacceptable passwords, but there is some other element y which is in our dictionary of unacceptable passwords. And these two elements, x and y, have the same hash value. h(x) = h(y). So, when we inserted y into our dictionary, then we set this bit at this point to 1. So then, when we do the query on x, this bit looks to y. So, we think or as far as we know, x might be in our dictionary. So we have to output yes because it might be there. But, in fact, it is NO because it was not inserted but somebody else was inserted with the same hash value. That's how false positives arise.

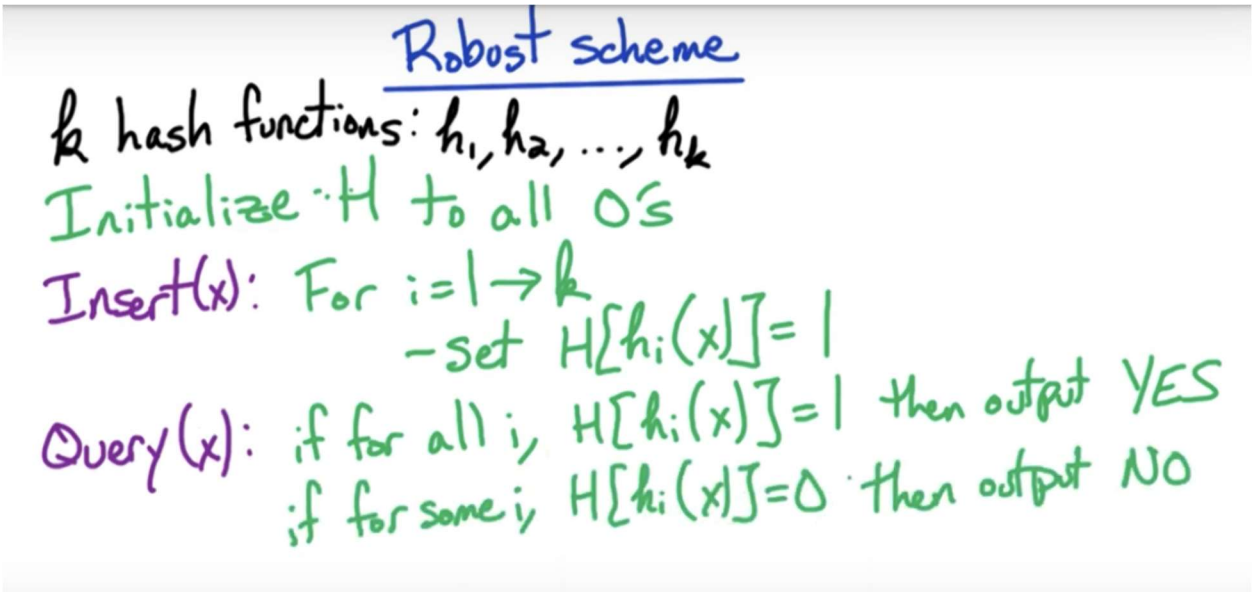
Now this scheme is not going to perform very well. How can we improve it? Well we can try to use our power of two choices idea that we used before in our traditional hashing scheme.

So, what are we going to do? Well instead of using one hash function, we're going to use two hash functions. Now, in the traditional balls and bins example, there was a big gain from going from one hash function to two hash functions, but then going from two to three or three to four, was not much of a gain. But, here, this is a slightly different setting and there'll be a big gain possibly going from one to two but even for two to three there might be a gain. And it's not clear how many hash functions to use and we're going to try to optimize that choice of number hash functions. So we're going to allow, instead of two hash functions, we're going to allow k hash functions.

So we want to generalize this scheme to allow k hash functions and then we're going to go back and figure out what is the optimal choice of k, the number of hash functions. So let's look at the more robust setting where we allow k hash functions, and how do we modify this data structure to accommodate k hash functions.

***

(Slide 16) Robust Scheme



So now we have k hash functions instead of just one,: h1, h2, up to hk. We're going to initialize our hash table at H to all zeroes. So all of the bits, the n bits of H are set to zero.

How do we do an insertion? --how do we add an element x, a possible password, into our dictionary S of unacceptable passwords. Previously, we computed this hash value h(x) and we set that bit to 1. What are we going to do now? Now we compute the k hash values and we set all of those k bits to 1.

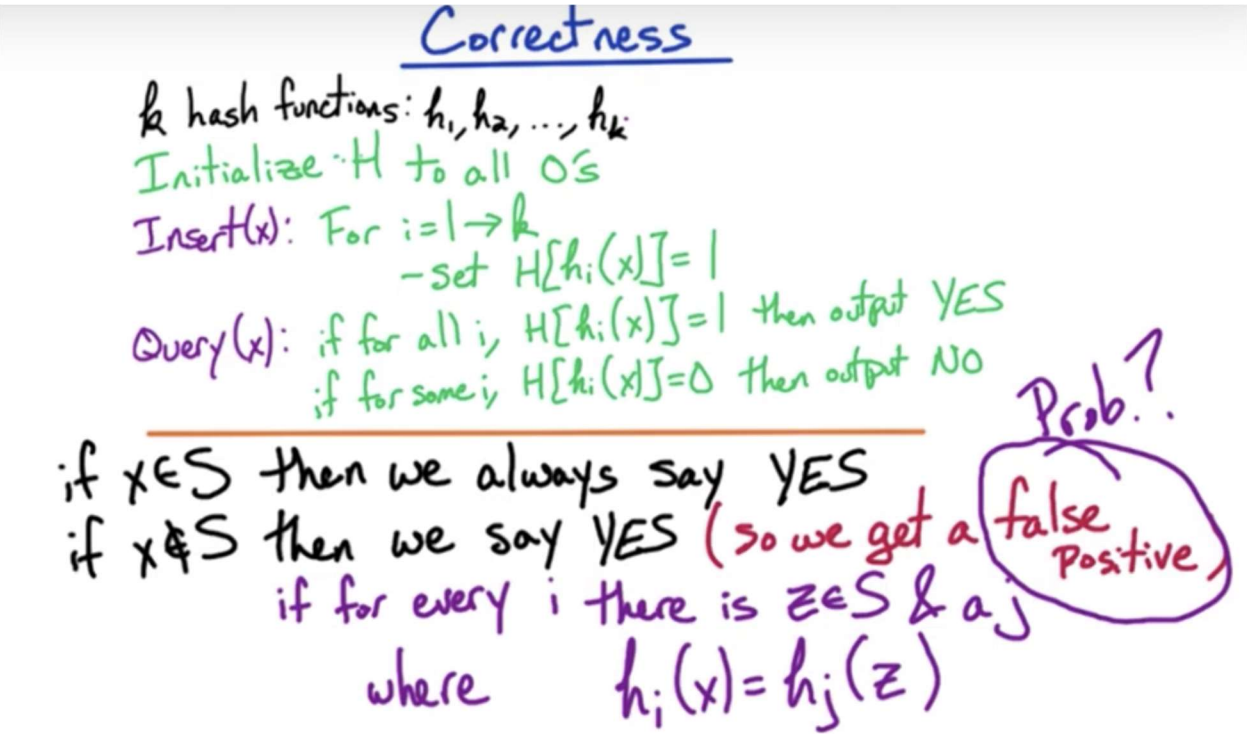
So we go through the hash functions 1 through k, and then we compute it's ith hash value ($h_i(x)$), and we set this bit to 1 (i.e., $H[h_i(x)] = 1$). It might already be 1 as before, but we always change bits from 0 to 1 just like before.

How do we do a query? --How do we check whether an element x was inserted into our dictionary S? Well, we compute it's k hash values and we check whether all of those k bits are set to 1 or not. If all of those k bits are set to 1, then our best guess is that x was inserted into the database S. If any of those are still 0, then we're guaranteed that x was not inserted into the database. So let's write that out more precisely.

If, for all of those k hash values, the hash function at those k bits is set to 1, then we output YES - we believe that this element x is in the database S. If any of these k bits is still 0, then we're guaranteed that this element X is not in the database. So, we output NO.

***

(Slide 17) Bloom Filters Correctness



Let's take a look at the correctness of this algorithm for our queries.

Suppose x was inserted into our database S, and we do a query on x. What do we output? -- well, when we inserted x into the database, we set all of these k bits to one. So, when we do a query, we're guaranteed that all of these bits are set to one, and so we're going to output YES because none of the bits ever change from ones to zeros. Bits only change from zeros to one. It's a one directional process.

So if x was inserted into the database, when we do a query on x, we always output YES -- it is in the database.

Now, suppose x was not inserted into the database and we do a query on x. Sometimes, we might say YES -- we believe it's in the database. In which case, we get a false positive. We falsely say that – YES, it's in the database. How can this occur? --this can occur if all of the k bits were set to one by other insertions.

So, for each of the k bits of x, so take the ith bit. So this is $h_i(x)$. There is some element, z, which was inserted into the database s and one of the k bits for z exactly matches the ith bit of x. Which of the k bits for z? Let's say the jth bit for z. So the jth bit for z matches the ith for x. In

other words, $h_i(x)$, as the ith for x, matches the jth bit of z. So $h_i(x) = h_j(z)$. This means that when z was inserted into the database: we did the insert of z; then, we set this bit which matches the ith bit of x to 1. And, if this is true for every bit of x – so, all the k bits of x are set to 1 by some other insertion -- then we're going to get a false positive on x.

So, this scheme has this extra robustness or redundancy. In order to get a false positive, we need all of these k bits to be set to 1 by some other insertions, whereas the previous scheme only had one bit which we're checking. Now we have k bits which need to get set to 1 in order to get a false positive. So, it seems like things improve that the false positive rate goes down as k increases.

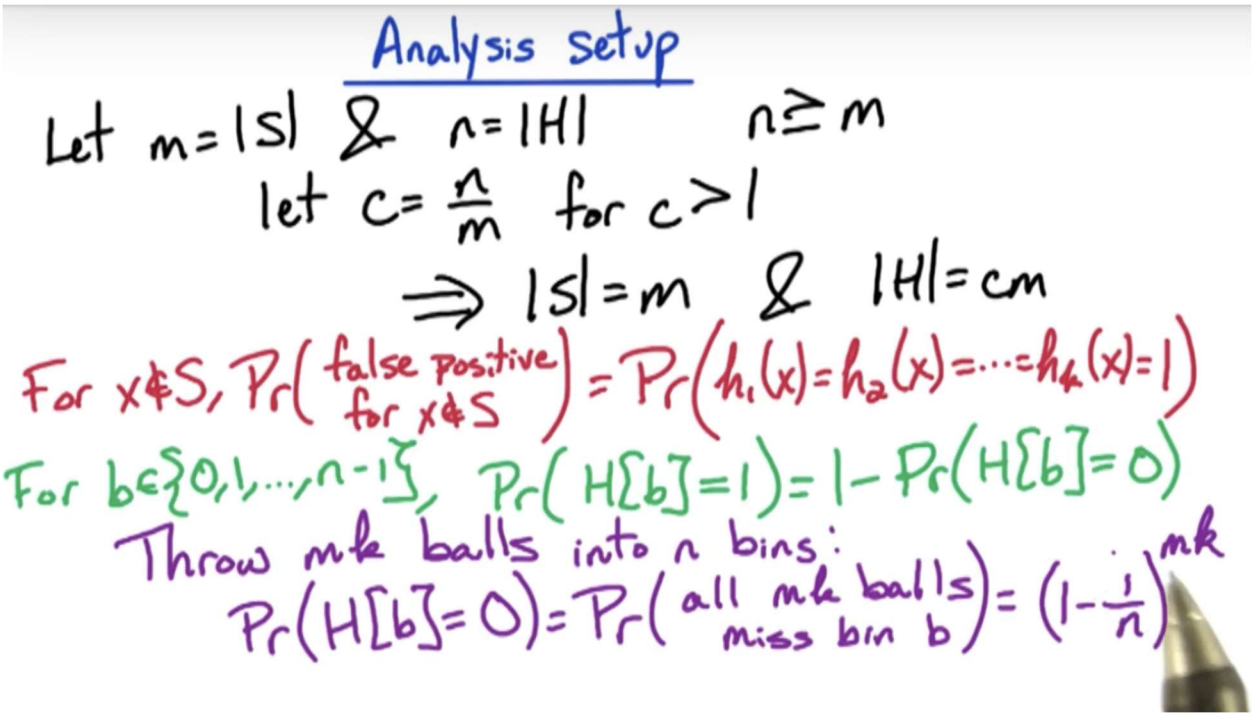
But, in fact, there's an optimal choice of k. If k gets too large, the false positive rate starts to shoot up again. Why is that? --well, if k is huge, then for every insertion you're setting k bits to one. So, you're setting many bits to 1 if k is huge.

So that means that for each of these and insertions -- each of these elements in S -- they have many bits, many choices of j which are set to 1. So, it's more likely if k is big, that one of these k bits is going to match up with one of the bits of x. So if k is too large, every insertion is setting too many bits to 1. If k is small, then when we're checking it -- when we're doing the query on x -- we're checking too few bits. So there's some optimal choice of k -- not too large and not too small.

What we want to do now is more precisely analyze these false positives. What's the probability of a false positive? We want to look at it as a function of k and then we can figure out the optimal choice of k in order to minimize the false positive rate. And then we can compare and see what that false positive rate looks like to see whether this is a good data structure to use or not.

***

(Slide 18) Analysis Setup



Let's start analyzing the false positive rate.

As before, let m denote the size of our database or dictionary that we're maintaining. And, let n denote the size of our hash table. Now, presumably, n, the size of our hash table, is going to be at least the size of the database that we're maintaining. So, the number of bins is at least the number of balls that we're inserting. So, the important parameter is going to be the ratio of these sizes. So, let c denote this ratio -- the size of the hash table compared to the size of the database. So, c is going to be at least one. And our goal is to try to get the smallest c possible.

So once again, the size of our database as a dictionary of unacceptable passwords is m. And the size of our hash table is c x m. There's a constant c which is at least one, and our hash table is this constant c -- bigger than the database that we're maintaining.

Now, for an element x, which is not in our database -- so, we didn't do an insertion on X -- let's look at the probability of a false positive for this x.

So x was not inserted into the database. And what is the probability of a false positive? So, we incorrectly say that x was inserted into the database. So, in order for this to occur, we need that all the k bits for x -- So h1(x), h2(x), up to hk(x) were all set to one. If all of these bits are 1, but x was never inserted into the database, then we'll get a false positive. We'll incorrectly say yes, it

is in the database because all of the k bits are 1, but it was never in fact inserted into the database.

So, let's try to analyze the probability that all the k bits are set to one. Let's first look at a simpler problem for a specific bit, b. What's the probability that this specific bit is set to one? So, for a specific bit, b (this ranges between 0 and n-1), what is the probability that this specific bit b is set to 1?

It would be slightly easier to look at the complimentary event that this specific bit is set to zero. So, the probability that this specific bit is 1 is 1 minus the probability that this bit is still zero. Now, to analyze the probability that it's still set to zero, what we have to do is we have to check that all of the insertions miss this one bit. So, let's go back and think about our Balls and bins analogy in order to analyze this probability.

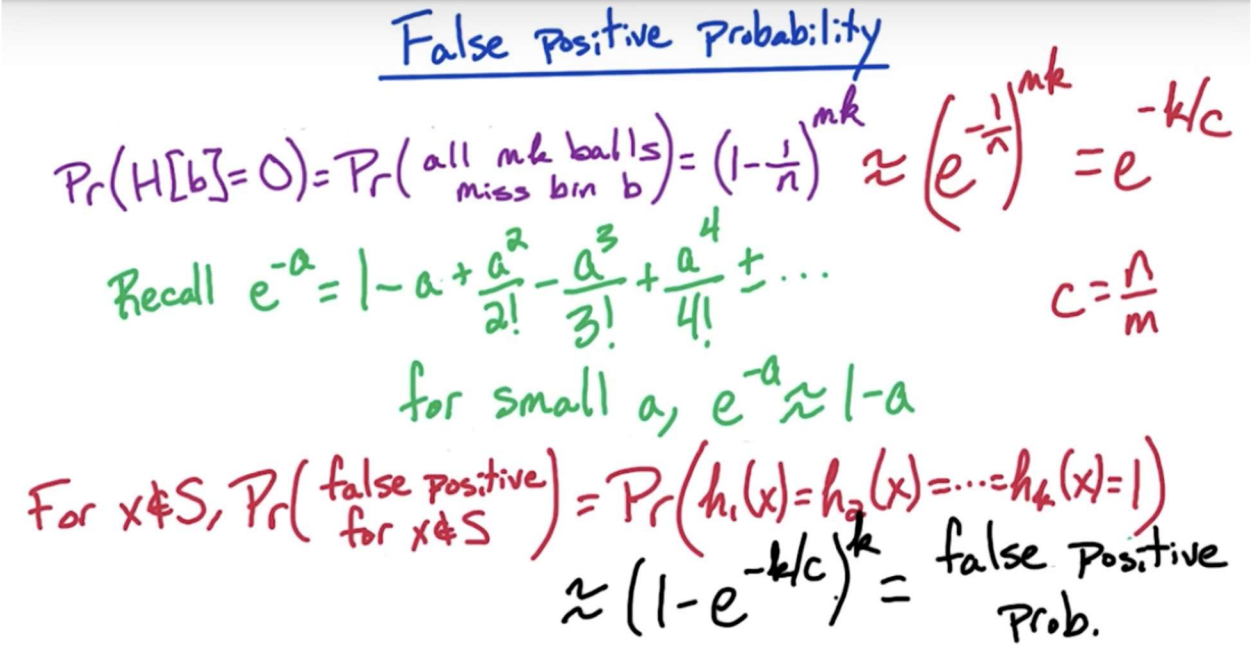
Now, we have m insertions. In our simple hashing scheme, these insertions correspond to throwing a ball into a bin. So, this corresponds to throwing m balls into bins. But, notice for each insertion, we have k hash values that we look at. And we set k of these values set to one. So, each insertion corresponds to k balls. So, actually we're throwing mk (m x k) balls into bins. So, we're throwing these mk balls and we're throwing them into n bins.

Now, what is this specific bit being set to zero correspond to in this Balls and bins example? In order for this bit to still be zero, we need that all of these mk balls miss this specific bin, b. So this probability that this bit is zero is equivalent to the probability that all mk balls miss this specific bin.

For one ball, what's the probability that it misses a specific bin? The chance that it hits the specific bin is 1/n; the chance it misses this bin is $1 - 1/n$. And we're doing this for mk balls. Now, this expression is not very complicated but it will be much more convenient for us to have a slightly simpler expression.

***

(Slide 19) False Positive Probability



So, what have we done so far? We've shown that the probability that a specific bit is zero is equal to $(1 - 1/n)^{mk}$. Let's try to manipulate this to get a slightly more convenient expression.

I want to replace this by an exponential. Suppose I look at e^{-a}, for a number a. Let's take a look at the Taylor series for the exponential function. Let me remind you of that expression:

$e^{-a} = 1 - a + a^2/2! - a^3/3! + a^4/4! +-$ and so on

Notice -- it's alternating signs – the current term is $a^4/4!$, and the next term is $-a^5/5!$, and so on. You have this infinite series. Now, for small a, this series is decreasing, and as a goes to zero, then this series is approximated by 1 - a. So, $e^{-a} \approx 1 - a$ is a good approximation when a is sufficiently small. That's going to correspond in our case to n being sufficiently large and we're looking at a asymptoticly as n grows to infinity.

So, let's use this approximation to simplify our analysis of the false positive rate. So -- what can I do here? Here (for the expression $(1 - 1/n)^{mk}$), I have $a = 1/n$, and as n grows, 1/n goes to zero. So $e^{-a} \approx 1 - a$ is a reasonable approximation -- so, I can replace $1 - 1/n$ by $e^{-1/n}$. So, I have $(1 - 1/n)^{mk} \approx (e^{-1/n})^{mk}$. Recall that c is the ratio of the size of the hash table to the size of the database ($c = n/m$). Therefore, this expression can be simplified to $(e^{-1/n})^{mk} = e^{-k/c}$. So now I have a very simple expression for the probability that a specific bit is zero.

Recall our original problem. We have an element x which was not inserted into our database. We want to look at the probability of a false positive for this element. So what's the chance we output yes, even though x was not inserted into the database? So what's the probability that all of these k bits corresponding to x were set to one by some other insertions? Well, the probability of a specific bit being set to zero (which we've just analyzed and shown that it's approximately $e^{-k/c}$) -- So what's the probability one of these specific bits is set to one? -- It's one minus the probability that is set to zero. The probability that it's set to zero is $e^{-k/c}$; the probability that it is set to 1 is $(1 - e^{-k/c})$. And we want k specific Bits all set to one. So the probability of that, is $(1 - e^{-k/c})$ raised to the power k.

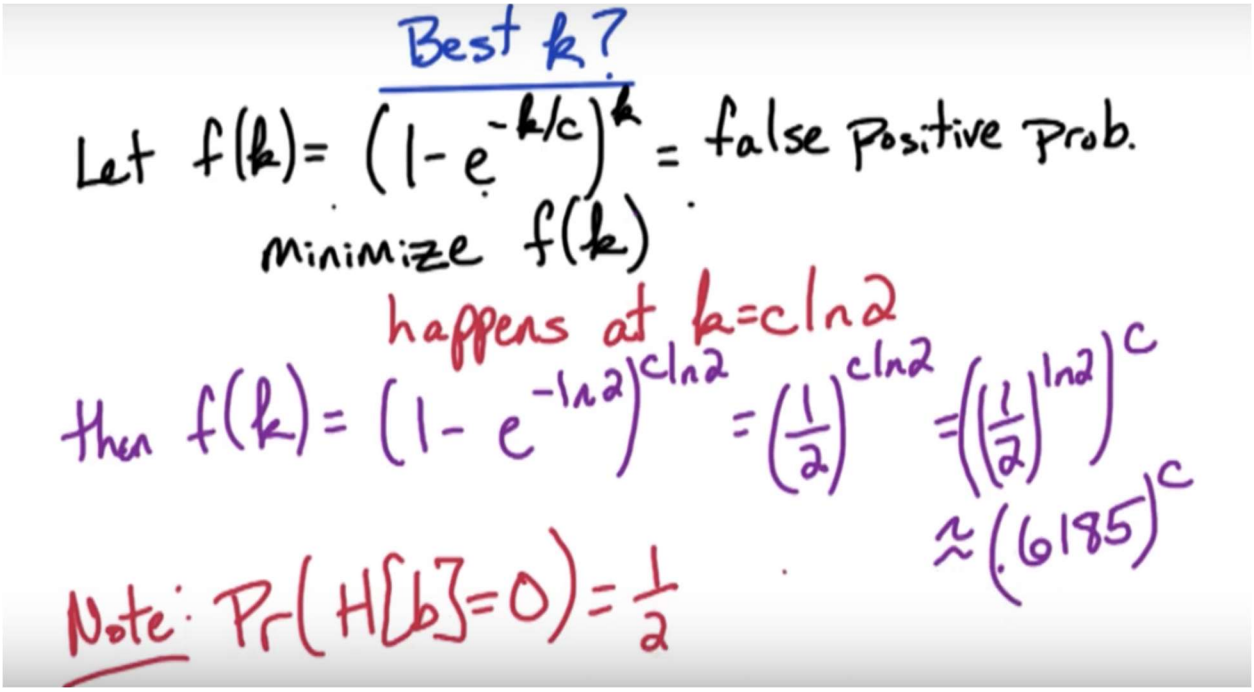
This expression $(1 - e^{-k/c})^k$ is the false positive probability. It's not very nice right now because we have this (exponential) k. Can we simplify this by eliminating k? Can we figure out what is the optimal k in order to minimize this false positive probability?

Recall our intuition from before, we wanted to have k not too small. If k is small, then when we do a query, we're checking too few bits. But if k was big, if it's too large, then when we do an assertion we're setting too many bits to one, for each insertion. So there's some middle ground and we want to figure out the optimal choice of k in order to minimize this false positive probability.

So what are we going to do? We're going to take this function $(1 - e^{-k/c})^k$. We're going to take its derivative. Set it equal to 0 and find the optimal choice of k in order to minimize this expression. So, a bit of calculus -- we're going to skip it. But I'll tell you the punchline.

***

(Slide 20) Optimal k



Recall the expression that we just obtained for the false positive probability. The expression was $(1 - e^{-k/c})^k$. Let's look at this as a function of k. So, look, let f(x) denote this expression. $f(k) = (1 - e^{-k/c})^k$. This f(k) is a false positive probability for a specific choice of k - the number of hash functions. Our goal is to figure out what's the optimal choice for the number of hash functions, in order to minimize this false positive probability.

So, what do we do? We are going to minimize this function f(k). How do we find the optimal choice of k? Well, we take it's derivative; set it equal to zero; check whether it's a global minimum -- and then we find that choice of k, which sets the derivative equal to zero.

Where does that optimal happen? It happens at k = c ln2 (That's the natural log of 2) -- the log base e of 2. (I'm going to skip this calculus, so that I don't embarrass myself. I've forgotten my calculus too, so don't worry).

But let's look at this choice of k which is the optimal choice in order to minimize the false positive probability. It's quite interesting.

Actually, first off, let's plug this choice of k back into this expression.

And then we can simplify it. Well, when you plug in k = c ln2 here, then the c is cancelled, and you're left with ln2. And then on the outside we have c ln2.

$f(x) = (1 - e^{-k/c})^k = (1 - e^{-c \ln 2/c})^{c \ln 2} = (1 - e^{-\ln 2})^{c \ln 2}$

What is $e^{-\ln 2}$? That's exactly 1/2. And what's 1 – 1/2? It's a half. So the inside is $1/2^{c \ln 2}$. Let's separate the c from ln2. So we have $((1/2)^{\ln 2})^c$.

$f(x) = (1 - e^{-k/c})^k = (1 - e^{-c \ln 2/c})^{c \ln 2} = (1 - e^{-\ln 2})^{c \ln 2} = (1/2)^{c \ln 2} = ((1/2)^{\ln 2})^c$

Now what is this inside? This is just a fixed constant. It's one half raised to the power ln2. You can plug it in on your calculator. Turns out that the inner expression is .6185 approximately. So we have that raised to the power c -- c is the ratio of the size of our hash table compared to our database.

Now we have a simple expression for the false positive probability. It's $(.6185)^c$.

$f(x) = (1 - e^{-k/c})^k = (1 - e^{-c \ln 2/c})^{c \ln 2} = (1 - e^{-\ln 2})^{c \ln 2} = (1/2)^{c \ln 2} = ((1/2)^{\ln 2})^c \approx (.6185)^c$

So, if you tell me how large of a hash table you're willing to do, I can tell you what the false positive probability is.

Before we look at that, let's just take a look at this expression right here:

$f(x) = (1 - e^{-\ln 2})^{c \ln 2}$

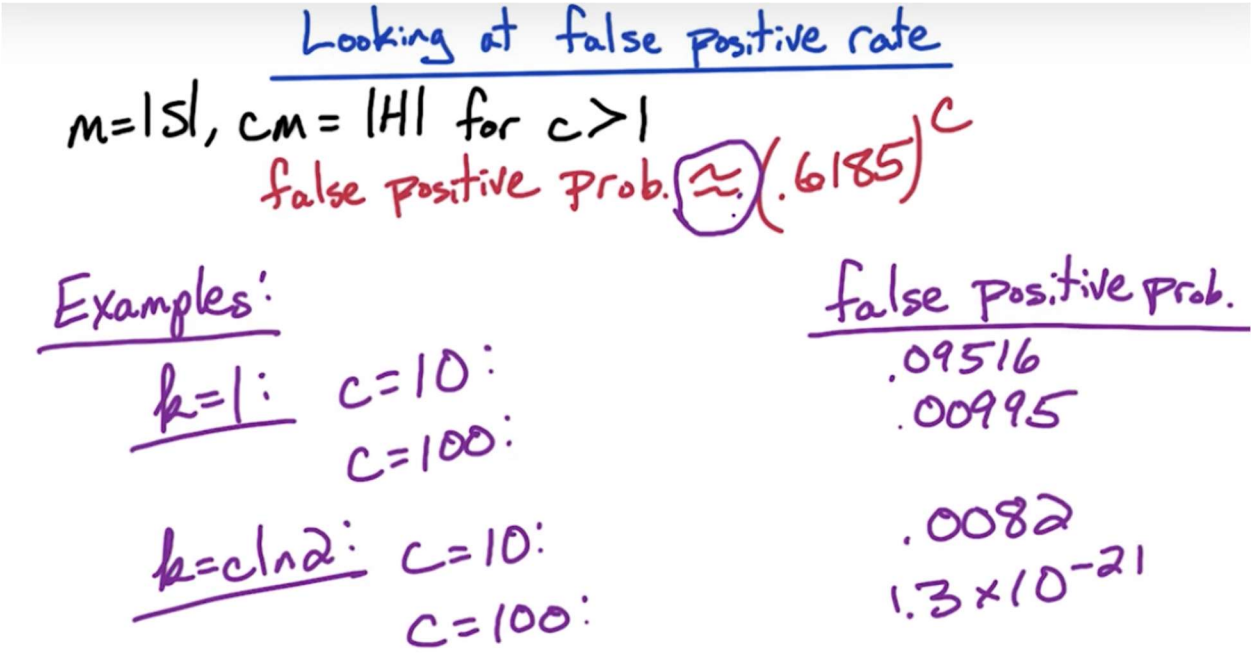
I want to point out something interesting about this choice, k = cln2 -- this optimal choice of k. What's the probability a specific bit is zero or one? The chance it's one is this inner expression $(1 - e^{-\ln 2})$. Chance is zero is this expression $(e^{-\ln 2})$ right here. Both of those are one half. So the chance we're setting a bit to zero is one half. The chance we're setting it to one is a half.

So what does this binary string look like after we've done m insertions? So it's a binary string of length n, and we've done m insertions into it. What does it look like? Well, each bit afterwards is set to zero with probability half. So, each bit is randomly set to zero or one. So this string H corresponds to a random string, where each bit is independently flipped to zero or one.

So what's interesting, is that the optimal choice of k means that if we just look at what this random string looks like, it's going to correspond to a random binary string where each bit is independently set to zero or one. So the optimal choice of k corresponds to balancing out the zeros and ones in H, so that H looks like a random string.

***

(Slide 21) Looking at False Positive Rate



Recall our setting. We have a database of size m and we have a hash table of size c times m for some c strictly greater than one. What we just showed is that the false positive probability is approximately $(.6185)^c$. This .6185 corresponded to $(1/2)^{\ln 2}$. Let's now look at some specific examples to see how this performs.

Examples:

- Let's suppose we did the naive scheme, where k = 1. So, we didn't do the optimal choice of k. We just set one hash function. And let's look at the case where we do 10 times larger (c = 10) or 100 times larger (c = 100).

  Now, the expression for the false positive probability was assuming the optimal choice of k. In order to analyze this case where k = 1, we have to go back to our expression of f(k). If you look back at that expression and you plug in k = 1 and c = 10, or c = 100, equal's a 100, you get the following:

  In the first case (k=1, c=10), the false positive probability is .095. And in the second case (k=1,c=100), it's point.00995.

- Now, suppose we do the optimal choice of k, k = c ln2. So, then our false positive probability is going to be this expression $(.6185)^c$.

Let's look at c = 10. What do we get? We get.0082. A reasonable gain. But not that much better than c=100 with this simple k = 1 case.

Let's try c = 100. So, the hash table is a 100 times bigger than the database we're trying to store. But this is just a binary string, right? So, it's very reasonable to consider a hash table which is 100 times bigger. Now, the false positive probability is 1.3×10^{-21}.

The key thing is that this is exponential in c. So, taking c = 100, it's tiny. This is really a minuscule probability. And if this is not small enough for you, you can go c = 200, or 300 and you're going to get a really, really tiny probability of a false positive.

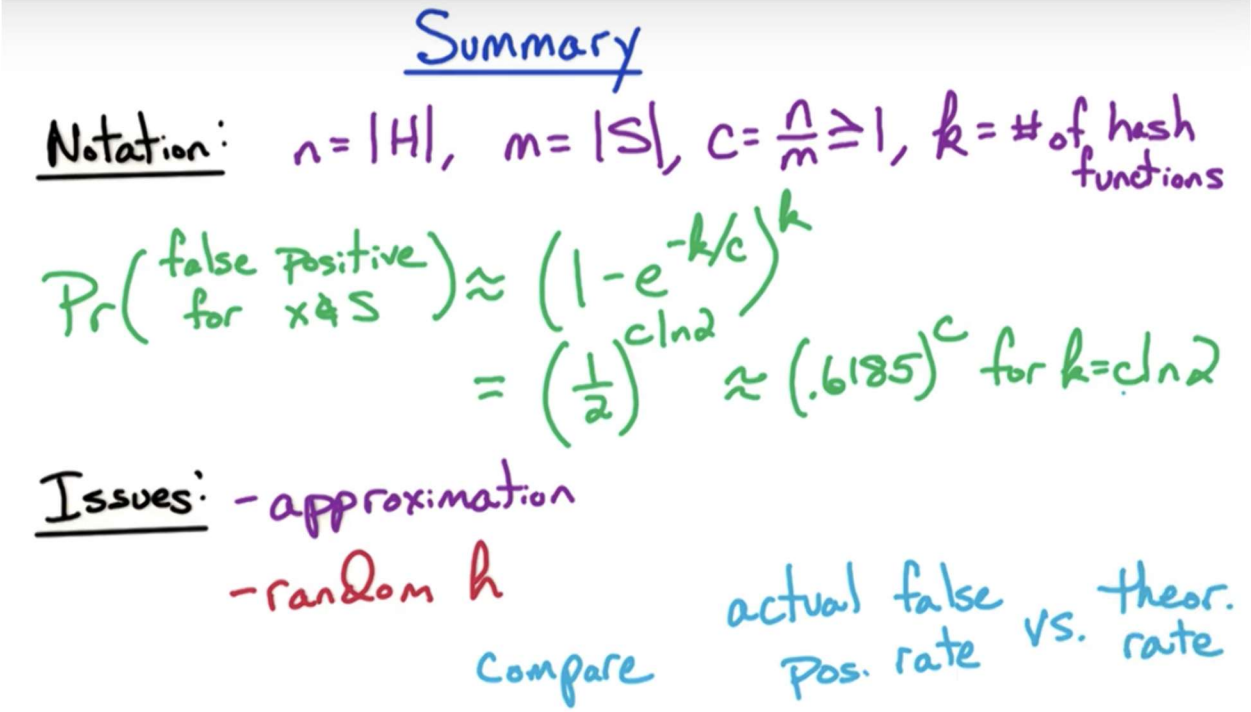
So, if you're willing to have a very small probability of a false positive, then you have this very simple data structure which just corresponds to having a binary string. It's very simple to maintain and is very fast query times and the false positive probability is very small.

The downside of this data structure is that occasionally you might have some false positives and also it doesn't easily allow for deletions from the database. Though, there are some heuristics for allowing deletions, these are modifications which are called Counting Bloom Filters.

Well, that completes our description of Bloom Filters. I look forward to seeing your projects where you're going to implement Bloom Filters and you're going to explore whether these approximations that we did in our analysis were reasonable or not.

***

(Slide 22) Bloom filters Summary



Let's summarize a few key points about Bloom filters.

- First off. Let's recall the notation:
  - little n denotes a hash table size and know that it's simply little n bits;
  - m is the database size – this is the number insertions into our hash table;
  - and we used c to use to denote the ratio of the sizes. So c = n/m. This is at least one
  - and, finally k is the number for hash functions that were using

- Let's recall what we proved about the false positive rate for an element x, which is not in our database. So, we never inserted x into our set S. What's the probability of a false positive that we falsely report that x is in the database?

  We proved that this is approximately $(1 - e^{-k/c})^k$ but it's a bit of a monster but we proved that for the optimal choice of k, this simplifies the $(½)^{c\ln 2}$. This is approximately $(0.6185)^c$. This is for the optimal choice of k, which is c ln2. Notice that this may not be an integer value. So, you may not achieve this false positive rate. We might have to look at the closest integer.

- There are a few issues I'd like to point out. First, $(.6185)^c$ is not a rigorous bound. This is simply an approximation.

  What was the approximation we used? We replaced $(1 - x)$ by e^{-x}. For us, x was 1/n. If we look at this for big n, then x is tiny and when x is small this is a good approximation.

  So, this was a reasonable approximation. We do expect a false positive rate for these hash functions to behave like this.

About your project

The big issue is that we assumed that these hash functions are random hash functions. So, for each element x in the universe, their hash values are mutually independent of each other. Now it's impossible to achieve -- the use of such a random hash function. And the question is how does an actual implementation of a hash function compare to the theoretical results for this random hash function and that's what you're going to address in your project.

For a simple choice of a hash function H, we're going to compare the actual false positive rate for this simple implementation of a hash function. We're going to compare it to the theoretical false positive right which we predict for a random hash function. So it's a quite intriguing question. If we look at an actual hash function and we look at as a function of k, does the actual false positive rate behave similar to this theoretical result and does it turn out that the minimum false positive rate occurs at $k = c \ln 2$ (or the closest integer to it).

Now, there's quite a lot of choices you can use for your choice of a hash function and we're just going to use a simple hash function in order to simplify this comparison.