MIT OpenCourseWare
http://ocw.mit.edu

6.046J Introduction to Algorithms, Fall 2005

Please use the following citation format:

> Erik Demaine and Charles Leiserson, *6.046J Introduction to Algorithms, Fall 2005*. (Massachusetts Institute of Technology: MIT OpenCourseWare). http://ocw.mit.edu (accessed MM DD, YYYY). License: Creative Commons Attribution-Noncommercial-Share Alike.

Note: Please use the actual date you accessed this material in your citation.

For more information about citing these materials or our Terms of Use, visit:
http://ocw.mit.edu/terms

MIT OpenCourseWare
http://ocw.mit.edu

6.046J Introduction to Algorithms, Fall 2005
Transcript – Lecture 8

Hashing. Today we're going to do some amazing stuff with hashing. And, really, this is such neat stuff, it's amazing. We're going to start by addressing a fundamental weakness of hashing. And that is that for any choice of hash function There exists a bad set of keys that all hash to the same slot. OK. So you pick a hash function. We looked at some that seem to work well in practice, that are easy to put into your code. But whichever one you pick, there's always some bad set of keys. So you can imagine, just to drive this point home a little bit.

Imagine that you're building a compiler for a customer and you have a symbol table in your compiler and one of the things that the customer is demanding is that compilations go fast. They don't want to sit around waiting for compilations. And you have a competitor who's also building a compiler and they're going to test the compiler, both of your compilers and sort of have a run-off. And one of the things in the test that they're going to allow you to do is not only will the customer run his own benchmarks, but he'll let you make up benchmarks for the other program, for your competitor. And your competitor gets to make up benchmarks for you. So and not only that, but you're actually sharing code.

So you get to look at what the competitor is actually doing and what hash function they're actually using. So it's pretty clear that in this circumstance, you have an adversary who is going to look at whatever hash function you have and figure out OK, what's a set of variable names and so forth that are going to all hash to the same slot so that essentially you're just chasing through a linked list whenever it comes to looking something up.

Slowing down your program enormously compared to if in fact they got distributed nicely across the hash table which is, what after all, you have a hash table in there to do in the first place. And so the question is, how do you defeat this adversary? And the answer is one word. One word. How do you achieve? How do you defeat any adversary in this class? Randomness. OK. Randomness. OK. You make it so that he can't guess. And the idea is that you choose a hash function at random. Independent. So he can look at the code, but when it actually runs, it's going to use a random hash function that he has no way of predicting what the hash function is that will actually be used.

OK. So that's the game and that way he can provide an input, but he can't provide an input that's guaranteed to force you to run slowly. You might get unlucky in your choice of hash function, but it's not going to be because of the adversary. So the idea is to choose a hash function -- -- at random, independently from the keys that you're, that are going to be fed to it. So even if your adversary can see your code, he can't tell which hash function is going to be actually used at run time. Doesn't get to see the output of the random numbers.

And so it turns out you can make this scheme work and the name of the scheme is universal hashing, OK, is one way of making this scheme work. So let's do some

math. So let U be a universe of keys. And let H be a finite collection -- -- of hash functions -- -- mapping U to what are going to be the slots in our hash table. OK. So we just have H as some finite collection. We say that H is universal --

-- if for all pairs of the keys, distinct keys -- -- so the keys are distinct, the following is true. So if the set of keys, if for any pair of keys I pick, the number of hash functions that hash those two keys to the same place is a one over m fraction of the total set of keys. So let m just, so to view that, another way of viewing that is if H is chosen randomly -- -- from the set of keys H, the probability of collision between x and y is what?

What's the probability if the fraction of hash functions, OK, if the number of hash functions is H over m, what's the probability of a collision between x and y? If I pick a hash function at random. So I pick a hash function at random, what's the odds they collide? One over m. Now let's draw a picture for that, help people see that that's in fact the case. So imagine this is our set of all hash functions.

OK. And then if I pick a particular x and y, let's say that this is the set of hash functions such that H of x is equal to H of y. And so what we're saying is that the cardinality of that set is one over m times the cardinality of H. So if I throw a dart and pick one hash function at random, the odds are one in m that the hash function falls into this particular set. And of course, this has to be true of every x and y that I can pick. Of course, it will be a different set, a different x and y will somehow map the hash functions differently, but the odds that for any x and y that I pick, the odds that if I have a random hash function, it hashes it to the same place, is one over m.

Now this is a little bit hard sometimes for people to get their head around because we're used to thinking of perhaps picking keys at random or something. OK, that's not what's going on here. We're picking hash functions at random. So our probability space is defined over the hash functions, not over the keys. And this has to be true now for any particular two keys that I pick that are distinct. That the places that they hash, this set of hash functions, I mean this is like a marvelous property if you think about it.

OK, that you can actually find ones where no matter what two elements I pick, the odds are exactly one in m that a random hash function from this set is going to hash them to the same place. So very neat. Very, very neat property and we'll see the mathematics associated with this is very cool. So our theorem is that if we choose h randomly from the set of hash functions H, and then we suppose we're hashing n keys into m slots in Table T --

-- then for given key x -- -- the expected number of collisions with x -- -- is less than n over m. And who remembers what we call n over m? Alpha, which is the, what's the term that we use there? Load factor. The load factor of the table. OK, load factor alpha. So the average number of keys per slot is the load factor of the table. So we're saying, so what is this theorem saying? It's saying that in fact, if we have one of these universal sets of hash functions, then things perform exactly the way we want them to.

Things get distributed evenly. The number of things that are going to collide with any particular key that I pick is going to be n over m. So that's a really good property to have. Now I haven't shown you, the construction of U is going, sorry, of the set of hash functions H, that that construction will take us a little bit of effort. But first I

want to show you why this is such a great property. Basically it's this theorem. So let's prove this theorem. So any questions about what the statement of the theorem is? So we're going to go actually kind of fast today. We've got a lot of good stuff today. So I want to make sure people are onboard as we go through.

So if there are any questions, make sure, you know, statement of theorem of whatever, best to get them out early so that you're not confused later on when the going gets a little more exciting. OK? OK, good. So to prove this, let's let C sub x be the random variable denoting the total number of collisions -- -- of keys in T with x. So this is a total number and one of the techniques that you use a lot in probabilistic analysis of randomized algorithms is recognizing that C of x is in fact a sum of indicator random variables.

If you can decompose things into indicator random variables, the analysis goes much more easily than if you're left with aggregate variables. So here we're going to let our indicator random variable be little c of x., which is going to be one if h of x equals h of y and 0 otherwise. And so we can note two things. First, what is the expectation of C of x.. OK, if I have a process which is picking a hash function at random, what's the expectation of C of x.?

One over m. Because that's basically this definition here. Now in other words I pick a hash function at random, what's the odds that the hash is the same? It's one over m. And then the other thing is, and the reason we pick this thing is that I can express capital C sub x, the random variable denoting the total number of collisions as being just the sum over all the keys in the table except x of C of x..

So for each one that would cause me a collision, with x, I add one and if it wouldn't cause me a collision, I add 0. And that adds up all of the collisions that I would have in the table with x. Is there any questions so far? Because this is the set-up. The set-up in most of these things, the set-up is where most students make mistakes and most practicing researchers make mistakes as well, let me tell you.

And then once you get the set-up right, then working out the math is fine, but it's often that set-up of how do you actually translate the situation into the math. That's the hard part. Once you get that right, well, then, algebra, we can all do algebra. Of course, we can also all make mistakes doing algebra, but at least those mistakes are much more easy to check than the one that does the translation. So I want to make sure people are sort of understanding of how that's set up. So now we just have to use our math skills.

So the expectation then of the number of collisions is the expectation of C sub x and that's just the expectation of just plugging the sum of y and T minus the element x of c_xy. So that's just definition. And that's equal to the sum of y and T minus x of expectation of c_xy. So why is that? Yeah, that's linearity. Linearity of expectation, doesn't require independence. It's true of all random variables.

And that's equal to, and now the math gets easier. So what is that? One over m. That makes the summation easy to evaluate. That's just n minus one over m. So fairly simple analysis and shows you why we would love to have one of these sets of universal hash functions because if you have them, then they behave exactly the way you would want it to behave. And you defeat your adversary by just picking up the hash function at random. There's nothing he can do. Or she.

OK, any questions about that proof? OK, now we get into the fun math. Constructing one of these babies. OK. This is not the only construction. This is a construction of a classic universal hash function. And there are other constructions in the literature and I think there's one on the practice quiz. So let's see. So this one works when m is prime. So it works when the set of slots is a prime number. Number of slots is a prime number.

So the idea here is we're going to decompose any key k in our universe into r plus 1 digits. So k, we're going to look at as being a k 0, k one, k_r where 0 is less than or equal to k sub I, is less than or equal to m minus one. So the idea is in some sense we're looking at what the representation would be of k base m. So if it were base two, it would be just one bit at a time. These would just be the bits. I'm not going to do base two. We're going to do base min general and so each of these represents one of the digits. And the way I've done it is I've done low order digit first. It actually doesn't matter. We're not actually going to care really about what the order is, but basically we're just looking at busting it into a twofold represented by each of those digits. So one algorithm for computing this out of k is take the remainder mod m.

That's the low order one. OK, take what's left. Take the remainder of that mod m. Take whatever's left, etc. So you're familiar with the conversion to a base representation. That's exactly how we're getting this representation. So we treat, this is just a question of taking the data that we've got and treating it as an r plus one base m number. And now we invoke our randomized strategy. The randomized strategy is going to be able to have a class of hash functions that's dependent essentially on random numbers. And the random numbers we're going to pick is we're going to pick an a at random --

-- which we're also going to look at as a base mnumber. For each a_i is chosen randomly -- -- from -- -- 0 to m minus one. So one of our, it's a random if you will, it's a random base mdigit. Random base m digit. So each one of these is picked at random. And for each one we, possible value of A, we're going to get a different hash function. So we're going to index our hash functions by this random number. So this is where the randomness is going to come in. Everybody with me? And here's the hash function.

So what we do is we dot product this vector with this vector and take the result, mod m. So each digit of k of our key gets multiplied by a random other digit. We add all those up and we take that mod m. So that's a dot product operator. And this is what we're going to show is universal, that this set of h sub a, where I look over that whole set. So one of the things we need to know is how big is the set of hash functions here.

So how big is this set of hash functions? How many different hash functions do I have in this set? It's basic 6.042 material. It's basically how many vectors of length r plus one where each element of the vector is a number of 0 to m minus one, has m different values. So how many? m minus one to the r. No. Close. It's up there. It's a big number. m to the r plus one. Good. It's m, so the size of H is equal to m to the r plus one. So we're going to want to remember that. OK, so let's just understand why that is. I have m choices for the first value of A. m for the second, etc. m for the r th. And since there are plus one things here, for each choice here, I have this many same number of choices here, so it's a product.

OK, so this is the product rule in counting. So if you haven't reviewed your 6.042 notes for counting, this is going to be a good idea to go back and review that because we're doing stuff of that nature. This is just the product rule. Good. So then the theorem we want to prove is that H is universal. And this is going to involve a little bit of number theory, so it gets kind of interesting. And it's a non-trivial proof, so this is where if there's any questions as I'm going along, please ask because the argument is not as simple as other arguments we've seen so far.

OK, not the ones we've seen so far have been simple, but this is definitely a more involved mathematical argument. So here's a proof. So let's let, so we have two keys. What are we trying to show if it's universal, that if I pick any two keys, the number of hash functions for which they hash to the same thing is the size of set of hash functions divided by m. OK, so I'm going to look at two keys. So let's pick two keys arbitrarily. So x, and we'll decompose it into our base r representation and y, y_0, y_1 --

So these are two distinct keys. So if these are two distinct keys, so they're different, then this base representation has the property that they've got to differ somewhere. Right? OK, they differ in at least one digit. OK, and this is where most people get lost because I'm going to make a simplification. They could differ in any one of these digits. I'm going to say they differ in position 0 because it doesn't matter which one I do, the math is the same, but it'll make it so that if I pick some said they differ in some position i, I would have to be taking summations as you'll see over the elements that are not i, and that's complicated.

If I do it in position 0, then I can just sum for the rest of them. So the math is going to be identical if I were to do it for any position because it's symmetric. All the digits are symmetric. So let's say they differ in position 0, but the same argument is going to be true if they differed in some other position. So let's say, so we're saying without loss of generality. So that's without loss of generality. Position 0. Because all the positions are symmetric here. And so, now we need to ask the question for how many --

-- hash functions in our universal, purportedly universal set do x and y collide? OK, we've got to count them up. So how often do they collide? This is where we're going to pull out some heavy duty number theory. So we must have, if they collide -- -- that h sub a of x is equal to h sub a of y. That's what it means for them to collide. So that implies that the sum of i equal 0 to r of a sub i x sub i is equal to the sum of i equals 0 to r of a sub i y sub i mod m.

Actually this is congruent mod m. So congruence for those people who haven't seen much number theory, is basically the way of essentially, rather than having to say mod everywhere in here and mod everywhere in here, we just at the end say OK, do a mod at the end. Everything is being done mod, module m. And then typically we use a congruence sign. OK, there's a more mathematical definition but this will work for us engineers. OK, so everybody with me so far? This is just applying the definition. So that implies that the sum of i equals 0 to r of a i x i minus y i is congruent to zeros mod m.

OK, just threw it on the other side and applied the distributive law. Now what I'm going to do is pull out the 0-th position because that's the one that I care about. And this is where it saves me on the math, compared to if I didn't say that it was 0. I'd have to pull out x_i. It wouldn't matter, but it just would make the math a little bit

cruftier OK, so now we've just pulled out one term. That implies that a_0 x_0 minus y_0 is congruent to minus --

-- mod m. Now remember that when I have a minus number mod m, I just map it into whatever, into that range from 0 to m minus one. So for example, minus five mod seven is two. So if any of these things are negative, we simply translate them into by adding multiples of mbecause adding multiples of m doesn't affect the congruence. OK. And now for the next step, we need to use a number theory fact. So let's pull out our number theory --

-- textbook and take a little digression So this comes from the theory of finite fields. So for people who are knowledgeable, that's where you're plugging your knowledge in. If you're not knowledgeable, this is a great area of math to learn about. So here's the fact. So let m be prime. Then for any z, little z element of z sub m, and z sub m is the integers mod m. So this is essentially numbers from 0 to m minus one with all the operations, times, minus, plus, etc., defined on that such that if you end up outside of the range of 0 to m minus one, you re-normalize by subtracting or adding multiples of m to get back within the range from 0 to m minus one.

So it's the standard thing of just doing things module m. So for any z such that z is not congruent to 0, there exists a unique z inverse in z sub m, such that if I multiply z times the inverse, it produces something congruent to one mod m. So for any number it says, I can find another number that when multiplied by it gives me one. So let's just do an example for m equals seven. So here we have, we'll make a little table. So z is not equal to 0, so I just write down the other numbers. And let's figure out what z inverse is.

So what's the inverse of one? What number when multiplied by one gives me one? One. Good. How about two? What number when I multiply it by two gives me one? Four. Because two times four is eight and eight is congruent to one mod seven. So I've re-normalized it. What about three? Five. Good. Five. Three times five is 15. That's congruent to one mod seven because 15 divided by seven is two remainder of one. So that's the key thing. What about four? Two.

Five? Three. And six. Yeah. Six. Yeah, six it turns out. OK, six times six is 36. OK, mod seven. Basically subtract off the 35, gives m one. So people have observed some interesting facts that if one number's an inverse of another, then that other is an inverse of the one. So that's actually one of these things that you prove when you do group theory and field theory and so forth. There are all sorts of other great properties of this kind of math. But the main thing is, and this turns out not to be true if m is not a prime. So can somebody think of, imagine we're doing something mod 10.

Can somebody think of a number that doesn't have an inverse mod 10? Yeah. Two. Another one is five. OK, it turns out the divisors in fact actually, more generally, something that is not relatively prime, meaning that it has no common factors, the GCD is not one between that number and the modulus. OK, those numbers do not have an inverse mod m. OK, but if it's prime, every number is relatively prime to the modulus. And that's the property that we're taking advantage of. So this is our fact and so, in this case what I'm after is I want to divide by x_0 minus y_0.

That's what I want to do at this point. But I can't do that if x_0, first of all, if m isn't prime, I can't necessarily do that. I might be able to, but I can't necessarily. But if m

is prime, I can definitely divide by x_0 minus y_0. I can find that inverse. And the other thing I have to do is make sure x_0 minus y_0 is not 0. OK, it would be 0 if these two were equal, but our supposition was they weren't equal.

And once again, just bringing it back to the without loss of generality, if it were some other position that we were off, I would be doing exactly the same thing with that position. So now we're going to be able to divide. So we continue with our -- -- continue with our proof. So since x_0 is not equal to y_0, there exists an inverse for x_0 minus y_0. And that implies, just continue on from over there, that a_0 is congruent therefore to minus the sum of i equal one to r of a_i, x_i minus y_i times x_0 minus y_0 inverse.

So let's just go back to the beginning of our proof and see what we've derived. If we're saying we have two distinct keys, and we've picked all of these a_i randomly, and we're saying that these two distinct keys hash to the same place. If they hash to the same place, it says that a_0 essentially had to have a particular value as a function of the other a_i. Because in other words, once I've picked each of these a_i from one to r, if I did them in that order, for example, then I don't have a choice for how I pick a_0 to make it collide. Exactly one value allows it to collide, namely the value of a_0 given by this.

If I picked a different value of a_0, they wouldn't collide. So let m write that down. Thus, while you think about it So for any choice of these a_i, there's exactly one of the impossible choices of a_0 that cause a collision. And for all the other choices I might make of a_0, there's n collision. So essentially I don't have, if they're going to collide, I've reduced essentially the number of degrees of freedom of my randomness by a factor of m. So if I count up the number of h_a's that cause x and y to collide, that's equal to, well, there's m choices, just using the product rule again.

There's m choices for a_1 times m choices for a_2, up to m choices for a_r and then only one choice for a_0. So this is choices for a_1, a_2, a_r and only one choice for a_0 if they're going to collide. If they're not going to collide, I've got more choices for a_0. But if I want them to collide, there's only one value I can pick, namely this value. That's the only value for which I will pick. And that's equal to m to the r, which is just the size of H divided by m.

And that completes the proof. So there are other universal constructions, but this is a particularly elegant one. So the point is that I have m plus one, sorry, r plus one degrees of freedom where each degree of freedom I have m choices. But if I want them to collide, once I've picked any of the, once I've picked r of those possible choices, the last one is forced if I want it to collide. So therefore, the set of functions for which it collides is only one in m. A very slick construction.

Very slick. OK. Everybody with me here? Didn't lose too many people? Yeah, question. Well, part of it is, actually this is a quite common type of thing to be doing actually. If you take a class, so we have follow on classes in cryptography and so forth, and this kind of thing of taking dot products, modulo m and also Galois fields which are particularly simple finite fields and things like that, people play with these all the time.

So Galois fields are like using exor's as your, same sort of thing as this except base two. And so there's a lot of study of this sort of thing. So people understand these kind of properties. But yeah, it's like what's the algorithm for having a brilliant

insight into algorithms? It's like OK. Wish I knew. Then I'd just turn the crank. [LAUGHTER] But if it were that easy, I wouldn't be standing up here today. [LAUGHTER] Good. OK, so now I want to take on another topic which is also I find, I think this is astounding. It's just beautiful, beautiful mathematics and a big impact on your ability to build good hash functions.

Now I want to talk about another one topic, which is related, which is the topic of perfect hashing. So everything we've done so far does expected time performance. Hashing is good in the expected sense. A perfect hashing addresses the following questions. Suppose that I gave you a set of keys, and I said just build me a static table so I can look up whether the key is in the table with worst case time. Good worst case time. So I have a fixed set of keys. They might be something like for example, the hundred most common or thousand most common words in English.

And when I get a word I want to check quickly in this table, is the word that I've got one of the most common words in English. I would like to do that not with expected performance, but guaranteed worst case performance. Is there a way of building it so that I can find this quickly? So the problem is given n keys -- -- construct a static hash table. In other words, no insertion and deletion. We're just going to put the elements in there. A size --

-- m equal Order n. So I don't want it to be a huge table. I want it to be a table that is the size of my keys. Table of size m equals Order n, such that search takes O(1) time in the worst case. So there's no place in the table where I'm going to have, I know in the average case, that's not hard to do. But in the worst case, I want to make sure that there's no particular spot where the number of keys piles up to be a large number. OK, in no spot should that happen. Every single search I do should take Order one time.

There shouldn't be any statistical variation in terms of how long it takes me to get something. Does everybody understand what the puzzle is? So this is a great, because this actually ends up having a lot of uses. You know, you want to build a table for something and you know what the values are that you're going look up in it. But you don't want to spend a lot of space on it and so forth. So the idea here is actually going to be to use a two-level scheme.

So the idea is we're going to use a two-level scheme with universal hashing at both levels. So the idea is we're going to hash, we're going to have a hash table, we're going to hash into slots, but rather than using chaining, we're going to have another hash table there. We're going to do a second hash into the second hash table. And the idea is that we're going to do it in such a way that we have no collisions at level two.

So we may have collisions at level one. We'll take anything that collides at level one and put them into a hash table and then our second level hash table, but that hash table, no collisions. Boom. We're just going to hash right in there. And it'll just go boom to its thing. So let's draw a picture of this to illustrate the scheme. OK, so we have -- -- 0 one, let's say six, m minus one. So here's our hash table. And what we're going to do is we're going to use universal hashing at the first level, OK. So we find a universal hash function. We pick a hash function at random. And what we'll do is we'll hash into that level. And then what we'll do is we'll keep track of two things.

One is what the size of the hash table is at the next level. So in this case, the size of the hash table will only use the number of slots. There's going to be four. And we're also going to keep a separate hash key for the second level. So each slot will have its own hash function for the second level. So for example, this one might have a key of 31 that is a random number. The a's here. a's up there. There we go, a's up there. So that's going to be the basis of my hash function, the key with which I'm going to hash. This one say has 86. And let's say that this, and then we have a pointer to the hash table. This is say S_1. And it's got four slots and we stored up 14 and 27. And these two slots are empty.

And this one for example, had what? Two nines. So the idea here is that in this case if we look over all our top level hash function, which I'll just call H, has that H of 14 is equal to H of 27 is equal to one. Because we're in slot one. OK, so these two both hash to the same slot in the level one hash table. This is level one. And this is level two over here. So level one hashing, 14 and 27 collided. They went into the same slot here. But at level two, they got hashed to different places and the hash function I use is going to be indexed by whatever the random numbers are that I chose and found for those and I'll show you how we find those. We have then h of 31 of 14 is equal to one h of 31 of 27 is equal to two.

For level two. So I go, hash in here, find the, use this as the basis of my hash function to hash into whatever table I've got here. And so, if there are no, if I can guarantee that there are no collisions at level two, this is going to cost me Order one time in the worst case to look something up. How do I look it up? Take the value. I apply h to it. That takes me to some slot. Then I look to see what the key is for this hash function. I apply that hash function and that takes me to another slot. Then I go there. And that took me basically two applications of hash functions plus some look-up, plus who knows what minor amount of bookkeeping.

So the reason we're going to have no collisions at this level is the following. If they're n sub i items that hash to a level one slot i, then we're going to use m sub i, which is equal to n sub i squared slots in the level two hash table. OK, so I should have mentioned here this is going to be m sub i, the size of the hash table and this is going to be my a sub i essentially. So I'm going to use, so basically I'm going to hash n sub i things into n sub i squared locations here. So this is going to be incredibly sparse.

OK, it's going to be quadratic in size. And so what I'm going to show is that under those circumstances, it's easy for me to find hash functions such that there are n collisions. That's the name of the game. Figure out how can I make these hash functions so that there are no collisions. So that's why I draw this with so few elements here. So here for example, I have two elements and I have a hash table size four here. I have three elements. I need a hash table size nine.

OK, if there are a hundred elements, I need a hash table size 10,000. I'm not going to pick something so there's likely that there's anything of that size. And then the fact that this actually works and gives us all the properties that we want, that's part of the analysis. So does everybody see that this takes Order one worst case time and what the basic structure of it is? These things, by the way, are not in this case prime. I could always pick primes that were close to this. I didn't do that in this case.

Or I could use a universal hash function that in fact would work for things other than primes. But I didn't do that for this example. We all ready for analysis? OK, let's do

some analysis then. And this is really pretty analysis. Partly as you'll see because we've already done some of this analysis. So the trick is analyzing level two. That's the main thing that I want to analyze, to show that I can find hash functions here that are going to, when I map them into, very sparsely, into these arrays here, that in fact, such hash functions exist and I can compute them in advance.

So that I have a good way of storing those. So here's the theorem we're going to use. My hash and keys into m equals n squared slots using a random hash function in a universal set H. Then the expected number of collisions is less than one half. OK. The expected number of collisions I don't expect there to be even one collision. I expect there to be less than half a collision on average. And so, let's prove this, so that the probability that two given keys collide under h is what?

What's the probability that two given keys collide under h when h is chosen randomly from the universal set? One over m. Right? That's the definition, right, of, which is in this case equal to one over n squared. So now how many keys, how many pairs of keys do I have in this table? How many keys could possibly collide with each other? OK. So that's basically just looking at how many different pairs of keys do I have to evaluate this for. So that's n choose two pairs of keys.

n choose two pairs of keys. So therefore, the expected number of collisions is while for each of these n, not n over two. n choose two pairs of keys. The probability that it collides is one in n squared. So that's equal to n times n minus one over two, if you remember your formula, times one in n squared. And that's less than a half. So for every pair of keys, so those of you who remember from 6.042 the birthday paradox, this is related to the birthday paradox a little bit.

But here I basically have a large set, and I'm looking at all pairs, but my set is sufficiently big that the odds that I get a collision is relatively small. If I start increasing it beyond the square root of m, OK, the number of elements, it starts getting bigger in the square root of m then the odds of a collision go up dramatically as you know from the birthday paradox. But if I'm less than, if I'm really sparse in there, I don't get collisions. Or at least I get a relatively small number expected. Now I want to remind you of something which actually in the past I have just assumed, but I want to actually go through it briefly. It's Markov's inequality. So who remembers Markov's inequality? Don't everybody raise their hand at once. So Markov's inequality says the following.

This is one of these great probability facts. For random variable x which is bounded below by 0, says the probability that x is bigger than, greater than or equal to any given value T is less than or equal to the expectation of x divided by T. It's a great fact. Doesn't happen if x isn't bound below by 0. But it's a great fact. It allows me to relate the probability of an event to its expectation. And the idea is in general that if the expectation is going to be small, then I can't have a high probability that the value of the random variable is large. It doesn't make sense. How could you have a high probability that it's a million when my expectation is one or in this case we're going to apply it when the expectation is a half?

Couldn't happen. And the proof follows just directly on the definition of expectation, and so I'mdoing this for a discrete random variable. So the expectation by definition is just the sum from little x goes to 0 to infinity of x times the probability that my random variable takes on the value x. That's the definition. And now it's just a question of doing like the coarsest approximation you can imagine. First of all, let me

just simply throw away all small terms that can be greater to or equal to x equals T to infinity of x times the probability that x is equal to little x. So just throw away all the low order terms. Now what I'm going to do is replace every one of these terms is lower bounded by the value x equals T.

So that's just the summation of x equals T to infinity of T times the probability that x equals x. OK. Over x going from T larger. Because these are only bigger values. And that's just equal then to T, because I can pull that out, and the summation of x equals T to infinity of the probability that x equals x is just the probability that x is greater than or equal to T. And that's done because I just divide by T.

So that's Markov's inequality. Really dumb. Really simple. There are much stronger things like Chebyshev bounds and Chernoff bounds and things of that nature. But Markov's is like unbelievably simple and useful. So we're going to just apply that as a corollary. So the probability now of no collisions, when I hash n keys into n squared slots using a universal hash function, I claim is the probability of no collisions is greater than or equal to a half. So I pick a hash function at random.

What are the odds that I got no collisions when I hashed those n keys into n squared slots? Answer. Probability is I have no collisions is at least a half. Half the time I'm guaranteed that there won't be a collision. And the proof, pretty simple. The probability of no collisions is the same as the probability as, sorry, is one minus the probability that I have at most one collision. So the odds that I have at least one collision, the odds that I have at least one collision, probability greater than or equal to one collision is less than or equal to, now I just apply Markov's inequality with this. So it's just the expected number of collisions --

-- divided by one. And that is by Markov's inequality less than, by definition, excuse me, of expected number of collisions, which we've already shown, is less than a half. So the probability of at least one collision is less than a half. The probability of 0 collisions is at least a half. So we're done here. So to find a good level to hash function is easy. I just test a few at random. Most of them out there, OK, half of them, at least half of them are going to work. So this is in some sense, if you think about it, a randomized construction, because I can't tell you which one it's going to be. It's non-constructive in that sense, but it's a randomized construction.

But they have to exist because most of them out there have this good property. So I'mgoing to be able to find for each one of these, I just test a few at random, and I find one. Test a few at random, find one, etc. Fill in my table there. Because all that is pre-computation. And I'mgoing to find them because the odds are good that one exists. So -- -- we just test a few at random. And we'll find one quickly --

-- since at least half will work. I just want to show that there exists good ones. All I have to prove is that at least one works for each of these cases. In fact, I've shown that there's a huge number that will work. Half of them will work. But to show it exists, I would just have to show that the probability was greater than 0. So to finish up, we need to still analyze the storage because I promised in my theorem that the table would be of size order n.

And yet now I've said there's all of these quadratic-sized slots here. So I'mgoing to show that that's order n. So for level one, that's easy. We'll just choose the number of slots to be equal to the number of keys. And that way the storage at level one is just order n. And now let's let n sub i be the random variable for the number of keys

-- -- that hash to slot i in T. OK, so n sub i is just what we've called it. Number of elements that slot there. And we're going to use m sub i equals n sub i squared slots in each level two table S sub i.

So the expected total storage -- -- is just n for level one, order n if you want, but basically n slots for level one plus the expected value, whatever I expect the sum of i equals 0 to m minus one of theta of n sub i squared to be. Because I basically have to add up the square for every element that applies here, the square of what's in there. Who recognizes this summation? Where have we seen that before? Who attends recitation? Where have we seen this before? What's the --

We're summing the expected value of a bunch of -- Yeah, what was that algorithm? We did the sorting algorithm, right? What was the sorting algorithm for which this was an important thing to evaluate? Don't everybody shout it out at once. What was that sorting algorithm called? Bucket sort. Good. Bucket sort. Yeah. We showed that the sum of the squares of random variables when they're falling randomly into n bins is order n. Right?

And you can also out of this get a, as we did before, get a probability bound. What's the probability that it's more than a certain amount times n using Markov's inequality. But this is the key thing is we've seen this analysis. OK, we used it there in time, so there's a little bit, but that's one of the reasons we study sorting at the beginning of the term is because the techniques of sorting, they just propagate into all these other areas of analysis. You see a lot of the same kinds of things. And so now that you know bucket sort clearly so well, now you know that this without having to do any extra work.

So you might want to go back and review your bucket sort analysis, because it's applied now. Same analysis. Two places. OK. Good recitation this Friday, which will be a quiz review and we have a quiz next, there's no class on Monday, but we have a quiz on next Wednesday. OK, so good luck everybody on the quiz. Make sure you get plenty of sleep.