PROFESSOR: So we're going to do rolling caches then, we're going to go a little bit over amortized analysis and if we have a lot of time left, we're going to talk about good and bad hash functions. So can someone remind me what's the point of rolling hashes? What's the problem? What are we trying to solve in lectures? Be brave.

AUDIENCE: Gets faster, I think, because like--

PROFESSOR: So what are we trying to solve? You don't need to go ahead, tell me what's the big problem that we're trying to solve.

AUDIENCE: I don't remember/

PROFESSOR: OK, so let's go over that. So we have a big document, AKA a long string, and we're trying to find a smaller string inside it. And we're trying to do that efficiently. So say the big document is-- you might have seen this before. And we're trying to look for the here. How do I do that with rolling hashes? So the slow, nice solution is I get this the and then I overlap with the beginning of the document, I do a string comparison. If it matches, I say that it's a match. It's not, I overlap it here. String match, I overlap it here. String match, so on and so forth. The problem is this does a lot of string matching operations, and the string matching operation is how expensive? What's the running time?

AUDIENCE: Order n.

PROFESSOR: Order n, where n is the size of the string. So if we have a string, say this is the key that we're looking for and n is the document size then this is going to be order n times k. We want to get to something better. So how do I do this with rolling hashes?

AUDIENCE: We take the strings up, and you come up with a hash code for it.

PROFESSOR: OK so we're going to hash this. And let's say this is the key hash. OK, very good.

AUDIENCE: And then once you know that, then you'll need to compute the next letter hash, or just add it on to that pairing.

PROFESSOR: OK, so next letter for--

AUDIENCE: Yeah. So you compute the hash of the entire string, n, capital n--

PROFESSOR: Let's not do that. Let's compute the hash of the first key characters in the string.

AUDIENCE: Are we separating them by space [? inside? ?]

PROFESSOR: Yeah, so this is going to be a character.

AUDIENCE: The space will?

PROFESSOR: Sorry?

AUDIENCE: The space will be a character?

PROFESSOR: Yeah. So let's take the first three characters and compute the hash of that. And let's call this the our sliding window. So we're going to say that window has the and then we're going to compute the hash of the characters in the window, and we're going to see that this matches the hash of the key. And then we'll figure out what to do. That aside, we're going to slide the window to the right by one character so take out key and put in the space. And now the window has HE space, we're going to compute the hash of the window, see that it's not the same as this hash of the key. What do we know in this case? Different hashes means--

AUDIENCE: Not the same string.

PROFESSOR: For sure not the same string. So this is not the. OK, now suppose I'm sliding my window so after this I will slide my window again, and I would have e space f. Right, so on and so forth. Suppose I'm happy sliding my window and then I get here and I have my window be IN space, and the hash the window happens to match the hash of the key. So we're in the same situation as here. Now what?

AUDIENCE: [INAUDIBLE].

PROFESSOR: Very good. We have to check if the string inside the window is the same as the string inside the key. And if it is, we found a match. If it isn't, we keep working. All right? And we have to do the same thing here. So this is our string matching algorithm.

AUDIENCE: Can we somehow make sure that we make a hash function such that it will never [INAUDIBLE]--

PROFESSOR: Excellent question. Thank you, I like that. Can we make a hash function so that we don't have any false positives, right? Let's see. How do hash functions work? What's the argument to a hash function and what's the return value?

AUDIENCE: The argument is something that you want to hash.

PROFESSOR: So in this case we're working with three character strings. But let's say we're looking for a one megabyte string inside the one gigabyte string. Say we're a music company and we're looking for our mp3 file inside the big files off a pirate server or something.

So this is 1 million character strings, because that's the window size. And it's going to return what? What do hash functions return for them to be useful? Integers. Nice small integers, right? Ideally, the integer would fit in a word size, where the word is the register size on our computer. What are popular word sizes? Does anyone know?

AUDIENCE: [INAUDIBLE].

AUDIENCE: Excellent. 32-bits, 64-bits integers. OK, so what's the universe size for this function? How many one million character strings are there?

AUDIENCE: How many characters are there?

PROFESSOR: Excellent. Let's say we're old school and we're doing [? SG ?]. We don't care about the rest of the world.

AUDIENCE: 256 characters?

PROFESSOR: OK.

AUDIENCE: To the one millionth power.

PROFESSOR: Cool. Let's say we're working on an old school computer, since we're old school and we have a 32-bit word size. How many possible values for the hash function?

AUDIENCE: 2 to the 32.

PROFESSOR: The [? other ?] is bigger. You're messing with me, right? OK so this is 2 to the 8th.

AUDIENCE: 2 to the 8 million, right?

PROFESSOR: Yup. So this is a lot bigger than this. So if we want to make a hash function that gives us no false positives, then we'd have to be able to-- if we have the universe of possible inputs and the universe of possible outputs, draw a line from every input to a different output. But if we have-- 2 to the 32 by the way is about four billion.

If we have four billion outputs and a lot of inputs as we draw our lines, we're eventually going to run out of outputs and we're going to have to use the same output again and again. So for a hash function, pretty much always the universe size is bigger than the output size. So hash functions will always have collisions. So a collision for a hash function is two inputs, x1, x2, so that's x1 is not x2, but h of x1 equals h of x2.

So this will always happen. There's no way around it. What we're hoping for is that the collisions aren't something dumb. So we're hoping that the hash function acts in a reasonably randomly and we talked about ideal hash functions that would pretty much look like they would get a random output for every input. And we're not going to worry too much about that. What matters is that as long as the hash function is reasonably good, we're not going to have too many false positives.

So say the output set is O, so O is 2 to the 32. Then we're hoping to have false positives about one every O times. So 1 out of 2 to the 32 false positives. So what's the running time for when you slide the window and we're doing this logic here. What's the running time if the hashes aren't the same?

AUDIENCE: What's the running time of the hash function--

PROFESSOR: Of the whole matching algorithm.

AUDIENCE: No, no no, of the hash function itself. Can we make any assumptions about that?

Very good. What's the running time of the hash function? So we're going to have to implement-- if we implement the hash function naively, then the running time for hashing a key character string is order key. But we're going to come up with magic way of doing it in order one time. So assume hashing is order 1. What's the running time for everything else? So if the hashes don't match, we know it's not a candidate. So we're going to keep going. So this is order 1. What if the hashes do match?

AUDIENCE: [INAUDIBLE] characters.

PROFESSOR: Order--

AUDIENCE: I mean, but it depends on how many ones match, but it will be--

PROFESSOR: So for one match, what's the running time for one match?

AUDIENCE: Order k--

PROFESSOR: Order k. Excellent. So the total running time is the number of matches times order k plus the number of non matches times order 1. So as long as the number of false positives here is really tiny, the math is going to come out to be roughly order 1 per character.

AUDIENCE: So the whole thing is order in.

PROFESSOR: Everything should be order n, yeah, that's what we're hoping for. OK so let's talk about the magic because you asked me what's the running time for the hash function and this is the interesting part. How do I get to compute these hashes and order 1 instead of order k? We have this data structure called rolling hash. So rolling hash-- does anyone remember from lecture what it is?

AUDIENCE: Isn't that what we're doing right now?

PROFESSOR: So this is a sliding window. And the data structure will compute fast hashes for the strings inside the sliding window. So how does it work? I mean not how does it work functionally, what are the operations for a rolling hash, let's try that.

AUDIENCE: Oh [INAUDIBLE].

PROFESSOR: OK, so we have two updates. One of them is pop. For some reason, our notes call it skip, but I like pop better, so I'm going to write skip and think pop. And the other one is?

AUDIENCE: Always [INAUDIBLE].

PROFESSOR: A pen with a new character, OK? Cool. So these are the updates. Now what's the point of those updates? What's the query for a rolling hash?

AUDIENCE: [INAUDIBLE]. You just grab the next character, append that, and then skip [INAUDIBLE].

PROFESSOR: OK, so this is how I update the rolling hash to contain to reflect the contents of my sliding window. And what I do after that? What's the reason for that?

AUDIENCE: You skip your [INAUDIBLE].

PROFESSOR: So don't think too hard, it's a really easy question. I moved a sliding window here. What do I want to get?

AUDIENCE: You want to get the hash of those characters.

PROFESSOR: The hash of those characters, very good. So this is the query. So a rolling hash has a sequence of characters in it, right? Say t, h, e. And it allows us to append the character and pop a character. Append a character, pop a character. And then it promises that it's going to compute the hash of whatever's inside the rolling hash really fast. Append goes here, skip goes here. How fast do these operations need to be for my algorithm to work correctly?

AUDIENCE: Order 1.

PROFESSOR: I promised you that computing hash there is order 1, right? So I have to-- OK. Let's see how we're going to make this happen. So these are letters. These make sense when we're trying to understand string matching. But now we're going to switch the numbers, because after all, strings are sequences of characters, and characters are numbers. And because I know how to do math on numbers, I don't know how to do math on characters. So let's use this list. Let's say that instead of having numbers in base 256 which is [INAUDIBLE], we're going to have numbers in base 100, because it's really easy to do operations in base 100 on paper.

So 3, 14, 15, 92, 55, 35, 89, 79, 31. So these are all base 100 numbers. And say my rolling window is size 5. One, two, three, four, five. So I want to come up with a way so that I have the hash of this and then when I slide my window, I will get the hash of this. What hashing method do we use for rolling hashes? Does anyone remember?

AUDIENCE: [INAUDIBLE].

PROFESSOR: Mod, you said-- I heard mod something.

AUDIENCE: Yeah, that's what I said.

PROFESSOR: OK, so? So? So the hash is? The hash of a key is?

AUDIENCE: It's k mod m or m k. [INAUDIBLE].

PROFESSOR: OK. I'm going to say k mod something, and I'm going to say that something has to be a prime number and we'll see why in a bit. Let's say our prime number is 23. So let's compute the value of the hash for the sliding window of the first sliding window and then we'll compute the hash for the second sliding window. Oh, there is some at the computer, sweet. 314159265 modulo 23 is how much? OK, while you're doing that, can someone tell me what computation will he need to do for the second sliding window?

AUDIENCE: 1519265359.

PROFESSOR: 159265359.

AUDIENCE: That's a third sign.

AUDIENCE: There's a 1-4 before that.

AUDIENCE: The first one is 11.

PROFESSOR: OK. And what's the second one?

AUDIENCE: [INAUDIBLE] adding--

PROFESSOR: 1415926335 modulo 23.

AUDIENCE: 5.

PROFESSOR: I heard a 5 and a 7. OK.

AUDIENCE: Hold on, hold on.

PROFESSOR: I'll take the average of those two and we can move on, right?

AUDIENCE: Three five, and arguably 6.

PROFESSOR: All right, so let's implement an operation called slide. And slide will take the new number that I'm sliding in. And the old number that I'm sliding out, making my life really easy. So in this case, the numbers would be--

AUDIENCE: The new one is 35.

PROFESSOR: And the old one?

AUDIENCE: 3.

PROFESSOR: Excellent. And I want to have an internal state called hash that has 11 and I want to get 6 after I'm done running slide. This is still too hard for me, so before we figure out hash, let's say that we have an internal state called n. And n is this big number here. So I want to get from this big number to this big number. What am I going to do?

AUDIENCE: Mod 3,000 [INAUDIBLE].

PROFESSOR: OK, so you want to take the big number 159265 and mod it.

AUDIENCE: [INAUDIBLE]

PROFESSOR: So if I mod it to by a big number, that's going to be too slow. So I can't mod it.

AUDIENCE: Can't you just divide it?

PROFESSOR: Division is also slow, I don't like the division. I like subtraction, someone said subtraction. So what I want to do is I want to get from here to a number that-- to this number, right? So I want to get rid of the 3 and I want to add 35 at the end. To get rid of the 3, what do I subtract?

AUDIENCE: 3 with a bunch of 0s.

PROFESSOR: 3 with a bunch of 0s. Excellent. 1, 2, 3, 4, 5, 6, 7, 8. How many of them are there?

AUDIENCE: 8.

PROFESSOR: OK, how many digits conveys 100?

AUDIENCE: Oh, 2 right?

AUDIENCE: 4.

AUDIENCE: Oh, oh.

PROFESSOR: 4. So base 100, so two numbers--

AUDIENCE: One base 100 number is two digits. Yep.

PROFESSOR: So 8. yeah, OK, 4. Cool. So let's try to write this in a more abstract way. So n is the old n minus old, right, so that 3 is old times what do I have to multiply it by to get all those zeros?

k minus 1? [INAUDIBLE] to that base whatever.

PROFESSOR: OK, so-- base to the size to something. K minus 1. So K is 5 in this case, right? My window is 5. And I see a 4 there, so I'm going to add the minus 1 just because that's what I need to do. OK, so then I get 14159265. What do I do to tack on a 35 at the end?

AUDIENCE: [INAUDIBLE] 35.

PROFESSOR: OK, times the base, so that's going to give me the zeroes. And then this is a minus here. And then I'm going to add 35. Right? 1415926535. Look, it's right. So what do I write here?

AUDIENCE: The base first.

PROFESSOR: Good point. OK. Let me play with this a little bit before we go further. I'm going to distribute the base here. So this is n times base minus old times base to the k plus mu. And let's rename base to size to be the size of the window, I don't like k. And I'm renaming it because later on we're going to break our slide into appends and skip and the size won't be constant anymore. OK so does this make sense? It's all math. So this math here becomes abstract math here. But nothing else changes.

OK, so now I want to get hash-- I want to get hash out of n, how do I do that?

AUDIENCE: Mod 23.

PROFESSOR: Mod 23, very good. So in a general way, I would say mod p. OK so hash is n times base minus old times base to the size plus new mod p. Now let's distribute this. I know I can distribute modulo across addition and subtraction, so I have n mod p times base minus old times base to the size mod p plus new. And everything still has to be a mod p. So can someone tell me where did I add the mod p? Why did I put it here and here?

AUDIENCE: [INAUDIBLE] the original?

PROFESSOR: OK, nmodp is hash, let's do that. So what's true about both n and base to the size?

AUDIENCE: Constant.

PROFESSOR: Constant?

AUDIENCE: Like can you please repeat it?

AUDIENCE: You could [INAUDIBLE] base to the size but you can't [INAUDIBLE] hash, I mean [INAUDIBLE]--

PROFESSOR: Hm. OK, so keep this in mind that we can compute this, because we're going to want to do that later. But what I had in mind is the opposite of constant, because n is huge. Right? And base to the size is also huge, right? N is this number. Base to the size is this number here. 1 followed by this many zeros, so these numbers are big. All the other numbers are small. Base is small, old is small, new is small, p is small.

PROFESSOR: So I want to get rid of the big numbers, because math with big numbers is slow. So unless I get rid of the big numbers, I'm not going to get to order 1 operation. So we already got rid of this one because it's hash and how do I get rid of this one?

AUDIENCE: [INAUDIBLE]

AUDIENCE: There's some 6042 algorithm that does that quickly.

AUDIENCE: Well, we definitely just went over this in class today.

AUDIENCE: Which is why you needed the prime number, right?

PROFESSOR: Not quite. There is an algorithm that does it quickly. That algorithm is called repeated squaring and the quickest-- wait, I'm not done, I promise I'm not done. So the quickest that this guy can run if you do everything right is order of [? log ?] size. If the window size is 1 megabyte, 10 megabytes, if the window size keeps growing, if the window size is part of the input size, is this constant? Nope. So I can't do that. Someone else gave me the right answer before.

What did you say before?

AUDIENCE: Pre-compute it?

PROFESSOR: OK. It's a constant, so why don't we pre-compute it? Take it out of here, compute it once, and after that, we can use it all the time. And unless someone has a better name for it, I'm going to call this magic. The name has to be short, by the way, because I'll be writing this a few times.

OK, so now we have hash equals hash times base minus old times magic plus new modulo p. Doesn't look too bad, right? Pretty constant time. Now let's write the pseudo code for the rolling hash, and let's break this out into an append and a skip at the same time.

AUDIENCE: What if hash is bigger than your word size?

PROFESSOR: So hash is always going to be something modulo p.

AUDIENCE: Oh that's true, OK.

PROFESSOR: So as long as p is decent, it's not going to get too big.

AUDIENCE: All right. What if old and new [INAUDIBLE]--

PROFESSOR: So old and new--

AUDIENCE: P is a big number . 314159269 is possibly bigger than your word size, right?

PROFESSOR: Definitely. So that's why we're getting rid of it.

AUDIENCE: That is true. [INAUDIBLE]

PROFESSOR: So this is k digits in base b. Too much. Not going to deal with it. Hash is one digit in base p, because we're doing it mod p. Old and new are one digit base b. So hopefully small numbers. OK, I haven't seen a constructor in CLRS, so I'm going to say that when you write pseudocode, the method name for a constructor is in it because we've seen this before.

And let's say our constructor for a rolling cache starts with the base that we're going to use. And it builds an empty rolling hash, so first there's nothing in it. And then you append and you skip and you can get the hash.

AUDIENCE: What about p? Shouldn't you also do p?

PROFESSOR: Sure. Do that. So let's say base and p are set, so somethings sets base and p. And we need to compute the initial values for hash and magic. What's hash? Zero. There's nothing in there, right? The number is 0. What's magic?

AUDIENCE: [INAUDIBLE]. Well, I mean, you can calculate it, right?

PROFESSOR: So magic is based to the size mod p. What size?

AUDIENCE: [INAUDIBLE] 0. Just one mod p.

PROFESSOR: Yep. So when I start, I have an empty sliding window. Nothing in there, size is 0, base to the size is 1, whatever the size is. Very good. Let's write append. Hash is? So here, we're doing both an append and the skip. We have to figure out which operation belongs to the append, which operations belong to the skip. So someone help me out.

AUDIENCE: We know subtraction would [INAUDIBLE]--

AUDIENCE: Multiply mod base [INAUDIBLE].

PROFESSOR: Yup. So this is the append, right? And this is the skip. So hash equals hash. Times base plus new mod p. Very good. This is important. If you don't put this in, Python knows how to deal with big numbers. So it will take your code and it'll run it, and you'll get the correct output. But hash will keep growing and growing and growing because you're computing n instead of hash. And you'll wonder why the code is so slow. So don't forget this. What else do I need to update? OK, I don't have a constant for that, but I have a constant I for something else. Magic. So magic is base to the size mod p. So what happened to the window size?

AUDIENCE: Oh. Times base [INAUDIBLE].

PROFESSOR: Excellent. The window size grows by 1, therefore, I have to multiply this by base. Magic times base mod p.

AUDIENCE: Does p always have to be less then the base, or can it be anything?

PROFESSOR: It can be bigger than the base. So if I want to not have a lot of false positives, then suppose my base is 256, because that's an extra character. I was arguing earlier that the number of false positives that I have is 1/P basically. So I want p to be as close to the word size as possible. So p will be around 2 to the 4 billion. So definitely bigger. It can work either way. It's better if it's bigger for the algorithm that we're using there. All right, good question, thank you. Skip. Let's implement skip. Hash is?

AUDIENCE: Hash minus old [INAUDIBLE] then comes magic [INAUDIBLE].

PROFESSOR: OK, can I write this in Python? What happens if I write this?

AUDIENCE: [INAUDIBLE] magic is, [INAUDIBLE] We won't be able to find it.

PROFESSOR: OK so-- sorry, not in Python. So assume all these are instance variables done the right way, but what happens if old times magic is bigger than hash? I get a negative number. And in math, people assume that if you do something like minus 3 modulo 23, you're going to get 20. So modulo is always positive in modular arithmetic, but in a programming language, if you do minus 3 modulo 20, I'm pretty sure you're going to get minus 3. And things will go back. So we want to get to a positive number here so that the arithmetic modulo p will work just like in math. So we want to add something to make this whole thing positive.

AUDIENCE: That's something times [INAUDIBLE].

PROFESSOR: OK, so if we're working modulo p then we can add anything to our number, any multiple of p, and the result modulo p doesn't change. For example, here to get from minus 3 to 20, I added 23. Right? OK, so I want to add a correction factor of p times something. So what should that be? I want to make sure that this whole thing is positive.

AUDIENCE: [INAUDIBLE]

PROFESSOR: So let's see. How big are these guys, by the way? Magic is something mod p, right? So it's definitely smaller or equal to p. How about old?

AUDIENCE: [INAUDIBLE]

PROFESSOR: OK. So smaller or equal than?

AUDIENCE: Base.

PROFESSOR: Base. Very good. So this whole thing is definitely going to be smaller than [INAUDIBLE]. So this is definitely going to be smaller than base time p, right? So let's put that in here. You can get fancy and say hey, this is smaller than p, and this is old, so you can put old here instead, same thing. OK so we have hash. Now what do we do to magic?

AUDIENCE: [INAUDIBLE] divide it by the base and mod p. It seems base [? and p ?] don't share factors. You're allowed to do that?

PROFESSOR: OK, so skip part two. Magic equals-- So what if my magic is something like 5 and my base is 100? How is this going to work? This is where we use fancy math. And I call it fancy math because I didn't learn it in high school. So I'm assuming at least some of you do not know how this works. So if we're working modulo p, you can think 23 if you prefer concrete numbers instead.

For any number between 1 and p minus 1, there's something called the multiplicative inverse, a to the minus 1, that also happens to be an integer between 1 and p minus 1. And if you multiply, say, a times b, that's another number. And then you multiply this by a minus 1, you're going to get to b modulo p. So a minus 1 cancels a in a multiplication. Now let's see if you guys are paying attention. What's a times a to the minus 1 modulo p?

AUDIENCE: 1.

PROFESSOR: OK. Sweet. So suppose I want to find the multiplicative inverse of 6. What is it?

AUDIENCE: Is that the mod 23?

PROFESSOR: Yeah. Can someone think of what it should be?

AUDIENCE: 4.

PROFESSOR: 4, wow, fast. So 6 times 4 equals 24, which is 1 modulo 23. Now let's see if this magic really works, this math magic. So 6 times 7 equals?

AUDIENCE: 42.

PROFESSOR: Which is what mod 23? Computer guys.

AUDIENCE: Negative 4, so 5. Ah, just kidding. Yeah.

PROFESSOR: OK now let's multiply 19 by 4. What is this?

AUDIENCE: 76.

PROFESSOR: All right, 76 modulo 23?

AUDIENCE: 7 maybe.

PROFESSOR: Are you kidding? Did you compute it, or did you use--

AUDIENCE: 69 [INAUDIBLE]

PROFESSOR: OK. Started with 7, ended with 7. So this works. So as long we're working modulo a prime number, we can always compute multiplicative inverses. And Python has a function for that, so I'll let you Google its standard library to find out what it is. But it can be done, that's what matters as far as we're concerned. So we're going to say that magic is magic times base minus 1 mod p, which is the multiplicative inverse everything mod p.

Now suppose this base minus 1 modulo p, this multiplicative inverse algorithm is really slow. What do we do to stay order 1? Pre compute it. Base is not going to change. Very good. So the inverse of base, I base, is base minus 1 mod p. So here I replace this with I base. OK so skip part one is there, skip part two is here. Does this make sense so far? I see some confusion.

AUDIENCE: A lot.

PROFESSOR: A lot to take in at once?

AUDIENCE: Yes.

PROFESSOR: OK. So remember this concept. So this is where we started from. Then we computed n, then after n, we worked modulo p to gets to hashes. So by working module p, we're able to get rid of all the big numbers and we only have small numbers in our rolling hash. And there's that curveball there, there is that inverse, multiplicative inverse, but Python computes it for you, so as long as it's in the initializer, here you don't need to worry about it, because it's not part of the rolling hash operations.

By the way, what's the cost of the rolling hash operations? What's the cost of new? Sorry, what's the cost of append? Not thinking here. Constant. All these are small numbers, so the arithmetic is constant, right? What's the cost of skip? Skip part 1 here, skip part two there. What's the cost of skip? Constant. . All the numbers are small. We went through a lot of effort to get that, so skip is order 1. We're missing hash. How would we implement the hash operation? A hash query. It's easy. Sorry?

AUDIENCE: [INAUDIBLE] lookup [INAUDIBLE].

PROFESSOR: So a rolling hash has append, skip, and hash. I want to implement that hash function. Hash. We're computing hash all the time. Return. Sorry, I didn't understand what you meant by lookup.

AUDIENCE: It's one of our states.

PROFESSOR: Yeah. Exactly. So the hash function returns hash, right? What's the cost of that? Constant. So append is constant time, yes? Skip is constant time. Hash is constant time. We're done. This works. Any questions on rolling hashes before you have to implement one of your own?

AUDIENCE: [INAUDIBLE] wouldn't it be easier to use a shift function? Then you don't have to think about plus and minus.

PROFESSOR: A shift function.

AUDIENCE: Well I mean like, you can shift bit-wise, right?

PROFESSOR: OK.

AUDIENCE: So you can just use shift instead of thinking about where to add this, where to subtract this.

PROFESSOR: Well so I do bit operations if I'm willing to work with these big numbers.

AUDIENCE: But then you have to compute the mod of some big number, right? Like just like that. For this one, you don't have to, because you have the original hash [INAUDIBLE].

AUDIENCE: Oh, you mean the big number being the actual word you're looking at?

AUDIENCE: Yeah.

PROFESSOR: So doing shift is equivalent to maintaining a list and pushing and popping things into the list. And then you have to do a hash, it's equivalent to looking over the entire list and computing the hash function. Because you'd have a big number and you have to take it modulo 23. And that's order of the size of the big number. But we're not allowed to do that. Hash has to be constant time, otherwise this thing is slow.

AUDIENCE: Why do we compute magic numbers then?

PROFESSOR: Why do we compute magic? We compute magic because somewhere here, we had this base to the size mod p and this could get big. So I can't afford to keep it around and do math with it all the time. So I can't compute base to the size every time I want to do append.

AUDIENCE: Would it be worth it if you're computing 100 different values for matching and [INAUDIBLE], so all you'd have to do is, when reassigning magic, just look up--

PROFESSOR: So if you do that, then you have to compute values for all the sizes, right? For all the window sizes.

AUDIENCE: Right. So if we assume that window sizes will be less than 100, it doesn't take very long.

PROFESSOR: Well what if the window size is 1 million? What if I'm looking for a 1 million character in a 1 gigabyte string?

AUDIENCE: But wouldn't after all, wouldn't the size just be around the string, like plus or minus the size of the base? So--

AUDIENCE: Only if [INAUDIBLE]

So why would the size change again? Why wouldn't it just be-- I mean, if you're looking at one character.

PROFESSOR: So if I have a sliding window like this, then it doesn't change. But if I want to implement a rolling hash, that's a bit more general and that supports append and skip. Whenever I append, the size increases. Whenever I skip, the size decreases.

AUDIENCE: Oh, you're not doing those at every time step. You're doing them as needed.

PROFESSOR: So I'm trying to implement that, that can do them in any sequence.

AUDIENCE: Oh.

AUDIENCE: OK. I thought we were just doing sliding window.

PROFESSOR: So if we're just doing sliding window, you can--

AUDIENCE: This is really more caterpillar hash instead of rolling hash, like it's more general.

PROFESSOR: Yeah. It's a bit more general. So let's look at rolling hash for the window. And what you're saying is, hey, the window size is constant, so--

AUDIENCE: Why do we repeat magic [INAUDIBLE]?

PROFESSOR: Yeah, if the window size is constant, then we wouldn't re compute it. It wouldn't change. But with this thing, it's not. OK.

AUDIENCE: But I guess it doesn't really matter, but even if you call these in the same order, then isn't that wasting a lot of computing cycles because just shrinking and then growing every single operation?

PROFESSOR: Oh, well it turns out that a lot of computing cycles is still order one, right? Everything is order one. So as algorithms people, we don't care. If you're doing it in a system and you actually care about that, then OK. But you're still going to have to compute the initial value at some point.

AUDIENCE: But if you know window's staying the same, you don't need to that computation every time?

PROFESSOR: If you-- sorry?

AUDIENCE: If you know you're actually just doing a window rolling hash--

PROFESSOR: Yup. So then you would initialize magic here to be whatever you want it to be, right? But then when you add the first few characters to the window, you have to figure out how to add them. So the code gets more messy. It turns out that this is actually simpler than doing it that way.

AUDIENCE: [INAUDIBLE] magic I guess I'm just confused because it seems like we're still working with the large numbers every time [INAUDIBLE].

PROFESSOR: Oh. Let's see. Mod p, mod p.

AUDIENCE: That's not-- so even though you're still multiplying magic times base, it doesn't matter.

PROFESSOR: After I'm going that, I'm reducing it modulo p. Yeah.

AUDIENCE: And then because we're only working with the smaller values.

PROFESSOR: Yup. So everything here stays between 0 and base or 0 and p. Actually hash is between 0 and p and magic is between 0 and p. OK.

AUDIENCE: How big does p usually get?

PROFESSOR: How big does p usually get. So And let me get back to this. So I was arguing that the number of false positives here is one over O, right? is the number of values that the hash function can output. How many hash functions can we output using a rolling hash?

AUDIENCE: P.

PROFESSOR: P. OK. So the number of false positives is 1/P. So what do we want for p?

AUDIENCE: We want p to be the word size, because but if p's the word size, then--

PROFESSOR: So p can't be the word size, because it has to be prime, right? But we want it to be big, because as p becomes bigger, 1/P becomes smaller. So there are two constraints. We want p to be big so that we don't have a lot of false positives. And we want p to be small so that operations don't take a lot of time. So in engineering, this is how things work. We call it a tradeoff because there are forces pushing in opposite directions, and it turns out that a reasonable answer to the trade off is you make p fit in a word so that all those operations are still implementable by one CPU instruction.

You can't have it be the word size. So if we're working on a 32-bit computer, I can't have this be 2 to the 32. But I can have a prime number that's just a little bit smaller than 2 to the 32.

AUDIENCE: Wait, why can't it be the word size? Or why can't it be 2 to the 32?

PROFESSOR: So if p would be this instead of a prime, then I can't do this.

AUDIENCE: Oh, right right right, yeah I knew that.

PROFESSOR: There are a lot of moving parts here and they're all interconnected.

AUDIENCE: You could do that for any prime number, right?

PROFESSOR: Yup. So this works for prime numbers, but it doesn't work for non prime numbers.

AUDIENCE: You could find the multiplicative inverse for any prime number in base 32. Is that true? I mean any odd number is what I'm trying to say. No, that's not true.

PROFESSOR: I refuse to answer hard math questions.

AUDIENCE: They need to be relatively prime. They need to share no factors.

PROFESSOR: Yes, it might be true.

AUDIENCE: So an odd will not share a factor with 2 to the 32?

PROFESSOR: You're forcing me to remember hard math.

AUDIENCE: Yeah, I totally just thought about this as [INAUDIBLE] number.

PROFESSOR: So, no, it turns out that there's no-- if you're working modulo and non prime base, then there's no multiplicity inverses. So some numbers have no multiplicative inverses, and other numbers have more than one multiplicative inverse. And then the whole thing doesn't work. So let me see if I can make this work without having an example by hand. Let's say we're working mod 8, right? Mod 8. So 2 to the minus 1 mod 8 is not going to exist, right?

AUDIENCE: Right, but 3 will.

PROFESSOR: 3. Let's see what do we use? 3 times 3 is 9, right? So this is 1. How about 3 times 5? 15 mod 8 is 7. So 3 and 5, and then--

AUDIENCE: 11.

PROFESSOR: OK. So 3 times 7 would be 21, 5. OK so 3 and-- 3 is the multiplicative inverse of itself, and 5 and 7 are-- yeah. I have to build a more complicated example, but this breaks down in some cases. I'll have to get back to you. I will look at my notes for modular arithmetic and I'll get back to you guys over email for why and how that breaks. Yes.

AUDIENCE: Sorry, can you tell me again why we did the part 2 in skip? Like why did we do that? I'm not really sure [INAUDIBLE].

PROFESSOR: So we started with magic 1 and then we-- in order for this to work, we agree that magic will be base to the size modulo p all the time. So this has to be [INAUDIBLE] invariant for my rolling hash. When I do an append, the size increases by 1. And then I multiply by base to modulo p. When I do a skip, the size decreases by 1. So I have to change magic, because magic is always base times size, so I have to update it.

So this is why this happened. Because initially, I wanted to update by dividing it by base, right? Magic divided by base. But if magic is 5 and base is 100, we're not going to get an integer. And we want to stay within integers, so that's when I pulled out fancy math and-- OK. OK. So how are we doing with rolling hashes? Good?

AUDIENCE: All this math will be in the notes, right?

PROFESSOR: Everything. Oh, yeah, everything else will be in the notes. Before we close out, I want to show you one cute thing. Who remembers amortized analysis? I know there's one person that said they understood.

All

PROFESSOR: The growing, shrinking thing is what we did in lecture. I want to show something else. I want to show you a binary tree. A binary search tree, because you've seen this on the PSAT and you already hate it.

AUDIENCE: Why'd they call it amortization? Because I looked it up online, it means to kill, and so I'm like, why not say like, attrition or something else that's a little bit less--

PROFESSOR: Amortization is also used in accounting to mean you're--

[INTERPOSING VOICES]

PROFESSOR: Let's use the growing hash example, because that's good for why this is the case. So when you're growing your table, you're inserting. If you still have space, that's order one. If not, you have to grow your table to insert. And that is more expensive. That's order n where n is how many elements you had before. So if you graph this costs, if you start off with a table of size one, you can insert the first element for a cost of one. For the second element, you have to resize the table, so it's a cost of two.

Now when you're trying to insert the third element, you have to resize the table again to a size of 4. But when you insert the fourth element, it's free. Well, cost of one. When you insert the fifth element, you have to resize a table to the size of eight, right? So The table size is one, two, four, four, and now it's eight. But because they resized this to eight, the next three assertions are going to be order one. And then the one after that is going to make the table be 16. So I can do seven insertions for free and then I'm going to have to pay a lot more for the next one.

Someone said dampening. I like dampening because the idea behind amortization is that you can take-- you have these big costs and they don't occur very often. So you can think of it as taking these big costs and chopping them up. For example, I'm going to chop this up into four and I'm going to take this piece and put it here. This piece and put it here. This piece and put it here. And then I'm going to chop this guy into two, and then take this piece and put it here. And the beginning's a little bit weird, let's not worry about that but this guy, if I chop this guy up into eight, it's going to happen, is it?

Well we can put-- so this guy grows exponentially, right? Every time it's multiplied by 2. But the gap size here also is multiplied by 2. So when I chop this up and I re distribute the pieces, it turns out that the pieces are the same size. So if I apply a dampening function that does this, then the costs are going to look-- they're not going to be on one, they there are going to be three or something.

And they look like this. Now, my CPU time is going to look like this, right? That's not going to change, because that's what's really happening. But what I can argue for is that if I look for a bunch of operations, say if I look at the first 16 insertions, the cost of those is the sum of these guys. So it's not been squared, which is what you would get if you look at the worst cases here, but it's order n. So this is what's being dampened, the amount of time an operation takes. Does this make some sense?

All right, I want to show you a cute example for amortization. And I'll try to make it quick. So how do you list the keys in a binary search tree in order?

AUDIENCE: [INAUDIBLE]

PROFESSOR: In order traversal, right? OK, there's another way of doing it that makes perfect intuitive sense. Get the minimum key, right? And then output the minimum key, then while you can get the next largest, which is the successor-- so while this is not, now output that key, right?

If you do the thing within order traversal, you get order end running time. What's the running time for this?

AUDIENCE: For n. You're going through all the keys, too.

PROFESSOR: Yeah, but next largest-- what's the running time for next largest?

AUDIENCE: Log n.

PROFESSOR: So this guy's log in, right? So I have n keys, so this whole thing is O of n logn. So it's definitely not bigger than n logn. But now, let's look at what happens using the tree. When I call min, I go down on each edge. And then I call successor and it outputs this guy. Then I call successor and it goes here. Than I call successor and it goes up here and here and outputs this guy. Successor goes here. Successor goes here. Successor goes all the way down here, successor goes up here, successor goes here, and then successor goes all the way up to the roots and gives up.

AUDIENCE: [INAUDIBLE]

PROFESSOR: So how many times do I traverse each edge? Exactly twice, right? How many edges in the tree? If I have n nodes, how many lines do I use to connect them? [INAUDIBLE]. So 1 node, zero lines. 2 nodes, one line. Three nodes, two lines. So n nodes, n minus one. N asymptotically, good. Good answer. Order, n, edges. Right, each edge gets traversed exactly twice. So amortized cost for n next largest operations is order n. So you can do this instead.

This code makes a lot more sense than in order traversal. OK, and the last part is remember that list query that was on the PSAT? Turns out you can do a find for the lowest element and then call successor until you see the highest element for the same argument. Well, I couldn't tell you this for the PSAT because we hadn't learned amortized analysis, so you wouldn't be able to prove that your code is fast.

But now if you get the intuition, you can write it that way. And your code will still be fast. Same running time. So the intuition for that is a bit more complicated. The proof is more complicated. But the intuition is that say this is l and this is h. Then I'm going to go in this tree here. So the same edge magic is going to happen, except there will be logn edges that are unmatched here and logn edges that aren't unmatched here.

Because once I find the node that's next to h, I'll stop, right? So some edges will not be matched. So then I'll say that the total running time is logn plus a.

AUDIENCE: i being the number of elements you pull out, right?

PROFESSOR: Yup. So this is amortized analysis. The list is hard. The traversal is easy. Remember the traversal. That's easy to reason about, so that's good. OK. Any questions on amortized analysis? So the idea is that you look at all the operations, you don't look at one operation at a time.

And you're trying to see if I look at everything, is it the case that I have some really fast operations and the slow operations don't happen too much, because if that's the case, then I can make an argument for the average cost, which is better than the argument that says this is the worst case of an operation, I'm doing an operation, the total cost is n times the worst cost. Make some sense? OK. Cool. All right. Have fun at the next p set.