Week 4 Module 7 Algorithm Analysis

Primality Testing 2.1

Hi there hope you are having a great day. Today we are going to talk about how to analyze algorithms. We talk about correctness of algorithms and how to assess the resources algorithms used. Whether it is time memory network or stuff like that. We will focus mostly on time but same concepts would work for other resources as well. We will start by talking about the problem of testing whether a number is prime or not. Before we look at the problem definition let's look at some other two definitions. First is defining what is a prime number. I believe you probably know that but let's make it formal. So assuming we have an integer num that is greater or equal to two. We say that num is prime if its only factors are one and num itself. So if it divides only by num and one that would make it a prime number. For example 13 is prime because 13's only factors are 1 and 13 no other number divides by 13. On the other hand 12 is not prime because besides 1 and 12 for example 6 is a factor of 12. That would make it not a prime number. Another definition I want us to note here is complimentary dividers. The definition goes something like that again let's have an integer number num greater or equal to two. And let's assume d and k are two dividers of num. We would call k and q complimentary dividers of num if when we multiply them we get num. So if d times k equals num. For example for num equals 100 4 and 25 are complimentary because if you multiply 4 by 25 you will get 100. And so are 5 and 20 both are dividers if you multiply 5 by 20 you will get 100. That would make 4 or 25 a complimentary couple of dividers and 5 and 20 complimentary dividers. For example 2 and 10 which are also two dividers of 100 are not complimentary dividers because if we multiply 2 and 10 we won't get 100. So let's have these two definitions we will use them later on. And let's take a look at the problem of testing primality. So let's write a program that reads from the user an integer let's assume that it is greater or equal to two. And the problem would need to determine whether this number is prime or not. So an execution could look something like. Assuming we have we will ask the user to enter an integer greater or equal to two. The user would enter 911 and then the program would respond by saying 911 is a prime number. You can try it out and check that 911 is indeed a prime number. Ok let's go ahead and solve it.

Primality Testing: Solution 1 2.2

Ok so let's solve this problem. Let's implement a function isPrime that given a parameter num an integer or parameter would determine whether our num is prime or not. Basically would return a Boolean value true or false. The most straightforward solution would say just iterate over the numbers 1 through num count how many of them are factors of num. And after you do all of that if the end you have only two factors one and num are obviously factors of num then these are the only two factors basically stating that num is prime. If you have more than two factors than you should return false and say that num is not prime. So we will iterate over the numbers and maintain some kind of divider counter something like that. So let's declare count divs as an integer set it to zero and then we will iterate from one to num. Each time we will check whether I is a factor of num basically we will check if num mod I equals zero. If we divide them and we don't have any remainder that means it is a factor. In this case we will increment this count divs counter. After we are done doing all of that for all the numbers we will just ask whether count divs equals two in that case we will return true. Otherwise we will return false. Yes so that's the straightforward solution.

Primality Testing: Solution 2 2.3

Ok so the first solution we suggested basically went over the entire range of numbers one through num. I want to suggest another an alternative implementation an alternative solution here. Where now instead of going with the entire range one through num we will go only on the first half of the range. One through num by two and we will skip iterating over the rest of the numbers. How come can we do how come are we allowed to do that? Let's take a look at a simpler example. For example if num is one hundred the factors of num are one two four five and so on. You can see that num by two which is fifty actually that's the last factor of 100 besides 100 itself. So in the case of 100 it was safe to go only on the first half of the range and skipping the second half. So it makes sense to do that kind of an algorithm. The question is is it a coincident that there aren't any dividers

past 50 for 100 or can we say safely that no matter what the value of num is all the dividers are in the first half of the range and there are none dividers besides the number itself on the second half. And that's exactly what we are trying to show in order to show the correctness of this strategy. So let's try to argue that thing. So let's assume we have a divider in the second half of the range. Let k be such divider of num in the second half of the range therefore k is greater than num by two. And we will have to argue that k does not exist or maybe we should show that k basically equals to num. That's the only possible divider on the second half of the range. So let's have this k. Now let's take a look at k's complimentary divider. Let's say d is k's complimentary divider so basically it means that d times k equals num. Or in other words d is num over k. So let's take a look at what we have. So d is num over k and let's try to compare num over k to num over num by two. Now we know that k is greater than num by two so these two expressions are basically dividing num by either k or num by two. So when we are dividing num by k since k is greater than num by two the result of num by k is less than num by num a two. Right when you are dividing a number by a greater value you will result with a small amount than when you are dividing the same number with a smaller value. So altogether we can say since k is greater than num a two num a k is less than num by num by two. Simple arithmetic num by num by two would be equal to two. Altogether we get taking that thing together we see that d is less than two. Now these are complimentary divider of num specifically it means that d is a divider of num. Now a divider that is less than two that leaves us with d must equal to one right. The only divider that is less than two is one itself. So d is we had it before num over k equals one basically meaning that k equals num. That's what we wanted to show. So the only divider in the second half k was a divider in the second half is num itself. There aren't any divider in the second half. So we can feel safe iterating over the first half of the numbers and skipping the second half. That would make our algorithm faster I believe. So if we take a look at the previous implementation let's see what changes we need to do here. So basically we will still have the counter variable counting how many dividers we are going to have. But instead of ranging from one to num we will range from one to num by two. And then at the end a prime number we are not expecting to have two dividers right we are not going up to the end so num won't be tested. So a prime number would have only one divider in the first half of the range only one actually that's the only divider we are expecting for a prime number. If our num has more than one dividers it means that it is not a prime value. If count divs equals one we are going to return true otherwise we will return false. So that is it seems like a better version or a better implementation for the function isPrime.

Primality Testing: Solution 3 2.4

Ok so far we have two solutions two versions. One iterating over the entire range from one through num. The second iterating over half of the range from one through num by two. I want to suggest another version a third version here. In this case we will iterate not on the second half we will iterate only on the first square root of num that first values. So instead of going from one through num by two we will go from one to the square root of num which is less than num by two. Let's take a look at 100 here as an example here these are the factors of 100. Now 50 is 100 by two is num by two. And the square root of 100 is less than 50 is 10. So actually when we are taking a look at these factors of 100 doesn't seem like such a great idea to go only on the first square root of num values. And to test how many to count how many of them are dividers. Because here in this case we have a few dividers that are less than the square root but not like num by two that there aren't any dividers greater than num by two. It seems like there are a few dividers that are greater than square root of num right 20 25 50 they are all dividers of 100 that are greater than the square root. So it makes me wonder whether it is a correct idea or a correct algorithm to count the dividers only in the first square root of num values. Maybe by going only in that small range we won't find any dividers and all of them are kind of in the other part or the other portion of the range. From the square root to num maybe we are missing some dividers if we go only on this partial range here. But if we think about it a bit more deeply for example for 100 so 10 is the square root we will see that each divider that is greater than the square root in this case 100 has a complimentary divider that is less than 100 in this case one. Same thing 50 is a divider greater than the square root 2 is its complimentary divider that is less than the square root. And so 25 and 4 and 20 and 5. So in some sense maybe we are skipping maybe we are not counting the entire number of dividers but going over the first part the first portion up to the square root we will find at least one divider in each complimentary pair of dividers. So makes us think that maybe it is a good idea to go only on the first square root values and count the number of dividers there. We kind of believe that if there are none dividers in the first portion there won't be any complimentary

dividers in the second portion. We would obviously need to prove that generally not only for 100. And that's exactly what we are going to do right now. So the proofing technique we are going to use here is called proof by contradiction where we want to show some property let's say we want to show A. In this technique proof by contradiction in order to show that A is true we assume that A is not true kind of assume the contradiction the negation of A. We assume A is false and then we will just do some arguments and get to some universal contradiction. I don't know that one equals zero or some other thing that cannot hold. And by having a premise of A is false and getting a contradiction we will conclude that this assumption that A is not true is false. In other words that A must be true. In this example let's see how we use proof by contradiction as a proofing technique. So let's assume k and d are two complimentary dividers of num and the thing we cannot allow since we want to show that at least one in each pair is less than the square root. Our assumption would be that let's assume that both of them are not less than a square root. Or in other words both of them are greater than the square root. And we will show that that can hold we can't have two dividers two complimentary dividers that are both greater than the square root. So let's assume that it is the case here both are greater than the square root. And since we know that they are complimentary dividers we know that when we multiply them they would result to num. So num equals k times d. Now we assume that they are both greater than the square root. Basically k is greater than the square root of num and d is greater than the square root of num. Therefore when we multiply k by d we get something that is greater than the square root of num times square root of num right. K is greater than the square root d is greater than square root k times d is greater than square root times square root. Which is equal to num. Altogether when we take it we get that num is greater than num now that's absurd right. No number is greater than itself. Seven is not greater than seven. Five is not greater than five. Num greater than num that's an obvious contradiction. Which implies that our initial assumption that both of our dividers are greater than num is false. Or in other words it implies that at least one on each pair of complimentary dividers is less than the square root of num. Ok so given that now that we know that it is good enough to go only on the first portion of the range here from one to the square root. Let's see how we update our implementation so yea it is kind of straightforward. Instead of just going from one to num or from one to the num by two we will go from num to the from one to the square root of num. And in order to say that our num is a prime we are expecting to have only one divider in this range which is the number one itself. So the implementation would look something like that.

Runtime Analysis 2.5

Ok so we suggested three different solutions to the primality testing problem. First going on the over the entire range from one through num. Second going over the half of the range from one through num by two. And the third going first from one through square root of num that portion of the range. We've shown the correctness of all three versions. First one is obvious but the other two were needed a bit of an argument in order to be certain that they all would work for all values of num. Let's try to estimate the resources these versions require which is better than which. And let's concentrate on the running time of these three versions these three implementations. So let's try and figure out what is the running time of the first version the second version and the third version. Let's name them t1 t2 and t3. The time for version one the time for version two and the time for version three. So what can we say? Is t1 equals 354. Is t2 equals 1270? Actually I don't even think we can have a number for t1 and number for t2 and so on. Let's make a few observations of what we need to do when we are analyzing the runtime of an algorithm. So the first observation I want to note here is that the running time depends on the size of the input. So we can't say t1 equals 250 because it depends. If the input is 100 maybe it would take one amount of time. If the input is one million maybe the running time would be different. So the running time depends on the size of the input. In order to make our analysis of runtime in general we would need to parameterized the running time by the size of the input. So for example in our case of the primality testing problem the size of the input we would typically name it N. The size of the input is the input itself. So in this case N equals num. So t1 t2 and t3 are not just constants they are functions of N. So when we want to analyze the running time of these three algorithm we would analyze t1 of N how many what's the time given a value of N. T2 of N and t3 of N. Now when we want to let's make another observation here. When we want to analyze the run time of an algorithm obviously it is a function of N. It is parameterized by N. Kind of depends what we do in the algorithm. So if we add numbers that's faster than I don't know dividing numbers. If we work with integers that's faster than working with floating points and doubles. And it is kind of a pain to

take all of that things into consideration. We want to create a model that would give us an estimation and more quality kind of an estimation. For that since the running time depends on the operators we use and the types of data we are applying them on. In order to avoid all of that what we do is we ignore the machine dependent constants and basically count each primitive operation as one. So plus is counted as one and division is counted as one. We don't care if we are adding integers or floating points. Each primitive operator would be counted as one. Informally if we take these two observations together what we want to compare the number we want to give some kind of running time analysis. We try to compare the number of primitive operations the process executes as a function as its input size. Let's try to take that and use that criteria in order to analyze the running time of the first second and third versions of our primality testing algorithms. So let's try to compare the number of primitive operations each one of them does as a function of the input size.

Runtime Analysis 2.6

Let's start with the first one. So t1 of N down here. Let's see t1 of N equals. So first thing we do is set coutn divs to zero that's one. Then we set I to one that's another one. And the new start iterating in this for loop. Each iteration we do five things right we compare I to num this less or equal to. We take the mod. We compare it to zero. And potentially we have two increments count divs plus plus and I plus plus. So we do these five things over and over how many times? N times. So all together it has it costs us five times N. So we set count to zero that was one. We set I to one that was another one and then 5n that's the cost of our for loop. After that we do compare count divs to two that's another primitive operation. And either return true or false. So altogether that's two additional things. All together one plus one plus 5n plus two equals 5n plus four. That would give us t1 to be 5n plus 4. Let's try to do the same for t2 the second and the third versions. So let's go over to the second version. Once again count divs is initialized to zero that's one operation. I is initialized to one that's another primitive operation. And then we have actually in this case six things that are repeated over and over. Less than we need to calculate the div that's another thing. Mod double equals and potentially two increments. That would give us six things that are repeated but the number of iterations here is not N as it was before it is N by two. So it is six times N by two. After we do that we have in the if outside of the for loop we have two additional operations. Comparison and a return of either true or false. All together that adds up to 3n plus four. So t1 was 5n plus 4 t2 is 3n plus 4. That's an improvement that's better than 5n plus 4. Let's quickly analyze the third version. Once again we start with setting count divs to zero that's one operation. I to zero that's another operation. And then we have six operations repeating over and over in our for loop. The comparison I to square root of num. The calculating of square root of num which is by the way not a primitive operation but for this analysis we will just assume it is. Mod comparison to zero and potentially two increments. That would give us six operations primitive operations that are repeated but the number of times that are repeated is not N. Not N by two but square root of N. That would add to six times square root of N. After that we have these two primitive operations in the next if statement. All together it adds up to six square root of N plus 4. So t3 of N would be six square root of N plus 4. That clearly shows that given that type of a criteria where we try to compare the number of primitive operations as a function the process executes as a function of the input size. That t3 is the fastest the lowest value and then t2 is a bit slower and t1 is the slowest of the three.

Runtime Analysis 2.7

One more note I want to add here when we analyze the running time. Now we as you can see we are not measuring the running time using seconds as units. We are using them as number of primitive operations as units. There this is what we are counting. And we want to try to ignore the running time based on the computer we are running it on. So we want to ignore the hardware technology that the algorithm is ran on. Obviously if we take the a computer twice as fast our algorithm would run twice as fast. And we want to as I said before we want to make some kind of quality kind of a criteria that separates the algorithm by their quality. And for that we would try to avoid and to move aside the hardware technology. In order to do that we make what is called an asymptotic analysis that would look only on the order of growth of T of N. Not at the number of primitive operations but the number of growth of the number of primitive operations. So if informally we said that we compare the number of primitive operations executed by the process of the function of the input size. Using the asymptotic analysis we compare the asymptotic order of the number of primitive operations executed by the

process of the function of the input size. Obviously you don't know what are the asymptotic order. Basically means I will talk about it in a much more formal way in a few minutes but first let me give you as a rule of thumb. T of N is three N squared plus six N minus fifteen. In order to get the asymptotic order of T of N which is in this case theta of N squared. We say asymptotic order is theta or sometimes you will hear big O of N squared. But for our conversation let's use the term theta. In order to figure out the asymptotic order of three N squared plus six N minus fifteen is theta of N squared we will do two things. We will drop lower order terms the six N minus fifteen will be dropped off. We will stick only with the high order term in this case three N squared. And we will also ignore the leading constant. The three would be ignored. So we will be left with N squared. So we say that T of N is three N squared plus six N minus fifteen but that is theta in asymptotic order of N squared.

Runtime Analysis 2.8

If we look back at our t1 t2 and t3 the number of primitive operations each one of them does as a function of the input size. And we will take or we will try to figure out the asymptotic order of these three expressions. So the first one dropping off the four and dropping the leading constant five. We will be left with theta of N. And same thing for t2 of N. Again the four is ignored and the leading constant of three is also dropped off. So we are also left with N. And for t3 we are left with the square root of N. That means that t1 and t2 are both theta of N and t3 is theta of square root of N. Now that's a bit surprising if we look at the results we got here if we are comparing not the actual expression of 5n plus four three and plus four and six square root of N plus four. But comparing their order for growth N N and square root of N if we are comparing the order of growth then we get that t1 and t2 are equivalent asymptotic speaking. That means that going the entire range or going half of the range gives us two algorithms that are considered to be equivalent to one another. Where the third algorithm where we go for the first portion of the square root of N first numbers that is asymptotically lower than N right. Square root of N is asymptotically lower than N that means that t3 of N is asymptotically better than t1 and t2 of N.

Order of Growth: Formal Definition 3.1

Ok so we said that when we are analyzing a runtime of an algorithm basically we compare the asymptotic order of the number of primitive operations as a function of the input size. So if T of N that represents the number of primitive operations is three N squared plus six N minus fifteen. In order to figure out the asymptotic order of it as a rule of thumb we said we are dropping lower terms and we are ignoring the constants and that would give us theta of N squared. Let's make the asymptotic order the definition of theta a bit more formal. So it goes something like that. Assuming we have two functions F of N and G of N that basically represents running time means it goes from positive integers to positive real numbers. We say that F of N is theta of G of N if there exists two real constants c1 and c2 and another positive integer constant named n0. Where for all N greater or equal to n0 F of N is bounded in between two multiplicands of G of N it is bounded in c1 G of N and c2 G of N. And that should be for all N greater or equal to n0. Now you should take a very close look at this definition let's try to look at it visually. So for example if you have F of N that is the function we are trying to figure out the order of right. We are trying to say F of N is theta of G of N. So assuming you have a function F of N and assuming you look at two multiplicands of G of N. One of them is four times G of N this green line here the other is a third times G of N this kind of blue line here. And let's look at this point here eight. And you can see that when you look over eight all the values of F the black line here is in between these two multiplicands of G. In between four times G and a third times G. So you have three constants here you have n0 as eight that's the position from where it is all this inequality here is true. So starting at zero F of N is less or equal to four times G of N and it is greater or equal to a third times G of N. So our c1 would be four and our c2 would be a third. So we have these three constants two multiplicands of G that bound F of N starting at a specific point in this case at eight. Therefore we can say that F of N is theta of G of N that is what this definition is trying to say. In order to argue that F is theta of G you should show that starting at this specific point starting at n0 F of N is in between two multiplicands of G. And you have to show these two or you have to give these two multiplicands these c1 and c2.

Asymptotic Analysis Example 3.2

Ok let's try to use the definition we just saw the theta definition in order to show that three N squared plus six N minus fifteen is indeed theta of N squared. Now this proposition here three N squared plus six N minus fifteen is theta of N squared is follows whatever the theta definition is trying to define. So it is of the form F of N is theta of G of N. Or F of N here is three N squared plus six N minutes fifteen and our G of N here is N squared. So it is of the form F of N is theta of G of N. Let's follow the definition the theta definition in order to prove exactly that. So by the definition in order for F to be theta of G we need to show that there exists three constants c1 c2 and n0 that for all values greater or equal to n0 our F is in between c1 and c2 as multiplicands of G. So I don't know what c1 c2 and n0 are now but let's first try to figure them out or when we are proving we should first show them and appoint to them. And then we should show that they are good enough constants. So let's assume we have these constants and keep on with our proof. So we need to show that for all N greater or equal to n0 F of N is greater or equal to one constant times G of N and less or equal to another constant times G of N. So let's take a look at three N squared plus six N minus fifteen. Now three N squared plus six N minus fifteen is obviously less or equal to three N squared plus six N right we are just dropping off the negative fifteen which makes the entire expression here smaller or less than three N squared plus six N without the negative fifteen. If we want to make three N squared plus six N even greater instead of adding six N let's add six N squared. So three N squared plus six N squared is obviously even greater than three N squared plus six N and that is greater than three N squared plus six N minus fifteen. So altogether we have that three N squared minus six N minus fifteen is less or equal to nine N squared. That's one multiplicand of G of N so our c1 nine is a great choice for our c1 right. Now let's try to find c2. So let's take a look back at three N squared plus six N minus fifteen. Now I can say that three N squared plus six N minus fifteen is greater or equal to three N squared dropping off the six N minutes fifteen. But that would be true only if six N minutes fifteen is a positive value otherwise we can't just drop it. If it is a negative value it would make it smaller right. So let's see when is six N minus fifteen greater or equal to zero. So six N minus fifteen is greater or equal to zero if and only six N is greater or equal to fifteen or in other words N is greater or equal to 2.5. So we can't say that always three N plus six N minus fifteen is greater or equal to three N squared. But for values greater or equal to 2.5 that would be true. So let's take our starting position to be three let's take our n0 to be three. And for values greater or equal to three we can safely say that three N squared plus six N minus fifteen is greater or equal to three N squared. So c2 can then be three. Altogether we can say that starting at three now we have that three N squared plus six N minus fifteen is less or equal to nine squared and it is greater or equal to three N squared. So we have two multiplicands of N squared that bound this our F of N. So by definition we can conclude that three N squared plus six N minus fifteen is theta of N squared. Now when we made this proof we didn't know in the beginning what are c1 c2 n0 are going to be. We figured them out during our process of proving. After we have done that let's try now to read it over and make sure that our proof works correctly. So in order to show that our F of N is theta of G of N we take c1 to be 9 c2 to be 3 and n0 to be 3. Then for all values for all Ns greater or equal to three we have that three N squared plus six N minus fifteen is less or equal to nine N squared and it is greater or equal to three N squared therefore it is theta of N squared. So now it makes sense as we read it that we really showed that there are three such constants that have what we one of them to have to bound F of N by these two multiplicands of G.

Runtime Analysis: Example 1 4.1

Ok let's try to analyze the runtime of two examples. Let's start with the first one. So let's take a look at this program here. It asks the user to enter an input N and then it has N iterations right. I iterating from one to N. Each time let's take a closer look here we are iterating from one to N printing a star and breaking the line. So N times we are printing N stars and breaking the line. N stars breaking the line. N times we are doing that. Altogether it comes to a square N by N stars here. But our question more interesting question is what is the running time of this program here? What is T of N? Right again we are analyzing the running time as the function of the input size in this case our N. So let's see. We read the input we are going to make an asymptotic analysis so we don't really need to care about each primitive operator operation because anyway it is going to be dropped off later on in the asymptotic analysis. So I am not even counting the one operation for cout and the single operation for the cin so they are both primitive operations but we are ignoring them. And then we have these nested loops here when we are analyzing running time it is the best good practice to go

inside out. From the inner part of the algorithm towards the outer parts. So let's take a look at the inner loop here. So this inner for with the cout if we try to figure out how many operations we have there. So we can try and count the primitive operations assigning j to one and then each iteration we have three operations that are repeated N times and then we have a break line. But again since we want to make an asymptotic analysis here we can safely say that this body here makes theta of N operations right. So basically N times the outer loop would repeat these N operations right. So together all these executions would do N times N operations. Again asymptotically. That would make the entire runtime here asymptotically N squared right. Now it makes a lot of sense since we are viewing exactly what it is printed here. We are printing a square of N by N stars. So yea it makes sense each one of them is a one single operation so we are doing a total of N squared operations. So altogether it is theta of N squared.

Runtime Analysis: Example 2 4.2

Ok let's take a look at another example very similar to the previous one. But implementation wise you can see that the only difference is that the range we are iterating in the inner loop is not one through N but one through I. Which is written in orange here. So again we are reading N from the user and then N times we are repeating a body of iterating from one to I and breaking the line. Printing I stars basically and breaking the line. So first iteration when I is one we are printing one star and breaking the line. Second iteration when I is two we are printing two stars and breaking the line and then three stars and breaking the line. Altogether we will print this kind of a triangle of stars. Once again let's try and analyze the running time of this algorithm so our T of N here. Now again I am ignoring the cout cin the return zero at the end I am focusing on the major part the major cost of our algorithm these two nested loops. Once again we are going to go from the inner parts to the outer parts adding them all together. In this case though we cannot say that each iteration we are doing the same exact thing. In the previous example each iteration we printed the same amount of stars. We did N operations each time so it was N times N. In this case one time we are printing one star then we are printing two stars and three stars. We can't multiply something by N. The number of operations we are doing here varies from iteration to iteration. We can say that each iteration we are doing the theta of I operations right. And then when we want to figure out what is the total number of operations we are doing here we are not multiplying I by something. We would need to add these values over the iterations the outer iterations we are doing here. So for I equals one we are doing one for I equals two we are doing two for I equals three we are doing three operations and so on up to N. In order to figure out the total number of operations here we would need to figure out what is the order of growth what is the asymptotic order of this sum here that goes from one through N. Now I guess you all know that arithmetic progression the sum of this arithmetic progression adds up to N times N plus one over two. Which is basically half N squared plus half N. And yea you can see that that is theta of N squared right. Again dropping the half N and dropping the half as the constant we are left with N squared. We can also formally prove by the definition of theta. But intuitively we can see that it is theta of N squared. Another way to view that is looking at the image that we are basically printing this triangle of stars here. Now we can see that this triangle here is half of the square right. It is half N squared so again intuitively N squared and half of N squared are only a constant apart. A factor constant apart and that would make them both be the same order of growth. The same squared order of growth. So that's a good thing to notice here. Another important thing I want you to remember memorize because we are going to use that thing a lot. And the visual image here of the triangle that is basically half of the square really demonstrates it that one plus two plus tree plus four plus so on up to N is the same order as N squared. It is half of the square. That's the result we are going to use quite a lot.