Week 7 Module 12 Recursions and Mathematical Induction

[PART 1]

Mathematical Induction Overview 1.2

Hi there, hope you're having a great day. Today we're going to talk about recursions; recursions are basically a very powerful problem solving technique that we're going to explore and see how it works. But before we do that, let's talk a bit about mathematical induction. I know the two are very related to one another; there is a very close relationship between mathematical induction and recursion. I'm not sure exactly how much you're familiar with mathematical induction so let me do a short overview of mathematical induction.

First, so methodical Induction is a technique to prove statements; it's a mathematical technique to prove statements. Basically it's used to prove universal statements that are of the form P of N is true for any natural number. So, for all of the natural numbers P of N is true. This is the form of statements we use mathematical induction proving technique. When we prove using mathematical inductions or there are basically two steps in our proof: there is the base case and there is the inductive step. In the base case, we typically prove that P of one is true, that our statement is true for the smallest natural number for one P of one is true, that's the thing we do in our base case.

In the inductive step, we show that P of N minus one implies P of N. So, if we know P of N minus one is true, we can also show that P of N is true too. And we show this statement P of N minus one implies P of N for any natural number greater than or equal to two. When we use these two steps to prove a universal statement as we said, basically we're not showing in a straightforward manner that P of N is true for all natural numbers but we say that after proving the base case and the inductive step we can conclude that P of N is true for all natural numbers. And let's get convinced that these two steps basically imply that P of N is true for any natural number.

So, let's see what we've proved and what conclusions can we make out of it. So, obviously when we prove that P of one is true we can say that P of one is true. But then we also prove that P of one implies P of two, now since we already know that one is true, this implication would lead that P of two is also true. And we've proved this P of N. minus one implies P of N, not only for one and two but for any natural number so we also prove that P of two implies P of three given that we already have concluded P of two is true. These two together would lead us to say that P of three is true and once again P of three implies P or four and P of three is true, so P of four is also true and this keeps on going. And that way we can just say that P of one, P of two, P of three, P of four, or basically any P. of N for all natural numbers are true. That's like the bases of mathematical induction.

Mathematical Induction Example 1.3

Okay. So, let's use this technique in order to prove the following claim. We'll show that for any natural number N: two N square plus five N minus six is greater or equal to zero. Let's go ahead and prove it.

Mathematical Induction Implementation 1.4

Okay. So we're going to prove that for any natural number N, two N square plus five N minus six is greater than equal to zero. Before we start just know that this statement here is of the form for any natural number and P of N is true, where this here are P of N: two N square plus five N minus six greater equal to zero, that's a P of N. Okay.

Let's start to disprove here; know there are a lot of ways to show this statement here. But we'll use induction, you know, in order to prove it. So, it's first state that we will prove using induction on N. And as we said there are two steps: there is the base and there is the inductive step.

So, let's start with the base. The base: we are supposed to show that P of one is true; basically, that this thing here is true for any cause one. And that's quite easy because for any cause one, we get that two times one square plus five times one minus six basically equals to one which is greater or equal to zero just as required. Now, let's try to show; let's try to do the inductive step.

So for the inductive step, we said that we're going to assume that P of N minus one is true, and under that assumption we'll show that P of N is also true. So we assume that P of N minus one is true, that means we should just take N minus one instead of N in our statement; two N minus one square plus five N minus one minus six is greater or equal to zero. So, that's our assumption. And we'll show that under that assumption two N square plus five N minus six is also greater or equal to zero. So, this here is our assumption and this here is what we need to show. And we'll use our assumption for, to show that thing.

So, okay. So let's see; two N square plus five N minus six. Let's see how we can make it using our assumption. So hopefully it would just be equal to two N minus one square plus five N minus one minus six. If these two expressions were equal, using our assumption we can just say that the right hand side is greater or equal to zero and therefore two N square plus five N minus six is greater or equal to zero. Unfortunately, they're not equal because of this thing here is, and I'll simplify it, it goes, gets to two N square plus N minus nine.

So, we can fix this expression for it to be equal and for that we would need, okay, so two N square, we have on both sides. On the left hand we have five N, on the right hand we have, and so we need to add four N. On the left hand we have negative six on the right hand we have a negative nine so let's just add three more, and now these two things are equal to one another. So, now that's very easy because, it's very useful actually, because we know that this thing here is greater or equal to zero; that's our inductive assumption. We also know that four N plus three, also, is greater or equal to zero because N is greater or equal to two, is a positive integer. So, if we multiplied by four and add three we would definitely get a positive number.
Therefore, these two add-ins, basically, are greater equal to zero plus zero, which is their zero. All together, we get that two N square five N minus six is greater or equal to zero. So, the inductive step showed that given our assumption: that's two N minus one squared plus five N minus one minus six is greater or equal to zero. Given that assumption, we can then say that two N square plus five N minus six, itself, is greater or equal to zero.

Strong Induction Overview 1.5

Let's talk about a variation, a more general form of induction, called strong induction. Strong induction is also a mathematical technique to prove universal statements of the same form, P of N is true for any natural number N, but then the technique is a bit different. This technique also has two steps: the base case and the inductive step.

Base case, same as the regular mathematical induction, we prove that P of one is true.
But the inductive step would be a bit different: in regular induction we just prove that a P of N minus one implies P of N, in the strong induction we prove that if P is true for all values more than N, that implies that P of N is true.

So, our assumption here is stronger instead of just assuming P of N minus one and using that a single assumption to prove that P of N is true, here we assume that P is true for all values smaller than N. Basically that P of N minus one is true, and P of N minus two is true, and P of N minus three is true and all values of P are true up to N; all of that together should imply that P of N is true. We have a much stronger assumption here to show that P of N is true; that's why it's named strong induction.

Let's see why this technique basically can be used in order to show that P of N is true for all natural numbers N. So, once again we prove that P of one is true, in our base case, that means that we can conclude that P of one and is true. And then we show that P of one implies P of two all values smaller than two are true, basically P of one, means P of two is true by showing this statement using the fact that P of one is already true, we can conclude that P. of two is true. Then we also show that if P of one and P of two are true, then P three is true. Our assumption, the stronger assumption for P of three was that all values smaller than three are true; P of one and two are true. That implies P of three is true and, in this point of time, we already know that P of one is true and that P of two is also true. And therefore, we can say that P of three is true. Moving on, we showed that P of one, P of two, and P of three, all of them together are true that implies P of four. Again, we already know that P of one and P of two and P of three are all true. Therefore, we can say safely that P of four is true. I believe you see that these things can go on, and that way we basically show that all P of N's are true for all natural number N's.

Strong Induction Example 1.6

Let's use the strong induction technique to prove some claim. So let's show that every natural number N can be written in the form of two to the power of I times J, where I is a non-negative integer and J is odd. Basically, we'll try to show that N can be written as a power of two times an odd number. For example, forty: we can write forty as two to the power of three, which is eight, times five. So, it's a power of two with the non-negative integer exponent times an odd number. That is true, we should show it not only for forty but for all natural number N. For example, for six: six can be written as two to the power of one times three, and seven, for example, can be written as two to the power of zero times seven. And we'll show that any number, any natural number, can be written in this form: some power of two with a non-negative exponent times an odd number. Let's go ahead and show it.

Strong Induction Example 1.7

Let's show that every natural number can be written as a power of two times an odd number. Once again you can see that this statement is of the form for any natural number some P of N is true; this is our P of N and know that N can be written in some form.

Let's go ahead and prove it. Actually before we start proving using strong induction, let's see that just the regular mathematical induction can't really help us here. So, in regular induction we would probably say that we assume that P of N minus one is true or basically N minus one can be written in the form of two to the power of some I times an odd number J, so let's assume we have that. And given that, we should show that N is also some power of two times an odd number. The only straightforward way I'm thinking of combining N with N minus one is just adding one to it, but then if N minus one is two to the I times J, if I'll add one to it, I won't get something of the form some power of I times J. But then if we know that this thing is true, not only for N minus one but for all other values of N. For example, I know that N is two times N by two. So, if I know that N by two, for example, is two times two to the power of I times J, then N would be two times two to the power of I times J, which would be some power of two times an odd number. So, using the assumption on a smaller value than N minus one, in this case N by two, seems to be helpful more than using the assumption of N minus one. That's why a strong induction here is better.

Let's use a strong induction to show exactly that. Okay. So, a proof will prove by a strong induction on N. And once again, we have two steps: the base and inductive step. So for the base, we should show that P of

one is true. So, for any cause one, we should show how we can represent one as some power of two times an odd number. So, that's very easy cause one is two to the zero times one. So, if we take our power of two to be zero, the exponent to be zero, and our odd number to be one we get that one is basically two to the zero times one, just as requested. And now, let's do the inductive step. In this case, we should assume that P of N is true for any value smaller than N. And using that assumption, we should show that P of N is true, so let's have this assumption ready. So we assume that every K smaller than N can be written in the form of K equals two to some power of I times J, where obviously I is a non-negative integer and J is odd. So, this is our assumption; it's cement. And now we show that N itself can also be written in this form.

Now we show that N can be written as two to the power of I times J, obviously other I's and J's here, with I and J as described. So, this is what we need to show. So, we'll use this assumption here in order to show that N can be represented this way and this assumption is for every value of K; for K equals N minus one, N minus two, N minus three, and so on. Also for K equals N by two, which would be the value of K that we would like to use this assumption on. Yeah, so let's do that. The only thing is that we need to take care, or in other words, not every N has an integer N by two; odd numbers you can't really divide by two and get another integer value after this division. So, let's separate to the two cases when our N is odd and when it's even and do all of that. So, we have two cases to take care of: the first is that N is odd; the second is that N is even. So, for the second case that's the main part of this proof is when we're going to use the inductive assumption on N by two.

The first case is quite easy when N is odd, obviously you can represent it as a power of two times an odd number, just take the power of two to be zero, so it would be one times N itself. So, in this case, if we take I to be zero and J to be N, we get that N equals two to the power of zero times N, which is obviously N, and I and J are as requested; so, we're fine.

In the case where N is even, now we're going to do what we basically said we'll do here in this case. Since N by two is less than N, by the inductive hypothesis, if we take K to be N by two, we get that N by two is then equal to two to the power of two to the power of some, let's say, I prime times J prime for our non-negative I prime and all J prime. And now we can, given that, we can present N in other requested form. So, since N is basically two times N by two, in the case where N is even, that is exactly the case. We get that N is two times two to the power of I prime times J prime, which is basically two to the power of I prime plus one times J prime. Therefore, if we take our I to be I prime plus one, the value that the inductive assumption gave us for the power of two for N by two plus one and J prime to be or our J to be the same as J prime. Then, we get a N written in the requested form. That concludes our proof here using strong induction.

[PART 2]

What is Recursion? 1.2

Recursion is basically a term used in several disciplines. In computer science, recursion is a problem solving technique where we solve a combination of problems. It is a very closely related to the mathematical induction that we just spoke about. The two are similar in some aspects; one of the main aspects is the fact that both kind of combine smaller instances for a larger one. So if we look at how we proved by induction. We used an assumption of P on smaller values in order to show that P is true on a larger value. Same thing kind of we're going to do here with recursions. So when we want to design a recursive algorithm, we're going to give you a very high level description of how we're going to design recursive algorithms; later on we'll see if you will demonstrate it using a few examples and you get the hang of it. But high level speaking, when we develop a recursive algorithm, just as induction, it has two steps: there is the base case/base step and inductive or recursive step.

Now when we make recursive algorithm when we use recursion, we use it as we said to solve problems. Not like induction where we use it to prove statements, in recursion with all problems. because in computer science we solve conventional problems were in mathematics we prove statements. So it would be very similar but then if in induction in the base case we proved the statement for the smallest possible value, in recursion we're going to solve the problem for the smallest possible input. So that's what we're going to do in the basis we're going to solve our problem for the smallest possible input. In the recursive step, we're going to assume that when we call the function on smaller inputs it does what it has to do it does its job. And using that assumption will try to figure out a way how to combine calls for smaller instances, to solve the problem on a given input. Let's see how we do that.

Print Ascending Problem 1.3

Okay. Let's try to use this technique in order to solve the following problem. Let's try to write a recursive implementation for the function of print ascending. Print ascending is supposed to get two arguments, a start and an end, both are integers and as the name kind of says, it should print the numbers from start to and you know ascending order. So we will assume that storage is less or equal to N, so there is like a range of numbers, a valid range of numbers to print and print ascending should then print the numbers from start to end.

For example if we'll call print ascending one and four, it would then print the values one, two, three, four in an ascending order. Once again, we would need to make a recursive implementation for this function, obviously an iterative one is very straightforward, but let's try to practice the recursion technique on this example here to see how we can then define recursive functions. So, as we said there are two steps in solving or creating/developing a recursive algorithm; first one is the base case where we are supposed to solve the problem for the smallest possible value/smallest possible input. So first we need to define what's the smallest possible input, or maybe before that we should even try to figure out how do we measure the size of the input. So we know that the inputs are start and end. These are the input themselves but the size of the input is a big difference so what should be the size of print ascending start and end.

For example, if we use print ascending three and six; what's the size of this instance? I would say that the size of print ascending three and six would be three, four, five, six, and equals four, the number of elements in the range from three to six. Print ascending seven nine, would be seven, eight, nine, the size would be three. Again, the number of elements in the range from seven to nine. Print ascending five and five, the size would be one. There is only one number in the range from five to five. So, let's define N as the input size to be the number of integers in the range from start to end. Saying that, it would be easier for us now to solve the problem for the smallest input; we know how to measure the size of the inputs. So, the smaller input would basically be the smallest valid range of numbers, basically a range with a single number in it. So if we want to solve the problem, solve print ascending, with the smallest values/the smallest size for start and end, if we try to identify this case it would be when start equals N. When start equals N, basically, we are in the case where there is a single element in the range from start to end.
So our solution for the smallest possible input would start, if this is the case of the smallest possible input. If start equals to N, then let's see what we should do in this case. So, when we call print ascending with a single element or a single integer range, the print ascending should just print the single element in that range, basically cout sort that is, right? So, this few lines of code basically solves the problem for the smallest input; quite straightforward, quite easy. By the way just the same as mathematical induction, the base case is a very easy step in the proof. It's very easy to show that P of one is true, most of the times. Same here, solving the problem for the smallest input is typically very easy; we just have to identify the smallest input and the solution is quite straightforward then. The recursive step that's a bit more tricky.

The recursive step, as we said, we first need to define the inductive assumption and then we would need to use it in order to solve the original of the given problem. So we said that, generally, the inductive

assumption is something of the form if we call the function on a smaller input it would do its job. for print ascending, if we try to make this assumption more specific here, I would say something like if we call printer sending on a smaller range it would print the numbers in that range in an ascending order. Instead of the word function, if we call the function I would say if we call print ascending instead of saying on a smaller input I would say on a smaller range. So if we call print ascending on a smaller range and instead of just saying it would do its job, I would say it would as print ascending should do print the numbers in the range in ascending order. So, this is a very powerful thing to assume; we're assuming basically that calling the function on a smaller input would do something, actually it would do what it has to do it would print the numbers in ascending order. Having this powerful assumption, let's see how we can use it in order to solve our original problem of printing the numbers the entire set of numbers from start to end. So let's try to use this assumption let's try to call print ascending on the smaller range.

So, our original range is, let's say, from start to end. Let's try to call this function on a smaller range, for example, not from start to end but from start to end minus one, that's a smaller range. What would happen if we call print ascending start, end minus one? Actually I don't know, but our assumption says that if we call print ascending on a smaller range it would print the numbers in an ascending order. So, since and end minus one is a smaller range, this call here should print the numbers from start to end minus one in ascending order. Surprisingly probably a lot this call would print the numbers from start to end minus one. That's most of the job we need to do here; we need to print the numbers from start to end in ascending order, and this single line of code already printed the line from start to end minus one in ascending order. The only thing left for us to do is just after that is print the value of end. These two lines combined together then would print the entire range from start to end. I know it is surprising; I know it is very not intuitive the next we are will try to show why it works. I think maybe before the second video just try it on your computer and make sure it does work and get surprised a bit.

Tracing printAsc with Runtime Stack 1.4

Okay. So logically I believe or I hope I convinced you that print ascending one and four works. Let's see it in a more formal way. Let's use our run time stack model, that basically follows the actual way that programs are executed inside our computer and let's see that using this model print ascending one and four, also prints one, two, three, four. So, at the beginning we have a single frame in our stack with a start equals one and end equals four and we are at the beginning of the function starting to execute it. So we are checking if start equals to end, in this case it does not, so we go to the else clause and we call print ascending with the start and end minus one. that would create a new frame in our stack with start equals one and end equals three, and after we create this frame we jump to the beginning of the function to store the new functions execution.

So once again we check if starts equals end, it does not so, we go to the else clause and once again we have another function call here. So we create a new frame in our stack, in this case start is one and end is two, after creating this frame we jump back to the beginning of the function to start the execution of this new call. So we ask whether the start equals end, it does not so we go to the else clause and unfortunately, there is another function call. That creates a new frame in our stack, in this case start and end are both equal to one; once again, we jump to the beginning of this function to execute this call. This time start is equal to end, so we see out start, we print one that's our output so far one. And this call basically ends, so this this frame is popped out and the one two frame turns active. We go back to where we came from we called, this call the just ended right here in the else clause. So we go one statement after that to see out end when we see out end in this active frame and is two, so two is printed. This call is also ended. So we pop out this frame go back to where we came from with one and three as the active frame; we see out three this call also ends, this frame pops out and we go back to where we came from with one and four as the active frame. We see end and in this case would print four and finally this last frame also pops out and this initial call of print ascending, one in four, basically ends. As you can see this call here just printed

one, two, three, and four. I know that this probably convinced you that this implementation works for this function, but then when we designed this algorithm we didn't think of the runtime stack evaluation that is going to occur. We thought of it kind of an inductive manner; how to combine a smaller instance of the problem to be a solution for original input.

Tracing printAsc 1.4

Let’s try to convince ourselves that this magical implementation here really works. So, I’ll do it in two separate ways. first you've executed it and probably saw that it basically works, but let me try to convince you a first more logically some kind of a bottom up argument, just like I show you that recursion basically prove that P of N is always true, so that would be one way for me to convince you that this implementation basically is fine. The second is a more formal tracing of using, I don’t know, runtime stack and stuff like that. but let's start with a more logical argument here to show that print ascending, for example, one four basically prints the values one, two, three, four. To do that, as I said, I'm going to make a bottom up argument. Let’s first convince ourselves that print ascending one and one basically prints the values in that range, basically prints one correctly. So if we’ll execute print ascending one one we’ll first evaluate the condition if start equals to end, it would be true. We would print start, in this case one and our function with the end after that. So, print ascending one one prints one which is exactly what it should do; remember that for the future, so print one one works fine.

Now let's try to trace the sending one and two. So when we look at the Boolean condition start equals end for one and two that would be false. We’ll go to the else clause in this case we will print the sending start and in minus one which would be one in one. We know that from the sending one one prints one we don't need to re execute it we already know that when we call print the sending one while we get one as an output. So this line would print one after that we would print the end which is two, both of them together would print one and two. So print ascending one two also works as we expected to; it would print one and two, so keep that in mind to.

Now let's see how/what print ascending one and three does. Once again start equals two and is false here, so we go to the else clause. We call print ascending with start and end minus one, in this case it's a one and two. We’ve already seen that print ascending one two prints one and two, so this line would print one and two. After that we are printing end for all so together we are printing one, two, three, four. Print ascending one and three prints the numbers one, two, three, just as required; keep that in mind so print ascending one and three prints one, two, three.

Let's finally see how print ascending one and four prints one, two, three, four. Once again, the Boolean condition is false; we go to the else clause. We call print ascending for one and three; we know that this call would print one, two, and three. After that we print end, which is four, all together it prints one, two, three, four. That shows that print ascending one to four prints the entire range of one two three four. I hope that this argument here convinced you not only for one and four that print ascending works properly, it would convince you that any range of size one works. Therefore, any range of size to works properly and equal for range of size three works properly. Therefore, any range of size four would work properly and so on. So, print ascending basically this implementation is basically fine. next video will talk about the runtime stack of this execution.

printAsc Alternative Implementations 1.6

Okay. So we are now convinced that our implementation really works. As we said we use the smaller instance of this problem, print ascending start and minus one. That would print in entire elements in the range from start to end minus one and followed by the end would basically print the entire range from

start to end. We can think of some other ways to use this assumption of calling print ascending on a smaller range and combining it in order to print from start to end. So given the range start and end taking the range start to end minus one is obviously a smaller range and calling this print ascending really worked as we did. But we can also, for example, call print ascending not from start to end minus one but from start plus one to end. This call is also a call for a function on an interval with a smaller range than the original start and end. By our assumption, this call should also do its job or in fact it should print the numbers from start plus one to end. that is great because all we have left to do here is just print start before we print from start plus one to end; so that's another version here. Another implementation, another recursive implementation, for the print ascending function.

We are first printing the value of start and then calling the recursive function to do the rest of the job: to print from start plus one to end. So if we have the range from start to end, we can reduce it by calling start and end minus one or by calling start plus one to end. Actually, we can think of another way or a lot of other ways, but in another very interesting way would say, instead of just reducing it by one maybe we can take half the size of the range. So let's print the first half of the numbers and after that we'll print the second half of the numbers, for that let's first, I don't know, calculate the middle point, which is basically the average of starts and ends. So, mid would be start plus end div to and then we'll make two recursive calls: one to print ascending from start up to the middle, and the second to print from the middle plus one, one after the middle, to end. The first call by our assumption since it's a smaller sized range; it would do the job it would print from start to the middle all of the numbers in ascending order. and the second call since it is also a call with a smaller size range, would bring the numbers from mid plus one up to end. Combining these two lines one after the other would print the entire set of elements from start to end in ascending order. That's kind of cool; you must say?

Print Ascending and Descending 1.7

We've seen that there are two steps we want to solve a computational problem using recursion. We've talked about the base case, the fact that we solve the problem for the smallest possible input and the recursive step where we combine solutions to smaller instances in order to solve the original of the given problem. Actually, we first define logically, it's not any line of code that we find our assumption; that calling the function on a smaller input does whatever it has to do. But given that in mind, we are trying to solve our problem for the given input. So we kind of make ourselves call the function on a smaller input and try to see what fixes we have to do in order to make it work for the given input. just the way we kind of proved by induction where we in the induction step when we want to prove that P of N is true we kind of make ourselves use the assumption that P of N minus one is true or all the values smaller than N are true. We kind of make yourself use this assumption in the proof that P of N is true, same thing in recursion we kind of makers of use or call the function smaller inputs and try to fix it, try to find a way to combine these calls for creating a solution for the given input.
Let's try to see some more examples. for example let's try to write a recursive implementation of the function print ascending and descending. once again this function could get two parameters, start and end; again we'll assume that start is less or equal to N. but this time the function given started and would print the numbers from start to end in ascending order, followed by the numbers in a descending order back to start. for example if we call print start and end with three and five, it would print three, four, five, four, three; it would printed in an ascending and descending order.

Let's try to create a recursive implementation of this problem here. Starting with the base case, starting to solve this problem on the smallest possible input. Once again, this input for this problem is measured by the number of elements in the range from start to end. Therefore, the smallest input possible is the range of size one; we can identify it by asking if start equals end, if it is equal to end, the solution for this very small instance, if we want to print ascending and descending a single element range, it is basically just printing that number; so cout start. Quite easy, as we said the base case are typically very straightforward.

But then let's see how we can solve all the other cases. First let's define the induction assumption. If we call in this case our function print ascending and descending on a smaller range, it would do the job. Basically it would print the numbers in the range in ascending and descending order so our assumption is if we call print ascending and descending on a smaller range, it would print the numbers as requested in an ascending and descending order. Let's try to use it in order to solve our problem for the range of start to end. So let's try to take a smaller range than start to end. For example, let's take start plus one to end; let's think, what the call print ascending and descending start plus one to end would do? by our assumption, we said that if we call this function on a smaller range, which obviously is a smaller range start plus one to end, it would print all the numbers in that range in ascending and descending order. So it would print from start plus one up to end then back to start plus one, that's basically most of what we need to do in order to print the entire range from start to end in an ascending and descending order. We just need to print start before this and print start after that so if we'll add two cout statements before this call and after this call, we would get a print of start and then a print descending the range start plus one to end then back to start plus one. Followed by a last print of start, which would all combine together, would print from start to end and back to start.

[PART 3]

Factorial 1.2

Let’s write another function using recursion. This time let’s write a recursive implementation for the function: factorial. This function, we already had an iterative version of it, this time we’ll write a recursive implementation. Factorial, as you probably remember, should get a positive integer as a parameter and then it should return the factorial of that value. For example, if we’ll call factorial of four, we’re expected to twenty four back. That’s because one times two times three times four is twenty four; four factorial is twenty four. Let’s take a look at factorial of N; so factorial of N is N times N minus one, times N minus two, and so on up to one. We can definitely have an iterative implementation for the factorial, as we already have, but when we’re thinking or trying to implement some sort of recursive implementation. We should try to see how we can combine or how we can define the factorial of N, with smaller instances of the factorial problem. In this case, we will probably see that one times two times up to N minus one, is basically N minus one factorial. So, factorial of N is basically N times the factorial of N minus one. So given that observation in mind, we can create a very easy recursive implementation. Let’s go ahead to the computer and do that.

Factorial Implementation 1.3

Let’s implement the factorial function. So before that I could be in factorial that the parameter here is n. and we should start with the base case; we should solve the problem for the smallest possible input. In the case of factorial the smallest possible input, we're assuming that N is a positive integer, so the smallest possible input is when N equals 1. Let’s identify this case when N equals 1, that the case we are trying to solve here. In this case the factorial of 1 is 1. So let's just return 1 as our output. Otherwise here we need to implement the recursive step. So our assumption would be that calling factorial on smaller value than N would return the factorial. in this case ,we already observe that pictorial of N is basically N times the factorial of N minus 1, so maybe we should just call factorial for N minus 1 and multiply that by N. so converting a single line of code but let me split it into a few, let's create res local variable. Let’s store in res, the result of calling factorial with N minus 1. that basically says that res now has factorial of N minus 1 and we should then just multiply res by N, that is the factorial of N. now res would hold factorial of N we could just return res for that matter. That is totally all the implementation for factorial.

Let's write a simple program that uses it. So I'll just declare factorial up here, and in my name let's cout factorial of four, basically prints 24. Let's execute it; yep, prints 24 which is exactly what we expected it to do. and when we're looking back at this implementation, we see that we used our assumption, we called our function on a smaller input, and it did what we asked it to do basically, to return the factorial of N minus 1 and then we update it we fixed this value in order to have the value of the factorial of N basically by each multiply by N. that was the whole idea here.

Are All of the Digits Even? 1.4

Let's write another recursive implementation. This time let's implement the function are all even; this function gets in the range of integers as a parameter and its logical size N. and it should return true if all of the elements in this array even, if not all of them are even it would just return false. For example, if we'll call this function with an array, let's say four, six, zero, and two and the logical size four, obviously it would return true because all of the elements here are even. If we'll call this function with the array four, seven, zero, and two with logical size four, it would return false because not of all of the elements here are even. For example, seven in this case is odd. Let's try to have a recursive implementation using a computer; let's do that now.

Are All of the Digits Even Implementation 1.5

Okay. So let's implement this function are all even so let's write first the prototype, Bool are all even, and it should get an array and it's logical size N. and let's start since we are looking for a recursive implementation, let's start with the base case. Base case we're basically trying to solve the problem for the smallest possible input. In this case the size of the problem would be the number of elements in the array. So the more the elements are in the array the bigger the problem is, the less elements in the array the smaller the problem is the smallest. The problem can be is when there is a single element in the rate. So let's check for this case: if we have, and N basically tells us the number of elements in the array so if N equals one that identifies the smallest possible input. Now, let's have the solution for this case; let's return the right value in case that it is a single element array. So in this case we just need to check whether this single element is even or not, we can use an IF statement modding it by two and checking whether it's zero and returning true or false. We can do the same in return statement in the return line, so we can return the value of A R R zero mod two equals zero. So if when we're dividing the single element by two and the remainder is zero, then this expression here would be true and we would pass; this true is a return value. But if the remainder when we're dividing A R R zero by two would not be zero, therefore it would be one, this expression would be false and then we would return the value false. So, that these two lines of code here basically solve correctly the base case, the scenario when we have the smallest possible input. the recursive step we are supposed to give the right value, the right output for all other inputs for basically arrays that are not single element arrays. and we should use our induction assumption that make it explicit, our assumption would be if we call this function, if we call are all even, this function, with a smaller instance basically within an array of less elements in it, then it would decide correctly it will return correctly whether all of the elements in this smaller array are all even or not. In order to create a smaller instance, we can just pass logically the first and minus one elements.

So let's create a temp variable here: result. And let's set result to be the result of calling this function with the same starting address but logically only N minus one elements. That is obviously a smaller instance for this problem; it is an N minus one array. And given this assumption here calling the function on a smaller instance would return the appropriate value so res would be true if all of the first and N minus one elements are all even, and it would be false otherwise. So res almost has all of the values we need here; all of the result we need here. We just need to check the last element; we need to check the last element actually only in case that res is true.

So, let's check if res is true then we're not sure yet whether our result should be true or false; we need to figure that out. But if resists is not true, basically if res is false, if not all of the elements in this first N minus one elements of array are all even that means that obviously not all of the elements in the entire array are all even. Therefore, we should return false. So if res itself is false, if this smaller instance is already false, obviously our value should be false. Only the case where the first N minus one elements are true is the case where we need to figure out whether or were entire array is also filled up with even elements. So in this case we should just check the last element and see whether it's even or not. We should do something very similar to what we've done for the single element array, but instead of going for the first element we'll check the last element. So, we'll check whether the last element A R R in the minus one index mod two equals zero; if the remainder is zero it means the last element is even, and since all the rest of the elements are also evens we would return that true. But if the last element is not even, even though all the rest of the elements are, we would still would return the false.

So, it seems like this implementation here is okay. Let's write a main program to test it. First, let's declare this function and our main, Let's keep it simple; let's create an array in A R R, int A R R size 4, let's put it in a few elements, two, twenty, forty six, fourteen. And then let's just call this function. So if are all even for this array and four equals true then let's print are all even. Otherwise, let's print not all are even. We're expecting, obviously, that all of them would be even in this case. Let's try to execute it; yeah, all are even.

Let's change it out; you know twenty three see that it still works. Not all are even; okay, good. Seems like our implementation is fine. Let's take a final look here at this implementation, as you can see we have the base case where we solve the problem for the smallest possible input, for a single sized array. and then we have our inductive, or recursive step, where we used our assumption that calling this function on the smaller instance would do the job would return true, if the smaller instance contains only even numbers. And we combine that for our original input just by you checking for the last element whether it's even or not. And that basically is our recursive implementation for this function.