CS125 : Introduction to Computer Science

Lecture Notes #21
Introduction to Algorithm Design and Recursion

Lecture 21 : Introduction to Algorithm Design and Recursion

Our focus now turns to *algorithm* design. An *algorithm* is a computational process for solving a problem. This definition has three important parts:

- Computational – the process must be something a computer could work through in a step-by-step manner, and all the necessary steps must be specified. Some things have no computational solutions. For example, even though *you* can detect that your program has an infinite loop, there is no generalized way for the computer to do this – meaning, yes, in certain special situations perhaps the computer *can* recognize there is an infinite loop...but usually those special situations don't apply and usually the computer cannot recognize an infinite loop *in general.*
- Solving – the process must actually produce the desired solution. usually what is desired is an exact solution to the problem; sometimes all that is wanted is a solution that is "close enough" to optimal, where "close enough" has some formal definition.
- Problem – the process must solve the general problem, not just a specific instance of it. If you want to return the sum of five numbers, just returning the integer literal "2" is not an accurate algorithm, even though that solution *does* work in some cases.

Algorithm design

Our main focus for much of the next few weeks will be on algorithm *design.* That is, we will be concerned with the creation of new algorithms first and foremost, as opposed to worrying about how to make the algorithm run as fast as possible, in a particular language, on a particular machine. Algorithms are mathematical, computational things. An algorithm is a process, a description of how to solve a problem. We can discuss whether the algorithm is sufficiently detailed to be translated into code, and we can discuss whether the algorithm solves our problem or not – all without needing to discuss actual code.

Recursion

Recursion is the language of algorithm design. It's not a new power in addition to what you have, but rather, a new way of looking at a problem, one that allows you to often focus on the major portion of the solution without worrying about every start-to-finish detail. Recursion is when a solution formula, or a method, is defined in terms of itself.

For example (the example works better when it's physically being given during lecture, but what the hey), consider crossing a room to knock on a door. If you are right next to the door, you can just knock on that door. If you are one step away, you take one step toward the door, and then knock. If you are two steps from the door, you take two steps and then knock. And so on. In general, if you are `n` steps from the door, take `n` steps and then knock on the door. This is the loop solution, and is one you could already write:

```
// start out n steps from door
void EventuallyKnock(int n)
{
   while (n > 0)
   {
      TakeOneStepTowardDoor();
      n = n - 1;
   }
   KnockOnDoor();
}
```

Notice that this required visualizing the entire solution at once, from start to finish. Every detail is accounted for by that loop. (Notice also that it makes no sense for `n` to be negative, so we'll assume for the rest of this discussion that `n` will always be non-negative.)

Now, instead, imagine that we handle this situation by referring to "simpler problems". As before, if you are right next to the door, just knock. Call that the "zero steps away" solution.

```
   # of steps away                        solution to that problem
   ---------------                        ------------------------
        0                                       1) knock on door
```

Now, let's cover up the solution to the "zero steps away" problem. This way, we'll know we *have* a solution; we just won't know what it is.

```
   # of steps away                        solution to that problem
   ---------------                        ------------------------
        0                                      [don't look]
```

Now, if you are one step away, well, if you took just one step toward the door, you'd be zero steps away and could run the "zero steps away solution". What is that solution? Who cares? Certainly, if we look back at the details we can read what the "zero steps away" solution was ("knock on door") but we don't have to bother. We know that we had previously created a "zero steps away" solution, and let's just run that, whatever it is. So, our "one step away" solution is to take one step, and then run the "zero steps away" solution.

```
# of steps away                        solution to that problem
---------------                        ------------------------
     0                                      [don't look]

     1                                      1) take a step
                                            2) run 0-steps-away solution
```

Now, we can cover up the “one step away” solution as well.

```
# of steps away                        solution to that problem
---------------                        ------------------------
     0                                      [don't look]

     1                                      [don't look]
```

So, what happens when we are two steps away? Well, now there’s a bit more work to do, but we don’t really need to think about all of it. All we need to do is realize that if we take just one step, we are now only one step away. And we have a solution for that problem already: the “one step away” solution. Let’s not worry what the details are; it’s enough to just say, “we discovered the one-step-away solution earlier, so just run that”. So, the “two steps away” solution is to take one step, and then run the “one step away” solution, whatever it might be.

```
# of steps away                        solution to that problem
---------------                        ------------------------
     0                                      [don't look]

     1                                      [don't look]

     2                                      1) take a step
                                            2) run 1-step-away solution
```

We can then cover this solution up as well:

```
# of steps away                        solution to that problem
---------------                        ------------------------
     0                                      [don't look]

     1                                      [don't look]

     2                                      [don't look]
```

and then, in trying to solve the “three steps away” problem, we can note that we have a “two steps away” solution, so we can simply take a step, and then run the “two steps away” solution, whatever it might be.

# of steps away	solution to that problem
0	[don't look]
1	[don't look]
2	[don't look]
3	1) take a step 2) run 2-steps-away solution

In general, we have the following pattern:

# of steps away	solution to that problem
0	1) knock on door
1	1) take a step 2) run 0-steps-away solution
2	1) take a step 2) run 1-step-away solution
3	1) take a step 2) run 2-steps-away solution
. . .	. . .
n	1) take a step 2) run (n-1)-steps-away solution

The general rule is that if we are `n` steps away, the solution to our problem is to first take one step, leaving us `(n-1)` steps away. Then, we can finish solving the problem, by running the solution appropriate for being `(n-1)` steps away, since that is now our situation, thanks to us having just taken one step. In every situation where we're a positive number of steps away from the door, we can solve that problem by taking one step – bringing us one step closer to the door – and then treating our new situation as a new problem to be solved, and applying our rule all over again.

Thus to write the solution to this problem, we don't *have* to have our code consider every detail from the start to the finish of the problem. We only need the first step and the last step – that is, we need to know how to finish the solution once we are zero steps away, and we need to know how to start the solution, by reducing the problem to a smaller *subproblem* – the same problem, just on fewer data values or smaller data values) – and then running the solution to that subproblem.

We could write things out formally as follows:

```
                        ------
                        |    TakeOneStepTowardDoor();        if n > 0
                        |    EventuallyKnock(n-1);
EventuallyKnock(n)    --|
                        |    KnockOnDoor(),                  if n == 0
                        |-----
```

The first case is called the *recursive* case – the general case we use to solve most problems. Since the problem can be solved in *exactly the same way* for almost every value of `n` – take one step, and then run the solution to the next smaller problem – this recursive case is what we run for almost every value of `n`. The recursive case is the case that is defined in terms of a solution to a subproblem. But we can't define every problem in terms of the solution to other problems; we need to eventually stop or we'd forever be solving!! And so the second case is the *base case.* This is the case whose answer is *not* defined in terms of a subproblem (i.e. whose answer is *not* defined in terms of the same algorithm run on a a smaller data quantity or smaller data value). `EventuallyKnock(0)` is not defined in terms of other solutions to `EventuallyKnock`. Whereas for any `n` greater than 0, we define that solution partly in terms another solution to `EventuallyKnock`.

Describing things recursively provides the advantage of not having to describe every single detail of the solution. We only need to explain two important things : how to reduce the current problem to a smaller one (the recursive case), and how to solve the smallest possible problem (the base case). All the details of figuring each solution out in turn is left to the definition itself. If you want to know the answer to `EventuallyKnocks(3)`, then by definition, you need to know the solution to `EventuallyKnocks(2)`, and to know that, you by definition need to know the solution to `EventuallyKnocks(1)`, and to solve that, you by definition need to know the solution to `EventuallyKnocks(0)`, which our definition already provides us.

We could then take that formal description above and create a recursive method from it:

```
void EventuallyKnock(int n)
{
   if (n > 0)
   {
      TakeOneStepTowardDoor();
      EventuallyKnock(n-1);
   }
   else
      KnockOnDoor();
}
```

When we pass this method a positive value as an argument, the method will hit the recursive case, and will *call itself* with a new argument. When a method calls itself, it is a recursive method. (For example, if we sent in `10` as an argument, because we want to solve the problem of being `10` steps away, we will take a step, and then make the call `EventuallyKnock(9);`, which will pass `9` as an argument to the same method, and which will solve the problem of being nine steps away.

Not having to provide every little detail to handle the entire problem from start to finish in one method call has three advantages – first, it is often easier to come up with solutions when you don't have to think all the way through every detail. For simple problems, this might not be the case – for example, the loop version (called an *iterative* version) for the above problem probably seems a bit clearer to you right now. But for more complex problems, it might not be at all clear how to solve it with a loop, while the recursive solution in the same circumstances could be quite obvious. Second, it makes solutions you do come up with easier to understand. Again, since the above problem is very simple and since you are just learning recursion, this might not seem true – but for more complex problems, describing just the first and last step makes for a more concise, clear description than trying to trace out the calculation from start to finish. And third, those solutions are also easier to prove correct, due to the proof technique of mathematical induction, which you will learn in CS173.

(Very briefly – proving something via the use of mathematical induction involves proving both a base case about that something, and proving a recursive case about it. For example, if you wanted to prove that the sum of all positive integers from `1` through `n` was given by the formula `(n * (n+1))/2`, you would first prove that the formula worked for `n==1` (the base case), and then you would prove that *if* the formula was true for some value `n==k`, then it must also be true for the value `n==k+1`. Once you've proved that second statement, well, then since you've already proved the formula is true for `n==1`, you know it's true for `n==2`. And then since it's true for `n==2`, it must also be true for `n==3`. And then since it's true for `n==3`, it must also be true for `n==4`. And so on. That is the idea of proof by mathematical induction – as you can see, there's a very close relationship between that proof technique, and the idea of recursion we are discussing. And that is why it is such a nice proof technique for proving that a recursive algorithm is correct. But as we said above, you will learn more about this in CS173.)

The last problem was somewhat abstract. Let's try again, but this time, with an actual real, mathematically-based problem. You might recall the definition of "factorial"; "n factorial" (notated `n!`) is the product of all the integers from `1` through `n`, inclusive:

```
          // product of all numbers from 1 through n.
          n! = n * (n-1) * (n-2) * (n-3) * ... * 2 * 1
```

We can write an iterative method (i.e. a method using loops) to calculate `n!`:

```
// Iterative version
public static int fac(int n)
{
   int product = 1;
   while (n > 0)
   {
      product = product * i;
      n = n - 1;
   }
   return product;
}
```

Notice that, in the above code, if we had sent `0` in as the argument, we would get `1` back as the return value. This is what we want to have happen; mathematically, `0!` is defined to be `1`. We do not define factorial for negative values, and thus we will assume that there is always a non-negative value passed as an argument to the parameter `n`.

Now, in this situation, we had the "..." in our definition of factorial above, to indicate "follow the same pattern", and we are assuming that anyone who looks at that definition can see that we are subtracting a value one larger from `n` each time, and so the missing terms should continue in that same pattern `n-4`, `n-5`, `n-6`, and so on.

We could also define `fac(n)` a different way, in terms of itself. If we are trying to calculate `n!`, then consider all possible subproblems: `1!`, `2!`, etc., all the way up to `(n-2)!` and `(n-1)!`. Which of these might help us in calculating `n!`? Well, if we have `(n-1)!`. we only need to multiply it by `n` to get the final result! That is to say:

```
         // product of all numbers from 1 through n.
         n! = n * (n-1) * (n-2) * (n-3) * ... * 2 * 1
                 |____________________________________|
                    this part is (n-1)!
```

giving us the shortened formula:

```
         n! = n * (n-1)!
```

Of course, we need a base case as well. Let's take `n==0` as our base case, since mathematically, we define `0!` to be equal to `1`, and since we don't define factorial for anything less than `0` anyway. That gives us the following algorithm for calculating factorial:

```
                 |
n!          =    | n * (n-1)!         if n > 0
                 |
 (n>=0)          | 1                  if n == 0
```

For example, you would compute `5!` as follows:

```
   5! = 5 * 4!
           /|\
            |
            4! = 4 * 3!
                    /|\
                     |
                     3! = 3 * 2!
                             /|\
                              |
                              2! = 2 * 1!
                                      /|\
                                       |
                                       1! = 1 * 0!
                                               /|\
                                                |
                                                0! = 1
```

Then once you know `0!` is `1`, you substitute `1` for `0!` in the expression `1 * 0!`...and that gives you `1`. So now that you know `1!` is `1`, you substitute `1` for `1!` in the expression `2 * 1!`...and that gives you `2`. And now that you know that `2!` is `2`, you substitute `2` for `2!` in the expression `3 * 2!`...and you get `6`. And now that you know that `3!` is `6`, you substitute `6` for `3!` in the expression `4 * 3!`...and you get `24`. And now that you know `4!` is `24`, you substitute `24` for `4!` in the expression `5 * 4!`...and you get `120`. And now you know that `5!` is `120`.

You can write the above algorithm as a recursive method `fac`. If you are trying to compute `n!` via the method call `fac(n)`, you would compute `(n-1)!` via the method call `fac(n-1)`, leading to the following code:

```
// reminder: we are assuming n is non-negative
public static int fac(int n)
{
   if (n > 0)
      return n * fac(n - 1);
   else  // n == 0
      return 1;
}
```

What we are going to do next is go step by step through the calculation of 4! using the recursive method we just wrote. We start with a `main()` method that makes a call to `fac(4)`:

```
---------------------------------------------------------------------
| main
|   int x;
|   x = fac(4);     // we call the fac method
|
|--------------------------------------------------------------------
```

Now, as with any method, when we call that method, we set up a “notecard” (some area of memory inside the machine) to store the parameters and local variables for that method call. It’s no different with recursive methods:

```
---------------------------------------------------------------------
| main
|   int x;
|   x = fac(4);     // we call the fac method, and in doing so,
|                   //  set up the "fac" notecard below:
|--------------------------------------------------------------------
| fac     (n: 4)   // this is the notecard for the first call to
|  (#1)            // fac; 4 was passed to this method as an
|                  // argument, and thus is stored in the parameter "n"
|
|---------------------------------------------------------------------
```

Now that we’ve set up the `fac` notecard, we run the `fac` method on the data in the `fac` notecard. Since `n` is not zero, we will take the recursive case, which requires we calculate $(n-1)!$, i.e. 3!.

```
---------------------------------------------------------------------
| main
|   int x;
|   x = fac(4);     // we call the fac method, and in doing so,
|                   //  set up the "fac" notecard below:
|--------------------------------------------------------------------
| fac     (n: 4)
|  (#1)
|
|  need to calculate: fac(3), so we can use its value to
|                              complete the multiplication in the
|                              expression
|
|                                  n * fac(n - 1)
|                                        i.e.
|                                   4 * fac(3)
|------------------------------------------------------------------
```

So, we make another call to our `fac` method. That is, from the call `fac(4)`, we will make the call `fac(3)`, and therefore we will start a second `fac` notecard, one where `n` is `3`. This is perfectly acceptable! In past method examples, when you made a method call, the method you were leaving and the method you were starting were two different methods; in this case, they are the same method. This does not matter. You can indeed have the same set of instructions run on two different notecards. Since the notecards have different values for `n`, you'll get different results.

If you're having trouble understanding this, imagine if I write the expression:

```
(a + b) * (c - d)
```

on two different pieces of paper. I could give one of those pieces of paper to one student in CS125, and I could give the other piece of paper to a different student in CS125. So, both students have the same expression in front of them. Now, if I tell the first student "`a` is `4`, `b` is `7`, `c` is `2`, and `d` is `3`", the student could plug those values into the expression, and get the result, `-11`. I could then tell the second student, "`a` is `5`, `b` is `1`, `c` is `9`, and `d` is `6`", and that student could plug those values into the expression, and get the result `18`. Both students can evaluate the expression; they have different values for `a`, `b`, `c`, and `d`, so they get different results – but you can indeed have two different students evaluate the same expression. Similarly, you can have two different notecards both setup to run the same method. If the values stored on the first notecard are different than the equivalent values stored on the second nodecard (such as `n` being `4` versus `n` being `3`), the method calls will accomplish different work...but there's no reason you can't have two different notecards, both of them being scratch areas for the same set of instructions, just like two different students can both work on evaluating their own copy of the same expression.

```
-------------------------------------------------------------------
| main
|   int x;
|   x = fac(4);
|__________________________________________________________________
| fac     (n: 4)
|  (#1)
|
|  need to calculate: fac(3), in order to calculate 4 * fac(3)
|
|___________________________________________________________________
| fac     (n: 3)
|  (#2)
|
|  need to calculate: fac(2), so we can use its value to
|                              complete the multiplication in the
|                              expression
|
|                                  n * fac(n - 1)
|                                        i.e.
|                                   3 * fac(2)
|___________________________________________________________________
```

As that last picture showed, within the `fac(3)` call, we will find that we need a `fac(2)` call, and so we will make that call:

```
-------------------------------------------------------------------
| main
|   int x;
|   x = fac(4);
|__________________________________________________________________
| fac     (n: 4)
|  (#1)
|
|  need to calculate: fac(3), in order to calculate 4 * fac(3)
|
|___________________________________________________________________
| fac     (n: 3)
|  (#2)
|
|  need to calculate: fac(2), in order to calculate 3 * fac(2)
|
|___________________________________________________________________
| fac     (n: 2)
|  (#3)
|
|  need to calculate: fac(1), so we can use its value to
|                              complete the multiplication in the
|                              expression
|
|                                  n * fac(n - 1)
|                                        i.e.
|                                   2 * fac(1)
|___________________________________________________________________
```

From within the `fac(2)` call, we will need a `fac(1)` call, so we make that call:

```
-----------------------------------------------------------------
| main
|   int x;
|   x = fac(4);
|________________________________________________________________
| fac     (n: 4)
|  (#1)
|
|  need to calculate: fac(3), in order to calculate 4 * fac(3)
|
|_________________________________________________________________
| fac     (n: 3)
|  (#2)
|
|  need to calculate: fac(2), in order to calculate 3 * fac(2)
|
|_________________________________________________________________
| fac     (n: 2)
|  (#3)
|
|  need to calculate: fac(1), in order to calculate 2 * fac(1)
|
|_________________________________________________________________
| fac     (n: 1)
|  (#4)
|
|  need to calculate: fac(0), so we can use its value to
|                              complete the multiplication in the
|                              expression
|
|                                  n * fac(n - 1)
|                                        i.e.
|                                   1 * fac(0)
|_________________________________________________________________
```

From within the `fac(1)` call, we will need a `fac(0)` call, so we make that call:

```
-------------------------------------------------------------------
| main
|   int x;
|   x = fac(4);
|__________________________________________________________________
| fac     (n: 4)
|  (#1)
|
|  need to calculate: fac(3), in order to calculate 4 * fac(3)
|
|___________________________________________________________________
| fac     (n: 3)
|  (#2)
|
|  need to calculate: fac(2), in order to calculate 3 * fac(2)
|
|___________________________________________________________________
| fac     (n: 2)
|  (#3)
|
|  need to calculate: fac(1), in order to calculate 2 * fac(1)
|
|___________________________________________________________________
| fac     (n: 1)
|  (#4)
|
|  need to calculate: fac(0), in order to calculate 1 * fac(0)
|
|___________________________________________________________________
| fac     (n: 0)
|  (#5)
|
|  base case!  (we have not returned yet)
|___________________________________________________________________
```

We make and pause four separate calls to `fac`, finally making our fifth call which is our base case.

Then, since the base case doesn't need to make any further recursive calls, we can start returning from there. The base case itself returns `1`:

```
-------------------------------------------------------------------
| main
|   int x;
|   x = fac(4);
|__________________________________________________________________
| fac     (n: 4)
|  (#1)
|
|  need: fac(3), in order to calculate 4 * fac(3)
|
|__________________________________________________________________
| fac     (n: 3)
|  (#2)
|
|  need: fac(2), in order to calculate 3 * fac(2)
|
|__________________________________________________________________
| fac     (n: 2)
|  (#3)
|
|  need: fac(1), in order to calculate 2 * fac(1)
|
|__________________________________________________________________
| fac     (n: 1)
|  (#4)
|
|  need: fac(0), in order to calculate 1 * fac(0)
|
|__________________________________________________________________
| fac     (n: 0)
|  (#5)
|
|  base case!    ------> returns 1
|__________________________________________________________________
```

Therefore, the value `1` is returned back to the previous call and is substituted for “`fac(1)`”, the method call expression that triggered the base case to be started:

```
-----------------------------------------------------------------
| main
|   int x;
|   x = fac(4);
|________________________________________________________________
| fac     (n: 4)
|  (#1)
|
|  need: fac(3), in order to calculate 4 * fac(3)
|
|_________________________________________________________________
| fac     (n: 3)
|  (#2)
|
|  need: fac(2), in order to calculate 3 * fac(2)
|
|_________________________________________________________________
| fac     (n: 2)
|  (#3)
|
|  need: fac(1), in order to calculate 2 * fac(1)
|
|_________________________________________________________________
| fac     (n: 1)
|  (#4)
|
|  need: fac(0), in order to calculate 1 * fac(0)
|                                         ------
|_________________________________________ /|\ __
                                            |
 base case returned 1; that is,             |
   the method call fac(0) returned 1...so now this expression
                                         is replaced by the value 1
```

Now that we have a value for `fac(0)`, we can finally complete the multiplication, since now we have both operands:

```
-------------------------------------------------------------------
| main
|   int x;
|   x = fac(4);
|__________________________________________________________________
| fac     (n: 4)
|  (#1)
|
|  need: fac(3), in order to calculate 4 * fac(3)
|
|___________________________________________________________________
| fac     (n: 3)
|  (#2)
|
|  need: fac(2), in order to calculate 3 * fac(2)
|
|___________________________________________________________________
| fac     (n: 2)
|  (#3)
|
|  need: fac(1), in order to calculate 2 * fac(1)
|
|___________________________________________________________________
| fac     (n: 1)
|  (#4)
|
|  this method call is supposed        1 * 1
|  to return 1 * fac(0) which          ------ <---- so now this expression
|  is 1 * 1 which is 1                                evaluates to 1
|______________________________________________________
```

Finally, we are supposed to return the result of that multiplication:

```
-------------------------------------------------------------------
| main
|   int x;
|   x = fac(4);
|__________________________________________________________________
| fac     (n: 4)
|  (#1)
|
|  need: fac(3), in order to calculate 4 * fac(3)
|
|___________________________________________________________________
| fac     (n: 3)
|  (#2)
|
|  need: fac(2), in order to calculate 3 * fac(2)
|
|___________________________________________________________________
| fac     (n: 2)
|  (#3)
|
|  need: fac(1), in order to calculate 2 * fac(1)
|
|___________________________________________________________________
| fac     (n: 1)
|  (#4)
|
|  this method call is supposed          1
|  to return 1 * fac(0) which          ------ <---- so now this value
|  is 1 * 1 which is 1                                is returned
|______________________________________________________
```

When the fourth `fac` call returns `1`, then the program returns back to the third `fac` call, and now that the `fac(1)` call is over, it is replaced by its return value, which is `1`:

```
-----------------------------------------------------------------
| main
|   int x;
|   x = fac(4);
|_________________________________________________________________
| fac     (n: 4)
|  (#1)
|
|  need: fac(3), in order to calculate 4 * fac(3)
|
|__________________________________________________________________
| fac     (n: 3)
|  (#2)
|
|  need: fac(2), in order to calculate 3 * fac(2)
|
|__________________________________________________________________
| fac     (n: 2)
|  (#3)
|
|  need: fac(1), in order to calculate 2 * fac(1)
|                                          ------
|__________________________________________ /|\ __
                                             |
                                             |
   the method call fac(1) returned 1...so now this expression
                                         is replaced by the value 1
```

And then this call works as the previous one did. First, the multiplication is completed:

```
-------------------------------------------------------------------
| main
|   int x;
|   x = fac(4);
|__________________________________________________________________
| fac     (n: 4)
|  (#1)
|
|  need: fac(3), in order to calculate 4 * fac(3)
|
|___________________________________________________________________
| fac     (n: 3)
|  (#2)
|
|  need: fac(2), in order to calculate 3 * fac(2)
|
|___________________________________________________________________
| fac     (n: 2)
|  (#3)
|
|  this method call is supposed        2 * 1
|  to return 2 * fac(1) which          ------ <---- so now this expression
|  is 2 * 1 which is 2                                evaluates to 2
|______________________________________________________
```

And then we were supposed to return the result of that multiplication:

```
-------------------------------------------------------------------
| main
|   int x;
|   x = fac(4);
|__________________________________________________________________
| fac     (n: 4)
|  (#1)
|
|  need: fac(3), in order to calculate 4 * fac(3)
|
|___________________________________________________________________
| fac     (n: 3)
|  (#2)
|
|  need: fac(2), in order to calculate 3 * fac(2)
|
|___________________________________________________________________
| fac     (n: 2)
|  (#3)
|
|  this method call is supposed          2
|  to return 2 * fac(1) which          ------ <---- so now this value
|  is 2 * 1 which is 2                                is returned
|_____________________________________________________
```

When the third fac call returns 2, then the program returns back to the second fac call, and now that the fac(2) call is over, it is replaced by its return value, which is 2:

```
------------------------------------------------------------------
| main
|   int x;
|   x = fac(4);
|_________________________________________________________________
| fac     (n: 4)
|  (#1)
|
|  need: fac(3), in order to calculate 4 * fac(3)
|
|__________________________________________________________________
| fac     (n: 3)
|  (#2)
|
|  need: fac(2), in order to calculate 3 * fac(2)
|                                          ------
|_________________________________________ /|\ __
                                            |
                                            |
   the method call fac(2) returned 2...so now this expression
                                         is replaced by the value 2
```

And then this call works as the previous two did. First, the multiplication is completed:

```
------------------------------------------------------------------
| main
|   int x;
|   x = fac(4);
|_________________________________________________________________
| fac     (n: 4)
|  (#1)
|
|  need: fac(3), in order to calculate 4 * fac(3)
|
|__________________________________________________________________
| fac     (n: 3)
|  (#2)
|
|  this method call is supposed        3 * 2
|  to return 3 * fac(2) which          ------ <---- so now this expression
|  is 3 * 2 which is 6                                evaluates to 6
|_____________________________________________________
```

And then we were supposed to return the result of that multiplication:

```
-------------------------------------------------------------------
| main
|   int x;
|   x = fac(4);
|__________________________________________________________________
| fac     (n: 4)
|  (#1)
|
|  need: fac(3), in order to calculate 4 * fac(3)
|
|___________________________________________________________________
| fac     (n: 3)
|  (#2)
|
|  this method call is supposed          6
|  to return 3 * fac(2) which          ------ <---- so now this value
|  is 3 * 2 which is 6                                is returned
|_____________________________________________________
```

When the second `fac` call returns `6`, then the program returns back to the first `fac` call, and now that the `fac(3)` call is over, it is replaced by its return value, which is `6`:

```
-------------------------------------------------------------------
| main
|   int x;
|   x = fac(4);
|__________________________________________________________________
| fac     (n: 4)
|  (#1)
|
|  need: fac(3), in order to calculate 4 * fac(3)
|                                         ------
|__________________________________________ /|\ __
                                             |
                                             |
   the method call fac(3) returned 6...so now this expression
                                         is replaced by the value 6
```

And then this call works as the previous three did. First, the multiplication is completed:

```
--------------------------------------------------------------------
| main
|   int x;
|   x = fac(4);
|___________________________________________________________________
| fac     (n: 4)
|  (#1)
|
|  this method call is supposed        4 * 6
|  to return 4 * fac(3) which          ------ <---- so now this expression
|  is 4 * 6 which is 24                                 evaluates to 24
|_______________________________________________________
```

And then we were supposed to return the result of that multiplication:

```
--------------------------------------------------------------------
| main
|   int x;
|   x = fac(4);
|___________________________________________________________________
| fac     (n: 4)
|  (#1)
|
|  this method call is supposed          24
|  to return 4 * fac(3) which          ------ <---- so now this value
|  is 4 * 6 which is 24                                 is returned
|_______________________________________________________
```

When the first `fac` call returns `24`, then the program returns back to `main()`, and now that the `fac(4)` call is over, it is replaced by its return value, which is `24`, which then gets written into the variable `x` by the assignment statement.

```
----------------------------------------------------------
| main
|   int x;
|   x = fac(4);
|       ---------
|________ /|\ __________________________________________
           |
           |
    this expression is replaced by 24, since the
     method call fac(4) has returned 24
```

That is how recursion works on the actual machine – as with any other method call, when you make a recursive method call, your calculation cannot proceed until that method call is finished. This is why the first `fac` call (the one where `n==4`) sits paused for a long time – until the `fac(3)` call it makes, finally returns.