CS125 : Introduction to Computer Science

Lecture Notes #22
Subproblems

Lecture 22 : Subproblems

To write a recursive function `f(x)`, you need to express `f(x)` in terms of the solution to one or more *subproblems* – i.e. in terms of the solution to one or more similar, but smaller problems. For example:

- If your parameters are integers, you might obtain a subproblem by reducing one or more of the integers, and running the same code with *those* values for the parameters. For example, in the factorial method from the last lecture, we took our one integer parameter and reduced it by `1`, and then calculated the factorial of *that* value. The original problem was to calculate $n!$, and our subproblem was to calculate $(n-1)!$ – that is, our subproblem was to run the exact same algorithm, just on a smaller value of n.

  Perhaps for some other problem with an integer parameter, it would be more useful to reduce the integer parameter by subtracting `2` from it, or dividing it by `3`. Or if we have multiple integer parameters, maybe only some of them get reduced in some way, or maybe all of them get reduced in some way. In any case, we want to then run the same algorithm, but on that set of "reduced" data, rather than the original data. That is our subproblem.

  We will see a few additional examples with integer parameters, later in this lecture.

- If we are trying to count the number of items in some pile of items, we might break that pile into a number of smaller piles, and make a recursive call to count the number of items in each of those smaller piles. Then, once we have the number of items in each of the smaller piles, we could add those values together to get the number of items in the original large pile. In this case, since our problem is to count the number of items in the overall pile, the subproblem is to count the number of items in one of the smaller piles – the same algorithm, just run on a smaller collection of data. We will examine this idea further in lecture 23.

- For dealing with graphics or pictures, a subproblem could be just drawing a smaller piece of the picture, or a smaller version of the picture. We will examine this idea further in lecture 24.

- If you have an array, a subproblem is the same algorithm run on a *subarray*. The idea of a subarray is that you take the original array, but only look at part of it. So if the original array has indices 0 through 9, perhaps the recursive call deals only with the cells from 1 through 9, or the cells from 0 through 8. Or perhaps the subarray is only half the size of the original array; in such a case, our subarray might consist of the cells 0 through 4, or the cells 5 through 9. If we have a multidimensional array, then the subarray could contain all the cells but one, or it could perhaps instead contain all the *rows* but one, or all the *columns* but one.

  The idea is that we need to define the low and high indices we are concerned with for each dimension of the array. Our algorithm will then be designed to run only on the section of the array that is defined by those indices. By increasing the low index for a particular dimension, or decreasing the high index for that dimension, we can reduce the number of indices we are working with for that particular dimension – meaning we have reduced the amount of data we are dealing with, and our subproblem can be to run the same algorithm, on that smaller amount of data. We will examine this idea further in lectures 25-27.

In any cases, coming up with the *correct* subproblem is one of the keys to designing a recursive algorithm. The subproblem is the part that you can "assume is solved already". For example, when we wrote the factorial algorithm in the last lecture:

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

we didn't then write separate a separate algorithm to evaluate `(n-1)!`. We simply used `(n-1)!` in our algorithm above, and assumed that when the value of `(n-1)!` was needed, its value would "automatically" exist and could be substituted right into the expression `n * (n-1)!` above, without there being a need for us to have to write any additional algorithms or add anything else to the above algorithm. That is, when we wrote the above algorithm, we just assumed that the calculation `(n-1)!` was already done and we could freely use the result of that calculation.

It works the same way for any recursive algorithm. The procedure for writing a recursive algorithm is to:

1. Choose a subproblem.
2. Assume that your chosen subproblem could be solved just by reapplying the same recursive algorithm you are writing for the original, larger problem. That is, assume you can just snap your fingers at any time, and the work represented by the subproblem will have "magically" been completed! Once you finish writing your algorithm, then in a sense, that's what happens – you make a recursive call to run the subproblem, and then wait; when that recursive call returns, the subproblem has been completed while you were just sitting there waiting.
3. Figure out how having that subproblem solved, would help you solve the original problem.
4. Figure out what the base case should be, so that you can eventually *stop* breaking your problem into a smaller subproblem.

The steps don't have to go in that *exact* order – you might be able to figure out the base case before you've got the subproblem figured out – but that's the basic idea.

Part of why this process is difficult is that there are generally many possibilities to choose from when selecting your subproblem, so you need to figure out which of those subproblems is a useful one. There are many subproblems available for you to choose, for which having that subproblem solved will be no help whatsoever in solving the larger, original problem. You want to find a subproblem for which having it solved *does* make the original problem easier to solve. If you've found a way to make use of a particular subproblem solution in order to solve the original problem...then you've got the key information needed to write your recursive algorithm. However, if you *can't* come up with any way to make use of the solution to a particular subproblem, you might have the wrong subproblem, and should consider a different one instead.

The first example we will look at in depth is exponentiation. That is, we want to calculate the values of expressions such as 2^5 or 13^7 or 8^{21}. More generally, we are trying to calculate:

$$base^{exp}$$

We'll assume for this problem that *base* and *exp* are integers. Further more, we'll assume that $exp \geq 0$, so that we don't have to deal with fractional results, and we'll assume that $base \geq 1$ so that we avoid having to deal with 0^0 in our algorithm.

For what values of $x \leq base$ and $i \leq exp$ can x^i be used to calculate $base^{exp}$? It sometimes helps to start with a concrete example. We might consider, say, 3^5, and try and figure out what subproblem solution might help us calculate 3^5. Some possible subproblems are:

- 2^5, i.e. reduce base by one
- 3^4, i.e. reduce exponent by one
- 2^4, i.e. reduce base and exponent by one
- 2^3, i.e. reduce base by one and reduce exponent by two

There are other possibilities as well, of course, but let us at least consider the ones above for a moment. Of those subproblems above, the one that could help the most would be to decrease the exponent but not the base. If we have 3^4, we only need to multiply it by the base (3) again to get 3^5. Trying to calculate 3^5 if given the value of 2^5, 2^4, or 2^3 wouldn't be anywhere near as simple. If you were to take the value 2^4, for example, and try and use that value to help calculate 3^5, you would likely conclude pretty quickly that knowing 2^4 didn't seem to be much help in calculating 3^5, and therefore that the "lower both the base and the exponent by one" subproblem wasn't much help. But on the other hand, if you were to take the value 3^4, then you'd probably see right away how easy it was to calculate 3^5 if you already had the value of 3^4 – and thus you'd conclude that the "lower the exponent by one" subproblem was indeed a useful one, since you'd *already* found a way to use the result of that subproblem!

So, if we generalize our result above, we can calculate $base^{exp}$ with the formula $base * base^{exp-1}$. (You might even recognize that as a special case of the $base^{exp} = base^y * base^{exp-y}$ exponent rule you likely learned in whichever early math class first taught you about exponents.) However, even though that formula works in most cases, it's not all we need; for recursion, we need a base case as well. The recursive case tells us how to solve any general problem in terms of a subproblem, and the base case tells us when to stop breaking the problem into yet another subproblem. In the case of exponentation, we said we'd allow our exponent to get as low as zero (anything less and our results are no longer integers) and any positive number raised to the zero power is 1, so our base case could be that when the exponent is `0`, we return 1.

That gives us the following mathematical formula:

```
                ____________  1, if exp == 0
     exp        |
 base      =    |
                |                            exp-1
(base >= 1,     |___________  base * (base       ),  if exp > 0
 exp >= 0)
```

The above is our algorithm. We could also have written the above out in some form of pseudocode, or in English, but since it's a mathematical computation, the above mathematical form is a nice way to express the algorithm. The point, however, is that the above tells us precisely how to calculate $base^{exp}$ for any legal values of *base* and *exp*.

Implementing the exponentiation algorithm

Since we will need the values of *base* and *exp* in order to calculate $base^{exp}$, it would follow that those two values should be arguments that we send to an exponentiation method, to be stored in parameters. That would give us the following first line for our method (the method signature, plus the `public static` tagged on to the beginning of the method):

```
public static int pow(int base, int exp)
```

We'll name the method `pow` since that is short for "power" and we are calculating "base to the exp power".

Given the mathematical description of exponentiation that we had earlier:

```
                ____________  1, if exp == 0
     exp        |
 base     =     |
                |                            exp-1
(base >= 1,     |____________  base * (base       ),  if exp > 0
 exp >= 0)
```

we can just convert this straight into a recursive method. Since we have two cases above, one where `exp == 0` and one where `exp > 0`, we will need a conditional to decide between those two cases:

```
public static int pow(int base, int exp)
{
   if (exp == 0)
      // [...we need to add some code here]
   else  //  exp > 0
      // [...we need to add some code here]
}
```

and then, in the `exp == 0` case in our mathematical description, we say the answer is `1`, so let us return `1` from our method in that case:

```
public static int pow(int base, int exp)
{
   if (exp == 0)
      return 1;
   else  //  exp > 0
      // [...we need to add some code here]
}
```

and likewise, the `exp > 0` case in the mathematical description above gives us the following as our answer:

```
                    exp-1
          base * (base      )
```

Since $base^{exp-1}$ will just be converted to `pow(base, exp-1)`, the recursive calculation in our mathematical description, translates to

```
          base * pow(base, exp - 1)
```

and thus we have the following recursive method, which is now complete:

```
// implementation of our exponentiation algorithm
public static int pow(int base, int exp)   // base >= 1; exp >= 0
{
   if (exp == 0)
      return 1;
   else  //  exp > 0
      return base * pow(base, exp - 1);
}
```

Note the comment at the end of the method signature – it's a reminder of what values are legal arguments for this method and what values are not. We might also have provided a more complete method header comment for this method, that would have told us the same information:

```
// pow
//    - parameters : base - the base of our exponentiation; we
//                      assume this value is >= 1
//                 : exp - the exponent to which we want to raise
//                      our base value; we assume this value is >= 0
//    - return value : an integer holding the value of base raised
//                       to the exp power
//    - calculates "base to the exp power" and returns that value
public static int pow(int base, int exp)
{
   if (exp == 0)
      return 1;
   else  //  exp > 0
      return base * pow(base, exp - 1);
}
```

If you make assumptions about the parameter values in your code, it's generally a nice idea to add a comment somewhere about that, so that other people using your code understand what values are and aren't legal to send as arguments to your code.

A run of the recursive method with base being 3 and exponent being 4 could start like this:

```
-------------------------------------------------------------------
| main
|   int x;
|   x = pow(3, 4);  // we call the pow method, and in doing so,
|                   //  set up the "pow" notecard below:
|-----------------------------------------------------------------
| pow     (base: 3)  // this is the notecard for the first call to
|  (#1)   (exp: 4)   // pow; 3 and 4 were passed to this method as
|                    // arguments, and are stored in the parameters
|                    // "base" and "exp", respectively
|-----------------------------------------------------------------
```

When we run the `pow` code on the data in the `pow` notecard, since `exp` is not zero, we will take the recursive case, which requires we calculate $base^{exp-1}$, i.e. 3^3.

```
-------------------------------------------------------------------
| main
|   int x;
|   x = pow(3, 4);
|-----------------------------------------------------------------
| pow     (base: 3)
|  (#1)   (exp: 4)
|
|  need to calculate: pow(3, 3), so we can use its value to
|                                complete the multiplication in the
|                                expression
|
|                                  base * pow(base, exp - 1)
|                                          i.e.
|                                     3 * pow(3, 3)
|
|                                hence the call to pow and the
|                                new notecard below this one, which
|                                stores the parameters for that call
|-----------------------------------------------------------------
```

So, we make another call to our `pow` method. That is, from the call `pow(3, 4)`, we will make the call `pow(3, 3)`, and therefore we will start a new notecard where the `base` and `exp` arguments are both `3`.

```
-------------------------------------------------------------------
| main
|   int x;
|   x = pow(3, 4);
|-----------------------------------------------------------------
| pow     (base: 3)
|  (#1)   (exp: 4)
|
|  need to calculate: pow(3, 3), in order to calculate 3 * pow(3, 3)
|
|-----------------------------------------------------------------
| pow     (base: 3)
|  (#2)   (exp: 3)
|
|  need to calculate: pow(3, 2), so we can use its value to
|                                complete the multiplication in the
|                                expression
|
|                                  base * pow(base, exp - 1)
|                                          i.e.
|                                     3 * pow(3, 2)
|
|                                hence the call to pow and the
|                                new notecard below this one, which
|                                stores the parameters for that call
|-----------------------------------------------------------------
```

As you can see above, from within the `pow(3, 3)` call, we will decide that we need a `pow(3, 2)` call, and so we will make that call:

```
-------------------------------------------------------------------
| main
|   int x;
|   x = pow(3, 4);
|-----------------------------------------------------------------
| pow     (base: 3)
|  (#1)   (exp: 4)
|
|  need to calculate: pow(3, 3), in order to calculate 3 * pow(3, 3)
|
|-----------------------------------------------------------------
| pow     (base: 3)
|  (#2)   (exp: 3)
|
|  need to calculate: pow(3, 2), in order to calculate 3 * pow(3, 2)
|
|-----------------------------------------------------------------
| pow     (base: 3)
|  (#3)   (exp: 2)
|
|  need to calculate: pow(3, 1), so we can use its value to
|                                complete the multiplication in the
|                                expression
|
|                                  base * pow(base, exp - 1)
|                                          i.e.
|                                     3 * pow(3, 1)
|
|                                hence the call to pow and the
|                                new notecard below this one, which
|                                stores the parameters for that call
|-----------------------------------------------------------------
```

From within the `pow(3, 2)` call, we will need a `pow(3, 1)` call, so we make that call:

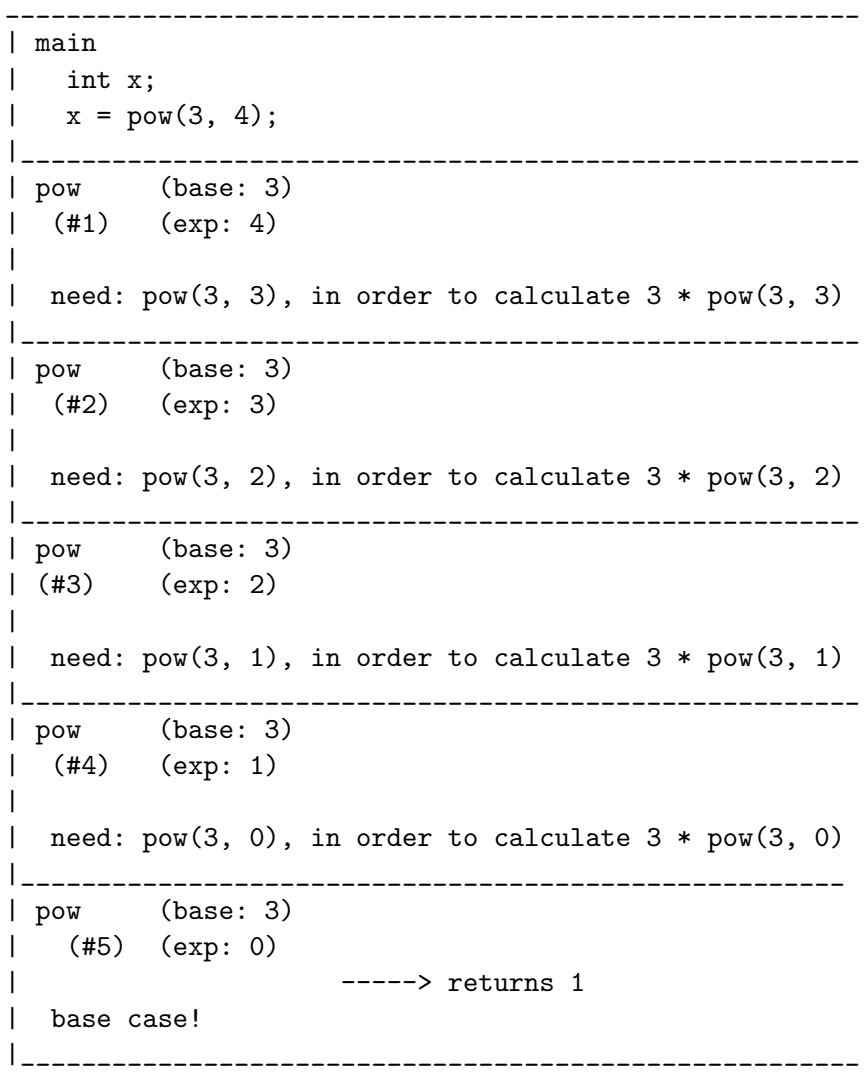
```
-------------------------------------------------------------------
| main
|   int x;
|   x = pow(3, 4);
|-----------------------------------------------------------------
| pow     (base: 3)
|  (#1)   (exp: 4)
|
|  need to calculate: pow(3, 3), in order to calculate 3 * pow(3, 3)
|
|-----------------------------------------------------------------
| pow     (base: 3)
|  (#2)   (exp: 3)
|
|  need to calculate: pow(3, 2), in order to calculate 3 * pow(3, 2)
|
|-----------------------------------------------------------------
| pow     (base: 3)
|  (#3)   (exp: 2)
|
|  need to calculate: pow(3, 1), in order to calculate 3 * pow(3, 1)
|
|-----------------------------------------------------------------
| pow     (base: 3)
|  (#4)   (exp: 1)
|
|  need to calculate: pow(3, 0), so we can use its value to
|                                complete the multiplication in the
|                                expression
|
|                                  base * pow(base, exp - 1)
|                                          i.e.
|                                     3 * pow(3, 0)
|
|                                hence the call to pow and the
|                                new notecard below this one, which
|                                stores the parameters for that call
|-----------------------------------------------------------------
```

From within the `pow(3, 1)` call, we will need a `pow(3, 0)` call, so we make that call:

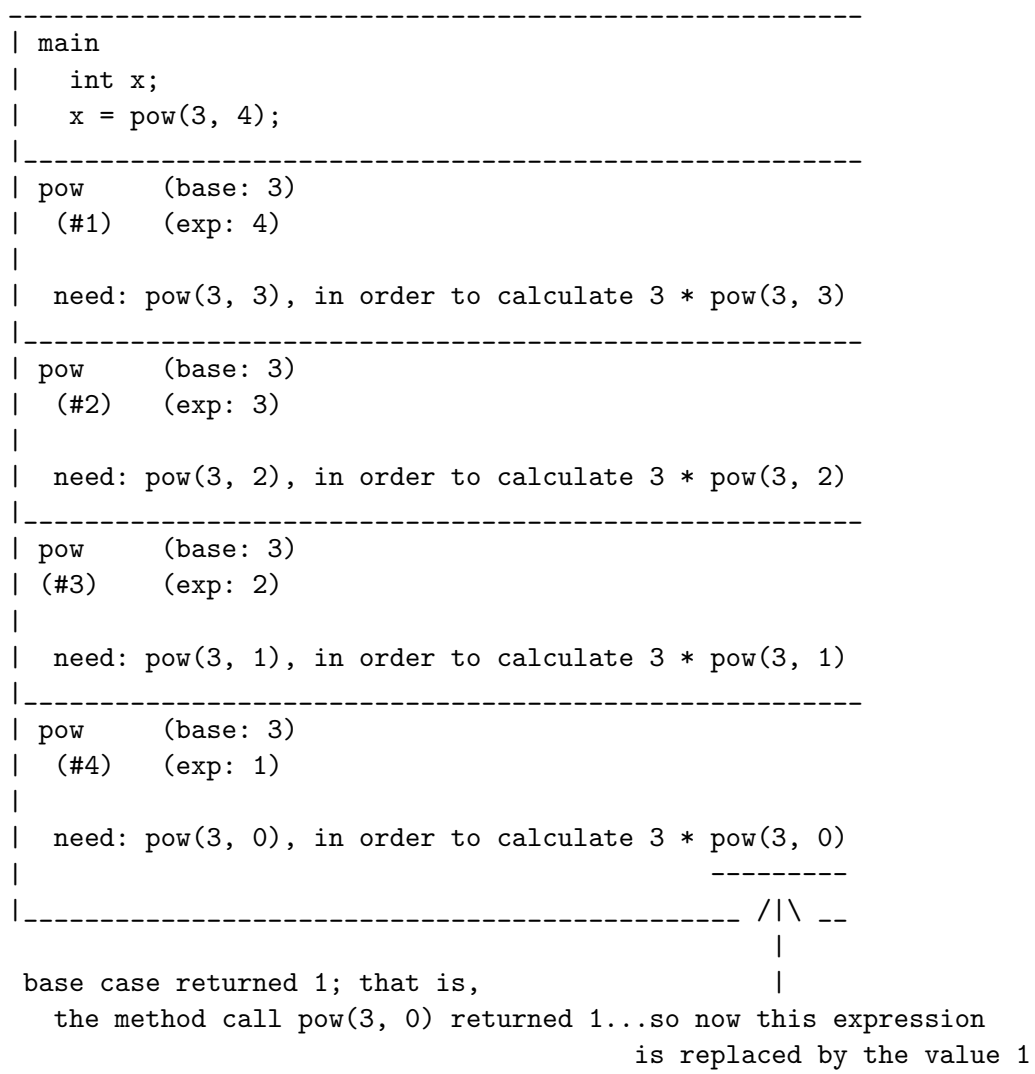
```
-------------------------------------------------------
| main
|   int x;
|   x = pow(3, 4);
|-------------------------------------------------------
| pow     (base: 3)
|  (#1)   (exp: 4)
|
|  need: pow(3, 3), in order to calculate 3 * pow(3, 3)
|-------------------------------------------------------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

We make and pause four separate calls to `pow`, 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`:



Therefore, the value 1 is returned back to the previous call and is substituted for "`pow(3, 0)`", the method call expression that triggered the base case to be started:



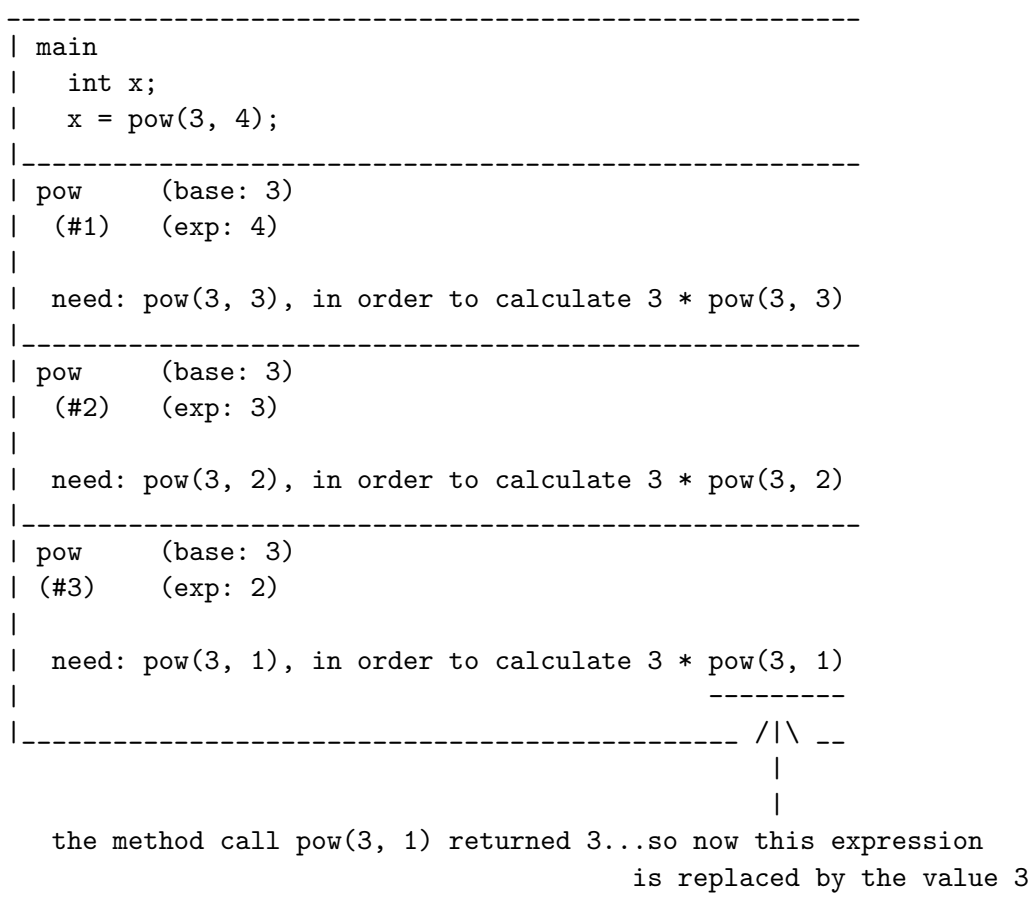
Now that we have a value for `pow(3, 0)`, we can finally complete the multiplication, since now we have both operands:

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

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

```
-------------------------------------------------------
| main
|   int x;
|   x = pow(3, 4);
|-----------------------------------------------------
| pow     (base: 3)
|  (#1)   (exp: 4)
|
|  need: pow(3, 3), in order to calculate 3 * pow(3, 3)
|-----------------------------------------------------
| pow     (base: 3)
|  (#2)   (exp: 3)
|
|  need: pow(3, 2), in order to calculate 3 * pow(3, 2)
|-----------------------------------------------------
| pow     (base: 3)
| (#3)    (exp: 2)
|
|  need: pow(3, 1), in order to calculate 3 * pow(3, 1)
|-----------------------------------------------------
| pow     (base: 3)
|  (#4)   (exp: 1)
|
|  this method call is supposed            3
|  to return 3 * pow(3, 0) which          ------ <--- so now this expression
|  is 3 * 1 which is 3                              is returned
|-----------------------------------------------------
```

When the fourth `pow` call returns `3`, then the program returns back to the third `pow` call, and now that the `pow(3, 1)` call is over, it is replaced by its return value, which is `3`:



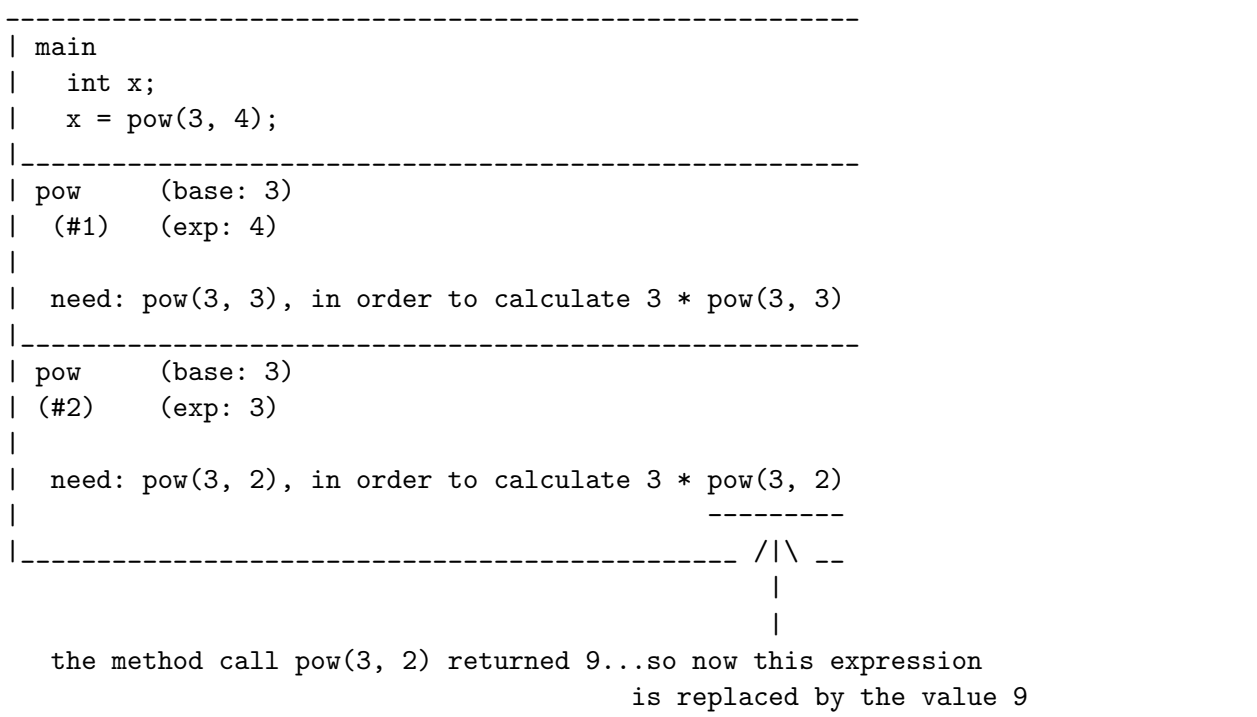
And then this call works as the previous one did. First, the multiplication is completed:

```
-------------------------------------------------------
| main
|   int x;
|   x = pow(3, 4);
|-----------------------------------------------------
| pow     (base: 3)
|  (#1)   (exp: 4)
|
|  need: pow(3, 3), in order to calculate 3 * pow(3, 3)
|-----------------------------------------------------
| pow     (base: 3)
|  (#2)   (exp: 3)
|
|  need: pow(3, 2), in order to calculate 3 * pow(3, 2)
|-----------------------------------------------------
|
| pow     (base: 3)
|  (#3)   (exp: 2)
|
|  this method call is supposed          3 * 3
|  to return 3 * pow(3, 1) which         ------  <--- so now this expression
|  is 3 * 3 which is 9                                 evaluates to 9
|-------------------------------------------------
```

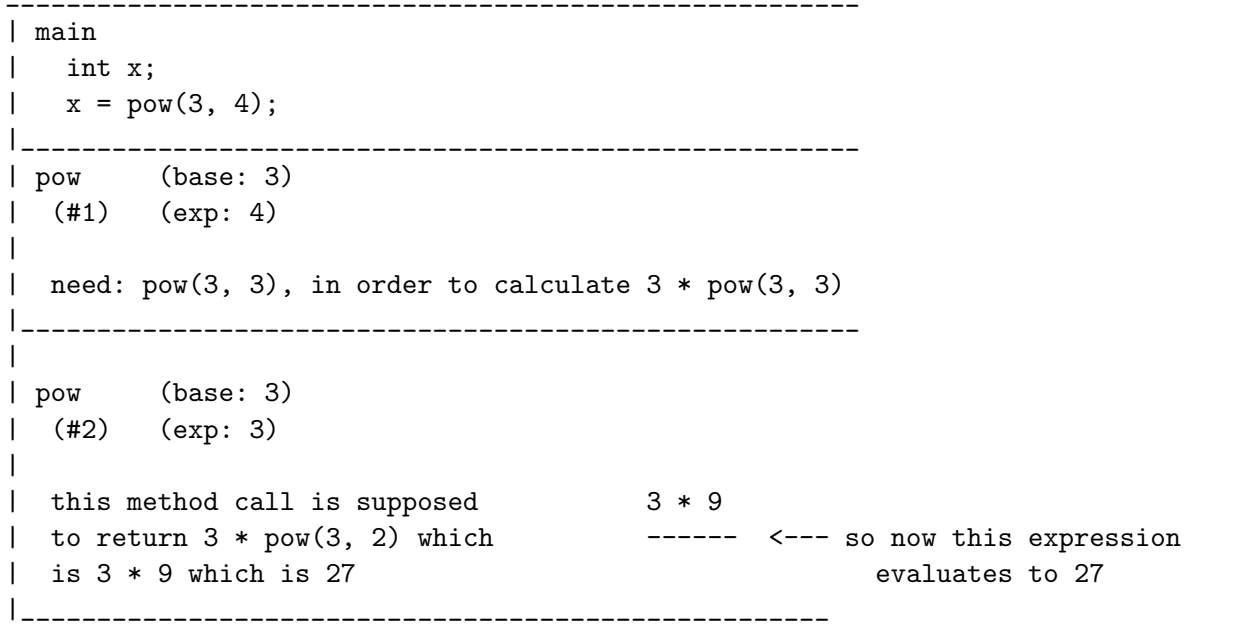
And then we were supposed to return the result of that multiplication:

```
-------------------------------------------------------
| main
|   int x;
|   x = pow(3, 4);
|-----------------------------------------------------
| pow     (base: 3)
|  (#1)   (exp: 4)
|
|  need: pow(3, 3), in order to calculate 3 * pow(3, 3)
|-----------------------------------------------------
| pow     (base: 3)
|  (#2)   (exp: 3)
|
|  need: pow(3, 2), in order to calculate 3 * pow(3, 2)
|-----------------------------------------------------
|| pow    (base: 3)
|  (#3)   (exp: 2)
|
|  this method call is supposed             9
|  to return 3 * pow(3, 1) which          ------ <--- so now this value
|  is 3 * 3 which is 9                                 is returned
|-----------------------------------------------------
```

When the third `pow` call returns `9`, then the program returns back to the second `pow` call, and now that the `pow(3, 2)` call is over, it is replaced by its return value, which is `9`:



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



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

```
-------------------------------------------------------
| main
|   int x;
|   x = pow(3, 4);
|-----------------------------------------------------
| pow     (base: 3)
|  (#1)   (exp: 4)
|
|  need: pow(3, 3), in order to calculate 3 * pow(3, 3)
|-----------------------------------------------------
|| pow    (base: 3)
|  (#2)   (exp: 3)
|
|  this method call is supposed            27
|  to return 3 * pow(3, 2) which          ------ <--- so now this value
|  is 3 * 9 which is 27                              is returned
|-----------------------------------------------------
```

When the second `pow` call returns 27, then the program returns back to the first `pow` call, and now that the `pow(3, 3)` call is over, it is replaced by its return value, which is 27:

```
-------------------------------------------------------
| main
|   int x;
|   x = pow(3, 4);
|-----------------------------------------------------
| pow     (base: 3)
| (#1)    (exp: 4)
|
|  need: pow(3, 3), in order to calculate 3 * pow(3, 3)
|                                                ---------
|------------------------------------------------ /|\ __
                                                   |
                                                   |
   the method call pow(3, 3) returned 27...so now this expression
                                          is replaced by the value 27
```

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

```
-------------------------------------------------------
| main
|   int x;
|   x = pow(3, 4);
|_______________________________________________________
|
| pow     (base: 3)
|  (#1)   (exp: 4)
|
|  this method call is supposed           3 * 27
|  to return 3 * pow(3, 3) which          ------  <--- so now this expression
|  is 3 * 27 which is 81                                 evaluates to 81
|_____________________________________________________
```

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

```
-------------------------------------------------------
| main
|   int x;
|   x = pow(3, 4);
|_______________________________________________________
|
| pow    (base: 3)
|  (#1)   (exp: 4)
|
|  this method call is supposed             81
|  to return 3 * pow(3, 3) which          ------ <--- so now this value
|  is 3 * 27 which is 81                                 is returned
|_____________________________________________________
```

When the first `pow` call returns `81`, then the program returns back to `main()`, and now that the `pow(3, 4)` call is over, it is replaced by its return value, which is `81`, which then gets written into the variable `x` by the assignment statement.

```
-------------------------------------------------------
| main
|   int x;
|   x = pow(3, 4);
|       ---------
|_______ /|\ ___________________________________________
          |
          |
    this expression is replaced by 81, since the
     method call pow(3, 4) has returned 81
```

Another example we can consider, is adding together the digits of a non-negative integer. For example:

```
Number              Sum of digits
----------          -------------
   0                     0
   4                     4
   92                    11      (9 + 2)
   305                   8       (3 + 0 + 5)
   45924                 22      (4 + 5 + 9 + 4)
   59480                 26      (5 + 9 + 4 + 8 + 0)
```

What subproblem is helpful in this case? Well, once again, let's try a concrete example. Imagine we are trying to find the sum of the digits in the integer `51460` (which is `16`, but we'll pretend we don't know that yet). If our number is `51460`, what smaller subproblem helps us out?

- 51459, i.e. $(number - 1)$ 24
- 51457, i.e. $(number - 3)$ 22
- 25730, i.e. $(number/2)$ 17

Those are three possibilities, and none of them seem particularly promising – all of them have a greater *sum of all digits* than the original sum.

What we really want to be doing is counting "fewer digits" for our recursive call:

```
sumOf(51460) == sumOf(5146) + 0

                sumOf(5146) == sumOf(514) + 6

                               sumOf(514) = sumOf(51) + 4

                                            sumOf(51) = sumOf(5) + 1

                                                        sumOf(5) == 5
```

Note that once our number drops below ten, we only have one digit, so the sum-of-digits of that number is just itself.

The above looks nice, but how can be extract individual digits from a number? Well, we can use the modulus and division operators:

```
num % 10 == the one's place of the number (ex: 45926 % 10 == 6)
num / 10 == all *but* the one's place of the number (ex: 45926 / 10 == 4592)
```

That gives us the following code:

```
public static int addDigits(int n) // assume n >= 0
{
   if (n <= 9)
      return n;
   else
   {
      int onesPlace = n % 10;
      int sumOfRest = addDigits(n/10);
      return sumOfRest + onesPlace;
   }
}
```