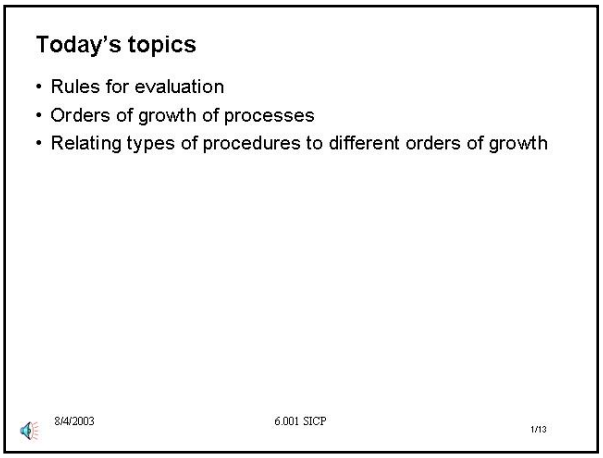
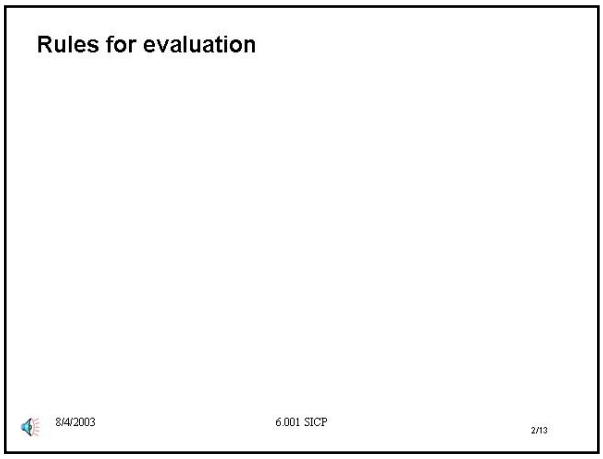
6.001 Notes: Section 4.1

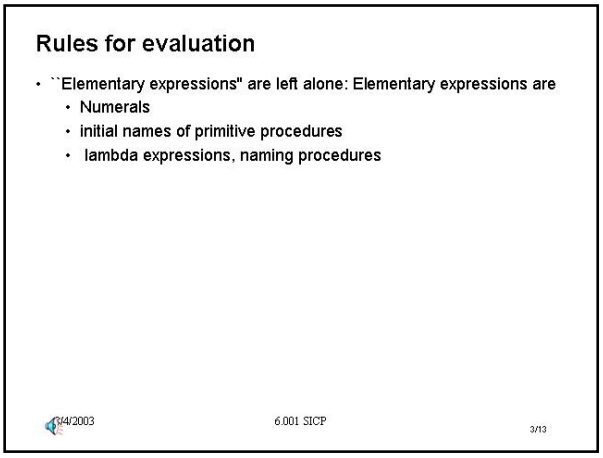
Slide 4.1.1
In this lecture, we are going to take a careful look at the kinds of procedures we can build. We will first go back to look very carefully at the substitution model, to see how the rules for evaluation help us determine the evolution of a process controlled by a procedure. We will then see how different kinds of process evolution can occur, with different demands in terms of computational resources. And we will see how to characterize those differences formally in terms of orders of growth of a process. Finally, we will explore examples of procedures from different classes of growth, helping you to begin to relate how choices in procedure design may affect performance of the actual use of the procedure.

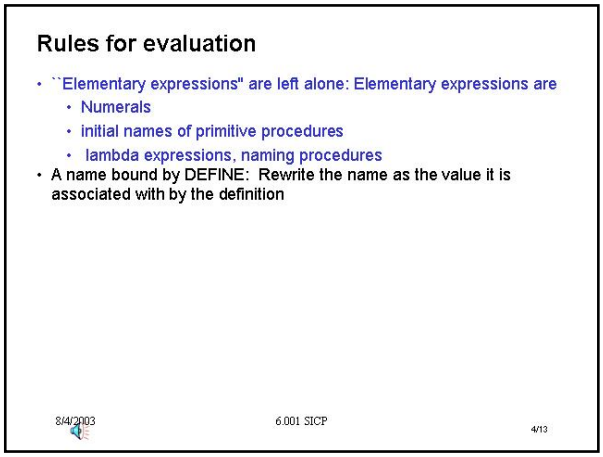




Slide 4.1.2
Now that we have seen two different implementations of the same idea, with different behaviors, we need a way of more formally characterizing why that difference occurs. To do that, we are going to go back to our substitution model.
Now we are going to think of our model as a set of **rewrite rules**, that is, as a set of rules that formally specify, given an expression of a particular form, rewrite it in the following form.

Slide 4.1.3
So elementary expressions in this viewpoint are just "left alone", that is, numbers, names of built-in procedures, and lambda expressions are left as is.



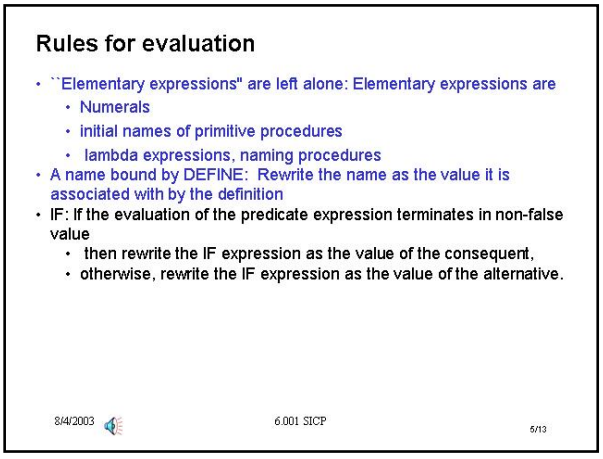


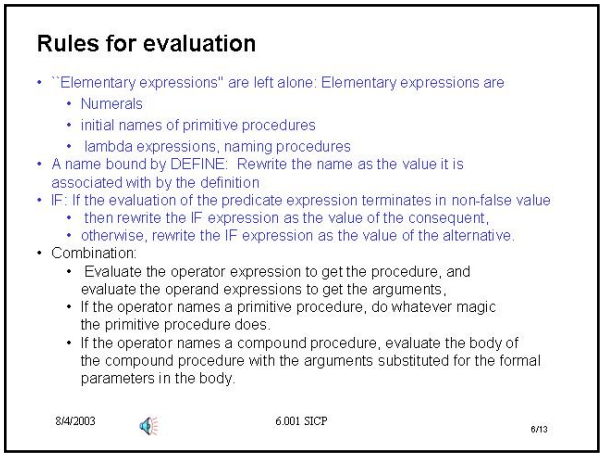
Slide 4.1.4

If our expression is a name that we created ourselves, we will rewrite in place of it the value that was associated with that name during the definition.

Slide 4.1.5

For the special form `if` we use the rule we just saw. We evaluate the predicate clause, and based on its value, we either rewrite the `if` expression by its consequent or its alternative.





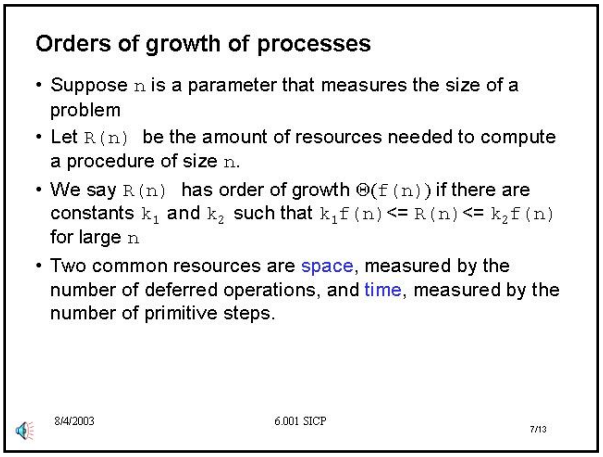
Slide 4.1.6

And finally: combinations. We use the rule that we first evaluate the operator expression to get the procedure, and we evaluate the operands to get the set of arguments. If we have a primitive procedure, we are just going to "do the right thing". Otherwise we are going to replace this entire expression with the body of the compound expression, with the arguments substituted for their associated parameters.

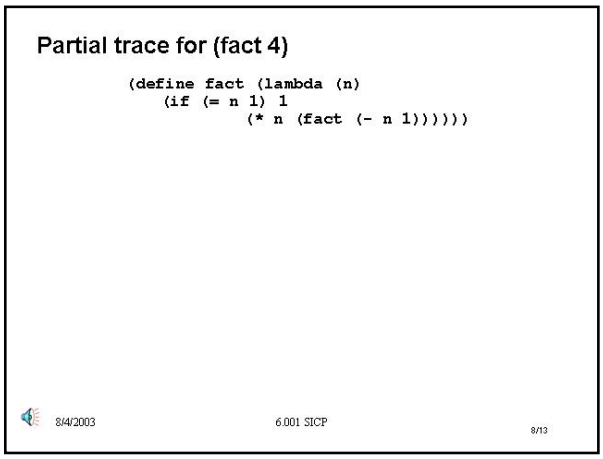
Slide 4.1.7

Given that model, we can more formally capture the evolution of a process. We can talk about the order of growth of a process as these rewrite rules evolve. This measures how much of a particular resource a process is going to take as these rules evolve, where typically we measure either space or time as the resource.

More formally, let **n** be a parameter that measures the size of the problem of interest. In the case of factorial, this would be the size of the input argument. We let R(n) denote the amount of resources we will need to solve a problem of this size, where as noted, the resources are usually space and time. We are interested in characterizing R(n) for large values of N, that is, in



the asymptotic limit. We say that R(n) has order of growth, Theta of **f** of **n** if we can find a function **f(n)** that is roughly of the same order as the needed resource, where the more formal definition is shown in the

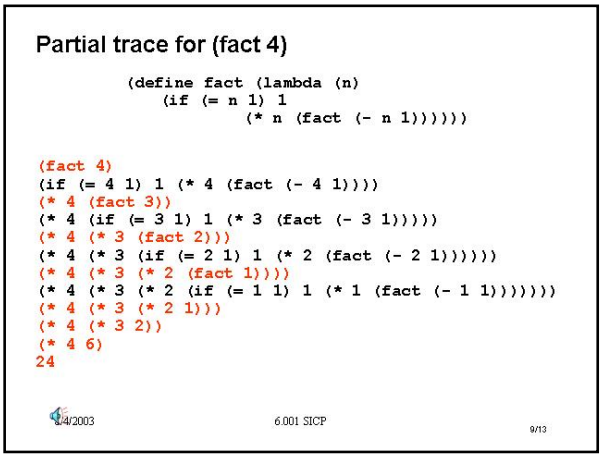


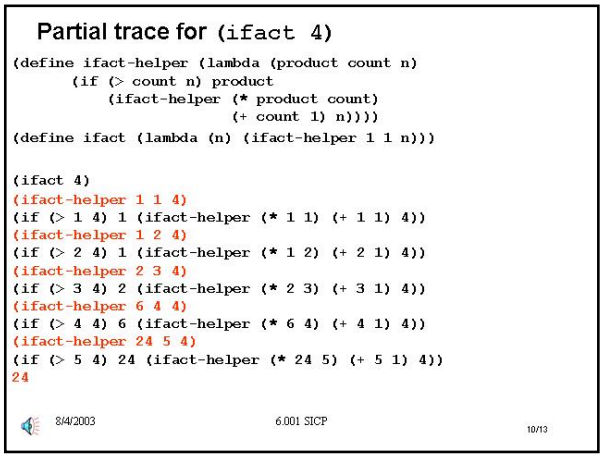
Slide 4.1.8

In more common terms, this says that when measuring the amount of space we need, we want to find a function that measures the number of deferred operations, as a function of the size of the problem, up to a constant factor. For measuring the amount of time, we want to find a function that measures the number of basic or primitive steps that the rewrite rules go through, again as a function of the size of the problem.

Slide 4.1.9

So let's use our rewrite rule idea, together with this notion of measuring the amount of resource we need as an order of growth in the size of the problem. Here is our recursive factorial procedure ...





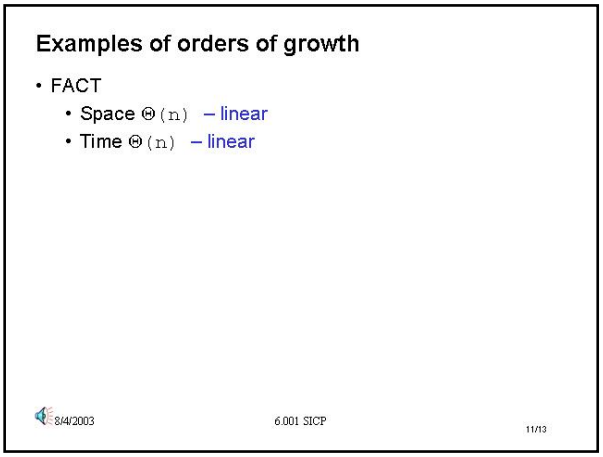
Slide 4.1.10

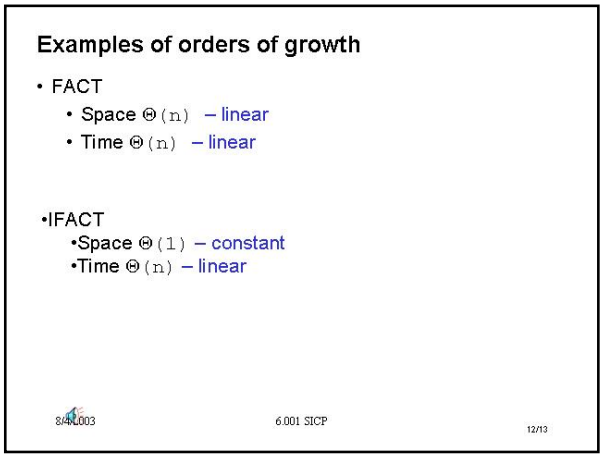
..and here is a partial trace of `fact` using those rewrite rules. We start with (fact 4). That reduces to evaluating an *if* expression. Since the predicate is not true, this reduces to evaluating the alternative statement, which is a combination of a multiplication and another evaluation of fact.
Notice how the orange colored lines capture the evolution of the rewrite rules. Note how (fact 4) reduces to a deferred operation and an evaluation of (fact 3) and this further reduces to another deferred operation and a subsequent call to fact. Notice the shape of this process: it grows out with a set of deferred operations until it reaches a base case, and then it contracts back in, completing each deferred operation. Using this, we can see that the amount of space grows **linearly** with the size of the argument. That is, with each increase in the argument size, I add a constant amount of space requirement. In terms of the number of operations, we can see that we basically need twice as many steps as the size of the problem, one to expand out, and one to reduce back down. This is also a **linear** growth, since the constant 2 does not change with problem size.

Slide 4.1.11

And let's compare that our iterative factorial. Here is our code again, and here is a partial trace of the rewrite evolution of that process.

Here we see the different shape of the process, and our order of growth analysis captures that difference. In particular, there are no deferred operations, and you can see that the maximum amount of space I need at any stage, that is, the maximum amount of space used by any rewrite, is independent of the size of the problem. We say that it is **constant**, as it does not grow with **n**. In terms of time, there is basically one operation for each increment in the argument, so this is again a **linear** order of growth.





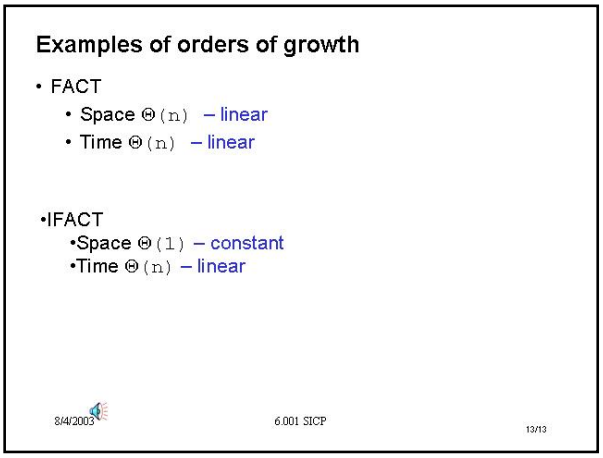
Slide 4.1.12

So we can formally capture this, as shown. `Fact` has linear growth in space, written as shown, because the maximum amount of space needed by any rewrite stage is a linear multiple of the size of the argument. It also has linear growth in time, because as we saw it takes a number of basic steps that is also a linear multiple of the size of the argument.

Slide 4.1.13

On the other hand, `iterative-fact` has no deferred operations, and is constant in space, written as shown, while as we argued, the time order of growth is linear. Notice that the fact that the recursive version took **2n** steps and the iterative version took **n** steps doesn't matter in terms of order of growth. Both are said to be linear.

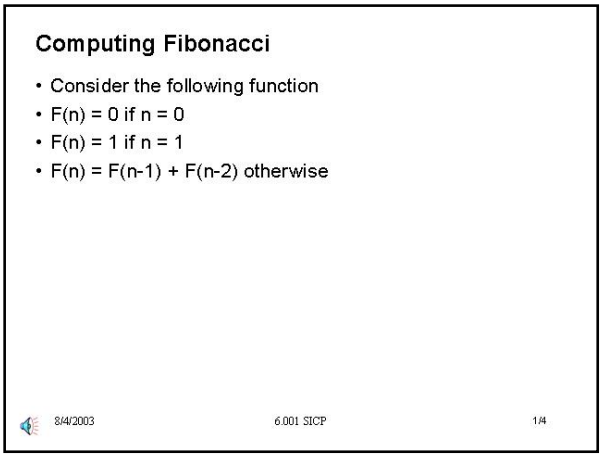
Thus we see that we can formally characterize the difference between these two processes, which have different shapes of evolution.

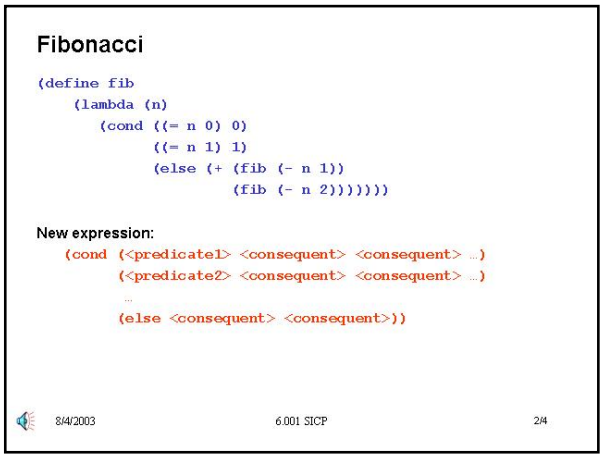


Slide 4.1.14
So why have we done this? The primary goal is to allow you to start to recognize different kinds of behaviors, and the associated procedure forms that give rise to those behaviors. This will start to allow you to work backwards, in designing a procedure, by enabling you to visualize the consequences of performing a computation in different ways.

6.001 Notes: Section 4.2

Slide 4.2.1
Having seen two different kinds of processes, one linear, and one constant, we want to fill out our repertoire of processes, by looking at other kinds of processes. The next one is an example of an exponential process, and deals with a classic function called **Fibonacci**. Its definition is that if its argument is 0, its value is 0, if its argument is 1, its value is 1, and for all other positive integer arguments, its value is the sum of its values for the two preceding arguments.
We would like to write a procedure to compute Fibonacci, and in particular see how it gives rise to a different kind of behavior.

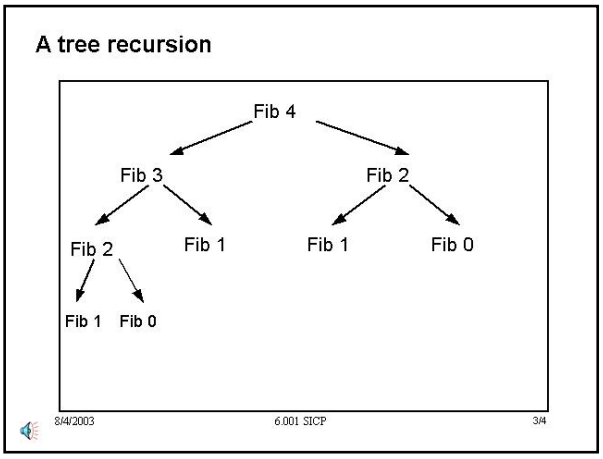


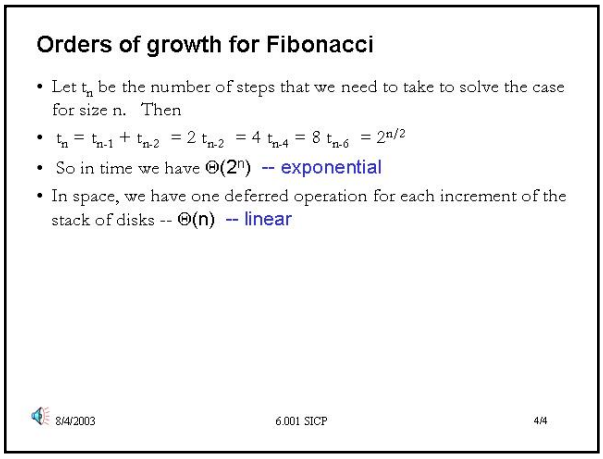


Slide 4.2.2
To solve this problem, let's use our tool of **wishful thinking**. Here, that wishful thinking says, let's assume that given an argument **n**, we know how to solve Fibonacci for any smaller sized argument. Using that idea, we can then work out a solution to the problem. With this in hand, it is clear that the solution to the general problem is just to solve two smaller sized problems, then just add the results together. Note that in this case we are using wishful thinking twice, not once, as in our previous examples.
Here is a procedure that captures this idea.
First, we introduce a new expression, called a `cond` expression. `Cond` uses the following rules of evaluation. The `cond` consists of a set of clauses, each of which has within it, a predicate clause, and one or more subsequent expressions. `Cond` proceeds by first evaluating the predicate of the first clause, in this case `(= n 0)`. If it is true, then we evaluate in turn each of the other expressions in this clause, returning the value of the last one as the value of the whole `cond`. In the case of the first clause of this `cond` that is the expression `0`. If the predicate of the first clause is false, we move to the next

clause of the cond and repeat the process. This continues for however many clauses are contained in the `cond` until either a true predicate is reached, or we reach a clause with the special keyword `else`. In this latter case, that predicate is treated as true, and the subsequent expressions are evaluated.
Given that form for `cond`, we can see how our `fibonacci` procedure captures the idea we expressed. We keep recursively solving smaller sized problems until we get to a base case. Notice that here we have two base cases, not one.
Also notice that while this is a recursive procedure, it is different from our earlier ones. Here, there are two recursive calls to the procedure in the body, rather than just one. Our question is whether this leads to a different kind of behavior, or a different order of growth.

Slide 4.2.3
This does give rise to a different kind of behavior, which we can easily see with the illustrated diagram. To solve a problem of size 4, notice that we have to solve two problems, one of size 3 and one of size 2, and each of these requires solving two smaller problems, and so on.
This gives rise to a kind of tree of things we have to do, and each recursive call gives rise to two subproblems of smaller size. This leads to a different order of growth.



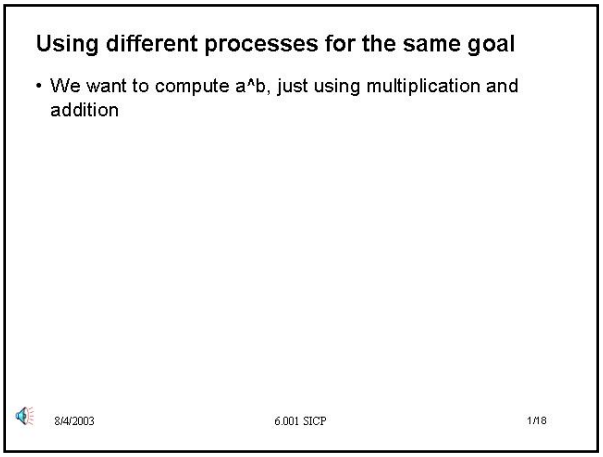


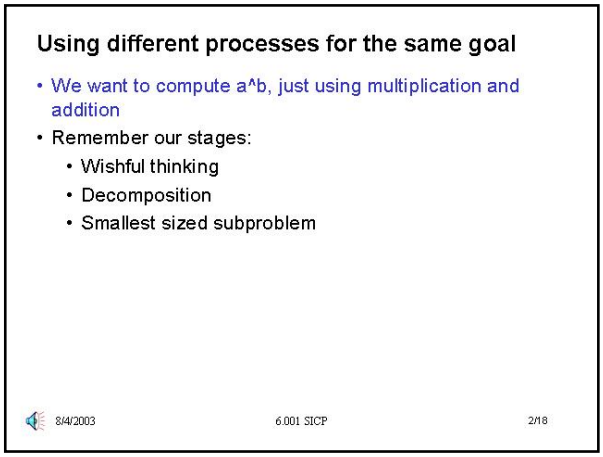
Slide 4.2.4
To measure the order of growth, let's let **t** of **n** denote the number of time steps we need to solve a problem of size **n**. From our tree, we see that to do this, we need to solve a problem of size **n-1**, and a problem of size **n-2**. We could actually work out a detailed analysis of this relationship, but for our purposes, we can approximate this as roughly the same as solving two problems of size **n-2**. Expanding a step further, this is roughly the same as solving 4 problems of size **n-4** and roughly the same as solving 8 problems of size **n-6**. A little math shows that in general this reduces to **2** to the power of **n/2** steps.
This is an example of an exponential order of growth, and this is very different from what we saw earlier. To convince yourself of this, assume that each step takes one second, and see how much time it would take for an exponential process as compared to a linear one, as n gets large.
In terms of space, our tree shows us that we have basically one deferred operation for step, or in other words, the maximum depth of that tree is linear in the size of the problem, and the maximum depth exactly captures the maximum number of deferred operations.

6.001 Notes: Section 4.3

Slide 4.3.1

Let's take another quick look at how we can create procedures with different orders of growth to compute the same function. As a specific example, suppose we want to compute exponentials, such as a raised to the b power, but to do so only using the simpler operations of multiplication and addition. How might we use the tools we have been developing to accomplish this?



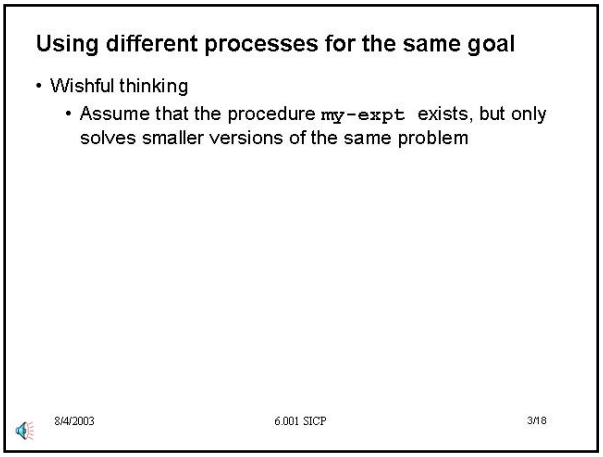


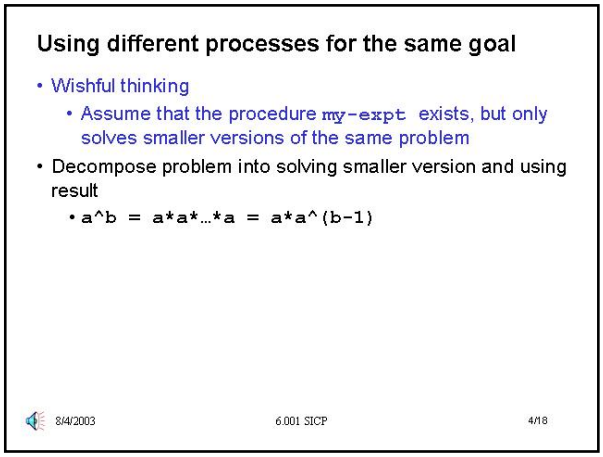
Slide 4.3.2

So, recall the stages we used to solve problems like this: we will use some wishful thinking to assume that solutions to simpler versions of the problem exist; we will then decompose the problem into a simpler version of the same problem, plus some other simple operations, and we will use this to construct a solution to the more general problem; and finally, we will determine the smallest sized subproblem into which we want to do decomposition. Let's look at using these tools on the problem of exponentiation.

Slide 4.3.3

Wishful thinking is our tool for using induction. It says, let's assume that some procedure `my-expt` exists, so that we can use it, but that it only solves smaller versions of the same problem.





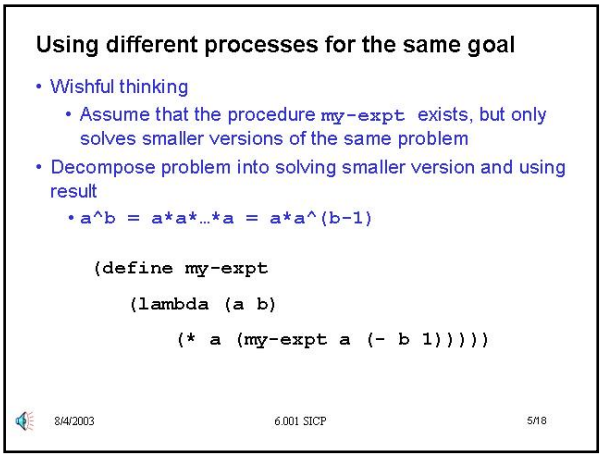
Slide 4.3.4

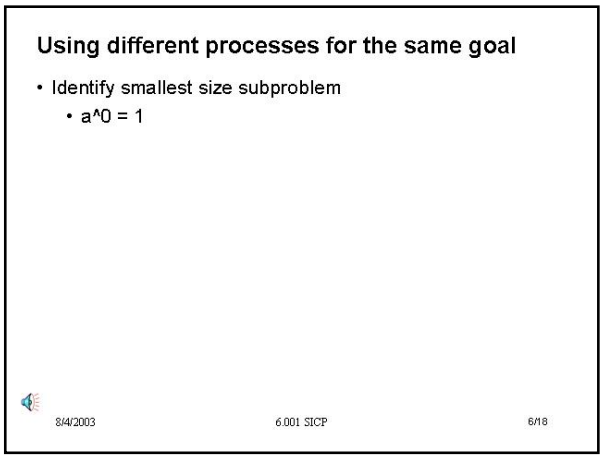
Given that assumption, we can then turn to the tricky part, which is determining how to decompose the problem into a simpler version of the same problem.

Here is one method: Notice that a to the power b mathematically is just b products of a. But this we can also group as a times b-1 products of a, and that latter we recognize as a smaller version of the same exponentiation problem. Thus, we can reduce a to the bth power as a times a to the b-1st power. Thus we can reduce the problem to a simpler version of the same problem, together with some simple operations, in this case, an additional multiplication.

Slide 4.3.5

Given that idea, here is the start of our procedure for `my-expt`. Notice how it recursively uses `my-expt` to solve the subproblem, then multiplies the result by `a` to get the full solution.





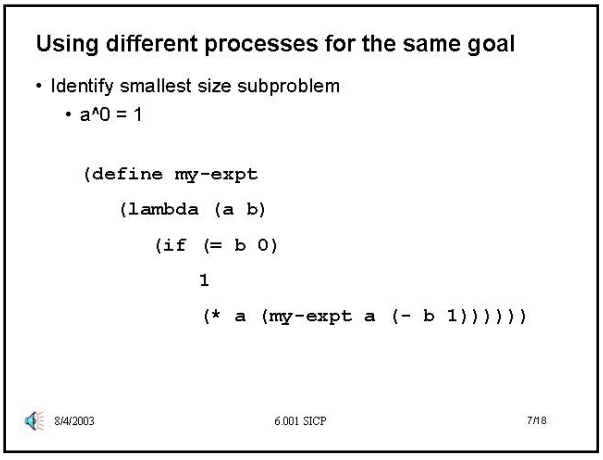
Slide 4.3.6

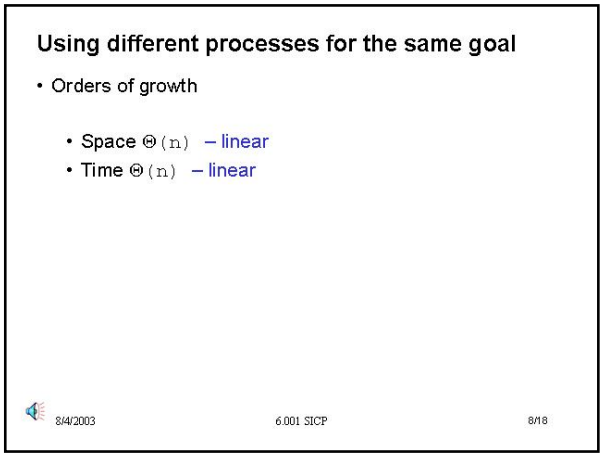
Of course, as stated, that procedure will fail, as it will recurse indefinitely. To stop unwinding into simpler versions of the same problem, we need to find a smallest size subproblem. Here, that is easy. Anything to the `0`th power is just 1!

Slide 4.3.7

And thus we can add our base case to our procedure, creating the same kind of procedure we saw earlier for `factorial`.

Note how the base case will stop unwinding the computation into simpler cases.



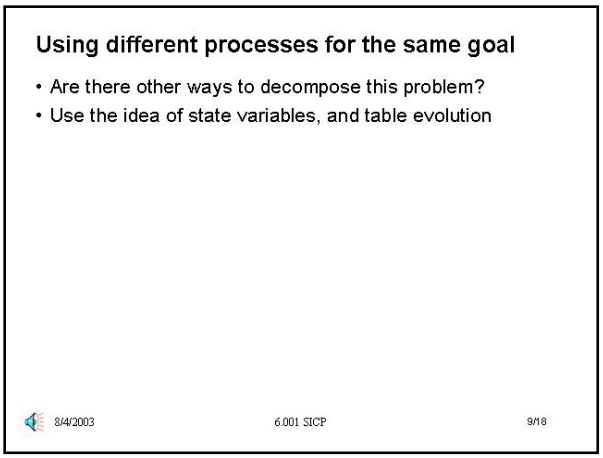


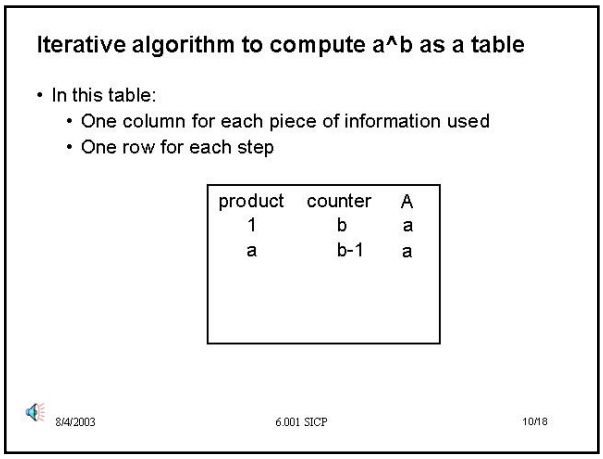
Slide 4.3.8

You could run the substitution model on this procedure to both verify that it works, and to determine its growth pattern. However, the form is very familiar, and we can thus deduce by comparison to `factorial` that this procedure has linear growth in both space and time. The time is easy to see: there is one additional step for each increment in the size of the argument. And for space, we see that there is one deferred operation for each recursive call of the procedure, so we will stack up a linear number of such deferred operations until we get down to the base case, at which point we can start completing the deferred multiplications and gathering up the answer.

Slide 4.3.9

As we saw earlier, we expect there are other ways of creating procedures to solve the same problem. With factorial, we used the idea of a table of state variables, with update rules for changing the values of those variables. We should be able to do the same thing here.



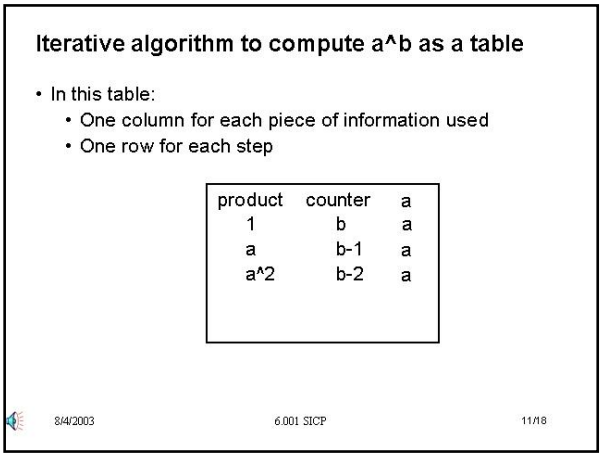


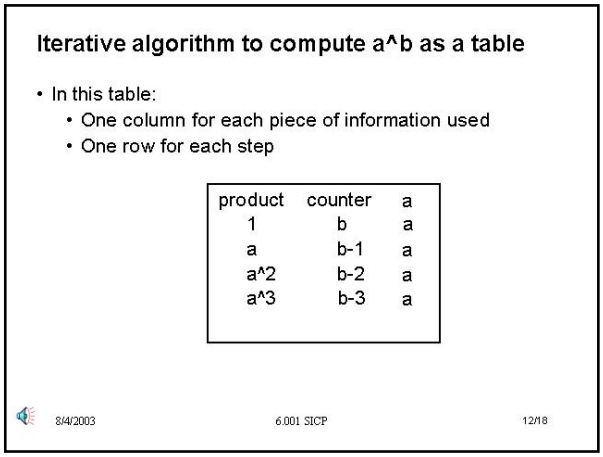
Slide 4.3.10

So recall the idea of our table. We set up one column for each piece of information that we will need to complete a step in the computation, and we use one row for each such step. The idea here is pretty straightforward. We know that `a` to the `b`th power is just `b` successive multiplications by `a`. So we can just do the multiplication, keep track of the product accumulated so far, and how many multiplications we have left. Thus after one step, we will have a product of `a` and `b-1` things left to do.

Slide 4.3.11

After the next step, we will multiply our current product by a and keep track of that new result, plus the fact that we have one less thing to do.



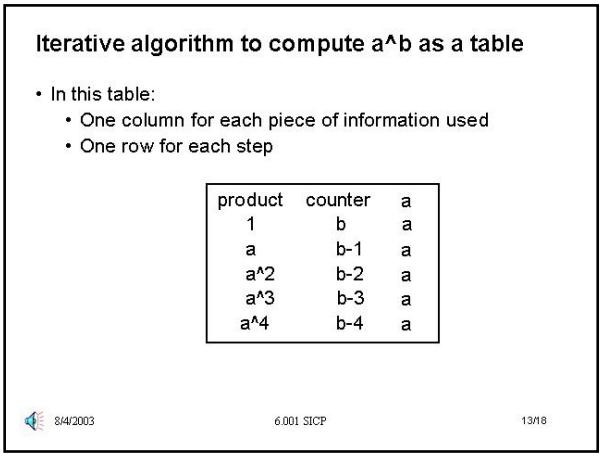


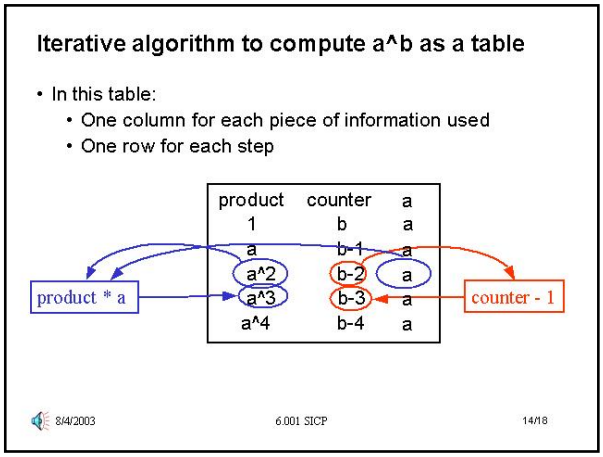
Slide 4.3.12

And this process we can continue until ...

Slide 4.3.13

... we reach a point where we have no further multiplies to do.



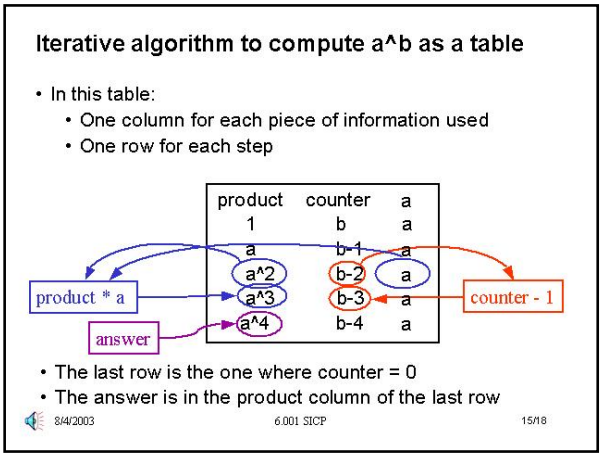


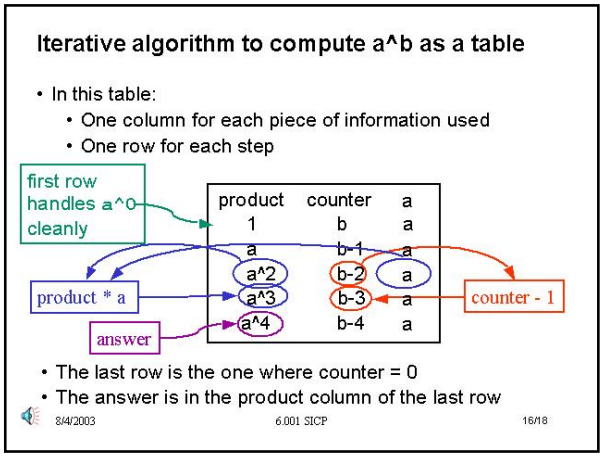
Slide 4.3.14

Now, what are the stages of this computation? To get the next value for the product, we take the current value of the product and the value of a, multiply together, and keep. That updates one of the state variables. To get the next value of counter, we simply subtract 1, since we have done one more of the multiplications. That updates the other state variable.

Slide 4.3.15

We know that when the counter gets down to zero, there are no more multiplications to do, so we are done. And when we get there, the answer is in the product column.



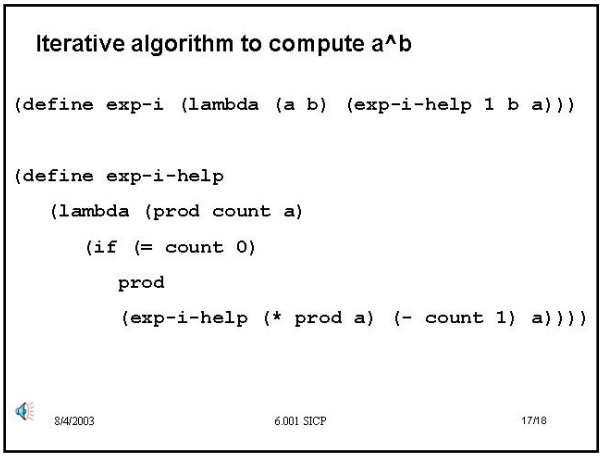


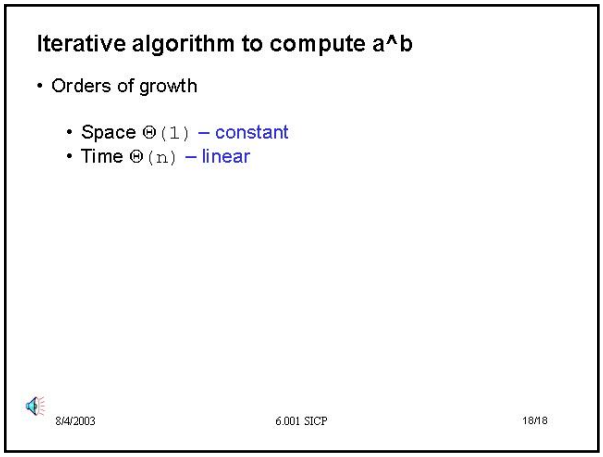
Slide 4.3.16

And finally, we see that the starting point is just that anything to the zeroth power is 1.

Slide 4.3.17

So now we can capture this in a procedure. As with `factorial`, we are going to use a helper procedure. And as in that case, we can see that this procedure checks for a base case, and if there, just returns the value of `prod`. Otherwise, it reduces the computation to a simpler version of the same computation, with a new value for `prod` and a new value for `count`, both obtained by using the update rules we just saw.



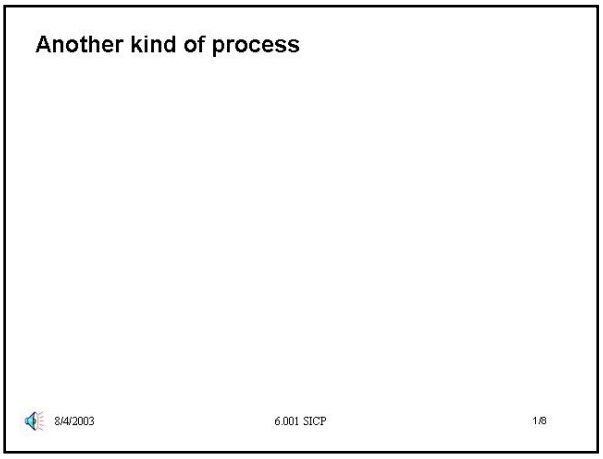


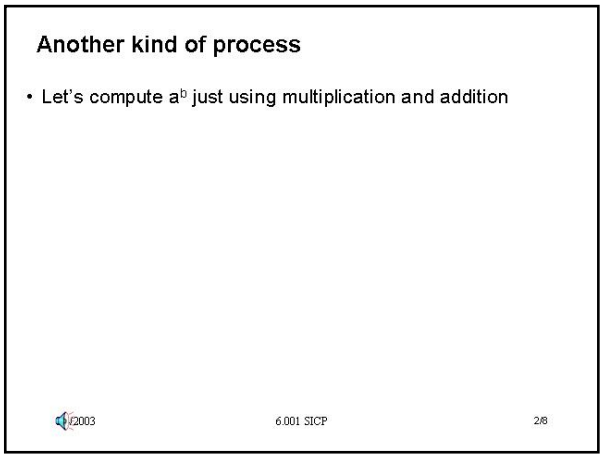
Slide 4.3.18

And what is the order of growth here? In time, this is still linear, as there is one subcomputation to perform for each increment in the size of `b`. In space, however, we again see that there are no deferred operations here, so this is constant. Thus, just as with `factorial`, we see that we can create different procedures to compute exponentiation, with different classes of behaviors.

6.001 Notes: Section 4.4

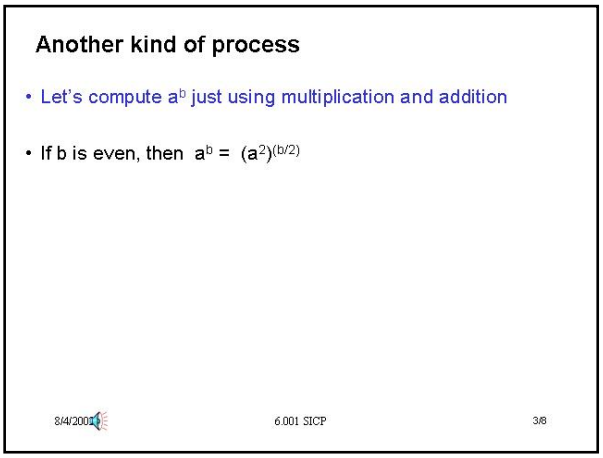
Slide 4.4.1
So now we have seen three very different kinds of procedures: ones that give rise to constant behavior, ones that give rise to linear growth, and ones that give rise to exponential growth. Let's finish up by looking at a fourth kind of procedure, with yet a different kind of behavior and thus a different cost in terms of time and space requirements.

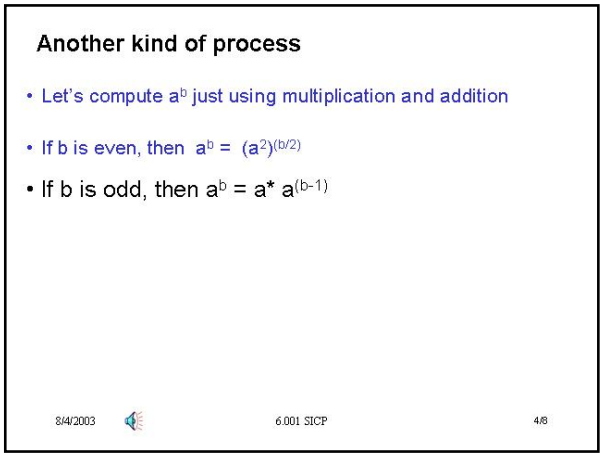




Slide 4.4.2
Here the problem is that we want to compute exponentials, that is a to the power of b, where b is an integer, but where our basic primitive operations are just multiplication, addition, and simple tests.

Slide 4.4.3
As with previous algorithms, the key is to find a way to reduce this problem to a combination of simpler problems, either simpler versions of the same problem or more basic operations. For this problem, here is the trick I can play. If b is an even integer, then I can recognize that raising a to the power of b is the same as first square a, then raising that result to the power of $b/2$. Notice what I have done. Squaring a is just a single multiplication, and reducing b by 2 is a simple operation. But by doing this, I have reduce the size of the problem by a half. I have only $b/2$ things left to consider. Of course, notice that I am relying on b as an even integer here, since in that case $b/2$ is also an integer, and I can use the same machinery to solve this smaller problem.



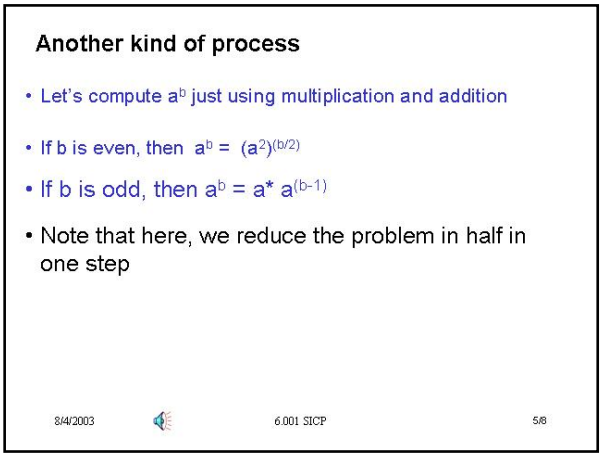


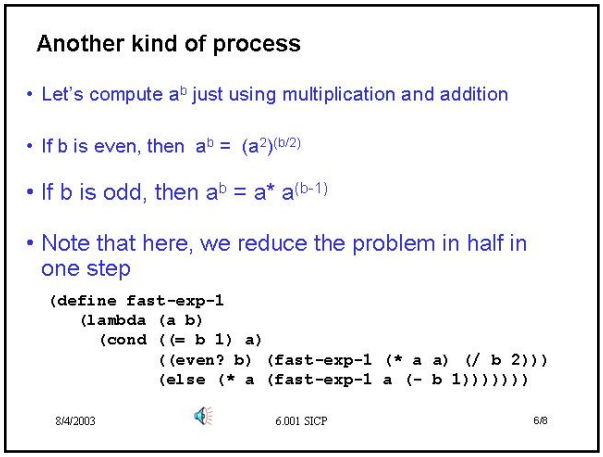
Slide 4.4.4

But what if b is odd? In that case, the same trick won't work; as I would reduce the exponentiation to a problem I don't know how to solve. So instead I can use a different variant on wishful thinking, similar to our factorial case. Here, I can reduce the problem to multiplying a by the result of raising a to the power b-1, that is to a simpler, or smaller, version of the same problem.

Slide 4.4.5

Notice the effect of doing this. If b is odd, then in one step I reduce the problem to the case where b is even, which means in the next step, I reduce the problem size by half. Thus, no matter what value b has, after at most two steps, the problem size is halved, and after at most another two steps, it is halved again, and so on.





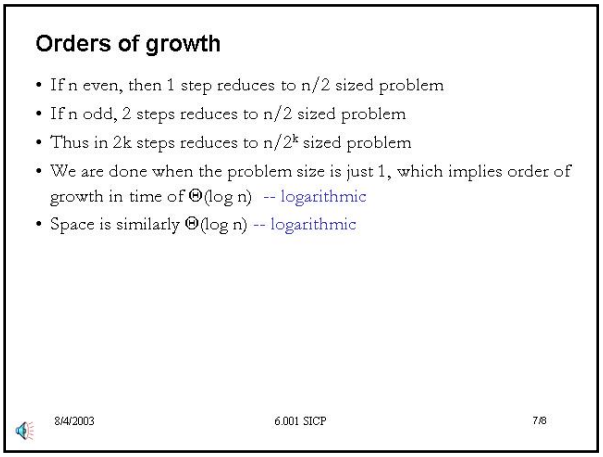
Slide 4.4.6

With those ideas, we can build the procedure shown. The cases clearly reflect the reasoning I just used, using one recursive application of the procedure for even cases, and a different one for odd cases. Of course, there is a base case to terminate the reduction of the problem to simpler versions.

The form of this procedure is a bit different. In one case, it looks like an iterative call (with just a change in the parameters of the procedure), in the other; it looks like a recursive call (with a deferred operation and a reduction in parameters). Thus I expect that there will be some deferred operations in the application of this procedure, but perhaps not as many as in previous examples.

But what happens in terms of time? Since I am reducing the problem in half, I would hope that this leads to better performance than just reducing the problem by 1.

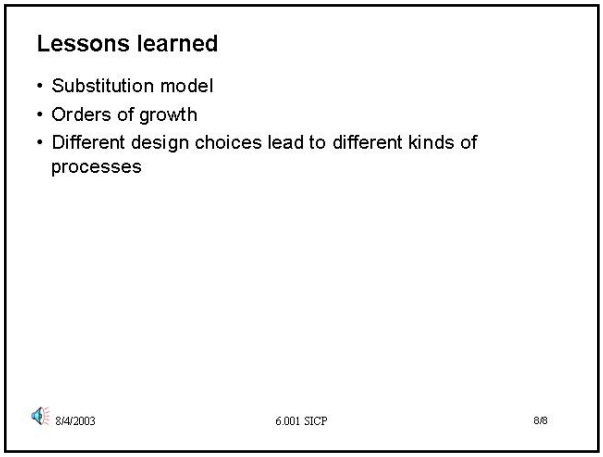
Slide 4.4.7



So let's measure the order of growth of this procedure. Here, **n** measures the size of the problem, which means the size of the argument b. We know that if the argument is even, in one step the problem size is reduced by 2. If the argument is odd, in one step it is reduced by 1, making it even so that in a second step, it is reduced by 2. Thus in at most 2 steps, the problem size is cut in half. After another 2 steps, it is cut in half again, and thus after **2k** steps, the problem is reduced by a factor of **2^k**, or cut in half **k** times.

How do we find the number of times we need to do this? We are done when the problem size is just 1, and that happens when **k = log n**. So this procedure has a different behavior, it is **logarithmic**. This is in fact a very efficient procedure, and to convince yourself of this, try the same trick of seeing how long it would take to solve a problem of different sizes, using one step per second, comparing it to a linear and an exponential process.

The same kind of reasoning will show you that the space requirements also grow logarithmically with the size of the problem.



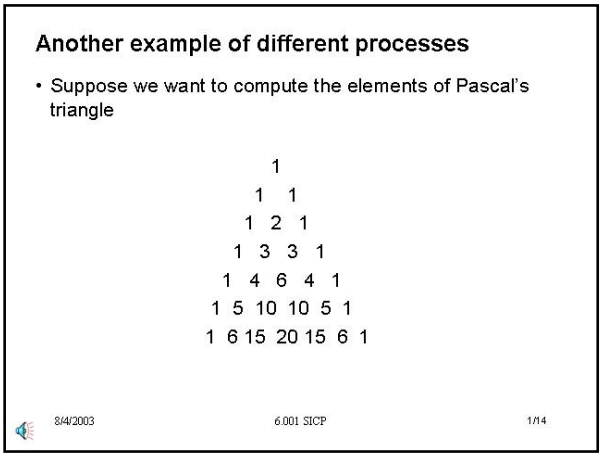
Slide 4.4.8

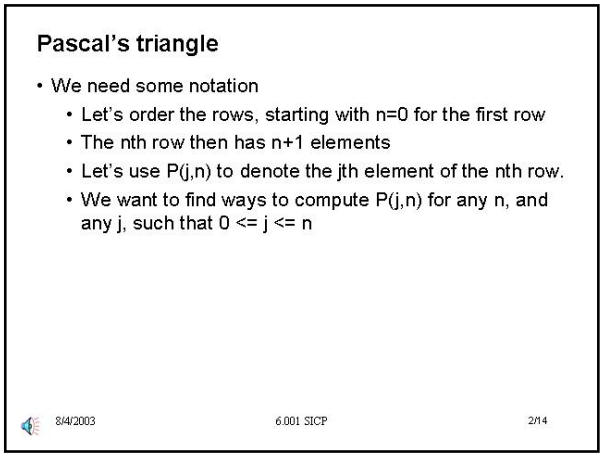
To summarize, we have now seen how our substitution model helps to explain the evolution of a process that occurs when a procedure is applied. Using this model, we have seen that very different kinds of behavior can occur, even for procedures computing the same abstract function, and we saw how to characterize those differences in terms of orders of growth in space and time. Already, you can see that there are different classes of algorithms: constant, linear, exponential, and logarithmic. Part of your task is to learn how to recognize the kinds of procedures that are associated with these different behaviors, and to learn how to use that knowledge in designing efficient procedures for particular problems.

6.001 Notes: Section 4.5

Slide 4.5.1

Let's take one more look at this idea of creating procedures to solve a problem, that have different behaviors in terms of how their underlying process evolves. For a different example, let's look at a mathematical problem known as Pascal's triangle. The elements of Pascal's triangle are shown here. The first row has just a single element, the second row has two elements, the third row has three elements, and so on. Clearly there is a structure to these elements, which we need to understand.



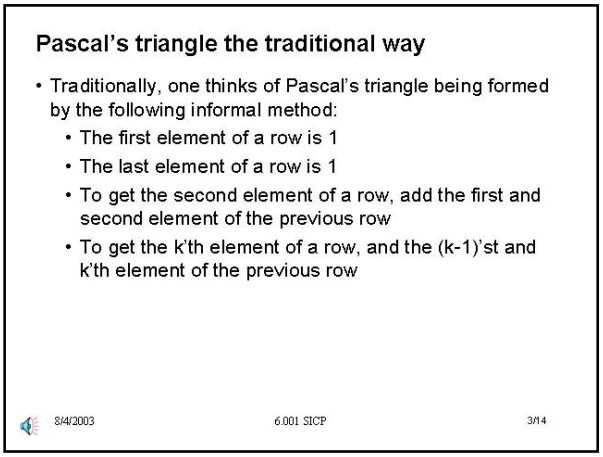


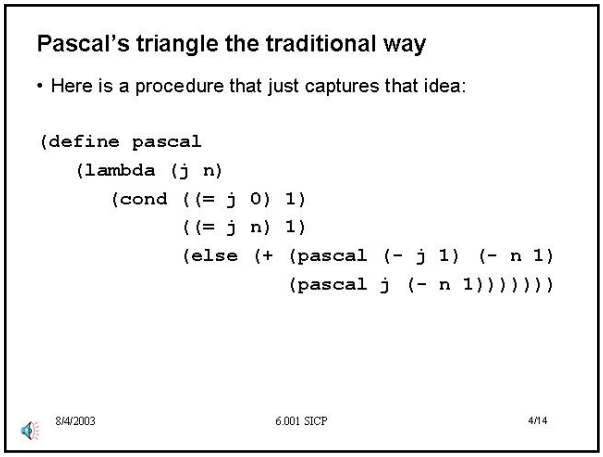
Slide 4.5.2

So, let's structure this problem a bit. Let's order the rows, and enumerate them, labeling the first row n=0, the second row n=1, and so on (we will see that this choice of labeling leads to a cleaner description of the problem). Using this labeling, we also see that the n'th row has n+1 elements in it. Let's use the notation P(j,n) to denote the j'th element of the n'th row. Our goal is to determine how to compute all elements of each row.

Slide 4.5.3

Traditionally, Pascal's triangle is constructed by noting that the first and last element of each row (except the first) is a 1, and by noting that to get any other element in a row, we add the corresponding element of the previous row and its predecessor together. If you use this rule of thumb you can verify that this works by generating the first few rows of the triangle. So we have an informal specification of how to generate the triangle.

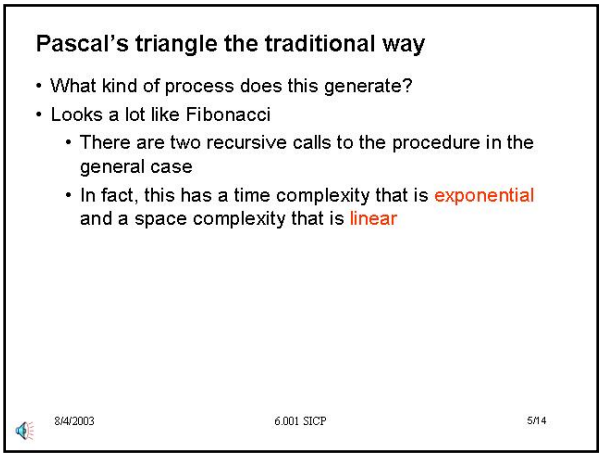


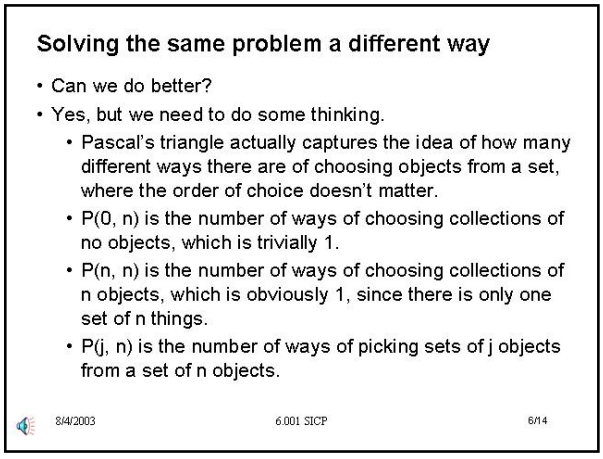


Slide 4.5.4

So we can proceed in a straightforward manner by capturing that idea in a procedure. Note the form: we have two base cases, one for when we are generating the first element of a row, and one for the last element of the row.
Otherwise, we simply rely on smaller versions of the same computation to do the job: we compute the same element of the previous row, and its predecessor, and then add them together. You can verify that this is correct by trying some examples, though clearly the

Slide 4.5.5





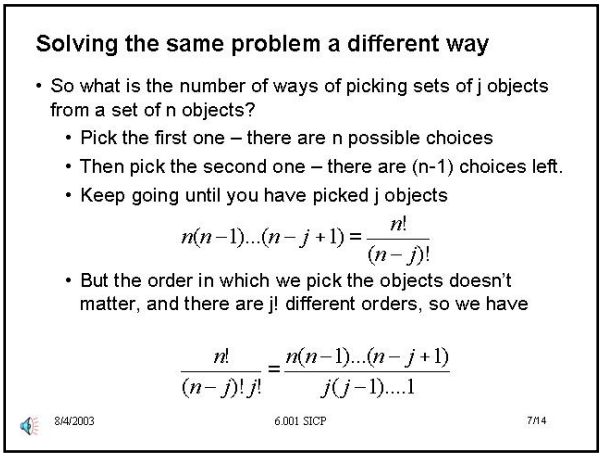
Slide 4.5.6

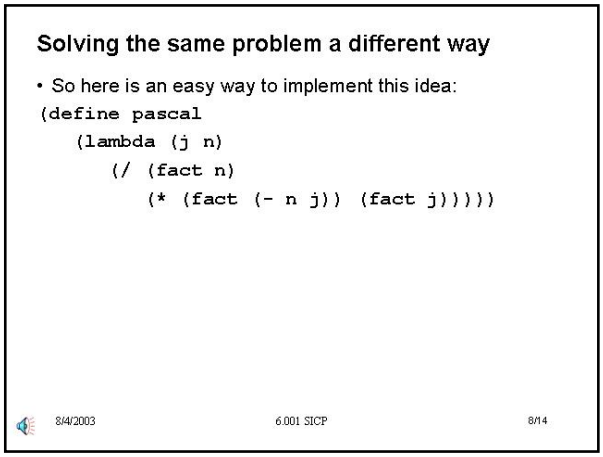
So what kind of process does this procedure engender? Well, it looks a lot like our computation of Fibonacci, and that is a pretty good clue. Just like Fibonacci, there are two recursive calls to smaller versions of the same process at each stage. In fact, a similar analysis will show that this is process that is exponential in time, and linear in space. We have already suggested that exponential algorithms are costly. In the case of Fibonacci, we didn't look for any other way of structuring the problem, but let's try to do better for Pascal.

Slide 4.5.7

To do better, we have to go back to the original problem. A little information from combinatorics tells us that in actuality, Pascal's triangle is capturing the number of different ways of choosing a set of j objects from a set of n objects (this isn't obvious, by the way, so just accept this as a fact). Thus, the first element of a row is the number of ways of picking no objects, which is by definition 1. The last element of a row is the number of ways of picking a set of n objects from a set of n objects. Since the order in which we pick the objects doesn't matter, there is only one size n subset of a set of size n, hence this element is 1.

The general case is the number of different subsets of size j that can be created out of a set of size n.





Slide 4.5.8

So what is that number?

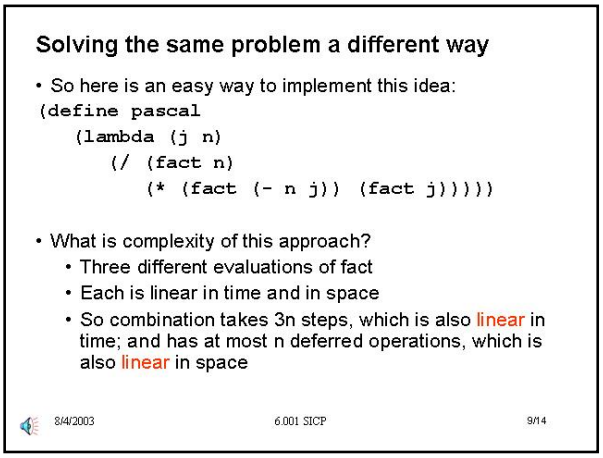
Well, we can pick the first element by selecting any of the n elements of the set.

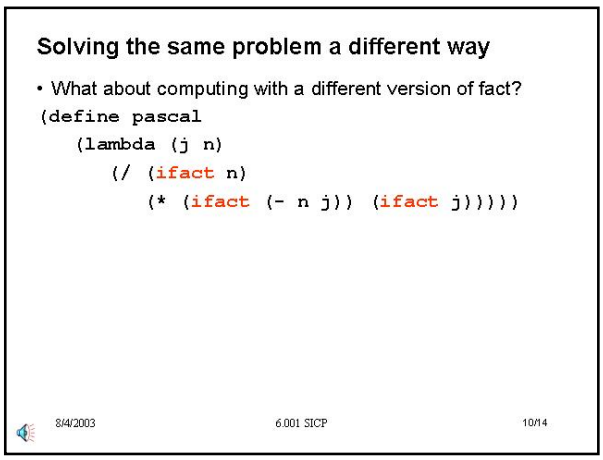
The second element has n-1 possibilities, and so on, until we have picked out j objects. That product we can represent as a fraction of two factorial products, as you can see.

Now, we are not quite done, because we said the order in which we pick the objects doesn't matter. Using the method we just described, we could pick the same set of j objects in j! different orderings, so to get the number of distinct subsets of size j, we factor this out, as shown.

Slide 4.5.9

Now this leads to a straightforward way to compute Pascal. We can just rely on the factorial code that we created earlier. Here we use factorial three times, once for the numerator, and twice in the denominator. Notice how our procedural abstraction nicely isolates the details of fact from its use in this case, and the code here cleanly expresses the idea of computing Pascal based on factorial.





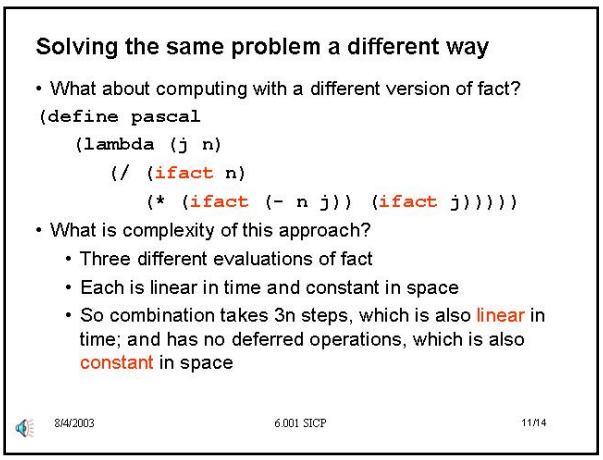
Slide 4.5.10

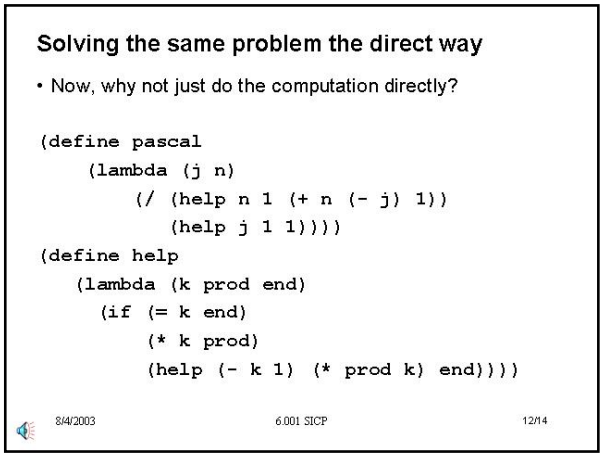
So do we do any better with this version of Pascal?

Sure! We know that this version of fact is linear. Our Pascal implementation uses fact three different evaluations of fact, but this is still linear in time, and a similar analysis lets us conclude that this is linear in space as well. While it might take three times as long to run than just running fact, what we are concerned with is the general order of growth; or how these processes' computational needs change with changing problem size, and both are linear in that change.

Slide 4.5.11

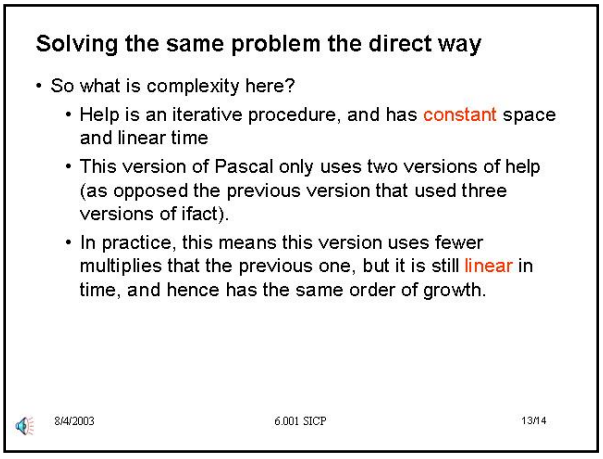
Well, earlier we saw that there were different ways of computing factorial. Suppose we use the iterative version, instead of the recursive one?

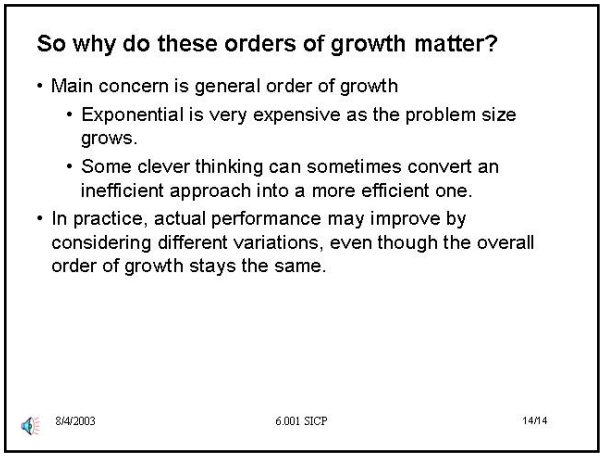




Slide 4.5.12
The same effect takes place. Now we have a version of Pascal that is linear in time and constant in space, relying on the fact that factorial under this version is also linear in time and constant in space.

Slide 4.5.13
Finally, if the use of an iterative factorial procedure gives us better performance, why not just do this computation directly as well? By that, we mean just compute the two products: one for the numerator and one for the denominator, and save those extra multiplications that we are doing in the extra call to factorial. After all, we are just computing a product of terms that we know we are simply going to factor out.





Slide 4.5.14
We can do the same analysis for this version. Our help procedure is clearly constant in space, and linear in time. Our version of Pascal uses it twice, so while it does take less actual time than the previous version, it still has the same general behavior: linear in time and constant in space. Thus, in practical terms, this may be the best version, but theoretically it has the same class of behavior as our previous version.

Slide 4.5.15
And that leads to our concluding point. First, we stress that the same problem may have many different solutions, and that these solutions may have very different computational behaviors. Some problems are inherently exponential in nature. Others may have straightforward solutions that have exponential behavior, but often some additional thought can lead to more efficient solutions. Part of our goal is to get you to recognize different classes of behavior, and to learn how to use the properties of standard algorithms to help you design efficient solutions to new problems.