This Lecture

- Substitution model
- An example using the substitution model
- Designing recursive procedures
- Designing iterative procedures
- Proving that our code works

Substitution model

- a way to figure out what happens during evaluation
  - not really what happens in the computer

Rules of substitution model:

- If expression is self-evaluating (e.g. a number), just return value
- If expression is a name, replace with value associated with that name
- If expression is a lambda, create procedure and return
- If expression is special form (e.g. if) follow specific rules for evaluating subexpressions
- If expression is a compound expression
  - Evaluate subexpressions in any order
  - If first subexpression is primitive (or built-in) procedure, just apply it to values of other subexpressions
  - If first subexpression is compound procedure (created by lambda), substitute value of each subexpression for corresponding procedure parameter in body of procedure, then repeat on body

Substitution model – a simple example

```
(define square (lambda (x) (* x x)))
```

1. (square 4)
   1. Square → [procedure (x) (* x x)]
   2. 4 → 4
2. (* 4 4)
3. 16

Substitution model details

```
(define square (lambda (x) (* x x)))
(define average (lambda (x y) (/ (+ x y) 2)))
```

```
(average 5 (square 3))
(average 5 (* 3 3))
(average 5 9)
```

first evaluate operands,
then substitute (applicative order)

```
(/ (+ 5 9) 2)
(/ 14 2)
7
```

if operator is a primitive procedure,
replace by result of operation

A less trivial procedure: factorial

- Compute n factorial, defined as $n! = n(n-1)(n-2)(n-3)...1$
- How can we capture this in a procedure, using the idea of finding a common pattern?

How to design recursive algorithms

- follow the general pattern:
  1. wishful thinking
  2. decompose the problem
  3. identify non-decomposable (smallest) problems

1. Wishful thinking

- Assume the desired procedure exists.
- want to implement fact? OK, assume it exists.
- BUT, only solves a smaller version of the problem.

Note – this is really reducing a problem to a common pattern, in this case that solving a bigger problem involves the same pattern in a smaller problem

2. Decompose the problem

- Solve a problem by
  1. solve a smaller instance (using wishful thinking)
  2. convert that solution to the desired solution

- Step 2 requires creativity!
  - Must design the strategy before coding.
  - n! = n(n-1)(n-2)... = n[(n-1)(n-2)...] = n * (n-1)!
  - solve the smaller instance, multiply it by n to get solution

```
(define fact
    (lambda (n) (* n (fact (- n 1)))))
```

3. Identify non-decomposable problems

- Decomposing not enough by itself
- Must identify the "smallest" problems and solve directly

- Define 1! = 1

```
(define fact
     (lambda (n)
          (if (= n 1) 1
               (* n (fact (- n 1))))))
```

General form of recursive algorithms

- test, base case, recursive case

```
(define fact
  (lambda (n)
    (if (= n 1)          ; test for base case
      1                  ; base case
      (* n (fact (- n 1)) ; recursive case
)))
```

- base case: smallest (non-decomposable) problem
- recursive case: larger (decomposable) problem

Summary of recursive processes

- Design a recursive algorithm by
  1. wishful thinking
  2. decompose the problem
  3. identify non-decomposable (smallest) problems

- Recursive algorithms have
  1. test
  2. recursive case
  3. base case

```
(define fact(lambda (n)
  (if (= n 1)1(* n (fact (- n 1))))))

(fact 3)
(if (= 3 1) 1 (* 3 (fact (- 3 1))))
(if #f 1 (* 3 (fact (- 3 1))))
(* 3 (fact (- 3 1)))
(* 3 (fact 2))
(* 3 (if (= 2 1) 1 (* 2 (fact (- 2 1)))))
(* 3 (if #f 1 (* 2 (fact (- 2 1)))))
(* 3 (* 2 (fact (- 2 1))))
(* 3 (* 2 (fact 1)))
(* 3 (* 2 (if (= 1 1) 1 (* 1 (fact (- 1 1))))))
(* 3 (* 2 (if #t 1 (* 1 (fact (- 1 1))))))
(* 3 (* 2 1))
(* 3 2)
6
```

The fact procedure is a recursive algorithm

- A recursive algorithm:
  - In the substitution model, the expression keeps growing

```
(fact 3)
(* 3 (fact 2))
(* 3 (* 2 (fact 1)))
```

  - Other ways to identify will be described next time

Iterative algorithms

- In a recursive algorithm, bigger operands => more space

```
(define fact (lambda (n)
  (if (= n 1) 1
      (* n (fact (- n 1))))))
```

```
(fact 4)
(* 4 (fact 3))
(* 4 (* 3 (fact 2)))
(* 4 (* 3 (* 2 (fact 1))))
(* 4 (* 3 (* 2 1)))
...
24
```

- An iterative algorithm uses constant space.

- We can implement an iterative algorithm or process with a recursive procedure in Scheme – no loss of efficiency when code is compiled.

Intuition for iterative factorial

- same as you would do if calculating 4! by hand:

  1. multiply 4 by 3 gives 12
  2. multiply 12 by 2 gives 24
  3. multiply 24 by 1 gives 24

- At each step, only need to remember:

  previous product, next multiplier

- Therefore, constant space

- Because multiplication is associative and commutative:

  1. multiply 1 by 2 gives 2
  2. multiply 2 by 3 gives 6
  3. multiply 6 by 4 gives 24

Iterative algorithm to compute 4! as a table

- In this table:
  - One column for each piece of information used
  - One row for each step

product	counter
1	2
2	3
6	4

Iterative algorithm to compute 4! as a table

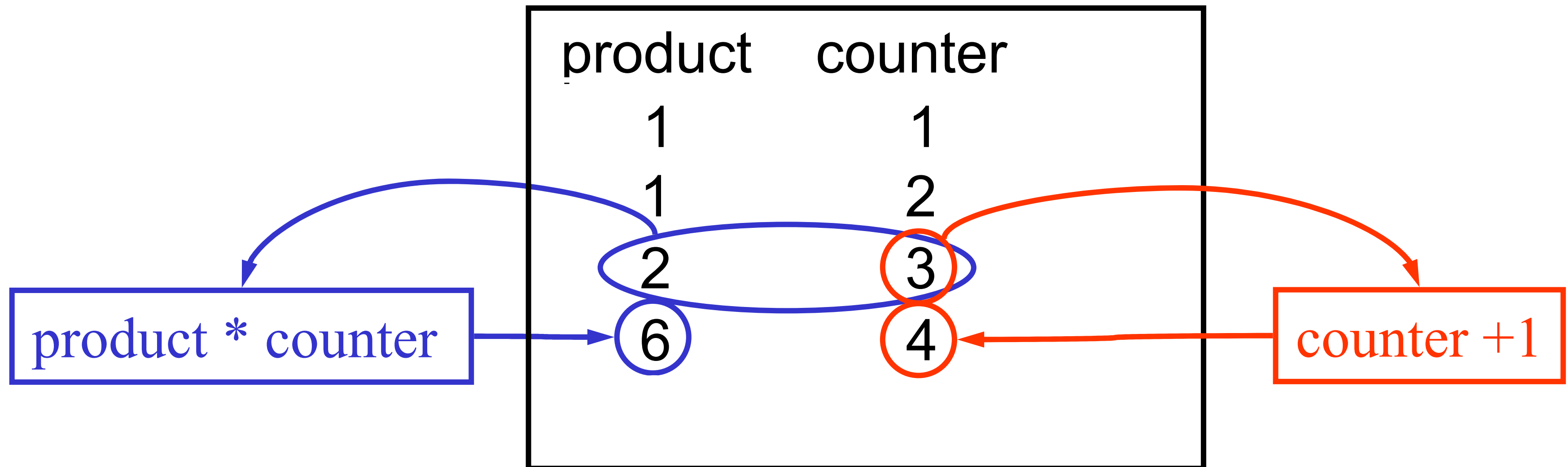
- In this table:
  - One column for each piece of information used
  - One row for each step

product	counter
1	1
1	2
2	3
6	4

product * counter

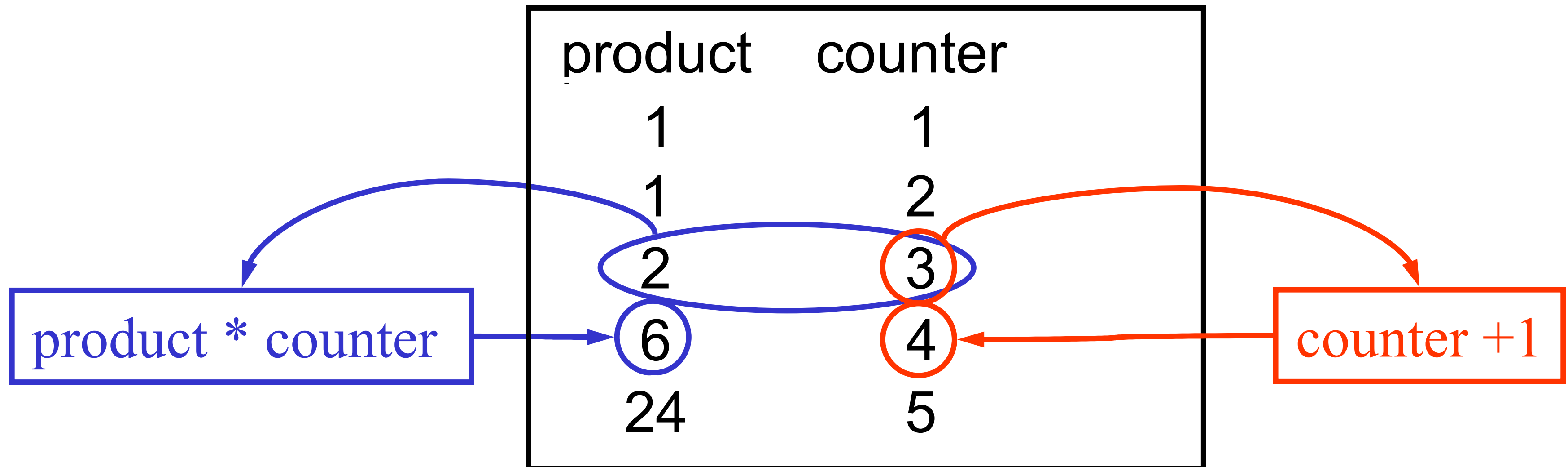
Iterative algorithm to compute 4! as a table

- In this table:
  - One column for each piece of information used
  - One row for each step



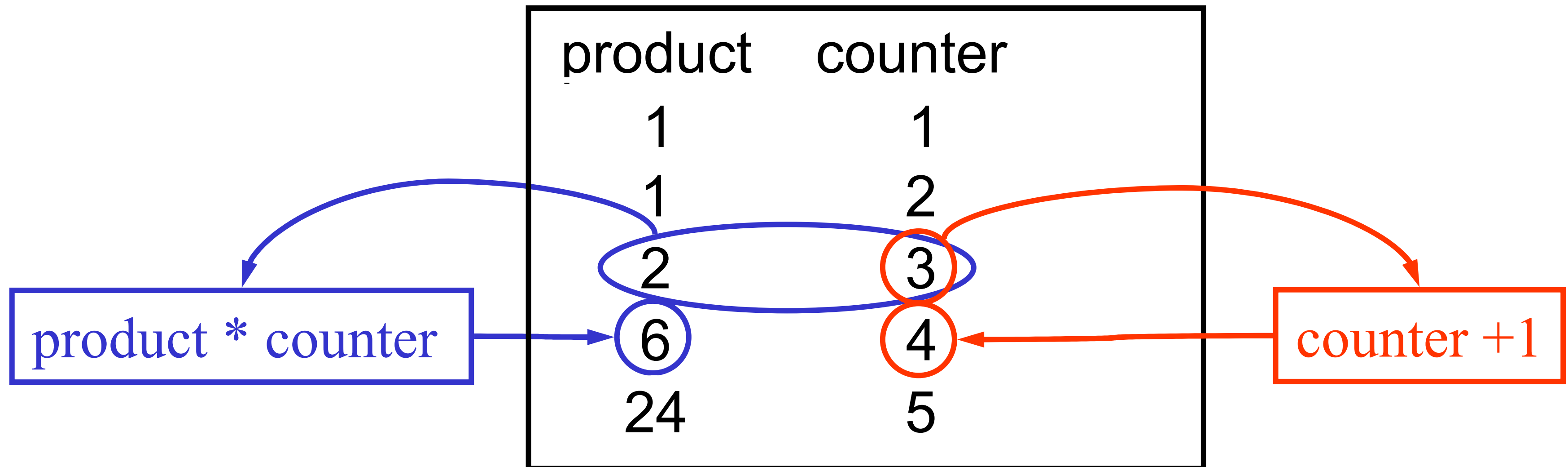
Iterative algorithm to compute 4! as a table

- In this table:
  - One column for each piece of information used
  - One row for each step



Iterative algorithm to compute 4! as a table

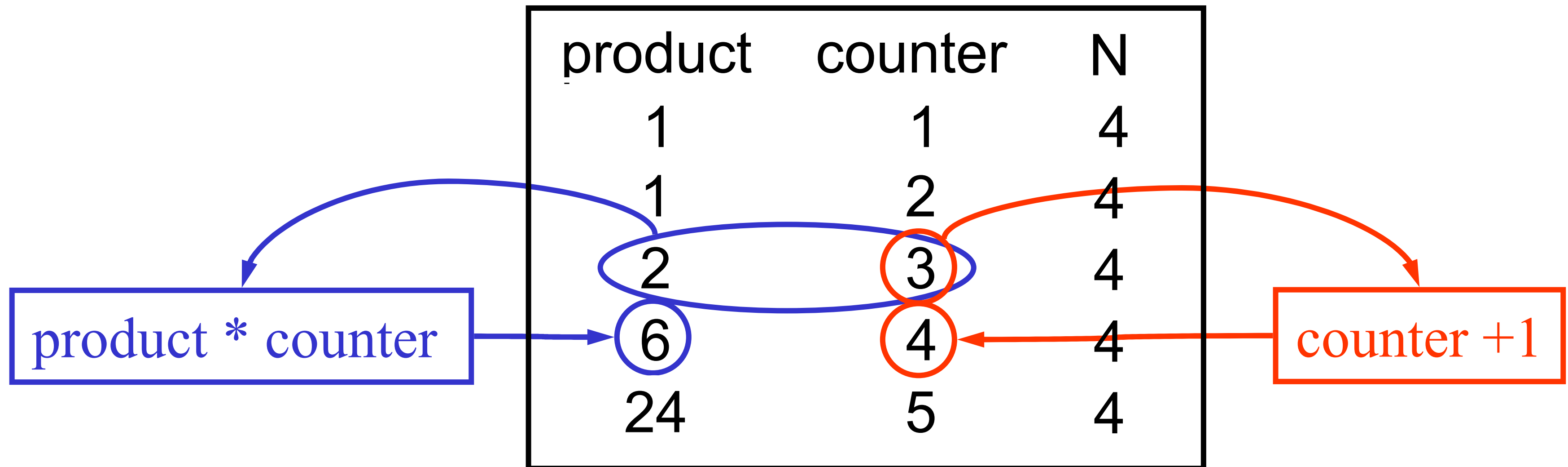
- In this table:
  - One column for each piece of information used
  - One row for each step



- The last row is the one where counter > n

Iterative algorithm to compute 4! as a table

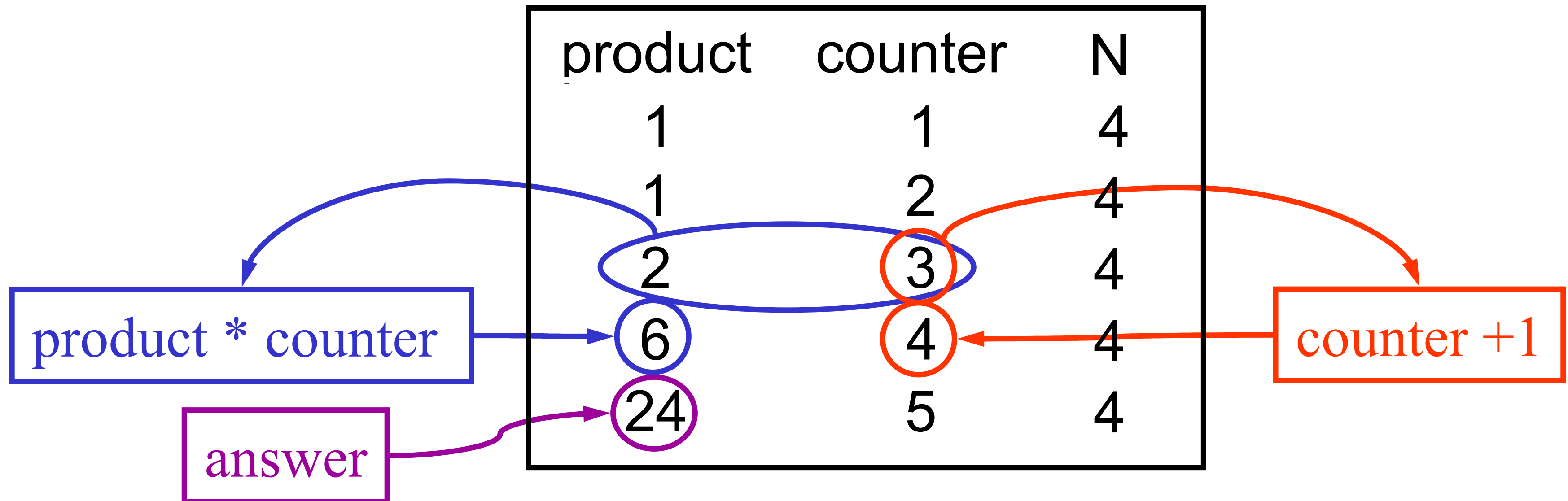
- In this table:
  - One column for each piece of information used
  - One row for each step



- The last row is the one where counter > n

Iterative algorithm to compute 4! as a table

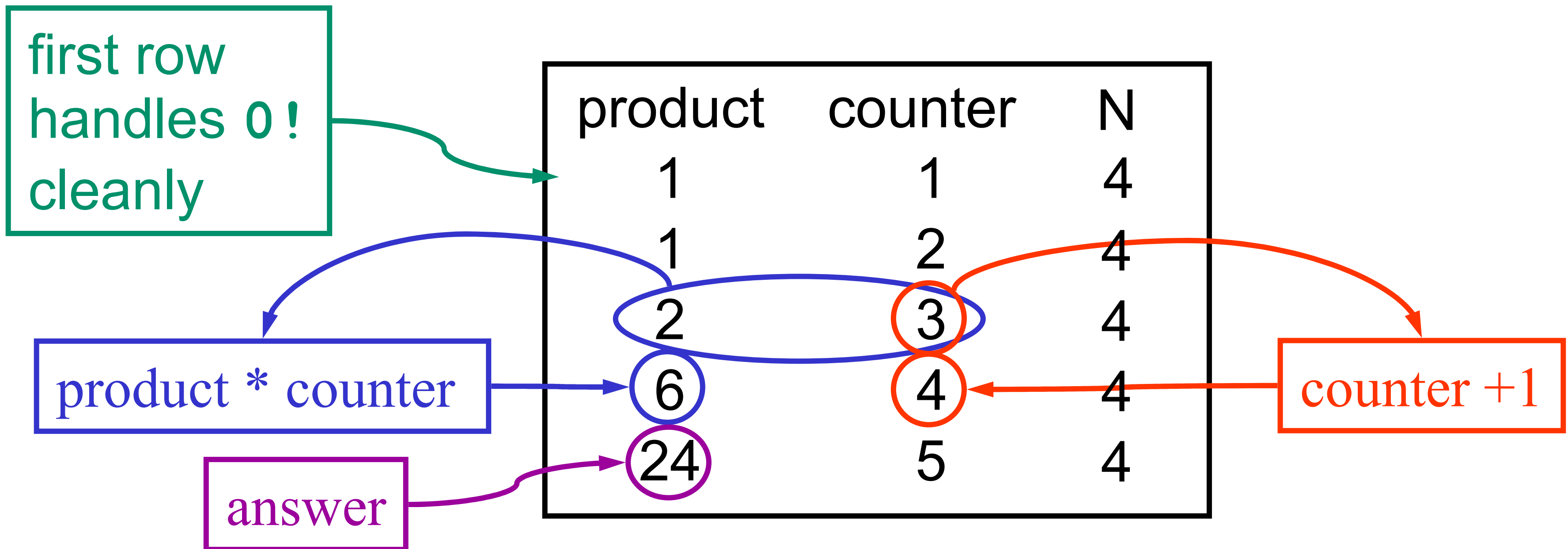
- In this table:
  - One column for each piece of information used
  - One row for each step



- The last row is the one where counter > n
- The answer is in the product column of the last row

Iterative algorithm to compute 4! as a table

- In this table:
  - One column for each piece of information used
  - One row for each step



- The last row is the one where counter > n
- The answer is in the product column of the last row

Iterative factorial in scheme

```
(define ifact (lambda (n) (ifact-helper 1 1 n)))

  (define ifact-helper (lambda (product counter n)

    (if    (> counter n)

           product

           (ifact-helper (* product counter) (+ counter 1) n))))
```

initial
row of table

compute next row of table

answer is in product column of last row

at last row when counter > n

Partial trace for `(ifact 4)`

```
(define ifact-helper (lambda (product count n)
      (if (> count n) product
          (ifact-helper (* product count)
                        (+ count 1) n))))
```

```
(ifact 4)
(ifact-helper 1 1 4)
(if (> 1 4) 1 (ifact-helper (* 1 1) (+ 1 1) 4))
(ifact-helper 1 2 4)
(if (> 2 4) 1 (ifact-helper (* 1 2) (+ 2 1) 4))
(ifact-helper 2 3 4)
(if (> 3 4) 2 (ifact-helper (* 2 3) (+ 3 1) 4))
(ifact-helper 6 4 4)
(if (> 4 4) 6 (ifact-helper (* 6 4) (+ 4 1) 4))
(ifact-helper 24 5 4)
(if (> 5 4) 24 (ifact-helper (* 24 5) (+ 5 1) 4))
24
```

Iterative = no pending operations when procedure calls itself

- Recursive factorial:

```
(define fact (lambda (n)
      (if (= n 1) 1
          (* n (fact (- n 1)) )
      )))
```

- `(fact 4)`

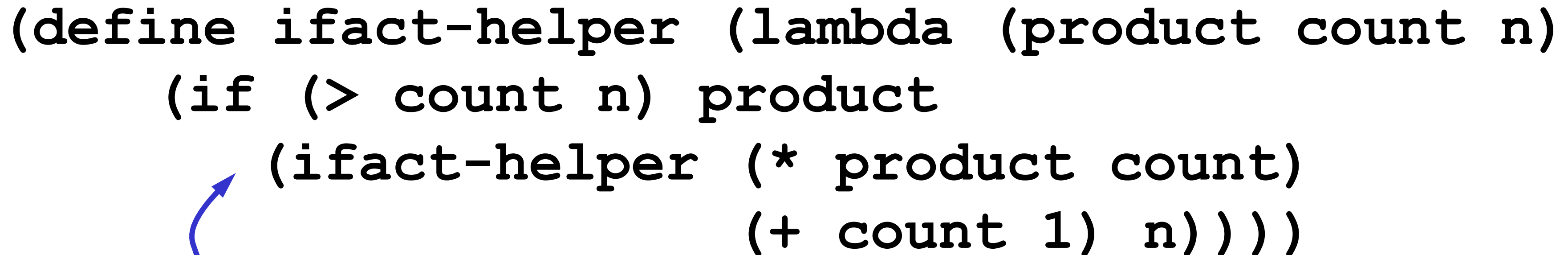
```
(* 4 (fact 3))
(* 4 (* 3 (fact 2)))
(* 4 (* 3 (* 2 (fact 1))))
```

- Pending ops make the expression grow continuously

Iterative = no pending operations

- Iterative factorial:

```
(define ifact-helper (lambda (product count n)
    (if (> count n) product
        (ifact-helper (* product count)
                      (+ count 1) n))))
```



no pending operations

- `(ifact-helper 1 1 4)`
  `(ifact-helper 1 2 4)`
  `(ifact-helper 2 3 4)`
  `(ifact-helper 6 4 4)`
  `(ifact-helper 24 5 4)`

- Fixed size because no pending operations

Summary of iterative processes

- Iterative algorithms have constant space
- How to develop an iterative algorithm
  - figure out a way to accumulate partial answers
  - write out a table to analyze precisely:
    - initialization of first row
    - update rules for other rows
    - how to know when to stop
  - translate rules into scheme code

- Iterative algorithms have no pending operations when the procedure calls itself

Why is our code correct?

- How do we know that our code will always work?
  - **Proof by authority** – someone with whom we dare not disagree says it is right!
  - For example – me!
  - **Proof by statistics** – we try enough examples to convince ourselves that it will always work!
  - E.g. keep trying
  - **Proof by faith –** we really, really, really believe that we always write correct code!
  - E.g. the Pset is due in 5 minutes and I don't have time
  - **Formal proof –** we break down and use mathematical logic to determine that code is correct.

Formal Proof

- A **formal proof** of a **proposition** is a chain of **logical deductions** leading to the proposition from a base set of **axioms**.
- A **proposition** is a statement that is either true or false.
  - Atomic propositions: simple statements of veracity
  - Compound propositions:
    - Conjunction (and): $P \wedge Q$
    - Disjunction (or): $P \vee Q$
    - Negation (not): $\neg P$
    - Implication (if P, then Q): $(P \rightarrow Q)$
    - Equivalence (P if and only if Q): $(P \leftrightarrow Q)$

Truth assignments for propositions

- A **truth assignment** is a function that maps each variable in a formula to True or False

P	Q	P and Q	P or Q	Not P	If P, then Q	P iff Q
F	F	F	F	T	T	T
F	T	F	T	T	T	F
T	F	F	T	F	F	F
T	T	T	T	F	T	T

Proof systems

- Given a set of propositions, we can construct complex statements by combinations.
- We can use **inference rules** to combine **axioms** (propositions that are assumed to be true) and true propositions to construct more true propositions.
- Example: modus ponens

$$P$$

$$P \rightarrow Q$$

$$\therefore Q$$

- Example: modus tollens

$$P \rightarrow Q$$

$$\neg Q$$

$$\therefore \neg P$$

Predicate logic

- We need to state propositions that will hold true for a range of values or arguments – a **predicate** is a proposition with variables. Example: P(x, y) could be the predicate "x*x=y"
- Predicates are defined over a **universe** (or set of values for the variables).
- **Quantifiers** can specify conditions on predicates
  - If predicate is true for all possible values in the universe

    $$\forall x Q(x)$$

  - If predicate is true for at least one value in the universe

    $$\exists x Q(x)$$

Proof by induction

- A very powerful tool in predicate logic is proof by induction:

$$P(0)$$
$$\forall n : P(n) \rightarrow P(n+1)$$
$$\therefore \forall n : P(n)$$

- Informally: If you can show that proposition is true for case of n=0, and you can show that if the proposition is true for some legal value of n, then it follows that it is true for n+1, then you can conclude that the proposition is true for all legal values of n

An example of proof by induction

$$P(n): \sum_{i=0}^{n} 2^i = 2^{n+1} - 1$$

e.g.,

n=0:	1	= 1 = 2-1
n=1:	1+2	= 3 = 4-1
n=2:	1+2+3	= 7 = 8-1

Base case: $$n = 0: 2^0 = 2^1 - 1$$

Inductive step: $$\sum_{i=0}^{n+1} 2^i = \sum_{i=0}^{n} 2^i + 2^{n+1}$$

$$= 2^{n+1} - 1 + 2^{n+1} = 2^{n+2} - 1$$

Stages in proof by induction

1. Define the predicate P(n), including what the variable denotes and the universe over which it applies (the induction hypothesis).
2. Prove that P(0) is true (the base case).
3. Prove that P(n) implies P(n+1) for all n. Do this by assuming that P(n) is true, while you try to prove that P(n+1) is true (the inductive step).
4. Conclude that P(n) is true for all n by the principle of induction.

Back to our factorial case.

P(n): our recursive procedure for fact correctly computes n! for all integer values of n, starting at 1.

```
(define fact
        (lambda (n)
           (if (= n 1)
               1
               (* n (fact (- n 1))))))
```

Fact works by induction

Base case: does this work when n=1? (Note that we need to adjust the base case to reflect the fact that our universe includes only the positive integers)

Sure – the IF statement guarantees that in this case we only evaluate the consequent expression: thus we return 1, which is 1!

```
(define fact
        (lambda (n)
           (if (= n 1)
               1
               (* n (fact (- n 1))))))
```

Fact works by induction

Inductive step: We can assume that our code works correctly for some value of n, we want to use this to show that the code then works correctly for n+1.

- In general case, value of expression (fact (+ n 1)) will reduce by our substitution model to
  - `(* (+ n 1)(fact n))`
- Our substitution model says to get the values of the subexpressions: * and (+ n 1) are easy. By induction, the remaining subexpression returns the value of n!
- Hence the value of the expression is (n+1)n! or (n+1)!
- By induction, this code will always compute what we expected, provided the input is in the right range (n > 0).

Lessons learned

Induction provides the basis for supporting recursive procedure definitions

In designing procedures, we should rely on the same thought process

- Find the base case, and create solution
- Determine how to reduce to a simpler version of same problem, plus some additional operations
- Assume code will work for simpler problem, and design solution to extended problem