Goals:

- Define
  - computational problem
  - instance of a computational problem
  - algorithm
  - pseudocode
- Briefly discuss
  - recursive procedure
  - big-O notation

Informal Definitions:

A computational problem is a description of (1) the form of the inputs (i.e., data), (2) the form of the outputs, and (3) the required relationship among them.
An instance of a computational problem is a set of inputs having the proper form for that computational problem.

For example, the problem of predicting genes in a genomic DNA sequence is a computational problem. Predicting genes on human chromosome 22 is an instance of that problem.

More Informal Definitions:

An algorithm for a given computational problem is a description of computational steps that solve any instance of that problem. The description can involve pictures, and it is intended to be read by a human.

Pseudocode is a very precise statement of an algorithm that explains low-level details. It is intended to be read by a human, not by a computer.

A computer program fills in even more details than does pseudocode, and is generally read by a computer, not a human.

Example 1

Problem: Given an integer n, print the number $1 + 2 + \ldots + n$.

Instance: Compute the sum of the first 15 integers.

Pseudocode:

SumIntegers(n)

```
1  sum ← 0
2  for i ← 1 to n
3  do
4        sum ← sum + i
5  return  sum
```

Example 2

Problem: Given an integer b, print the smallest integer i such that the sum of the first i integers is b or larger.

Instance: Find the smallest i such that $1 + 2 + \ldots + i \geq 25$. (The answer is 7.)

Pseudocode:

```
AddUntil(b)
1  i ← 0
2  total ← 0
3  while total < b
4  do
5       i ← i + 1
6       total ← total + i
7  return i
```

The (US) Change Problem

Problem:

Data: A positive dollar-cents amount.

Output: The smallest number of coins whose sum is that amount.

Instance: data = \$0.77 (output = 3 quarters and 2 pennies — assuming no half-dollars are available)

Algorithm: ???

The (US) Change Problem (continued)

Algorithm: Do the following, until the owed amount is 0: return the largest coin that does not exceed the owed amount, and subtract the value of that coin from the owed amount.

Pseudocode:

UsChange(M)

```
1  while M > 0
2  do c ← Largest coin ≤ M in value
3        Give that coin to the customer
4        M ← M − c
```

Thought Problem

If you go to a country with some other coin denominations, will the same basic algorithm work?

(Generally, with an algorithm we need to think about the assumptions that must be satisfied for the algorithm to work properly.)

Thought Problem (continued)

Suppose the available denominations are 4, 3 and 1. What is the best change for 6?

Recursive Procedure

A procedure is recursive if it calls itself. That is, for all but very small instances, it solves an instance by using solutions that it computes by calling itself on smaller instances.

Clearly, we need an example.

Towers of Hanoi Problem.

Input: Three wooden pegs, with n disks on the one of the pegs, no two disks of the same size and stacked smaller-on-larger.

Output: A series of moves, one disk at a time and never putting a disk on top of a smaller disk, that moves the disks to a specified peg. (See explanation at blackboard.)

Towers of Hanoi (continued)

HanoiTowers$(n, fromPeg, toPeg)$

1 **if** $n = 1$
2 **then** Move the disk from $fromPeg$ to $toPeg$
3 **else** HanoiTowers$(n-1, fromPeg, thirdPeg)$
4 Move the disk from $fromPeg$ to $toPeg$
5 HanoiTowers$(n-1, thirdPeg, toPeg)$

Towers of Hanoi, take 3

Can we think of a non-recursive algorithm for this problem? Doing this exercise will illustrate that a computational problem can frequently be solve by several completely different methods – not only where the pseudocode is different, but the way to think about the methods are completely different.

Hint: We only need to consider sequences of moves with the property that they never reach a configuration that can be reached in fewer moves. How many such sequences of moves are there?

Towers of Hanoi, take 4

Some reasoning shows that there are essentially only two sequences of moves. You can move the smallest disk either clockwise or counter-clockwise (thinking of the pegs as being around a circle). Every other move is by the small disk, and then there is no choice for the other move.

Fibonacci Problem

Input: An integer n.

Output: The nth Fibonacci number, F_n, where $F_1 = F_2 = 1$ and $F_n = F_{n-1} + F_{n-2}$ if $n < 2$.

Recursive procedure:

```
RecursiveFibonacci(n)
1  if n = 1 or n = 2
2     then return 1
3     else a ← RecursiveFibonacci(n − 1)
4          b ← RecursiveFibonacci(n − 2)
5          return a + b
```

Fibonacci Problem (continued)

Iterative algorithm:

FIBONACCI(n)

```
1  if n = 1 or n = 2
2     then return 1
3     else F_1 ← 1
4          F_2 ← 1
5          for i ← 3 to n
6          do F_i ← F_{i−1} + F_{i−2}
7          return F_n
```

Which is the faster way to compute the nth Fibonacci number — the recursive procedure or the iterative one?

Big-O Notation

Let $P(n)$ be a computational procedure (e.g., algorithm or pseudocode) whose execution time grows with the input parameter n. Let $f(n)$ be any function of n. That is, for any integer n, $f(n)$ is an integer. For instance $f(n) = 2n$. Or $f(n) = n^2$. Or $f(n) = 2^n$.

We say that $P(n)$ runs in time $O(f(n))$ if there is a constant c (which can depend on the details of how P is executed) such that the running time of $P(n)$ is less than $c \times f(n)$. Equivalently, we will sometimes say that $P(n)$ runs in time proportional to $f(n)$.

In big-O notation, what is the execution time of:

FIBONACCI(n)

```
1  if n = 1 or n = 2
2     then return 1
3     else F_1 ← 1
4          F_2 ← 1
5          for i ← 3 to n
6          do F_i ← F_{i−1} + F_{i−2}
7          return F_n
```

In big-O notation, what is the execution time of:

```
RecursiveFibonacci(n)
1  if n = 1 or n = 2
2     then return 1
3     else a ← RecursiveFibonacci(n − 1)
4          b ← RecursiveFibonacci(n − 2)
5          return a + b
```

How much worse is $O(2^n)$ than $O(n)$?

For $n = 20$, 2^n exceeds one million. For $n = 30$, 2^n exceeds one billion.