Definition: Algorithm

Example(s):
The Framework

1. ______________ — means that the solution can be described by an algorithm
   (a) ______________ — the algorithm is efficient
   (b) ______________ — no efficient solution algorithm is known

2. ______________ — no algorithm will ever describe the solution

Algorithm Characteristics (1 / 2)

Six Desirable algorithm characteristics:

1. Input —

2. Output —

3. Generality —
Algorithm Characteristics (2 / 2)

4. Definiteness —

5. Correctness —

6. Finiteness —

Example: Tooth-brushing Algorithm

1. Grab the toothpaste
2. Uncap the toothpaste
3. Grab your toothbrush
4. Squeeze toothpaste onto your toothbrush
5. Brush your teeth
Example: Decimal to Base X Conversion

**INPUT:**
- $n$: Base 10 value to be converted
- base: Destination number system

**OUTPUT:**
- digit(): digit(0) holds LSD of result

```plaintext
quotient <-- n
i <-- 0
while quotient does not equal 0:
    digit(i) <-- quotient modulo base
    quotient <-- the floor of quotient/base
    increment i by 1
end while
```

Some Sample Iterative Algorithms (2 / 3)

What is the cost to evaluate $f(x) = 2x^3 - 4x^2 + 3x + 6$?
Example: Horner’s Algorithm for Polynomial Evaluation

**Input:**
- \( x \) Value used to evaluate the polynomial
- \( n \) Largest exponent
- \( a(0) \ldots a(n) \) Coefficients of \( x^0 \ldots x^n \)

**Output:**
- result Evaluation of the polynomial

```
result <-- a(n)
index <-- n - 1
while index >= 0:
    result <-- x * result + a(index)
    decrement index by 1
end while
output result
```

Recursive Definitions (1 / 2)

**Definition: Recursive Definition**

A complete recursive definition has three parts:

(a) The ________________ determines how trivial cases are to be handled.

(b) The ________________ describes complex problem instances in terms of simpler instances

(c) The ________________ provides bounds on the definition
Recursive Algorithms

**Definition: Recursive Algorithm**
Example: Factorials (1 / 3)

**Definition: Factorial**

The factorial of \( n \in \mathbb{Z}^* \), denoted \( n! \), is the product of all integers 1 through \( n \), where \( 0! = 1 \).

An iterative factorial algorithm is easy to create:

```python
product <-- 1
while n is larger than 1:
    product <-- product * n
    n <-- n - 1
end while
output product
```

Example: Factorials (2 / 3)

Factorials can be easily computed recursively:

\[
4! = 4 \cdot 3 \cdot 2 \cdot 1
\]

\[
4! = 4 \cdot 3!
\]

But what are the Basis, Inductive, and Extremal clauses?
Example: Factorials (3 / 3)

Recursive pseudocode algorithm:

```pseudocode
subprogram factorial ( given: n ) returns: n!
    if n is 0
        return 1
    else
        answer <-- n * factorial(n-1)
        return answer
    endif
end subprogram
```

Can We Prove Our Algorithm? (1 / 2)

**Conjecture:** \(\text{factorial}(n)\) returns \(n!\).
Another Structural Induction Proof (1 / 4)

**Conjecture:** In a binary tree, the number of null references equals one more than the number of nodes in the tree, for all non-empty binary trees.
Example: Fibonacci Sequence (1 / 2)

**Definition: Fibonacci Sequence**

The $n^{th}$ term of the Fibonacci Sequence is the sum of terms $n - 1$ and $n - 2$; where $F(0) = 0$ and $F(1) = 1$:

$$F = 0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots$$

(A000045)

Recursively generating terms of the sequence is easy …

```plaintext
subprogram fibonacci ( given: n ) returns: n-th term
  if n is 0 or 1
    return n
  else
    return fibonacci(n-1) + fibonacci(n-2)
  end if
end subprogram
```
Example: Fibonacci Sequence (2 / 2)

...but inefficient!

Consider this tree of invocations resulting from \texttt{fibonacci}(5):

\[
\begin{align*}
\text{f(5)} & \quad \text{f(4)} + \quad \text{f(3)} \\
\quad \text{f(3)} & \quad \quad \quad \quad \quad \quad \text{f(2)} + \quad \text{f(1)} \\
\text{f(2)} + \text{f(1)} & \quad \text{f(1)} + \text{f(0)} \\
\text{f(1)} + \text{f(0)} & \quad \text{f(1)} + \text{f(0)}
\end{align*}
\]

Extra Slides

The remaining slides in this topic are some that I no longer cover in class. I won’t ask about them on a quiz or an exam, but they could be referenced on a homework or in section.
**Example: Euclidean Algorithm for GCDs**

**Theorem:** \( \text{GCD}(a, b) = \text{GCD}(b, a \mod b) \)

Recursive pseudocode algorithm:

```plaintext
subprogram GCD (given: a, b) returns: gcd(a, b)
    if a is 0, return b endif
    if b is 0, return a endif
    answer <-- GCD(b, a % b)
    return answer
end subprogram
```

**Example: Sums Of Odd Positive Integers (1 / 2)**

\[
\begin{align*}
Z^+ & : 1 \; 2 \; 3 \; 4 \; \ldots \; n \; \frac{(m+1)}{2} \\
o & : 1 \; 3 \; 5 \; 7 \; \ldots \; 2n - 1 \; m
\end{align*}
\]

Let \( \text{oddsum}(\text{term}) \) represent the sum of \( o(1) \) through \( o(\text{term}) \).

**Base:** \( \text{oddsum}(1) = 1 \)

**General:** \( \text{oddsum}(\text{term}) = \text{oddsum}(\text{term}-1) + 2*\text{term} - 1 \)

Recursive implementation, using pseudocode:

subprogram oddsum (given: term)
    returns: sum from 1 through term of (2i-1)

    if term is 1, return 1
    otherwise
        answer <-- oddsum(term-1) + 2*term - 1
        return answer
    end if

end subprogram

Proving oddsum () (1 / 2)

Conjecture: oddsum(t) produces \( \sum_{i=1}^{t} (2i - 1) \), \( \forall t \geq 1 \)

Proof (by structural induction):

**Basis:** At \( t = 1 \), the algorithm returns 1, and \( \sum_{i=1}^{1} (2i - 1) = 1 \). OK!

**Inductive:** If oddsum(t) returns \( \sum_{i=1}^{t} (2i - 1) \), then oddsum(t + 1) returns \( \sum_{i=1}^{t+1} (2i - 1) \).

(Continues …)
When given \( t + 1 \), \( \text{oddsum()} \) returns
\[
\text{oddsum}(t) + [2(t + 1) - 1] = \text{oddsum}(t) + (2t + 1).
\]

By the Inductive Hypothesis, \( \text{oddsum}(t) = \sum_{i=1}^{t} (2i - 1) \).

Substituting, \( \text{oddsum}(t + 1) \) returns \( \sum_{i=1}^{t} (2i - 1) + (2t + 1) \).

\( 2t + 1 \) is the \((t + 1)^{st}\) term of the sequence; thus
\[
\sum_{i=1}^{t} (2i - 1) + (2t + 1) = \sum_{i=1}^{t+1} (2i - 1).
\]

Therefore, \( \text{oddsum}(t) \) produces \( \sum_{i=1}^{t} (2i - 1) \), \( \forall t \geq 1 \).