
Algorithms

3.1, 5.3, 5.4

Algorithms

Definition: Algorithm

A finite set of instructions for performing a task

Example:

Is Binary Search an algorithm? **Yes!**

Is the Division Algorithm an algorithm? **No!**

(It's not a set of instructions)

The Framework

1. Computable - means that the solution can be described by an algorithm
 - (a) Tractable - the algorithm is efficient
 - (b) Intractable - no efficient solutions
2. Non-computable - no algorithm will ever describe the solution.

Algorithm Characteristics

- 1. Input** - Data is provided from outside of the algorithm
- 2. Output** - Information produced by the algorithm
- 3. Generality** - The instructions can solve a collection of similar problems

Algorithm Characteristics

- 4. Definiteness** - (a.k.a. Precision, Uniqueness) The instructions are not open to interpretation.
- 5. Correctness** - The output is the accepted answer for the given input.
- 6. Finiteness** - The complete output is produced by the execution of a finite quantity of instructions

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

Some problems with this algorithm:

What if the tube is empty? (Input)

Does this algorithm solve related problems? (Generality)

Brushing technique? (Definiteness)

When do we stop? (Finiteness)

Some Sample Iterative Algorithms

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**

```
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

What is the cost to evaluate $f(x) = 2x^3 - 4x^2 + 3x + 6$?

Naive evaluation:

$$f(x) = 2 \cdot x \cdot x \cdot x - 4 \cdot x \cdot x + 3 \cdot x + 6$$

1 2 3 1 4 5 2 6 3 3+'s, 6 ·'s

But can we do better?

$$f(x) = x(2x^2 - 4x + 3) + 6$$

$$= x(x(2x - 4) + 3) + 6$$

$$= x(x(x(2) - 4) + 3) + 6$$

3 2 1 1 2 3 3+'s, 3 ·'s

Some Sample Iterative Algorithms

Example: Horner's Algorithm for Polynomial Evaluation

Input:	x	Value used to evaluate the polynomial
	n	Largest Exponent
	a(0) .. a(n)	Coefficients of $x^0 \dots x^n$
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

Definition: Recursive Definition

A complete recursive definition has three parts:

- (a) The basis clause determines how trivial cases are to be handled
- (b) The inductive clause describes complex problem instances in terms of simpler instances
- (c) The extremal clause provides bounds on the definition

Recursive Definitions

Example:

Consider the sequence $S : 13, 10, 7, 4, 1$

Basis: $S_1 = 13$

Recurrence: $S_n = S_{n-1} - 3$

Extremal: $1 \leq n \leq 5$

Consider the non-negative integers (\mathbb{Z}^*)

Basis: $1 \in \mathbb{Z}$

Recurrence: if $n \in \mathbb{Z}$, then $n + 1 \in \mathbb{Z}$

Extremal: N/A

Consider general trees

Basis: Empty tree (0 nodes)

Recurrence: The root has ≥ 0 subtrees that are general trees


Extremal: N/A

Recursive Algorithms

Definition: *Recursive Algorithm*

A recursive algorithm express the solution to a task in terms of a simpler case of the same problem.

Aside: Control Structures in Programming Languages

1. Sequence
 2. Selection
 3. Iteration...or Recursion!
- 

Example: Factorials

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:

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

Example: Factorials

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?

Basis: $0! = 1$

Inductive: $n! = n \cdot (n - 1)!$

Extremal: $n!$ is defined $\forall n \in \mathbb{Z}^*$

Example: Factorials

Recursive pseudocode algorithm:

```
subprogram factorial (given: n) returns: n!  
  if n is 0  
    return 1  
  else  
    answer  $\leftarrow$  n * factorial(n-1)  
  end if  
end subprogram
```

(Basis)

(Inductive)

Extremal? Assumed!



Can We Prove Our Algorithm?

Conjecture: **factorial(n)** returns $n!$

Proof (structural induction):

Basis: Let $n = 0$. The algorithm returns 1, and by definition, $0! = 1$. Ok!

Inductive Step: If **factorial(n)** returns $n!$, then **factorial(n+1)** returns $(n + 1)!$.

When the input is $(n + 1)$, the algorithm will compute $(n + 1)!$ to be $(n + 1) * \text{factorial}(n)$

(Continues ...)

Can We Prove Our Algorithm?

By the Inductive Hypothesis, we know that `factorial(n)` computes $n!$. And, from the recursive definition of factorial, we know that

$$n! * (n + 1) = (n + 1)!. .$$

Therefore, `factorial(n)` computes $n!$

Another Structural Induction Proof

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.

Proof (structural induction):

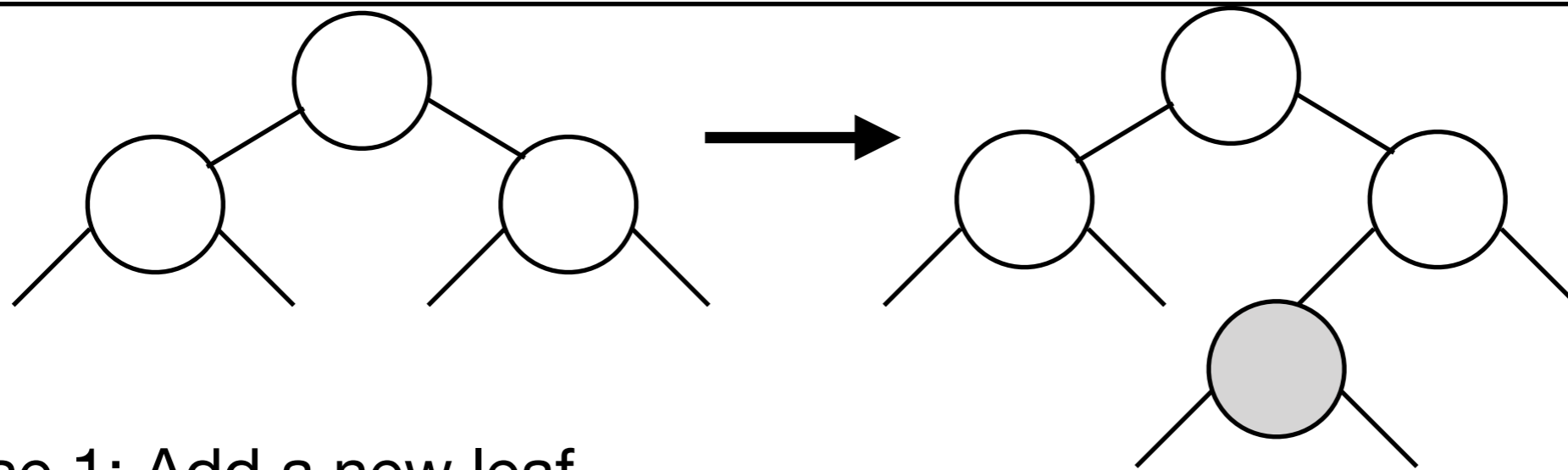
Basis: A binary tree with one node has 2 nulls. Ok!

Inductive Step: If a binary tree of n nodes has $n + 1$ nulls, then a binary tree of $n + 1$ nodes has $n + 2$ nulls.

There are three possible insertion situations

(Proof Continues ...)

Another Structural Induction Proof



Case 1: Add a new leaf.

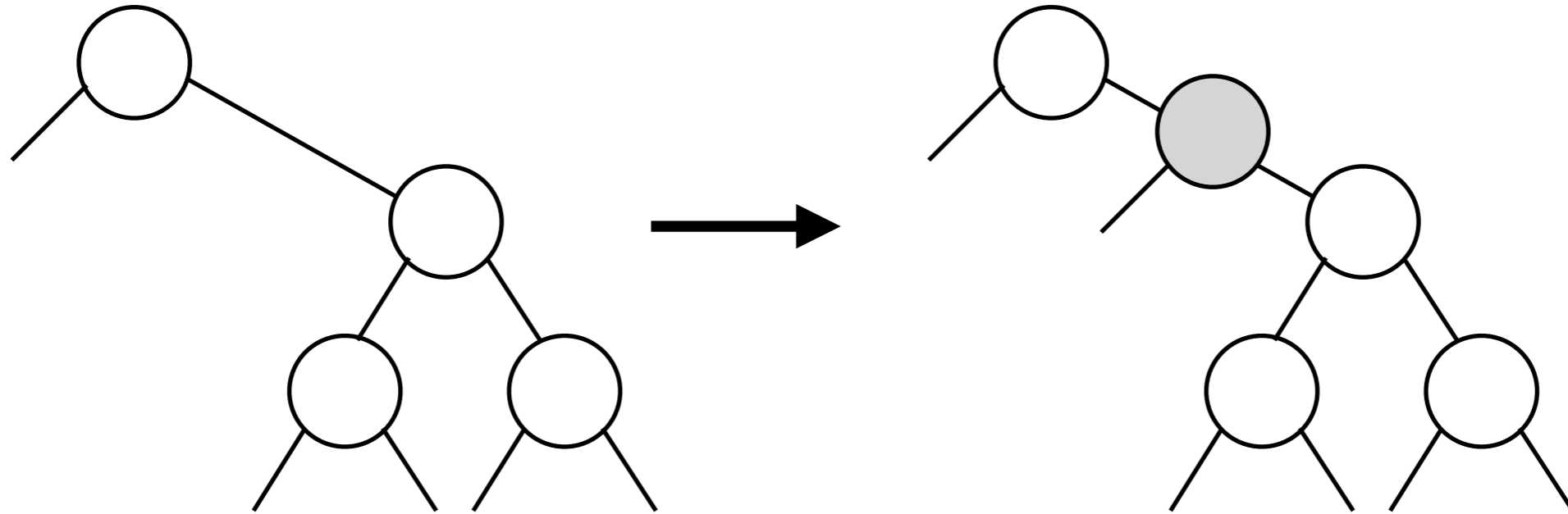
By the Inductive Hypothesis, we have n nodes and $n + 1$ nulls in our tree.

Adding a leaf adds one node and two nulls, but occupies (removes) an existing null.

This is a net gain of one node and one null, giving a total of $n + 1$ nodes and $n + 2$ nulls, as desired.

(Proof Continues)

Another Structural Induction Proof



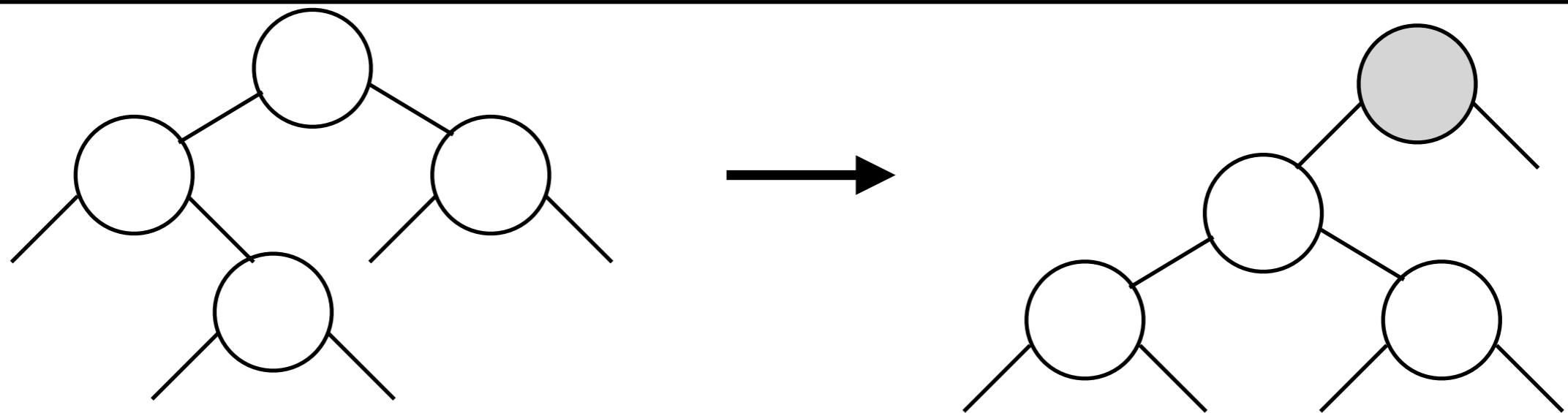
Case 2: Insert between nodes.

We add a node, occupy an existing null, and use one of its children, leaving one extra new null.

As before this is a gain of one node and one null.

(Proof Continues)

Another Structural Induction Proof



Case 3: Insert a new root.

We add a node and occupy one of its nulls in referencing the old root. Again, a net gain of one node and one null.

Therefore, $\# \text{-nulls} = 1 + \# \text{ nodes}$, for all non-empty binary trees

Example: Fibonacci Sequence

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 $F(1) = 1$

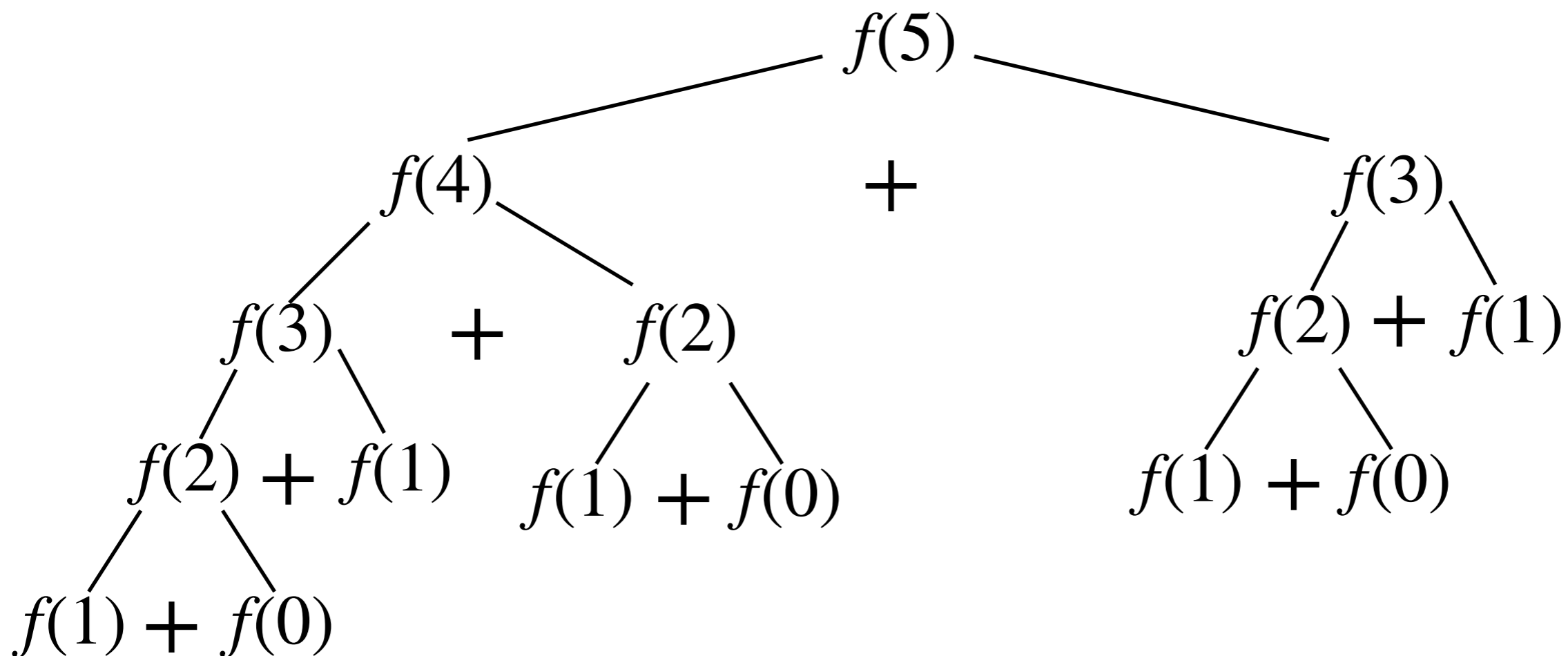
Recursively generating terms of the sequence is easy...

```
subprogram fibonacci (given: n) returns: nth term
  if n is 0 or 1
    return n
  else
    return fibonacci(n-1) + fibonacci(n-2)
  end if
end subprogram
```

Example: Fibonacci Sequence

... but inefficient!

Consider this tree of invocations resulting from `fibonacci(5)`:



**Note the three $f(2)$ trees and the two $f(3)$ trees
⇒ Repeated (and therefore wasted) effort!**

Extra Slides

Example: Euclidean Algorithm for GCDs

Theorem: $\text{GCD}(a,b) = \text{GCD}(b,a\%b)$

Proof: See Rosen 8/e p. 283

Recursive pseudocode algorithm:

```
subprogram GCD (given: a,b) returns: gcd(a,b)
  if a is 0, return b endif
  if b is 0, return a endif
  answer  $\leftarrow$  GCD(b, a%b)
  return answer
end subprogram
```

Question: Is this more or less efficient than the iterative algorithm presented earlier?

Example: Sums of Odd Positive Integers

$$\mathbb{Z}^+ : 1 \ 2 \ 3 \ 4 \ \dots \quad n \quad \frac{(m+1)}{2}$$

$$o : 1 \ 3 \ 5 \ 7 \ \dots \quad 2n-1 \quad m$$

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

Base: $\text{oddsun}(1) = 1$

General: $\text{oddsun}(\text{term}) =$

$$\text{oddsun}(\text{term}-1) + 2 * \text{term} - 1$$

Example: Sums of Odd Positive Integers

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  $\leftarrow$  oddsum(term-1)+2*term-1
        return answer
    end if
end subprogram
```

Proving oddsum()

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

Proof (structural induction):

Basis: Let $t = 1$. The algorithm returns 1, and $\sum_{i=1}^1 (2i - 1) = 1$. Ok!

Inductive Step: If oddsum(t) returns $\sum_{i=1}^t (2i - 1)$,

then oddsum(t+1) returns $\sum_{i=1}^{t+1} (2i - 1)$.

(Continues ...)

Proving oddsum()

When given $t + 1$, `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, `oddsum(t+1)` returns $\sum_{i=1}^t (2i - 1) + (2t + 1)$.

$2t + 1$ is the $(t + 1)^{\text{st}}$ term of the sequence; thus

$$\sum_{i=1}^t (2i - 1) + (2t + 1) = \sum_{i=1}^{t+1} (2i - 1).$$

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