FALL 2018 CISC 311

OBJECT & STRUCTURE & ALGORITHM I

– Chapter 3: Algorithm Analysis

Outline

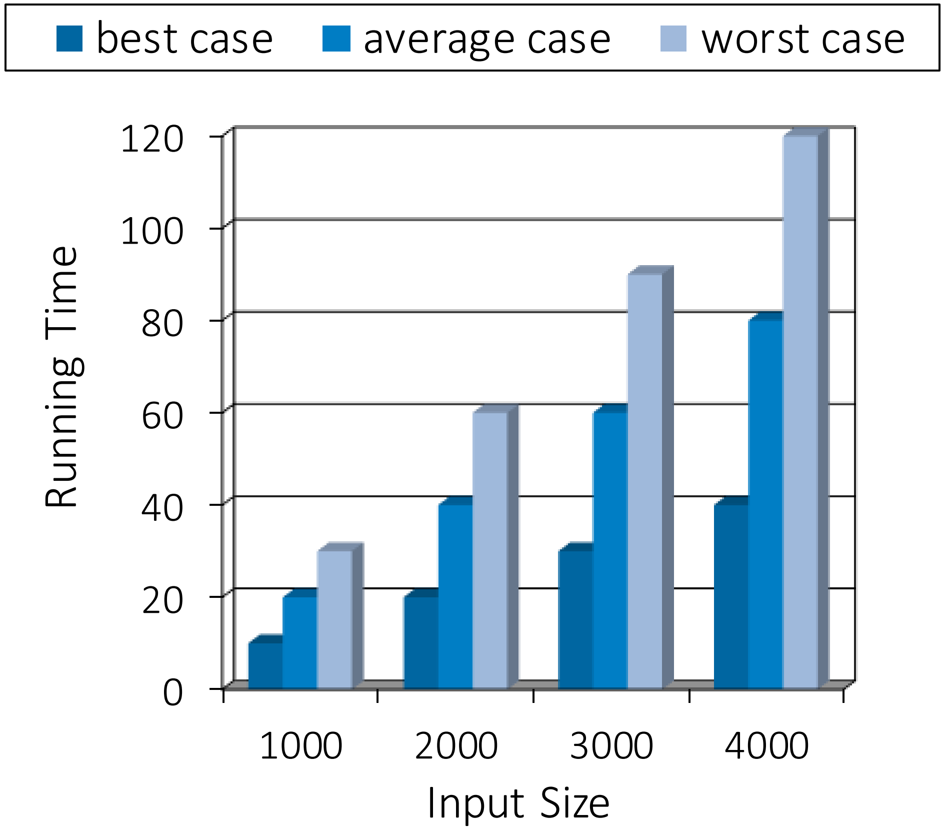
- Experimental study
- Theoretical analysis
- Asymptotic computational complexity *O*()
- Recursion
- Analyzing recursive algorithms
- Recursion with memoization
- Appendix: math review + pseudocode

Algorithm Analysis

- Algorithm: Step-by-step procedure for performing some task in a finite amount of time
- Complexity of algorithms:
  - Running time: primary measure of “goodness” of algorithms
    - Study the dependency of running time on the size of the input
    - Experimental studies:
      - Execute the program on various test inputs
      - Record actual time spent in each execution
    - Theoretical (analytical) analysis:
      - Study the high-level description of the algorithms without actually implementing it
  - Space usage: measure of “goodness” of algorithms
  - Network bandwidth usage: measure of “goodness” of algorithms

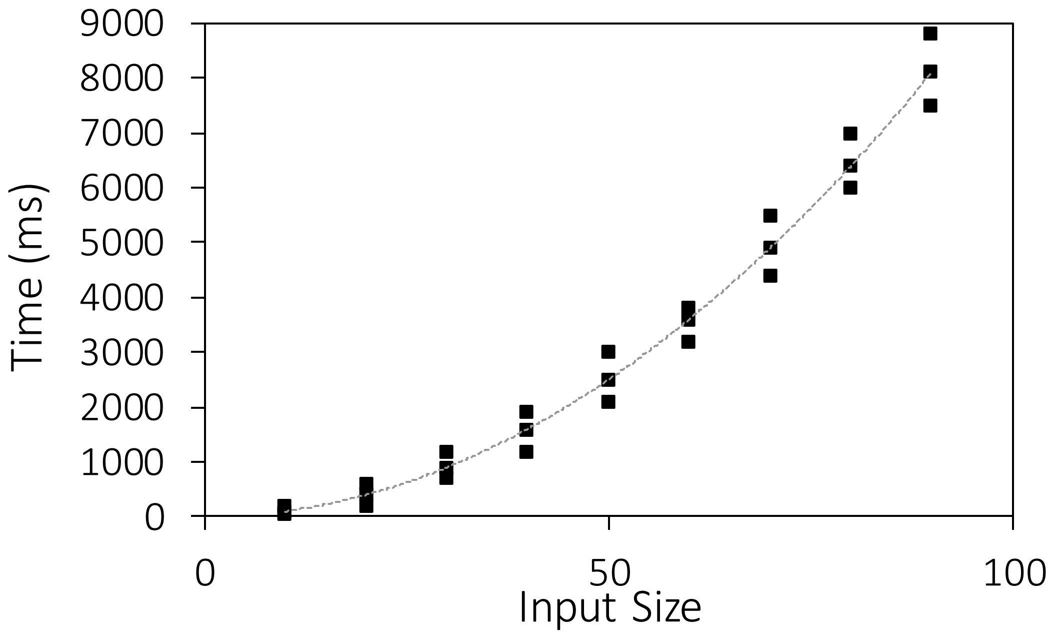
Running Time

- Most algorithms transform input objects into output objects
- The running time of an algorithm typically grows with the input size



Experimental Studies

- Experimental studies:
  - Write a program implementing the algorithm
  - Run the program with inputs of varying size and composition
  - Use a function, like the built-in `clock()` function, to get an accurate measure of the actual running time
  - Plot the results



Limitations of Experiments

- It is necessary to implement the algorithm, which may be difficult
- Results may not be indicative of the running time on other inputs not included in the experiment
- In order to compare two algorithms, the same hardware and software environments must be used

Theoretical Analysis

- Uses a high-level description of the algorithm instead of an implementation
- Associates with each algorithm a function $f(n)$ that characterizes running time as a function of the input size, n
- Takes into account all possible inputs
- Allows us to evaluate the speed of an algorithm independent of the hardware/software environment

Primitive Operations

- High-level primitive operations:
  - Independent from programming language
  - Each primitive operation corresponds to a low-level instruction with an execution time that depends on the hardware and software environment but is nevertheless constant
  - Include:
    - Assigning a value to a variable
    - Calling a method
    - Performing an arithmetic operation
    - Comparing two numbers
    - Indexing into an array
    - Following an object reference
    - Returning from a method

Analyzing Non-Recursive Algorithms

- Defines *t* as the number of primitive operations that an algorithm performs
- Uses *t* as a high-level estimate of the running time of the algorithm, which is proportional to the actual running time of that algorithm
- Counts the number of primitive operations are executed by the algorithm

Analyzing Non-Recursive Algorithms (cont'd.)

- Example: finding max element of an array

```
Algorithm arrayMax(A, n)
    Input A, n  array A of n integers
    Output currentMax  maximum element of A
    currentMax ← A[0]
    for i ← 1 to n – 1 do
        if A[i] > currentMax then
            currentMax ← A[i]
    return currentMax
```

At least: $2 + 1 + n + 4(n-1) + 1 = 5n$ $\longrightarrow$ The best case

At most: $2 + 1 + n + 6(n-1) + 1 = 7n - 2$ $\longrightarrow$ The worst case

Average-case and Worst-case Analysis

- Average-case analysis:
  - Requires the probability distribution on the set of inputs
  - Calculates the expected running time based on a given input distribution
- Worst-case analysis:
  - Easier to analyze
  - Crucial to applications such as games, finance and robotics

The Growth of Functions

- The growth of functions is usually described using the big-Oh notation
  - Definition: Let *f*(*n*) and *g*(*n*) be functions mapping nonnegative integers to real numbers. We say that *f*(*n*) is O(*g*(*n*)) if there are real constants *c* > 0 and *k* > 0 such that

$$|f(n)| \leq c|g(n)|$$

    for every integer $n \geq k$
- The idea behind the big-Oh notation is to establish an upper boundary for the growth of a function for *f*(*n*) large *n*

The Growth of Functions (cont'd.)

- Example: finding max element of an array

  At least: $2 + 1 + n + 4(n-1) + 1 = 5n \longrightarrow$ The best case

  At most: $2 + 1 + n + 6(n-1) + 1 = 7n - 2 \longrightarrow$ The worst case

  - We need to find real constants $c > 0$ and $k > 0$ such that

    $f(n) = 7n - 2 \leq cn$ for every integer $n \geq k \rightarrow$ for $c = 7$ and $k = 1$:

    $f(n) \leq cn \rightarrow f(n) \leq O(n)$

  - In asymptotic sense, n grows toward infinity

The Growth of Functions (cont'd.)

- Example: show that $f(n) = n^2 + 2n + 1$ is $O(n^2)$
  - For $n > 1$, we have: $n^2 + 2n + 1 \leq n^2 + 2n^2 + n^2$

    $\rightarrow n^2 + 2n + 1 \leq 4n^2$

    Therefore, for $c = 4$ and $k = 1$: $f(n) \leq cn^2$ for any $x > k$

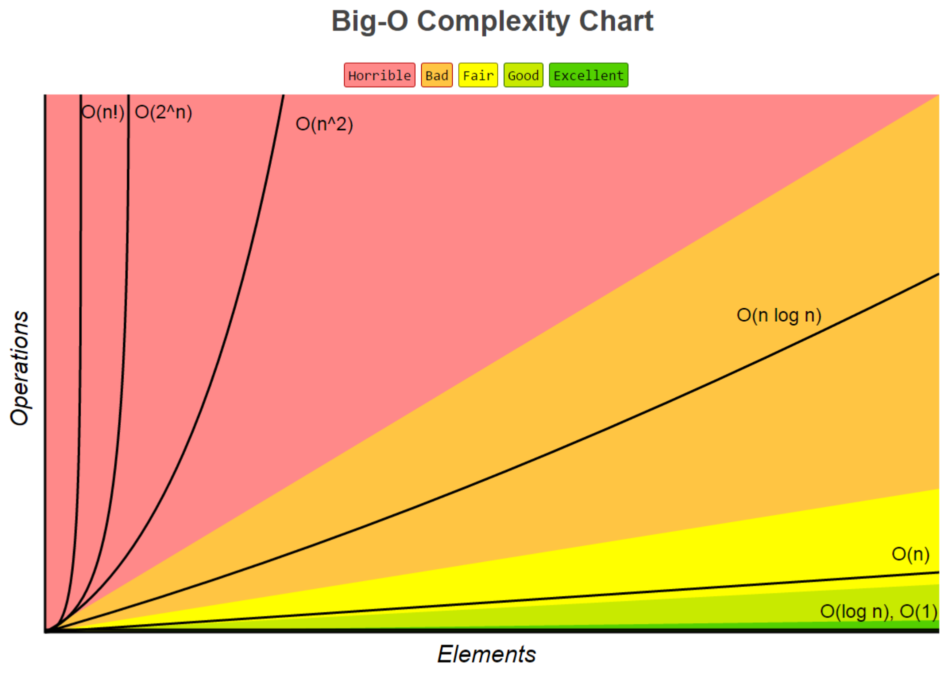
    $\rightarrow f(n)$ is $O(n^2)$
  - Also $f(n)$ is $O(n^3)$
  - n^3 grows faster than n^2, so n^3 grows also faster than $f(n)$. Therefore, we always have to find the smallest simple function $g(n)$ for which $f(n)$ is $O(g(n))$

The Growth of Functions (cont’d.)

- “Popular” functions $g(n)$ are

$$n\log n,\ 1,\ 2^n,\ n^2,\ n!,\ n,\ n^3,\ \log n$$

- Listed from slowest to fastest growth:
  - 1
  - $\log n$ $\longrightarrow$ logarithmic
  - n $\longrightarrow$ linear
  - $n\log n$
  - n^2 $\longrightarrow$ quadratic
  - n^3 $\longrightarrow$ polynomial
  - 2^n $\longrightarrow$ exponential
  - $n!$ $\longrightarrow$ factorial



Big-Oh Rules

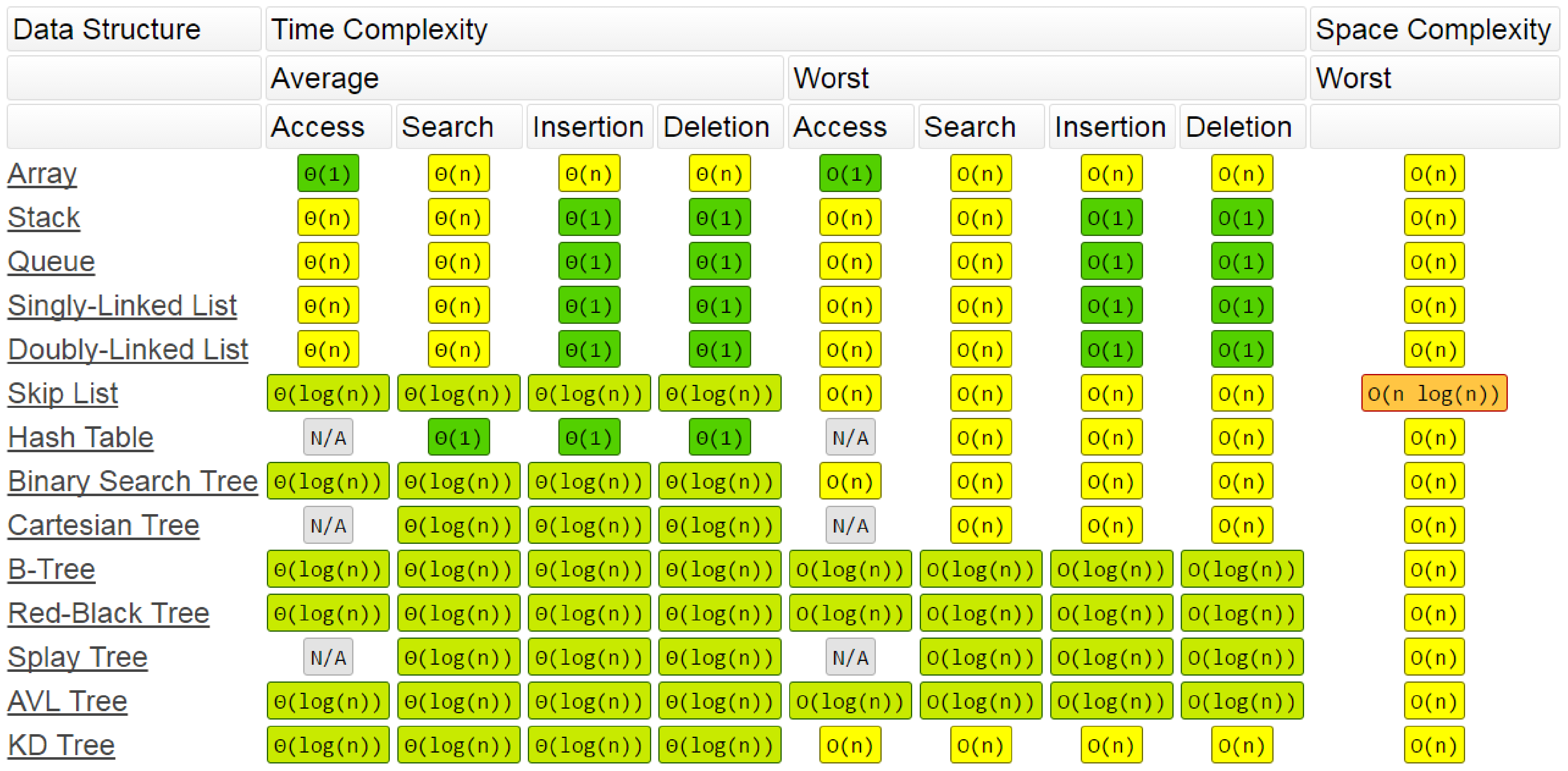
- If $f(n)$ is a polynomial of degree d, then $f(n)$ is $O(n^d)$
  1. Drop lower-order terms
  2. Drop constant factors
- Use the smallest possible class of functions
  - Say "$2n$ is $O(n)$" instead of "$2n$ is $O(n^2)$"
- Use the simplest expression of the class
  - Say "$3n + 5$ is $O(n)$" instead of "$3n + 5$ is $O(3n)$"

Why Growth Rate Matters

- Comparison: time complexity of algorithms A and B

Input Size	Algorithm A	Algorithm B
n	5,000n	1.1^n
10	50,000	3
100	500,000	13,781
1,000	5,000,000	2.5×10^{41}
1,000,000	5×10^9	4.8×10^{41392}

Complexity of Common Data Structures



Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

Useful Rules for Big-Oh

- If $f(n)$ is $O(g(n))$, then $af(n)$ is $O(g(n))$ is for any constant $a>0$
- For any polynomial $f(n) = a_d n^d + a_{d-1} n^{d-1} + \ldots + a_0$, where $a_0, a_1, \ldots a_d$ are real numbers, $f(n)$ is $O(n^d)$
- If $f_1(n)$ is $O(g(n))$ and $f_2(n)$ is $O(g(n))$, then $f_1(n) + f_2(n)$ is $O(g(n))$
- If $f_1(n)$ is $O(g_1(n))$ and $f_2(n)$ is $O(g_2(n))$, then $f_1(n) + f_2(n)$ is $O(\max(g_1(n), g_2(n)))$
- If $f_1(n)$ is $O(g_1(n))$ and $f_2(n)$ is $O(g_2(n))$, then $f_1(n)\, f_2(n)$ is $O(g_1(n) g_2(n))$

Exercise 3.1

- Exercise: $8n - 3$
- Exercise: $3n^3 + 20n^2 + 5$
- Exercise: $3\log n + 5$

Complexity Examples

- Find the maximum difference between any two numbers in the input sequence

```
Algorithm findMaxDiff1($a_1, a_2, \ldots. a_n$)
    Input $a_1, a_2, \ldots. a_n$  $n$ integers
    Output $m$ maximum difference
    $m \leftarrow 0$
    for $i \leftarrow 1$ to $n - 1$ do
        for $j \leftarrow i + 1$ to $n$ do
            if $|a_i - a_j| > m$ then
                $m \leftarrow |a_i - a_j|$
    return $m$
```

- Comparisons: $n - 1 + n - 2 + n - 3 + \ldots + 1 = \dfrac{(n-1)n}{2} = 0.5n^2 - 0.5n$

  $\rightarrow f(n)$ is $O(n^2)$

FindMaxDiff1.java

Complexity Examples (cont'd.)

- Another algorithm solving the same problem:

```
Algorithm findMaxDiff2($a_1, a_2, \ldots a_n$)
    Input $a_1, a_2, \ldots a_n$  $n$ integers
    Output $m$ maximum difference
    $min \leftarrow a_1$
    $max \leftarrow a_1$
    for $i \leftarrow 2$ to $n$ do
        if $a_i < min$ then
            $min \leftarrow a_i$
        else if $a_i > max$ then
            $max \leftarrow a_i$
    $m \leftarrow max - min$
    return $m$
```

- Comparisons: $2n - 2 \rightarrow f(n)$ is $O(n)$

`FindMaxDiff2.java`

Exercise 3.2

```
public void exercise1(int n) {
   for(int i = 0;  i < 100; i++) {
      System.out.println(i);
   }
   for(int i = 0; i < n; i++) {
      System.out.println(i);
   }
}

public void exercise2(int n, int m) {
   for(int i = 0;  i < n; i++) {
      System.out.println(i);
   }
   for(int i = 0; i < m; i++) {
      System.out.println(i);
   }
}
```

ComplexityExercise.java

Exercise 3.2 (cont'd.)

```
public void exercise3(int n, int m) {
    for(int i = 0; i < n; i++) {
        for(int j = 0; j < m; j++) {
            System.out.println(i + j);
        }
    }
}

public void exercise4(int n) {
    for(int i = 0; i < n; i++) {
        for(int j = 0; j < n; j++) {
            System.out.println(i + j);
        }
    }
    for(int i = 0;  i < n; i++) {
        System.out.println(i);
    }
}
```

ComplexityExercise.java

Exercise 3.2 (cont’d.)

```
public void exercise5(int n) {
    for(int i = 0; i < n; i++) {
        for(int j = n; j > n / 2; j--) {
            System.out.println(i + j);
        }
    }
}

public void exercise6(int n) {
    for(int i = 1; i < n; i = i * 2) {
        System.out.println(i);
    }
}
```

`ComplexityExercise.java`

Introduction to Recursion

- Recursion: a method that calls itself
- Recursive method must have a way to control the number of times it repeats
  - Usually involves an `if-else` statement which defines when the method should return a value and when it should call itself
- Depth of recursion: The number of times a method calls itself

Introduction to Recursion (cont'd.)

- Example:

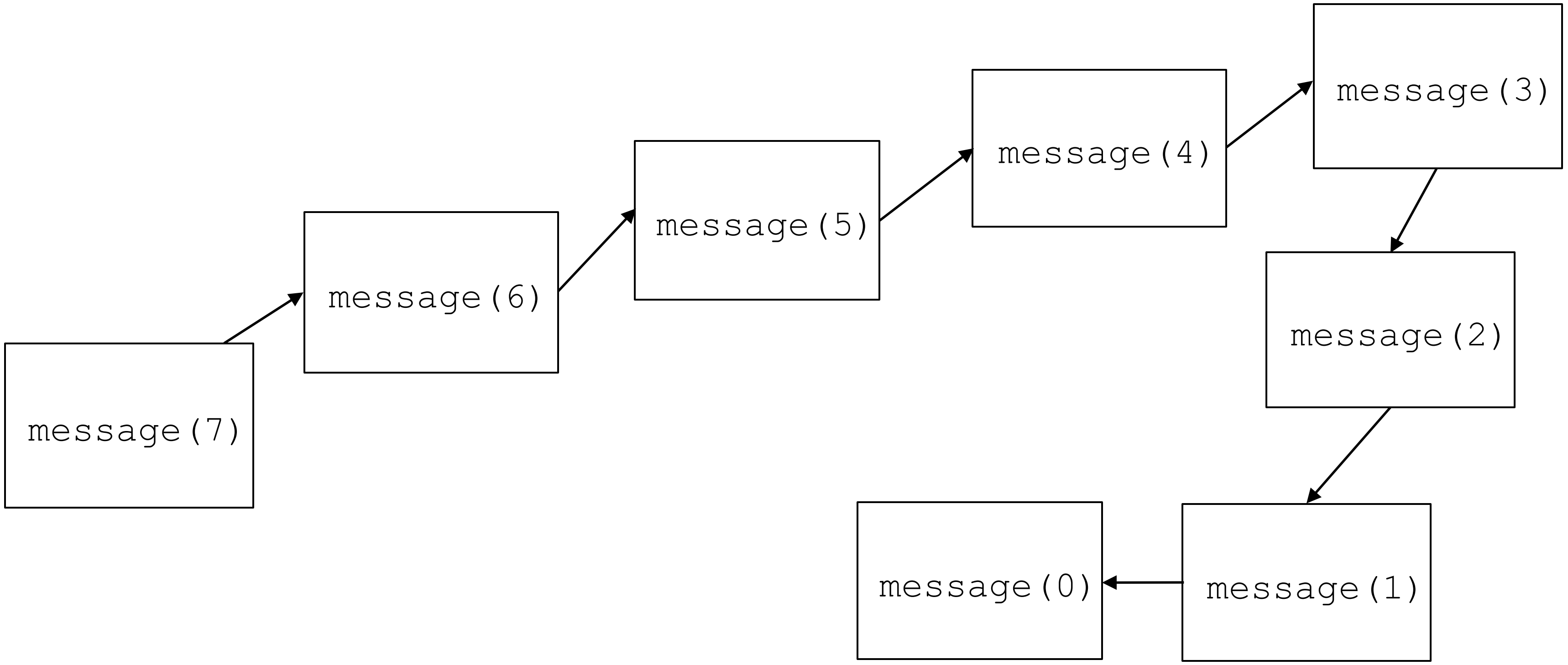
```
public class RecursionExample1 {
    void message(int times){
        if (times > 0){
            System.out.println("This is recursion!");
            message(times-1);}
    }

    public static void main (String[] args){
        RecursionExample1 rs = new RecursionExample1();
        rs.message(7);
    }
}
```

RecursionExample1.java

Introduction to Recursion (cont'd.)

- Call tree:



RecursionExample1.java

Problem Solving with Recursion

- Recursion is a powerful tool for solving repetitive problems
- Recursion is never required to solve a problem
  - Any problem that can be solved recursively can be solved with a loop
  - Recursive algorithms usually less efficient than iterative ones
    - Due to *overhead* of each method call

Problem Solving with Recursion (cont'd.)

- Some repetitive problems are more easily solved with recursion
- General outline of recursive function:
  - If the problem can be solved now without recursion, solve and return
    - Known as the *base case*
  - Otherwise, reduce problem to smaller problem of the same structure and call the function again to solve the smaller problem
    - Known as the *recursive case*

Example: Calculating Factorial

- In mathematics, the *n!* notation represents the factorial of a number *n*
  - For $n = 0$, $n! = 1$
  - For $n > 0$, $n! = 1 \times 2 \times 3 \times \ldots \times n$
- The above definition lends itself to recursive programming
  - `n = 0 is the base case`
  - `n > 0 is the recursive case`
    - `factorial(n) = n x factorial(n-1)`

Example: Calculating Factorial (cont'd.)

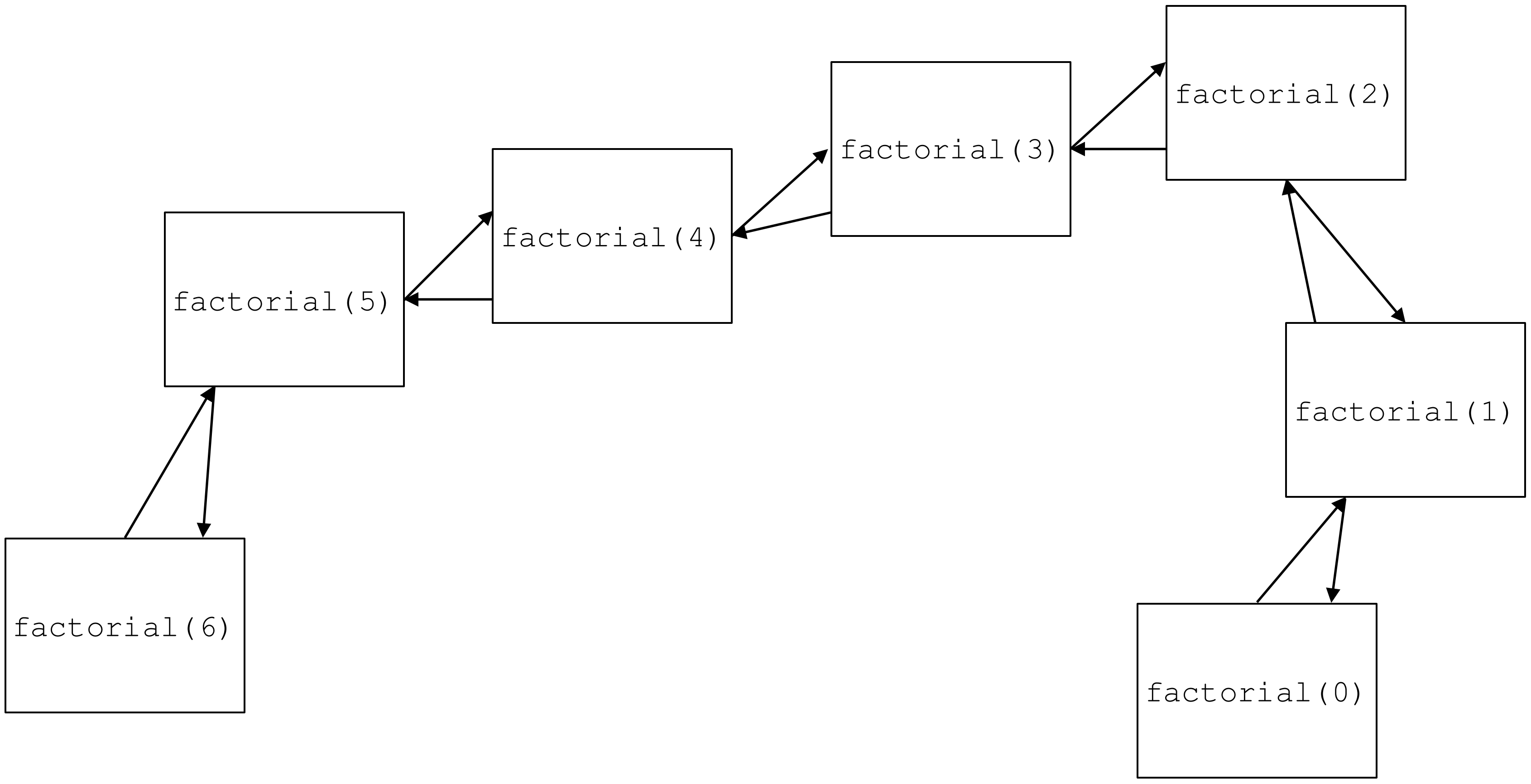
```
public class RecursionFactorial {
   public int factorial (int n) {
      int fac = 0;
      if (n == 0){
         fac = 1;
      } else if (n > 0) {
         fac =  n * factorial(n - 1);
      }
      return fac;
   }

   public static void main (String[] args){
      RecursionFactorial rf = new RecursionFactorial();
      int num = 6;
      int fac = rf.factorial(num);
      System.out.print("Factorial of "+num+" is "+fac);
   }
}
```

RecursionFactorial.java

Example: Calculating Factorial (cont'd.)

- Example: `factorial(6)`



Backward Substitution

- Recurrence equation: `factorial(n) = n x factorial(n-1)`
- Time complexity:

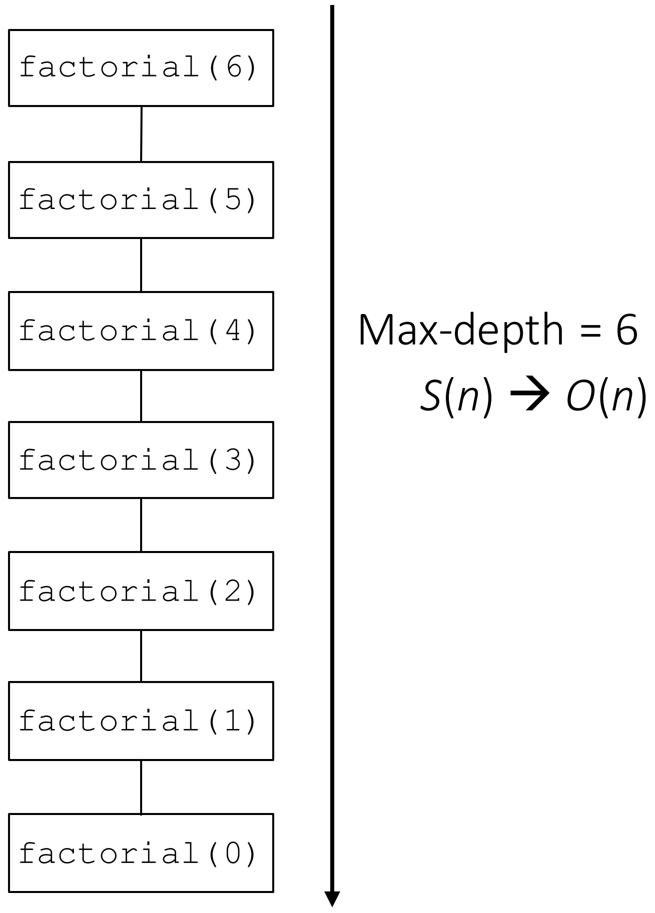
$$
\begin{aligned}
T(n) &= T(n-1) + c \\
&= T(n-2) + 2c \\
&= T(n-3) + 3c \\
&= \ldots \\
&= T(n-k) + kc, \text{ where } c \text{ is a constant}
\end{aligned}
$$

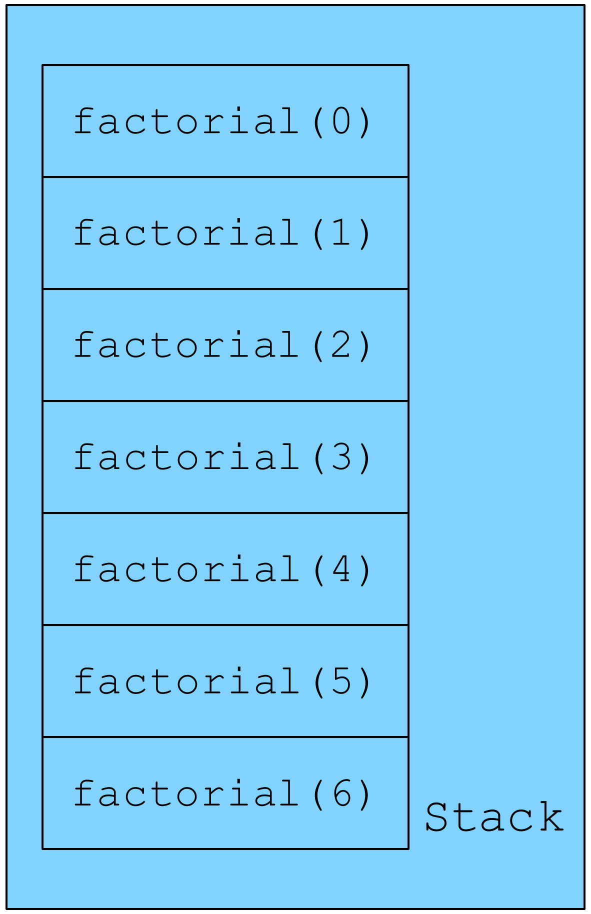
Base case $T(0)$ where $n = k$

$T(n) = T(0) + nc \rightarrow T(n) = 1 + nc \rightarrow O(n)$

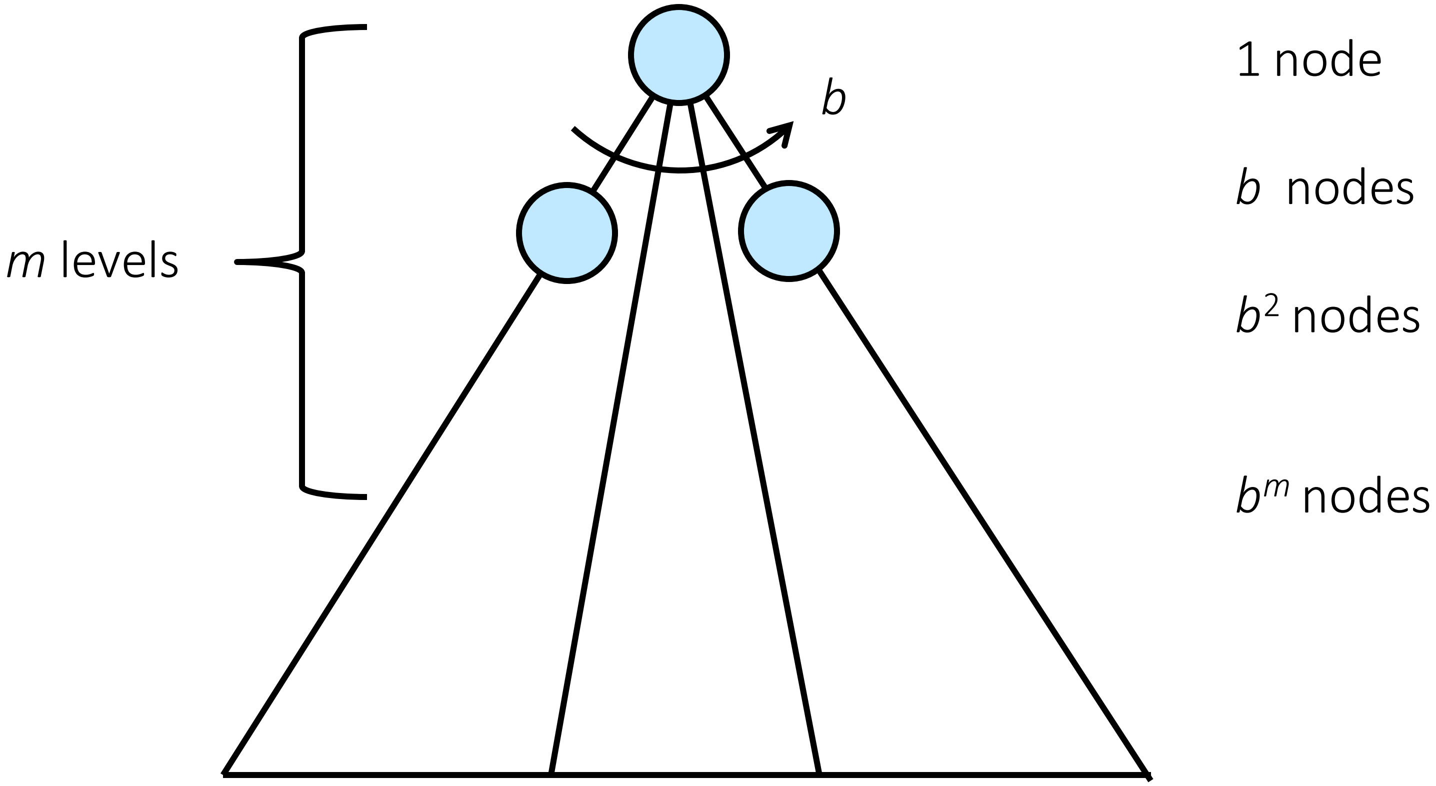
Recursive Tree

- Example: `factorial(6)`





Recursive Tree (cont'd.)



Recursive Tree (cont'd.)

- The idea of a recursion tree is to expand $T(n)$ to a tree with the same total cost
- Two things are important:
  - The height of the tree
  - The cost of the nodes at each level

Using Recursion

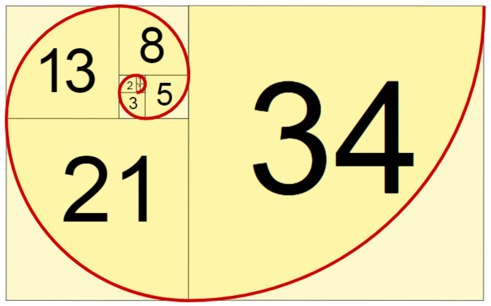
- Since each call to the recursive function reduces the problem:
  - Eventually, it will get to the base case which does not require recursion, and the recursion will stop
- Usually the problem is reduced by making one or more parameters smaller at each function call

Direct and Indirect Recursion

- Direct recursion: When a function directly calls itself
  - All the examples shown so far were of direct recursion
- Indirect recursion: When function *A* calls function *B*, which in turn calls function *A*

Example: The Fibonacci Series

- The Fibonacci sequence is a series of numbers where a number is found by adding up the two numbers before it. Starting with 0 and 1, the sequence goes 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, and so forth
- Written as a rule, the expression is $x_n = x_{n-1} + x_{n-2}$
- Fibonacci series: has two base cases
  - `if` n `= 0 then Fib(`n`) = 0`
  - `if` n `= 1 then Fib(`n`) = 1`
  - `if` n `> 1 then Fib(`n`) = Fib(`n`-1) + Fib(`n`-2)`

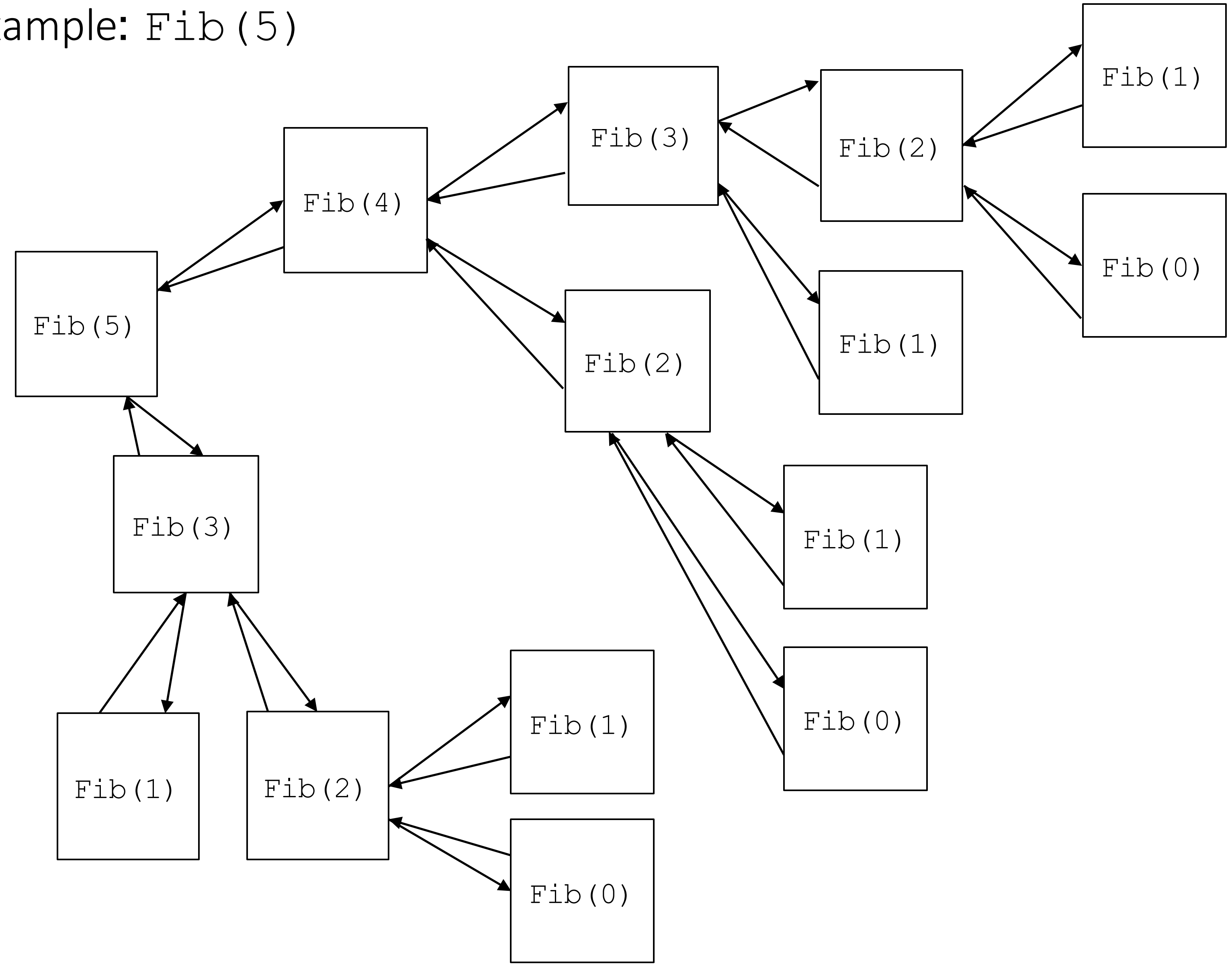


Φ= 1.618...
1
Golden Rectangle
1
Φ= 1.618...

```
public int fibonacci(int n){
   int fib = 0;
   if (n == 0){
      fib = 0;
   } else if (n == 1){
      fib = 1;
   } else if (n > 1){
      fib = fibonacci(n - 1) + fibonacci(n - 2);
   }
   return fib;
}
```

`RecursionFibonacci.java`

- Example: `Fib(5)`



Backward Substitution

- Recurrence equation: `Fib(n) = Fib(n-1) + Fib(n-2)`
- Time complexity:

$T(n) = T(n\text{-}1) + T(n\text{-}2) + c$

Then make $T(n\text{-}1) \approx T(n\text{-}2)$

Lower bound: $T(n) = 2T(n\text{-}2) + c$

$= 4T(n\text{-}4) + 3c$

$= 8T(n\text{-}6) + 7c$

$= \ldots$

$= 2^kT(n\text{-}2k) + (2^k\text{-}1)c$, where c is a constant

Base case $T(0)$, $T(1)$, where $n = 2k \rightarrow k = n/2$

$T(n) = 2^{n/2}T(0) + (2^{n/2}\text{-}1)c \rightarrow T(n) = (1+c)\ 2^{n/2} - c \rightarrow$

$O(2^{n/2})$

Backward Substitution (cont'd.)

- Recurrence equation: `Fib(n) = Fib(n-1) + Fib(n-2)`
- Time complexity:

$$T(n) = T(n\text{-}1) + T(n\text{-}2) + c$$

Then make $T(n\text{-}1) \approx T(n\text{-}2)$

Upper bound: $T(n) = 2T(n\text{-}1) + c$

$= 4T(n\text{-}2) + 3c$

$= 8T(n\text{-}3) + 7c$

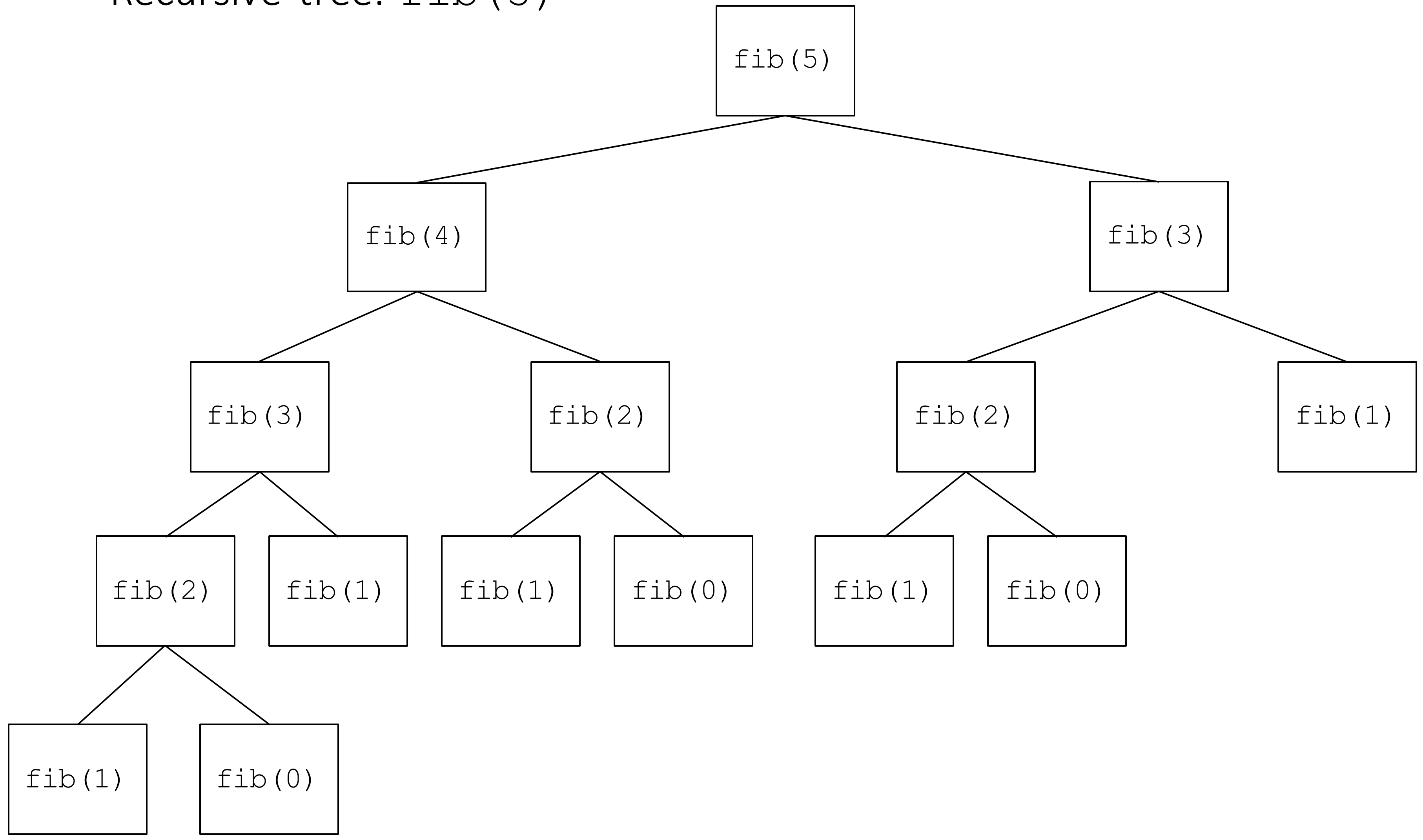
$= \ldots$

$= 2^k T(n\text{-}k) + (2^k\text{-}1)c$, where c is a constant

Base case $T(0)$, $T(1)$, where $n = k \rightarrow k = n$

$T(n) = 2^n T(0) + (2^n\text{-}1)c \rightarrow T(n) = (1+c)\ 2^n - c \rightarrow O(2^n)$

- Recursive tree: `fib(5)`



- Iterative solution by looping: $T(n) = O(n)$

```
public class LoopFibonacci {
   public int fibonacci (int n) {
      int num0 = 0;
      int num1 = 1;
      int fib = 0;
      for (int i = 0; i <= n - 2; i++){
         fib = num0 + num1;
         num0 = num1;
         num1 = fib;
      }
      return fib;
   }
}
```

`LoopFibonacci.java`

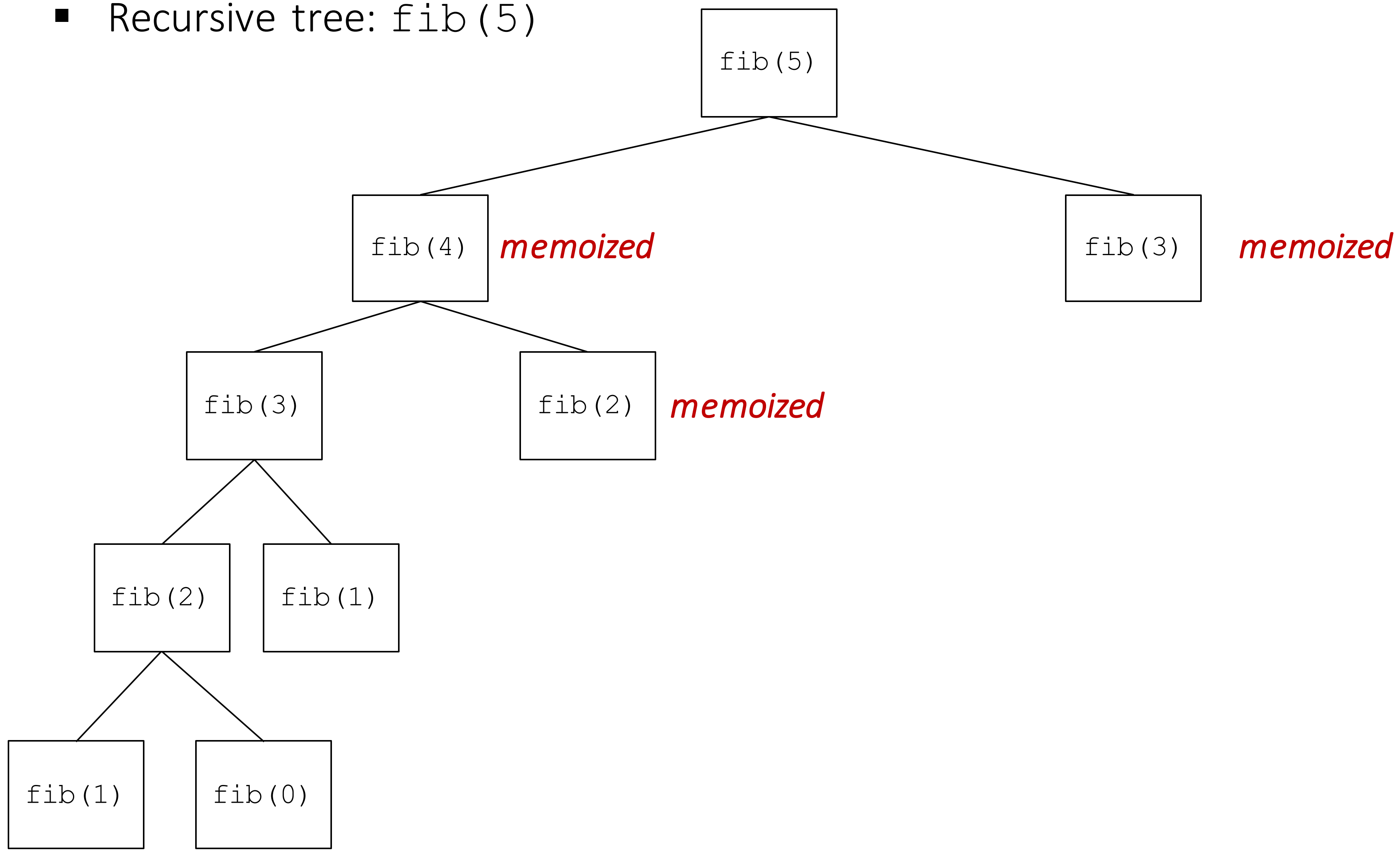
Recursion versus Looping

- Reasons not to use recursion:
  - Less efficient: entails method calling overhead that is not necessary with a loop
  - Usually a solution using a loop is more evident than a recursive solution
- Some problems are more easily solved with recursion than with a loop
  - Example: Fibonacci, where the mathematical definition lends itself to recursion

Recursion with Memoization

- We can use optimization techniques to address performance problems of recursions
- Memoization is one popular technique used primarily to speed up computer programs by storing the results of expensive method calls and returning the cached result when the same inputs occur again
- Example: `fib(5)` memorize the calculated results

- Recursive tree: `fib(5)`



```
int[] memory = new int[100];
public int fibonacci(int n){
    int fib = 0;
    if (memory[n] != 0) {
        fib = memory[n];
    } else if (n == 0) {
        fib = 0;
    } else if (n == 1) {
        fib = 1;
    } else if (n > 1) {
        fib = fibonacci(n - 1) + fibonacci(n - 2);
        // memorize the calculated result
        memory[n] = fib;
    }
    return fib;
```

RecursionFibonacciMemory.java

Recursion with Memoization (cont'd.)

- Memoization decreases the computation time of the optimized algorithm as it reduces the amount of calculation that is required by storing the calculated data into a data structure
- Memoization requires more space as another data structure is needed to store the values already calculated
  - In the case of the Fibonacci sequence, not much more space is used however in many cases memoization takes up too much extra space when used to be practical

Summary: Fibonacci

	Time complexity	Space complexity
Looping	$O(n)$	O(1)
Recursion	$O(2^n)$	$O(n)$
Recursion with Memoization	$O(2n)$	O(n)

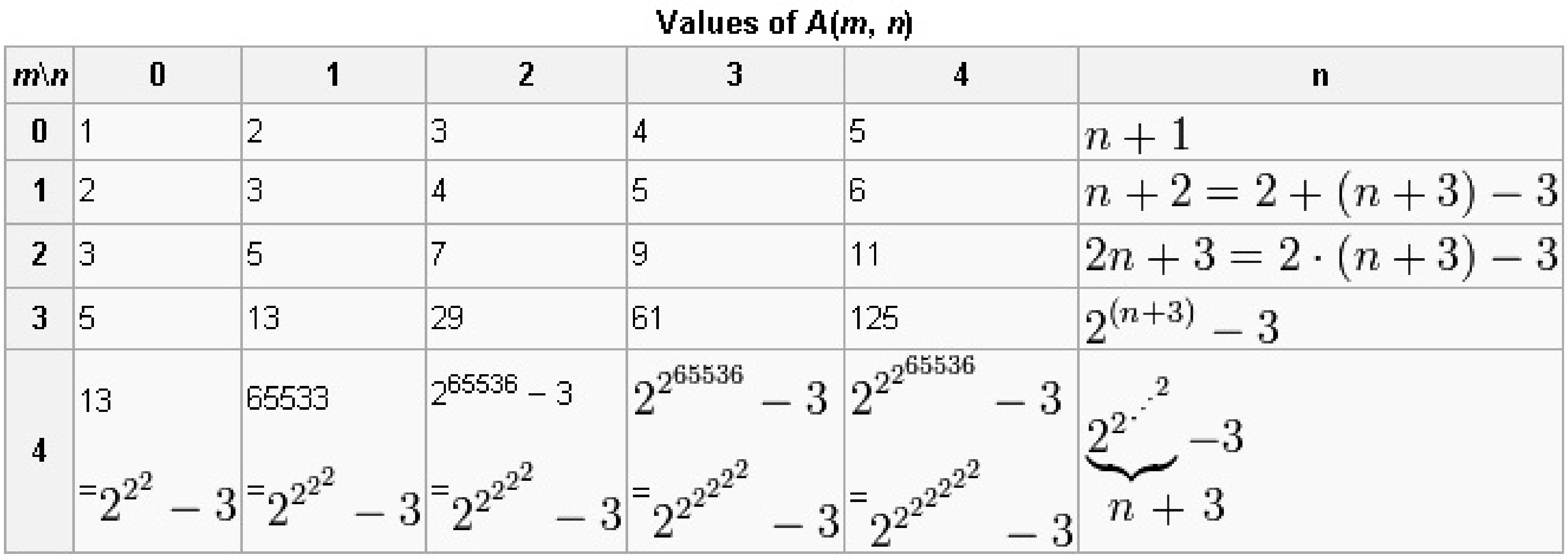
Exercise 3.3

- Ackermann’s Function is a recursive mathematical algorithm that can be used to test how well a system optimizes its performance of recursion
- The two-argument Ackermann–Péter function, is defined as follows for nonnegative integers *m* and *n*:

$$A(m,n) = \begin{cases} n+1 & \text{if} \quad m = 0 \\ A(m-1,1) & \text{if} \quad m > 0 \text{ and } n = 0 \\ A(m-1, A(m,n-1)) & \text{if} \quad m > 0 \text{ and } n > 0 \end{cases}$$

- Analyze the complexity of the program

Exercise 3.3 (cont'd.)



Values of $A(m, n)$

$m \backslash n$	0	1	2	3	4	n
0	1	2	3	4	5	$n+1$
1	2	3	4	5	6	$n+2=2+(n+3)-3$
2	3	5	7	9	11	$2n+3=2\cdot(n+3)-3$
3	5	13	29	61	125	$2^{(n+3)}-3$
4	13 $=2^{2^{2}}-3$	65533 $=2^{2^{2^{2}}}-3$	$2^{65536}-3$ $=2^{2^{2^{2^{2}}}}-3$	$2^{2^{65536}}-3$ $=2^{2^{2^{2^{2^{2}}}}}-3$	$2^{2^{2^{65536}}}-3$ $=2^{2^{2^{2^{2^{2^{2}}}}}}-3$	$\underbrace{2^{2^{\cdot^{\cdot^{2}}}}}_{n+3}-3$

Appendix: Mathematical Review

- Summations
  - Mathematical notation uses a symbol that compactly represents summation of many similar terms: the *summation symbol*, ∑, an enlarged form of the upright capital Greek letter Sigma. This is defined as:

$$\sum_{i=a}^{b} f(i) = f(a) + f(a+1) + f(a+2) + \cdots + f(b)$$

  - For any integer $n \geq 0$ and any real number $0 < a \neq 1$, consider

$$\sum_{i=0}^{n} a^i = 1 + a + a^2 + \cdots + a^n$$

if $a < 1$,

$$\sum_{i=0}^{n} a^i = \frac{1 - a^{n-1}}{1 - a}$$

Appendix: Mathematical Review (cont'd.)

- In mathematics, an arithmetic progression or arithmetic sequence is a sequence of numbers such that the difference between the consecutive terms is constant

$$\sum_{i=1}^{n} i = 1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2}$$

$$\sum_{i=1}^{n} i^2 = 1^2 + 2^2 + 3^2 + \cdots + n^2 = \frac{n(n+1)(2n+1)}{6}$$

$$\sum_{i=1}^{n} i^3 = 1^3 + 2^3 + 3^3 + \cdots + n^3 = [\frac{n(n+1)}{2}]^2$$

Appendix: Mathematical Review (cont'd.)

- Logarithms and exponents
  - $\log_b a = c$ if $a = b^c$
  - $\log_b ac = \log_b a + \log_b c$
  - $\log_b a/c = \log_b a - \log_b c$
  - $\log_b a^c = c\log_b a$
  - $\log_b a = (\log_c a) / \log_c b$
  - $b^{\log_c a} = a^{\log_c b}$
  - $(b^a)^c = b^{ac}$
  - $b^a b^c = b^{a+c}$
  - $b^a/b^c = b^{a-c}$

Appendix: Pseudocode

- High-level description of an algorithm
- More structured than English prose
- Less detailed than a program
- Preferred notation for describing algorithms
- Hides program design issues

Appendix: Pseudocode (cont'd.)

- Control flow
  - if … then … [else …]
  - while … do …
  - repeat … until …
  - for … do …
  - Indentation replaces braces
- Method declaration

  Algorithm method (*arg* [, *arg*…])

  Input …

  Output …
- Array indexing

  A[*i*] the *i*-th cell in the array *A*

- Method/Function call

  var.method (*arg* [, *arg*…])
- Return value

  return *expression*
- Expressions

  $\leftarrow$ Assignment (like =)

  $=$ Equality testing (like ==)

  n^2 Superscripts and other mathematical formatting allowed

Summary

- Experimental study versus Theoretical analysis
- Asymptotic computational complexity $O()$
  - Time complexity
  - Space complexity
  - Network bandwidth complexity
- Recursion
  - Factorial
  - Fibonacci
- Analyzing recursive algorithms
  - Backward substitution
  - Recursive tree
- Recursion with memoization