Java Programming

Zheng-Liang Lu

Department of Computer Science & Information Engineering
National Taiwan University

Java 415
Summer 2024

```
1 class Lecture5 {
2
3                 "Methods and Recursion"
4
5 }
6
7 // Keywords:
8 return, var
```

Introduction

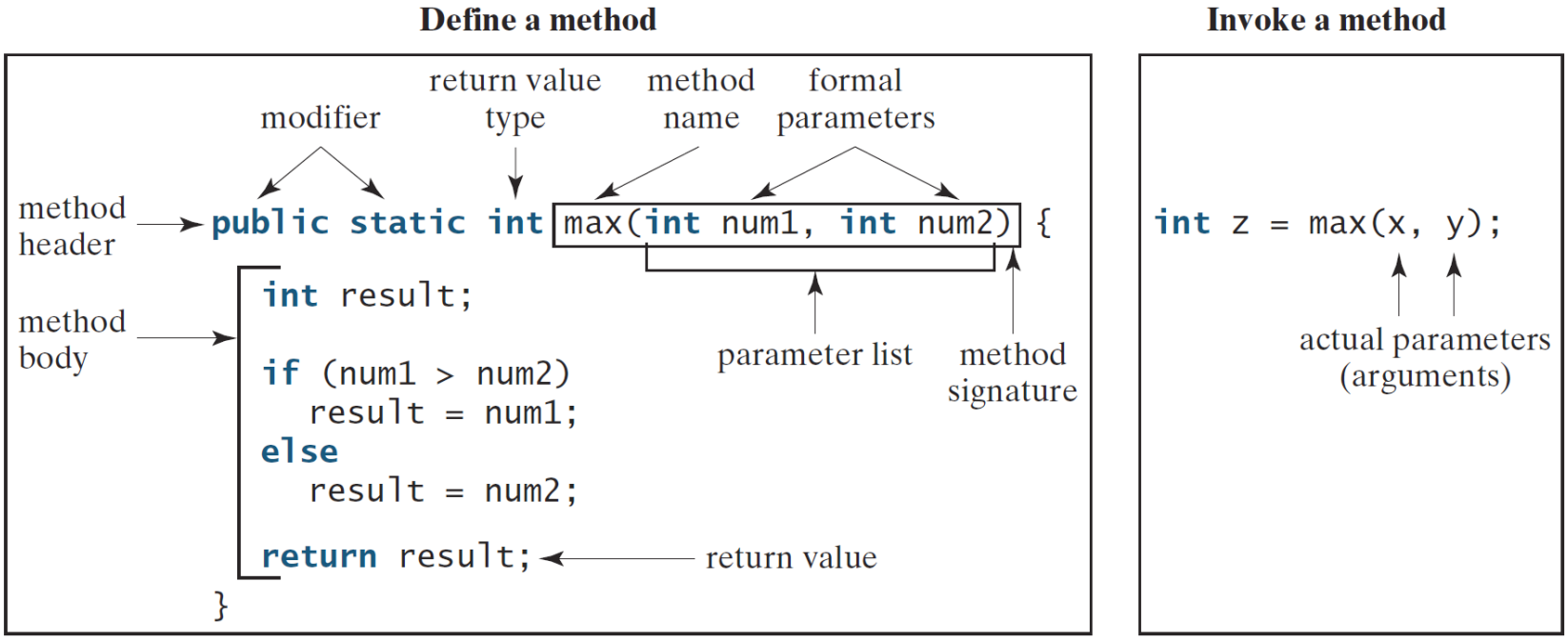
- Methods[1] are used to define reusable codes, so that it could organize and simplify your programs.
- The idea of methods originates from math, like

$$f(x, y),$$

  where x and y are its two input parameters.
- Every parameter should be declared with one specific type.
- Every method needs one return type even if it has no return!

[1] As known as functions, procedures and subroutine.

Example: max



- The method signature comprises its name and parameter list.

Alternatives?

```
1  ...
2      public static int max(int num1, int num2) {
3
4          if (num1 > num2) {
5              return num1;
6          } else {
7              return num2;
8          }
9
10     }
11 ...
```

```
1  ...
2      public static int max(int num1, int num2) {
3
4          return num1 > num2 ? num1 : num2;
5
6      }
7  ...
```

"All roads lead to Rome."

– Anonymous

"但如你根本並無招式，敵人如何來破你的招式？"

– 風清揚 (笑傲江湖。第十回。傳劍)

About the return Statement

- The return statement terminates the method.
- A caller invokes one method, called the callee.
- The caller and the callee should follow the method header, like a contract.
- The caller provides the callee with adequate inputs and receives one return value from the callee (or none if the return type is void).
- Note that a method could have more than one return statement.

Pitfalls

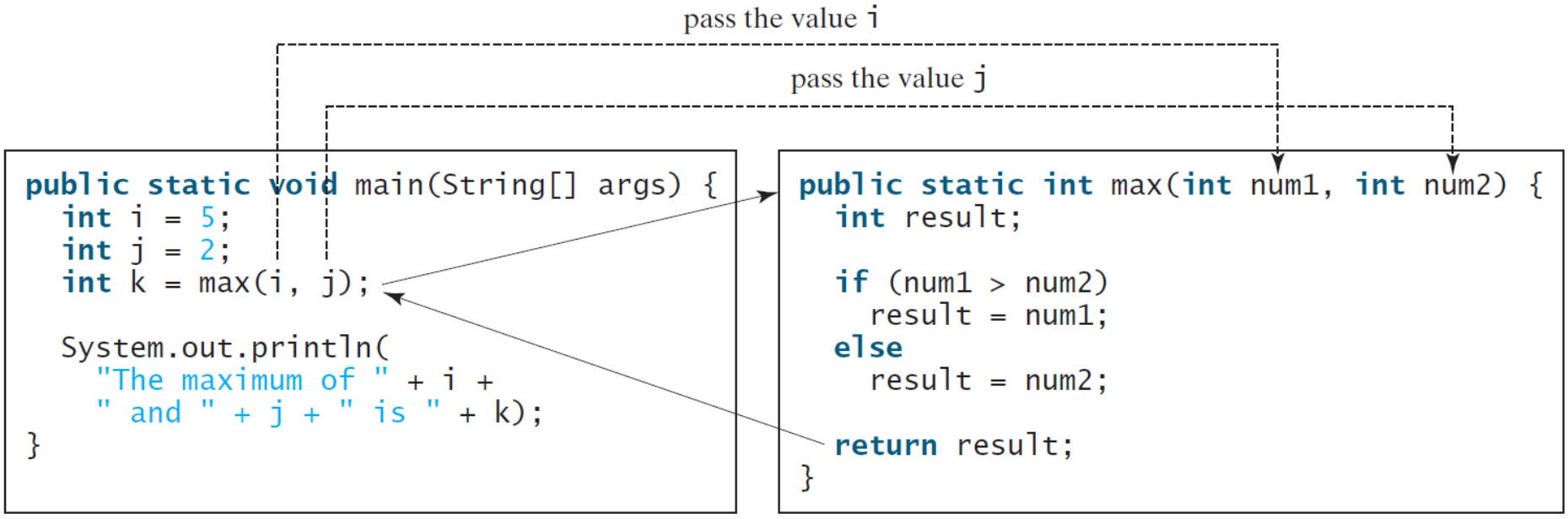
```
1  ...
2      public static int foo1() {
3
4          while (true);
5          return 0; // Unreachable code.
6
7      }
8
9      public static int foo2(int x) {
10
11         if (x > 0)
12             return x; // What if x <= 0? Not allowed.
13
14     }
15 ...
```

- Stick to the contract!

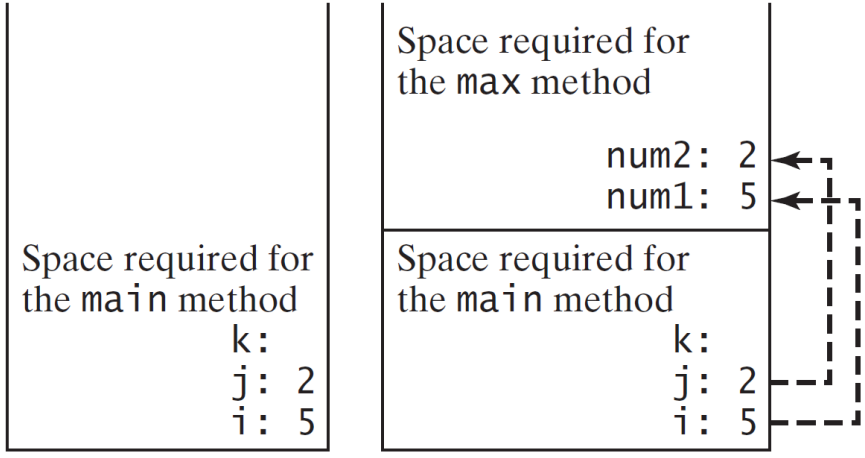
More Examples

```
1  ...
2      // Method w/o return.
3      public static void display(int[] A) {
4
5          for (int item : A)
6              System.out.printf("%d ", item);
7          System.out.println();
8
9      }
10
11     // Method returning array (reference)!
12     public static int[] arrayFactory(int size, int low, int high) {
13
14         int[] A = new int[size];
15         int numOfStates = high - low + 1;
16         for (int i = 0; i < A.length; i++)
17             A[i] = (int) (Math.random() * numOfStates) + low;
18         return A;
19
20     }
21 ...
```

Method Invocation

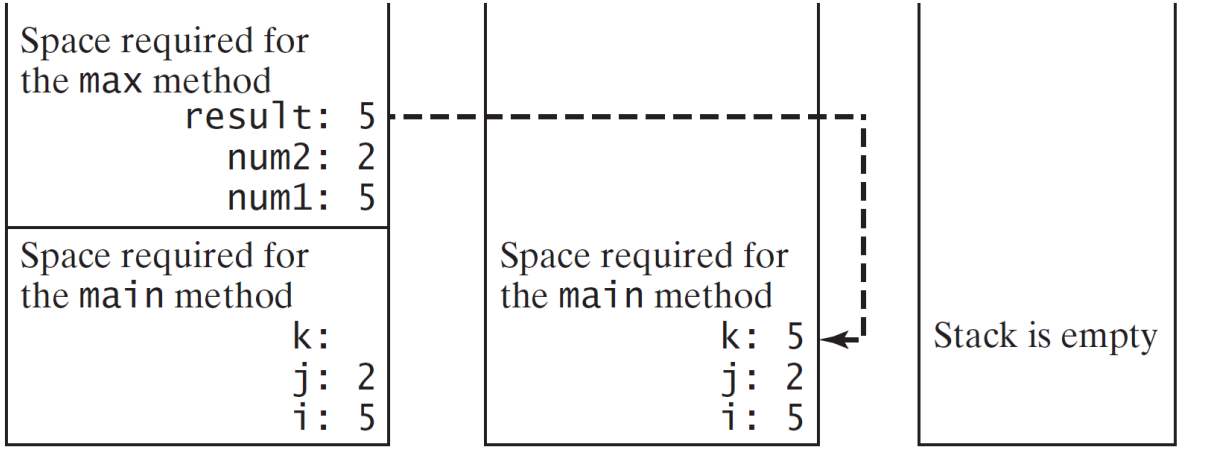


- The formal parameters are sort of variables declared within the method as placeholders.
- When invoking the method, the caller passes (copies) the arguments to the callee, in order and compatible type.
- This is called passing by value.



(a) The `main` method is invoked.

(b) The `max` method is invoked.



(c) The `max` method is being executed.

(d) The `max` method is finished and the return value is sent to `k`.

(e) The `main` method is finished.

- The JVM pushes a frame into the call stack[2], which stores the arguments and other necessary information for each method invocation.
- Once the method reaches any return statement (or the bottom of the method), the frame will be nullified and the JVM returns to where the caller jumps from.
- It also implies that the memory space occupied by the frame will be recycled for next method invocation.
- Note that the execution flow of method invocation is the central concept of recursions.

[2] A data structure with the first-in-last-out (FILO) property is called a stack.

Variable Scope

- A variable scope is the region where one variable is visible.
- A variable has one of the following three scopes: class level[3], (method) local level, and loop level[4].
- As a local variable, any changes made inside the method does not affect the original value.
- Note that one local variable can have its name identical to the one of class level.
- This is called the shadow effect because we favor the local one. (Why?)

[3] We will discuss about this kind later in the next chapter.

[4] We've discussed the loop variables in the chapter of flow control: Any variable declared in the loop is invisible when the loop is finished.

Example

```
1  public class ScopeDemo {
2
3      public static int x = 10;  // Class level; global variable.
4
5      public static void main(String[] args) {
6
7          System.out.println(x); // Output 10.
8          int x = 100; // Method level, aka local variable.
9          x++;
10         System.out.println(x); // Output 101.
11         addOne();
12         System.out.println(x); // Output? Why?
13
14     }
15
16     public static void addOne() {
17
18         x = x + 1;
19         System.out.println(x); // Output?
20
21     }
22 }
```

Local Variable Type Inference[6]

```
1 ...
2         var x = 100;                    // x will be an integer.
3         var y = "type inference";       // y will be a string.
4         var z = new ArrayList<>(); // z will be an ArrayList object.
5 ...
```

- Type inference is a compiler's ability to automatically infer unspecified data type parameters from contextual information.
- It allows us to write more concise Java code when it comes to generics and lambda expression[5].
- Note that this is applicable only for local variables.

[5] We will meet lambda expressions soon.

[6] Added in JDK 10. See Java 10 Local Variable Type Inference.

Manual for Math Toolbox: **Math** Class

- The **Math** class provides basic math functions and two global constants **Math**.PI and **Math**.E.
- Check out the official document for **Math**.[7]
- As you can see, its methods are all public and static.
- As a professional programmer, you should be capable to read documents (manuals) to survive in the future![8]

[7] See Math from Oracle's official document.

[8] You may hear about RTFM: https://en.wikipedia.org/wiki/RTFM.

Method Overloading

- Naming conflict is allowed when methods with the same name can be identified by method signatures.

```
1 ...
2         public static int max(int x, int y) { ... }
3
4         // Differ in types.
5         public static double max(double x, double y) { ... }
6
7         // Differ in numbers of inputs.
8         public static int max(int x, int y, int z) { ... }
9 ...
```

- Note that this mechanism does not relate to the return type.

Special Issue: Variadic Functions[9]

```
1  ...
2        // You don't have to do these below:
3        // public static int max(int n1, int n2) { /* Ignored */ }
4        // public static int max(int n1, int n2, int n3) { /* Ignored */
5
6        public static int max(int... nums) { /* Ignored */ }
7        // The above method definition is equivalent to
8        // public static int max(int[] nums) { /* Ignored */ }
9
10       public static void main(String[] args) {
11
12             int x = max(100, 200, 300);
13             int y = max(100, 200, 300, 400);
14
15       }
16 ...
```

- The ellipsis (...) allows the user to pass an arbitrary number of arguments to the method.

[9] Since JDK5. It is one of syntactic sugars.

The Entry Method: main(**String**[] args)

- You can start the program together with a series of strings.
- Those attached strings are stored in one **String** array as the program parameters.

```
1 ...
2     public static void main(String... args) {
3
4         for (String arg : args)
5             System.out.println(arg);
6
7     }
8 ...
```

- In Eclipse, you may turn on the input dialog by adding "${string_prompt}" as a program argument to JVM.
- You can also compile and run the program in the command line interface (CLI).

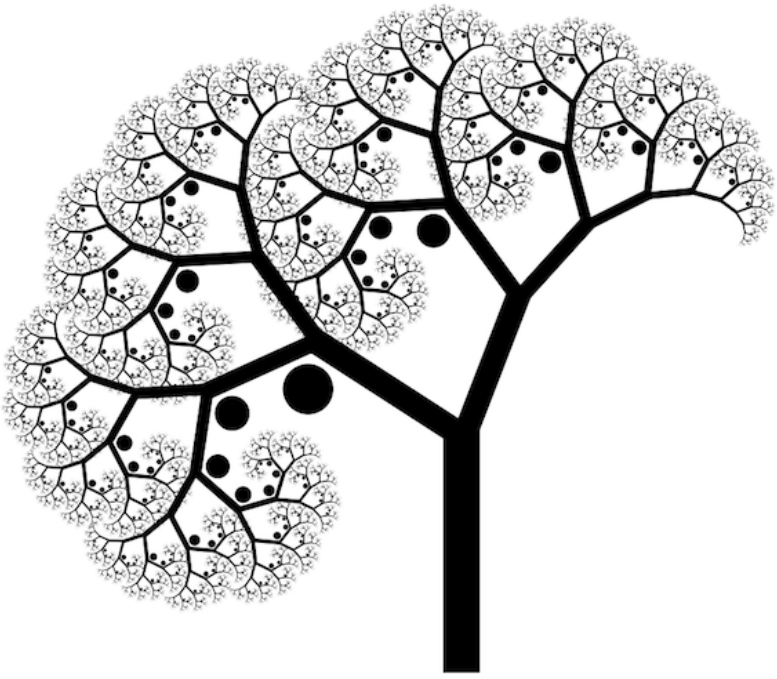
Recursion[10]

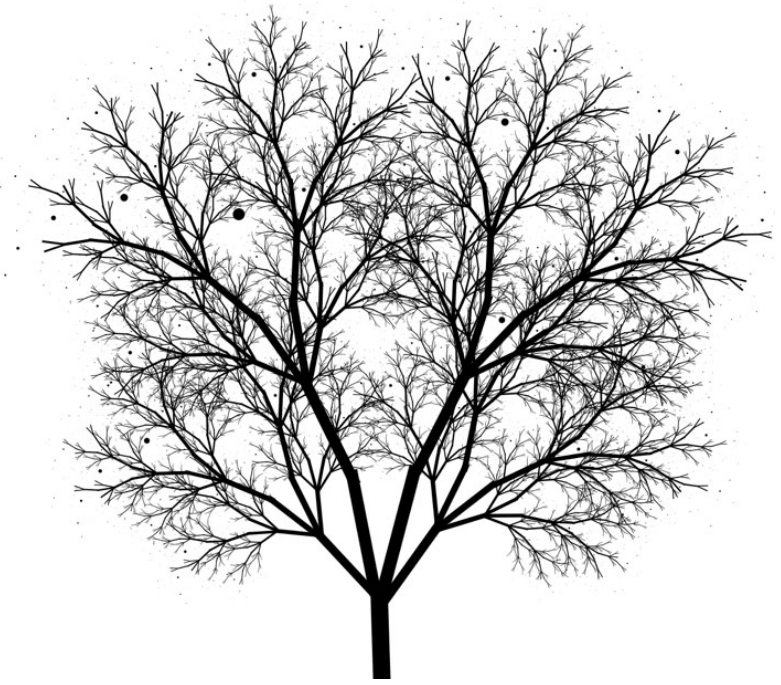
Recursion is a process of defining something in terms of itself.

- A method that calls itself in some way is recursive.
- Recursion is an alternative form of repetition without any loop.

[10] Recursion is a commom pattern in nature.

Examples of Natural World





- Try Fractal.

Example: Factorial (Revisited)

Write a program to determine $n!$ by recursion.

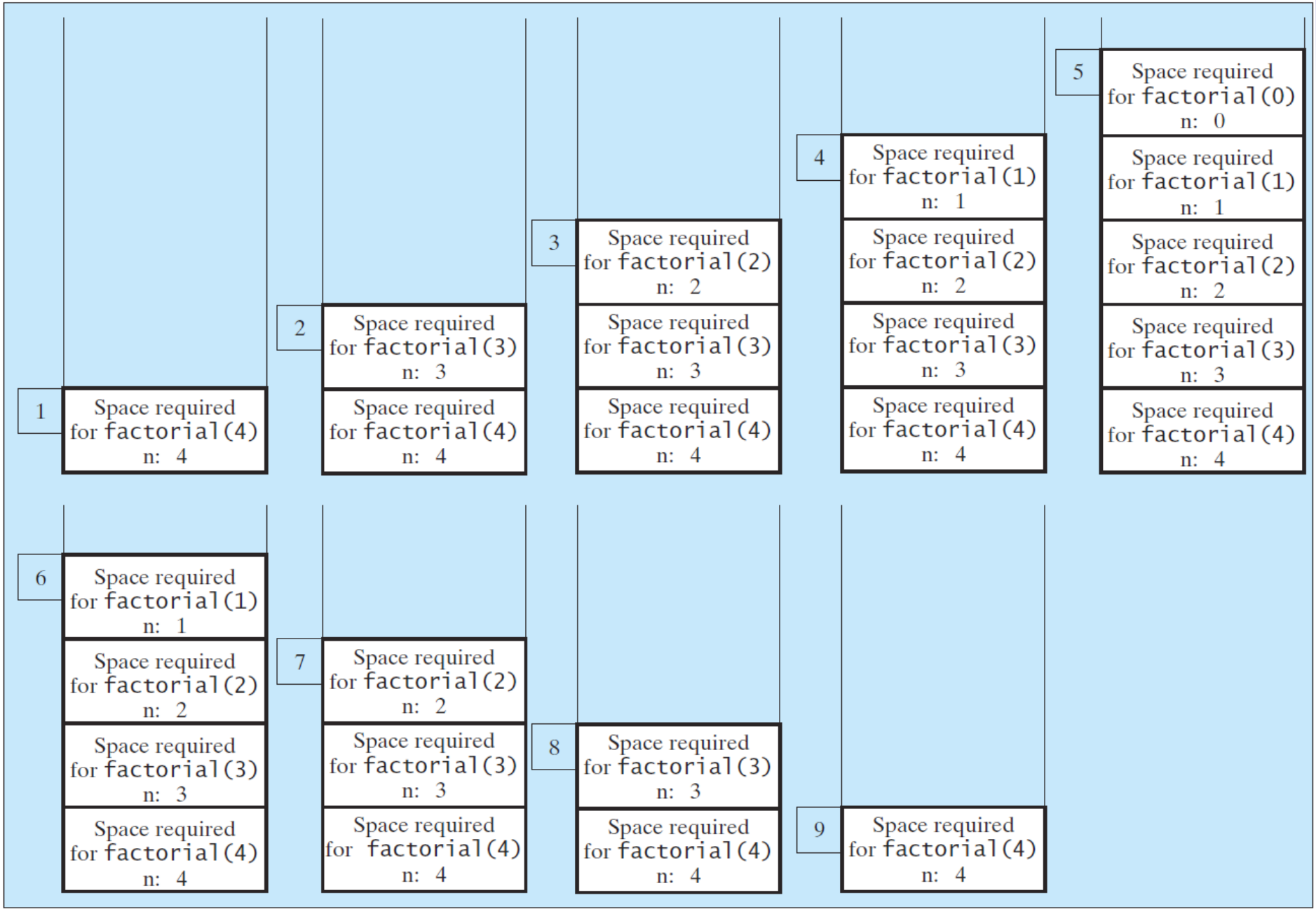
- For example,

$$
\begin{aligned}
4! &= 4 \times 3 \times 2 \times 1 && \text{(in view of loops)} \\
&= 4 \times 3! && \text{(in view of recursion)} \\
&= 4 \times (3 \times 2!) \\
&= 4 \times (3 \times (2 \times 1!)) \\
&= 4 \times (3 \times (2 \times (1 \times 0!))) \\
&= 4 \times (3 \times (2 \times (1 \times 1))) \\
&= 24.
\end{aligned}
$$

- Find any recursive pattern?

```
1  ...
2      public static int factorial(int n) {
3
4          if (n < 2)
5              return 1;                          // base case
6          else
7              return n * factorial(n - 1); // recurrence relation
8
9      }
10 ...
```

- Remember to set a base case in recursion. (Why?)
- What is the time complexity?


1
Space required for factorial(4) n: 4
2
Space required for factorial(3) n: 3
Space required for factorial(4) n: 4
3
Space required for factorial(2) n: 2
Space required for factorial(3) n: 3
Space required for factorial(4) n: 4
4
Space required for factorial(1) n: 1
Space required for factorial(2) n: 2
Space required for factorial(3) n: 3
Space required for factorial(4) n: 4
5
Space required for factorial(0) n: 0
Space required for factorial(1) n: 1
Space required for factorial(2) n: 2
Space required for factorial(3) n: 3
Space required for factorial(4) n: 4
6
Space required for factorial(1) n: 1
Space required for factorial(2) n: 2
Space required for factorial(3) n: 3
Space required for factorial(4) n: 4
7
Space required for factorial(2) n: 2
Space required for factorial(3) n: 3
Space required for factorial(4) n: 4
8
Space required for factorial(3) n: 3
Space required for factorial(4) n: 4
9
Space required for factorial(4) n: 4

```
1 ...
2         int s = 1;
3         for (int i = n; i > 1; i--) {
4             s *= i;
5         }
6 ...
```

- Both run in $O(n)$ time.
- One intriguing question is, Can we always turn a recursive method into a loop version of that?
  - Affirmative.
  - The Church-Turing Thesis[11] implies that both are equivalent.

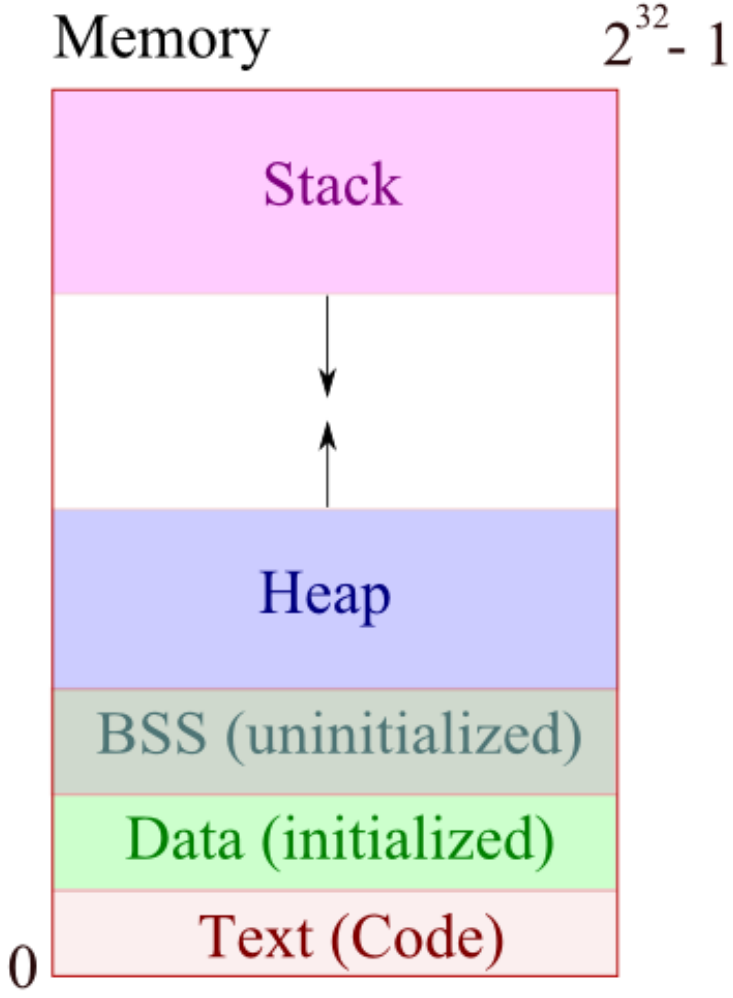
[11] See http://plato.stanford.edu/entries/church-turing/.

Remarks

- Recursion bears substantial overhead.
- So the recursive algorithm may execute a bit more slowly than the iterative equivalent.
- Moreover, a deep recursion depletes the call stack, which is limited, and causes the error **StackOverflowError**.[12]

[12] See https://stackoverflow.com/, https://www.oreilly.com/, and https://www.quora.com/Does-reading-Copying-and-Pasting-from-Stack-Overflow-mak

Memory Layout



Exercise: Summation (Revisited)

Write a function to calculate the sum from 1 to n by recursion.

- For example,

$$\begin{aligned}
\text{sum}(100) &= 100 + \text{sum}(99) \\
&= 100 + 99 + \text{sum}(98) \\
&= 100 + 99 + 98 + \text{sum}(97) \\
&\vdots \\
&= 100 + 99 + 98 + \cdots + 1.
\end{aligned}$$

- Can you find the recurrence relation?

```
1 ...
2     public static int sum(int n) {
3
4         if (n == 1)
5             return 1;
6         return n + sum(n - 1);
7
8     }
9 ...
```

- Time complexity?

Exercise: Greatest Common Divisor (GCD)

Let a and b be two positive integers. Calculate GCD(a, b) by recursion.

- We implement the Euclidean algorithm for GCD.[13]
- For example,

$$\begin{aligned} \text{GCD}(54, 32) &= \text{GCD}(32, 22) \\ &= \text{GCD}(22, 10) \\ &= \text{GCD}(10, 2) \\ &= 2. \end{aligned}$$

[13] See https://en.wikipedia.org/wiki/Euclidean_algorithm.

```
...
    public static int gcd_by_recursion(int a, int b) {

        int r = a % b;
        if (r == 0)
            return b;
        return gcd_by_recursion(b, r); // Straightforward?!

    }
...
```

```
...
    public static int gcd_by_loop(int a, int b) {

        int r = a % b;
        while (r > 0) {
            a = b;
            b = r;
            r = a % b;
        }
        return b;

    }
...
```

Example: Fibonacci Sequence[14]

Let n be a nonnegative integer. Calculate the n-th Fibonacci number F_n.

- Set $F_0 = 0$ and $F_1 = 1$.
- For $n > 1$, the Fibonacci numbers follows the recurrence relation

$$F_n = F_{n-1} + F_{n-2}.$$

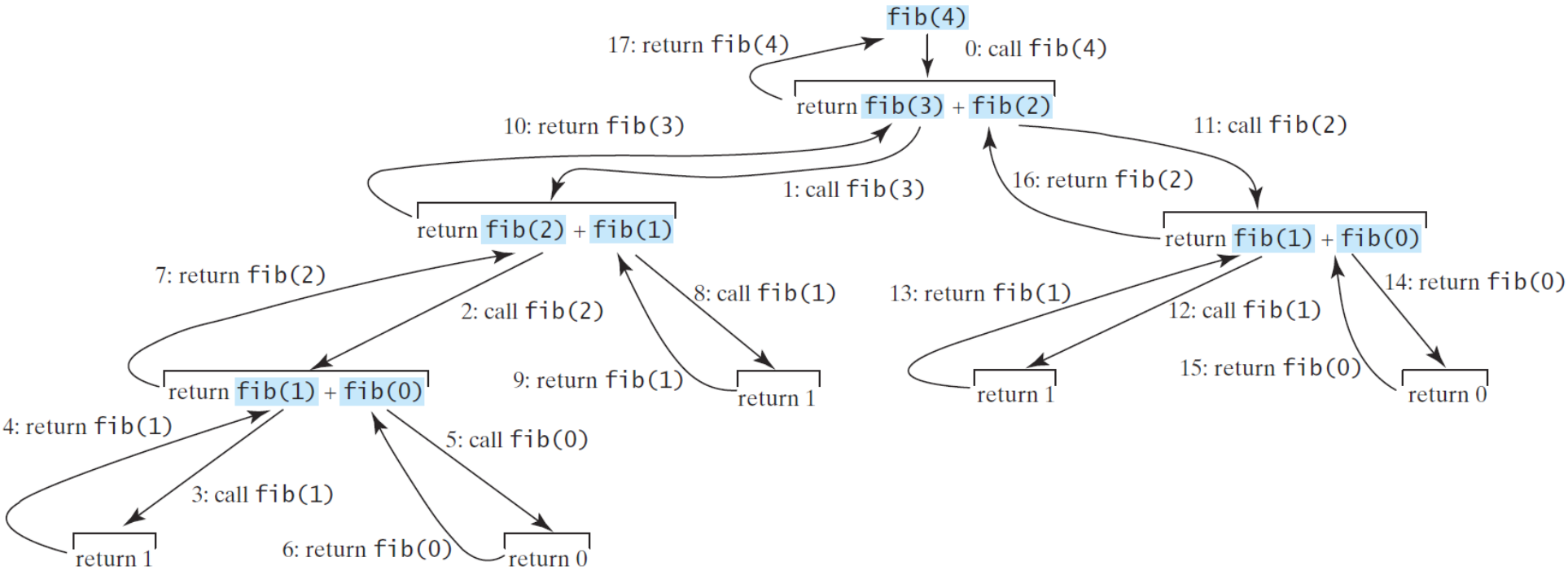
- The first 10 numbers are $0, 1, 1, 2, 3, 5, 8, 13, 21,$ and 34.

[14] See https://www.mathsisfun.com/numbers/fibonacci-sequence.html and https://en.wikipedia.org/wiki/Fibonacci_number.

```
1  ...
2      public static int fib(int n) {
3
4          if (n < 2) {
5              return n;
6          } else {
7              return fib(n - 1) + fib(n - 2);
8          }
9
10     }
11 ...
```

- Time complexity: $O(2^n)$. (Why!!!)
- This algorithm suffers from the performance issue.
  - Assume that the modern CPU can finish 10^9 times of method invocation per second.
  - Then it takes 36.6 years for F_{60}.[15]

[15] You could reproduce the number by calculating $2^{60}/(10^9 \times 86400)$.



- A binary tree with height level h has at most $2^h - 1$ nodes. (Why?)
- The algorithm runs in $O(2^n)$ time because its execution counts grow like a binary tree.
- Can we do better by avoiding recomputaions?

```
1  ...
2      public static double fib2(int n) {
3
4          if (n < 2) return n;
5
6          int x = 0, y = 1;
7          for (int i = 2; i <= n; i++) {
8              int z = x + y;
9              x = y;
10             y = z;
11         }
12         return y; // Why not z?
13
14     }
15 ...
```

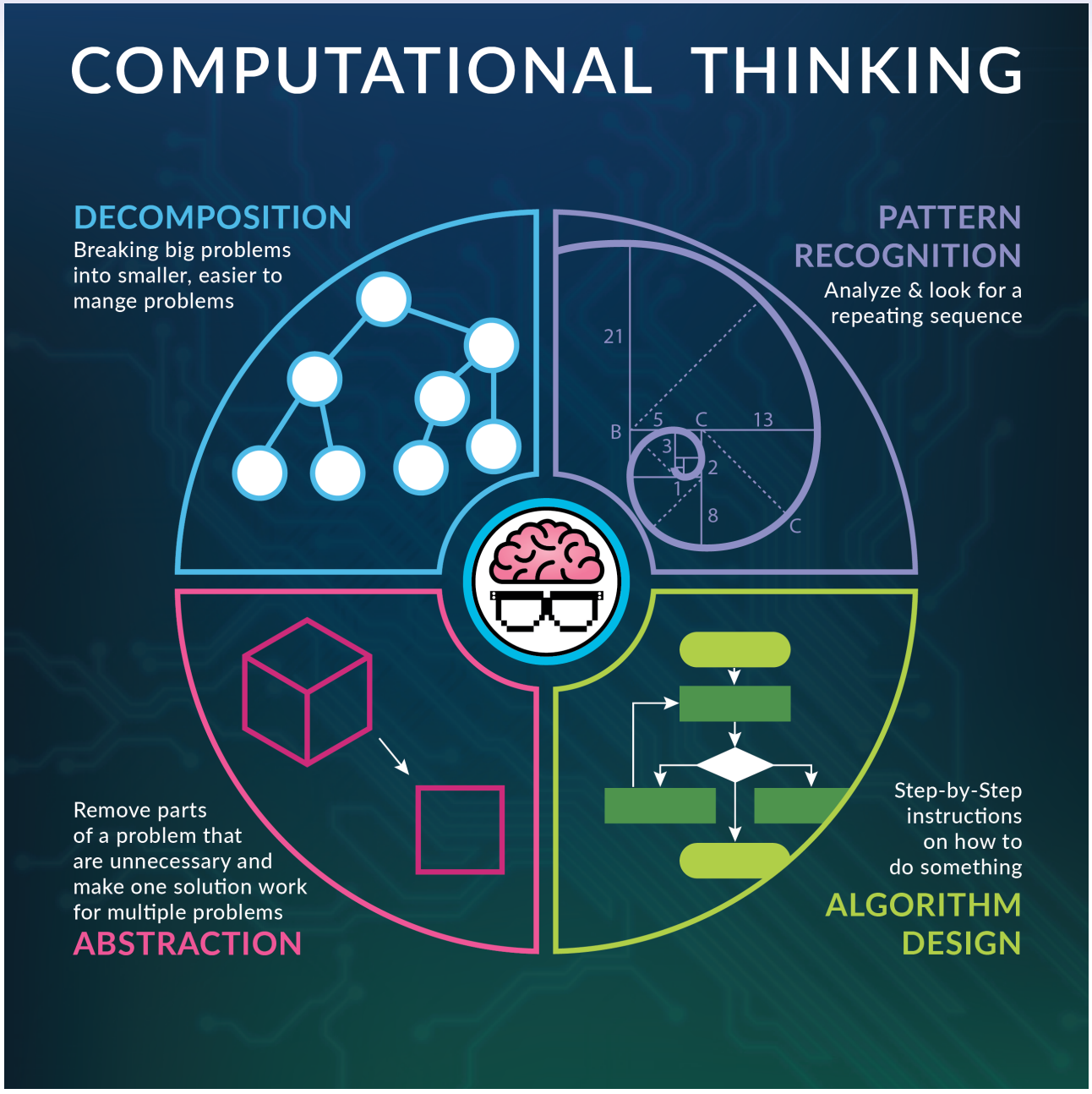
- The algorithm runs in $O(n)$ time!
- Could you find the $O(n)$-time recursive one?
- In fact, this problem can be solved in $O(\log n)$ time![16]

[16] See 509. Fibonacci Number of LeetCode.

Problem-Solving Skill: Divide & Conquer (DC)

- We often use the DC strategy to decompose the original problem into several manageable subproblems.
- It is also similar to do a study: narrow down to one doable topic and solve it.
- This approach benefits the program development, say easier to write, more possible to reuse, and better to facilitate teamwork.
- One more thing to note is that one method should not exceed, in principle, 20 lines of codes.[17]

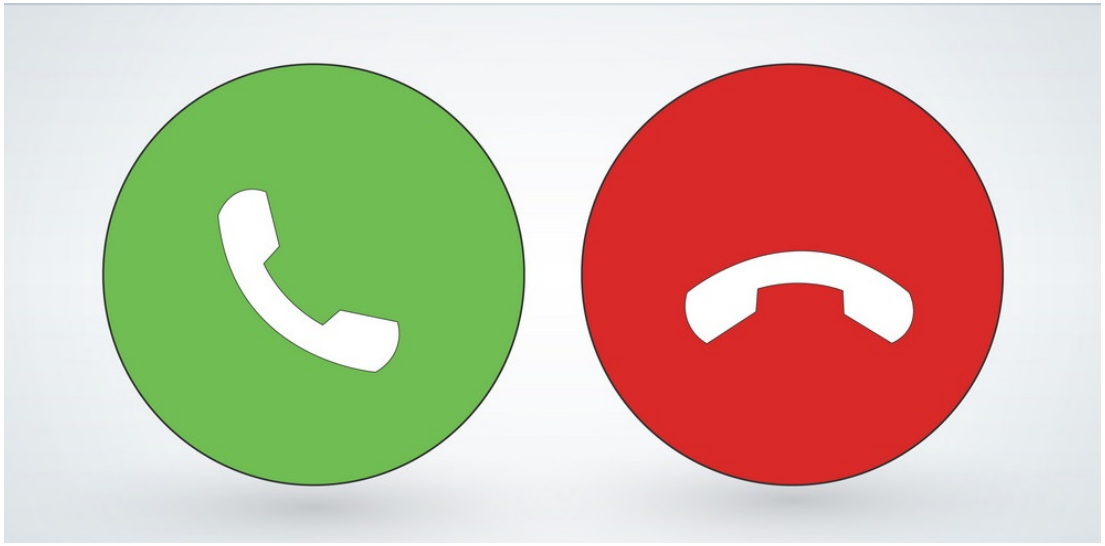
[17] See Clean Code.


COMPUTATIONAL THINKING
DECOMPOSITION
Breaking big problems into smaller, easier to mange problems
PATTERN RECOGNITION
Analyze & look for a repeating sequence
21
B
5
C
13
3
2
1
8
C
Remove parts of a problem that are unnecessary and make one solution work for multiple problems
ABSTRACTION
Step-by-Step instructions on how to do something
ALGORITHM DESIGN

Programming Concept: Abstraction

- Abstraction provides an interface to application programmers that separates policy from mechanism.
  - Policy: what the interface commits to accomplishing.
  - Mechanism: how the interface is implemented.
- This process enables us to build large and complex systems.
- Abstraction is everywhere, even in everyone's daily life.
- You can find a lot of similar experiences about abstraction.
  - For example, driving a car, writing Java programs.

Example: Graphical User Interface (GUI)



- You probably have no idea about electromagnetism and communication systems.
- However, you know how to make a phone call because you are familiar to its user interface!

Conclusions

- Methods are control abstractions while data structures are data abstractions.
- We can treat the notion of objects as a way to combine data and control abstractions.
- For example, try to enumerate the data with its associated controls in your cellphone.
  - Data: phone book, photo album, music library, clips, etc.
  - Controls? The buttons you can press in those apps.
- We will start with the object-oriented programming (OOP) paradigm in the next chapter.

"Abstraction is selective ignorance."

"We can solve any problem by introducing an extra level of indirection."

– Andrew Koenig (1952–)

"Being abstract is something profoundly different from being vague... The purpose of abstraction is not to be vague, but to create a new semantic level in which one can be absolutely precise."

– Edsger Dijkstra (1930–2002)