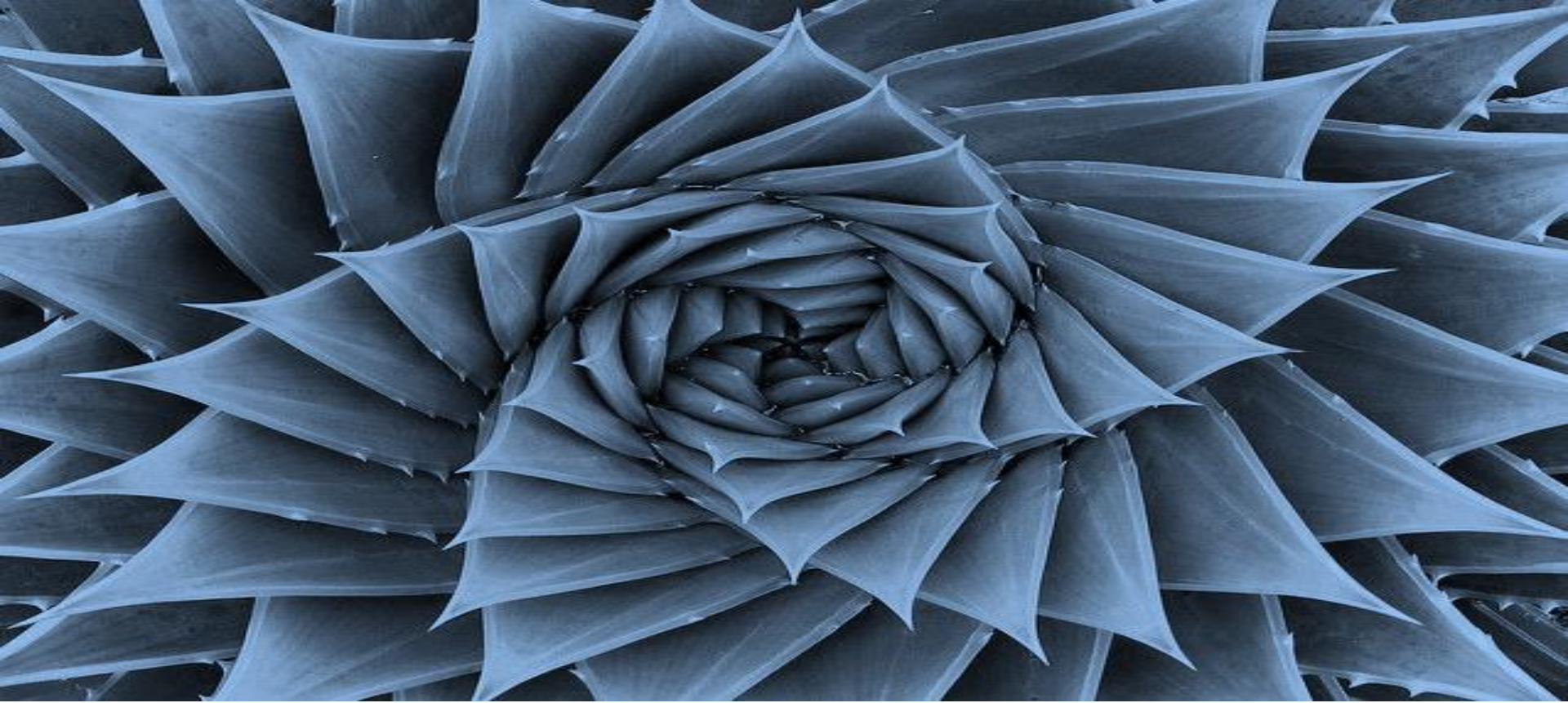


Advanced Object-Oriented Programming

CPT204 – Lecture 3

Erick Purwanto

Xi'an Jiaotong-Liverpool University

西交利物浦大学

CPT204 Advanced Object-Oriented Programming

Lecture 3

Coding Rules, Testing 3, Recursion

Welcome !

- Welcome to Lecture 3 !

- In this lecture we are going to
  - learn about Coding Rules
  - continue our discussion about Testing
    - how to choose the test suite
  - review Recursion and Recursion + Helper Method

Part 1: Coding Rules and Testing 3

There are sensible **coding rules** that help us to write good code: making it safe from bugs, easy to understand, and ready for change:

- Don't Repeat Yourself (DRY)
- Comments where needed
- Fail fast
- Avoid magic numbers
- One purpose for each variable
- Use good names
- Don't use global variables
- Return results, don't print them
- Use whitespace for readability

Don't Repeat Yourself

- Duplicated code is a risk to safety: if you have *identical* or *very similar code* in *two* places, then the fundamental risk is that there's a bug in both copies, and some maintainer fixes the bug in one place but not the other

- Avoid duplication like you'd avoid crossing the street without looking. Copy-and-paste is an enormously tempting programming tool, and you should be careful every time you use it!
  - The longer the block you're copying, the riskier it is

- Don't Repeat Yourself, or DRY for short, has become a programmer's mantra

- The dayOfYear example in the next slides is **full of identical code**
  - How would you DRY it out?

Don't Repeat Yourself

- The code below is full of identical codes: 1) why is this a BAD code? 2) how do you DRY it ?

```
public static int dayOfYear(int month, int dayOfMonth, int year) {
    if (month == 2) {
        dayOfMonth += 31;
    } else if (month == 3) {
        dayOfMonth += 59;
    } else if (month == 4) {
        dayOfMonth += 90;
    } else if (month == 5) {
        dayOfMonth += 31 + 28 + 31 + 30;
    } else if (month == 6) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31;
    } else if (month == 7) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30;
    } else if (month == 8) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31;
    } else if (month == 9) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31;
    } else if (month == 10) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30;
    } else if (month == 11) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30 + 31;
    } else if (month == 12) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30 + 31 + 31;
    }
    return dayOfMonth;
}
```

Don't Repeat Yourself

- The code below is full of identical codes: 1) why is this a BAD code? 2) how do you DRY it ?

```
public static int dayOfYear(int month, int dayOfMonth, int year) {
    if (month == 2) {
        dayOfMonth += 31;
    } else if (month == 3) {
        dayOfMonth += 59;
    } else if (month == 4) {
        dayOfMonth += 90;
    } else if (month == 5) {
        dayOfMonth += 31 + 28 + 31 + 30;
    } else if (month == 6) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31;
    } else if (month == 7) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30;
    } else if (month == 8) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31;
    } else if (month == 9) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31;
    } else if (month == 10) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30;
    } else if (month == 11) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30 + 31;
    } else if (month == 12) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30 + 31 + 31;
    }
    return dayOfMonth;
}
```

1) because:
there repeated values e.g. 28,
and code, e.g. the dayofMonth +=
if you change a value/code,
you have to change all of them!

Don't Repeat Yourself

- The code below is full of identical codes: 1) why is this a BAD code? 2) how do you DRY it ?

```
public static int dayOfYear(int month, int dayOfMonth, int year) {
    if (month == 2) {
        dayOfMonth += 31;
    } else if (month == 3) {
        dayOfMonth += 59;
    } else if (month == 4) {
        dayOfMonth += 90;
    } else if (month == 5) {
        dayOfMonth += 31 + 28 + 31 + 30;
    } else if (month == 6) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31;
    } else if (month == 7) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30;
    } else if (month == 8) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31;
    } else if (month == 9) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31;
    } else if (month == 10) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30;
    } else if (month == 11) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30 + 31;
    } else if (month == 12) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30 + 31 + 31;
    }
    return dayOfMonth;
}
```

2) use an array or list to store the number of days once, and use loop to add and return the value

Comments Where Needed: Specification

- Good comments should make the code easier to understand, safer from bugs (because important assumptions have been documented), and ready for change
- One kind of crucial comment is **a specification (spec)**, which appears above a method or above a class and *documents the behavior* of the method or class
  - In Java, this is conventionally written as a Javadoc comment, meaning that it starts with /** and includes @-syntax, like @param and @return for methods
  - Here's an example of a spec:

```
/**
 * Compute a hailstone sequence.
 * For example, hailstone(5) = [5 16 8 4 2 1].
 * @param n starting number for sequence. Assumes n > 0.
 * @return hailstone sequence starting at n and ending with 1.
 */
```

specs document assumptions

Comments Where Needed: Document Source

- Another crucial comment is one that specifies the source of a piece of code that was copied or adapted from elsewhere. This is important for practicing software developers, when you adapt code you found on the web, for example:

```
// read a web page into a string
// see http://stackoverflow.com/questions/4328711/read-url-to-string-in-few-lines-of-java-code
String homepage = new Scanner(new URL("http://www.aaa.com").openStream(), "UTF-8")
                                .useDelimiter("\\A").next();
```

- One reason for documenting sources is *to avoid violations of copyright*
  - Small snippets of code on Stack Overflow are typically in the public domain, but code copied from other sources may be proprietary or covered by other kinds of open source licenses, which are more restrictive
- Another reason for documenting sources is that *the code can fall out of date*; the Stack Overflow answer from which this code came has evolved significantly in the years since it was first answered

Comments Where Needed: Bad Comments

- Some comments are bad and unnecessary
- Direct transliterations of code into English, for example, do nothing to improve understanding, because you should assume that your reader at least knows Java
  - For example:

```
while (n != 1) { // test whether n is 1
   ++i; // increment i
   l.add(n); // add n to l
}
```

don't write comment like this !

- But, obscure code should get a comment:

```
// base cases
if (n == 0 || n == 1) {
    return 1;
}
```

Fail Fast

- Fail fast means that code should *reveal its bugs as early as possible*
  - The earlier a problem is observed (the closer to its cause), the easier it is to find and fix
  - As we saw in Lecture 2, static checking fails faster than dynamic checking, and dynamic checking fails faster than no checking at all, which may produce a wrong answer that could corrupt subsequent computation

```
public static int dayOfYear(int month, int dayOfMonth, int year) {
    ...
}
```

- The `dayOfYear` method ***doesn't fail fast*** — if you pass it the arguments in *the wrong order*, it will **quietly** return *the wrong answer*, as you will see in Lecture Quiz this week
  - It needs more checking — either static checking or dynamic checking

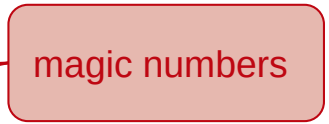
Avoid Magic Numbers

- There are really only two constants that computer scientists recognize as valid in and of themselves: 0, 1, and maybe 2 (ok, three constants -_-)

- All other constants are called **magic**, because they appear as if out of thin air with no explanation
  - One way to explain a number is with a comment,
    but a far better way is to declare the number as **a named constant**
    with ***a good, clear name***

- Our previous example, `dayOfYear`, is full of magic numbers

```
...
    } else if (month == 5) {
        dayOfMonth += 31 + 28 + 31 + 30;
...
```



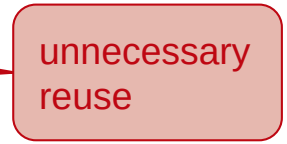
Avoid Magic Numbers: Better Ways

- The months `2`, …, `12` would be far more readable as `FEBRUARY`, …, `DECEMBER`
- The days-of-months `30, 31, 28` would be more readable (and DRY: eliminate duplicate code) if they were in a data structure like an array, list, or map
  - e.g. `MONTH_LENGTH[month]`
- The mysterious numbers 59 and 90 are particularly damaging examples of magic numbers!
  - Not only are they *uncommented* and *undocumented*,
    but they are also the result of a computation *done by hand* by the programmer
  - Don't hardcode constants that you've computed by hand!
    Java is better at arithmetic than you are!
    Explicit computations like `31 + 28` make the origin of these mysterious numbers much clearer
    `MONTH_LENGTH[JANUARY] + MONTH_LENGTH[FEBRUARY]` would be clearer still!

One Purpose for Each Variable

- In the `dayOfYear` example, the parameter `dayOfMonth` is *reused* to compute a very different value — the return value of the method, which is **not** the day of the month

```
public static int dayOfYear(int month, int dayOfMonth, int year) {
    if (month == 2) {
        dayOfMonth += 31;
    ...
    return dayOfMonth;
}
```



- Don't reuse parameters, and don't reuse variables
  - Variables are not a scarce resource in programming
  - Introduce them freely, give them good names, and just stop using them when you stop needing them
  - You will **confuse** your reader if a variable that used to mean one thing suddenly starts meaning something different a few lines down

One Purpose for Each Variable: Method Parameter

- Method parameters, in particular, should generally be *left unmodified*
  - This is important for being ready-for-change — in the future, some other part of the method may want to know what the original parameters of the method were, so you should *not* change them while you're computing

- It's a good idea to use `final` for method parameters, and as many other variables as you can
  - The `final` keyword says that the variable should *never be reassigned*, and the Java compiler will check it *statically*
  - For example:

```
public static int dayOfYear(final int month, final int dayOfMonth, final int year) {
    ...
}
```

Use Good Names

- Good method and variable names are long and self-descriptive
  - Comments can often be avoided entirely by making the code itself more readable, with better names that describe the methods and variables
  - For example, you can rewrite

```
int tmp = 86400;  // tmp is the number of seconds in a day
```

    as:

```
int secondsPerDay = 86400;
```

- In general, variable names like `tmp`, `temp`, and `data` are awful symptoms of extreme programmer laziness: every local variable is temporary, and every variable is data, so those names are generally meaningless
  - It is better to use a longer, more descriptive name, so that your code reads clearly all by itself

Use Good Names: Naming Conventions

- Follow the lexical naming conventions of the language

- In Java:
  - `methodsAreNamedWithCamelCaseLikeThis`
  - `variablesAreAlsoCamelCase`
  - `CONSTANTS_ARE_IN_ALL_CAPS_WITH_UNDERSCORES`
  - `ClassesAreCapitalized`
  - `packages.are.lowercase.and.separated.by.dots`

Use Good Names: Names

- Method names are usually verb phrases, like `getDate` or `isUpperCase`, while variable and class names are usually noun phrases
  - Choose short words, and be concise, but avoid abbreviations
  - For example, `message` is clearer than `msg`, and `word` is so much better than `wd`
    - Keep in mind that abbreviations can be even harder for non-native speakers

- `ALL_CAPS_WITH_UNDERSCORES` is used for static final constants

- All variables declared inside a method, including final ones, use `camelCaseNames`

- We have seen that a method named `leap` has a bad name
  - We named it `isLeapYear`

Don't Use Global Variables

- A global variable is:
  - a variable, a name whose meaning can be changed
  - that is global, accessible and changeable from *anywhere* in the program

- The following website has a good list of the dangers of global variables: http://web.archive.org/web/20160902115611/http://c2.com/cgi/wiki?GlobalVariablesAreBad

- In Java, a global variable is declared `public static`: the `public` modifier makes it accessible anywhere, and `static` means there is **a single instance** of the variable

- In general, **change** global variables into parameters and return values, or **put** them inside objects that you're calling methods on (make it instance/local variable)

Don't Use Global Variables: An Example

- How do you change this code, so that it doesn't use any global variables?

```
public static int LONG_WORD_LENGTH = 5;
public static String longestWord;

public static void countLongWords(List<String> words) {
    int n = 0;
    longestWord = "";
    for (String word: words) {
        if (word.length() > LONG_WORD_LENGTH) ++n;
        if (word.length() > longestWord.length()) longestWord = word;
    }
    System.out.println(n);
}
```

Return Results, Don't Print Them

- `countLongWords` in previous slide also isn't ready for change!
  - It sends some of its result to the console, `System.out`
  - That means that if you want to use it in another context — where the number is needed for some other purpose, like computation rather than human eyes — it would have to be rewritten!

- In general, only the highest-level parts of a program should interact with the human user or the console
  - Lower-level parts should take their input as parameters and return their output as results
  - The sole exception here is *debugging* output, which can of course be printed to the console. However, that kind of output shouldn't be a part of your design, only a part of how you debug your design

Use Whitespace for Readability

- Use ***consistent*** indentation
  - A very very bad example :

```
public static boolean leap(int y) {
    String tmp = String.valueOf(y);
    if (tmp.charAt(2) == '1' || tmp.charAt(2) == '3' || tmp.charAt(2) == 5 || tmp.charAt(2) == '7' || tmp.charAt(2) == '9') {
      if (tmp.charAt(3)=='2'||tmp.charAt(3)=='6') return true;
        else
            return false;
    }else{
        if (tmp.charAt(2) == '0' && tmp.charAt(3) == '0') {
            return false;
        }
            if (tmp.charAt(3)=='0'||tmp.charAt(3)=='4'||tmp.charAt(3)=='8')return true;
    }
    return false;
}
```

so painful to read, don't know where the if's end

- Never use tab characters for indentation, only space characters
  - Always set your programming editor to insert space characters when you press the Tab key

Testing and Bugs

- Testing means running the program on carefully selected inputs and checking the results
- Even with the best testing, it is very hard to achieve perfect quality in software
- Here are some **typical residual defect rates** (bugs left over after the software has shipped) per kloc (one thousand **l**ines **o**f source **c**ode):
  - 1 - 10 defects/kloc: Typical industry software
  - 0.1 - 1 defects/kloc: High-quality validation.
    The Java libraries might achieve this level of correctness.
  - 0.01 - 0.1 defects/kloc: The very best, safety-critical validation.
    NASA and aerospace companies can achieve this level.
- This can be discouraging for large systems
  - for example, if you have shipped a million lines of typical industry source code (1 defect/kloc), it means you missed 1000 bugs!

Why Software Testing is Hard (1)

Here are some approaches that unfortunately ***don't work well*** in the world of *software*:

- **Exhaustive testing** is *infeasible*
  The space of possible test cases is generally too big to cover exhaustively.
  Imagine exhaustively testing a 32-bit floating-point multiply operation, `a*b`.
  There are 2^{64} test cases!

- **Haphazard testing** ("just try it and see if it works") is *less likely to find bugs*, unless the program is so buggy that an arbitrarily-chosen input is more likely to fail than to succeed.
  It also doesn't increase our confidence in program correctness.

Why Software Testing is Hard (2)

- **Random or statistical testing** *doesn't work well for software*
  Other engineering disciplines can test small random samples (e.g. 1% of hard drives manufactured) and infer the defect rate for the whole production lot.
  - ***Physical systems*** can use many tricks to speed up time, like opening a refrigerator 1000 times in 24 hours instead of 10 years.
    These tricks give known failure rates (e.g. mean lifetime of a hard drive), but they assume continuity or uniformity across the space of defects.
    This is true for physical artifacts.
  - But, it's **not** true for ***software***.
    Software behavior varies discontinuously and discretely across the space of possible inputs.
    Although the system may seem to work fine across a broad range of inputs, it could still abruptly fail at a single boundary point.

Why Software Testing is Hard (3)

The famous Pentium division bug affected approximately ***1 in 9 billion divisions***. Stack overflows, out of memory errors, and numeric overflow bugs tend to happen abruptly, and always in the same way, not with probabilistic variation.

That's different from *physical systems*, where there is often visible evidence that the system is approaching a failure point (cracks in a bridge) or failures are distributed probabilistically near the failure point (so that statistical testing will observe some failures even before the point is reached).

- Instead, test cases must be chosen ***systematically***
  - **A test case** is a particular choice of inputs, along with the expected output behavior required by the specification
  - **A test suite** is a set of test cases for an implementation

Systematic Testing (1)

Systematic testing means that we are choosing test cases in a principled way, with the goal of designing a test suite with three desirable properties:

- **Correct**. A correct test suite is a legal client of the specification, and it accepts all legal implementations of the spec without complaint
  - This gives us the freedom to change how the module is implemented internally without necessarily having to change the test suite

- **Thorough**. A thorough test suite finds actual bugs in the implementation, caused by mistakes that programmers are likely to make

Systematic Testing (2)

- **Small**. A small test suite, with few test cases, is faster to write in the first place, and easier to update if the specification evolves
  - Small test suites are also faster to run
  - You will be able to run your tests more frequently if your test suites are small and fast

- By these criteria,
  - Exhaustive testing is thorough but infeasibly large
  - Haphazard testing tends to be small but not thorough
  - Randomized testing can achieve thoroughness only at the cost of large size

Putting on Your Testing Hat

- Testing requires having the *right attitude*
- When you're coding, your goal is to make the program work,
  but as a tester, you want to **make it fail**
- That's a subtle but important difference
  - it's all too tempting to treat code you've just written as a *precious* thing, a fragile eggshell, and test it ever so lightly just to see it work
- Instead, you have to be *brutal*
  - A good tester wields a sledgehammer and beats the program everywhere it might be vulnerable,
    so that those vulnerabilities can be eliminated

Test-First Programming

In test-first-programming, you write tests before you even write any code

The development of a *single* function proceeds in this order:

1. **Spec**: Write a *specification* for the function
2. **Test**: Write *tests* that exercise the specification
3. **Implement**: Write the actual code

   Once your code passes the tests you wrote, you're done

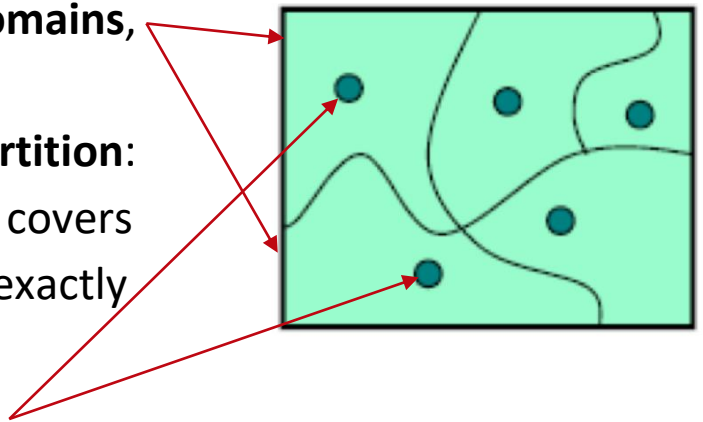
The Specification

- **The specification (spec)** describes the input and output behavior of the function
  - It gives the types of the parameters and any additional constraints on them (e.g. sqrt's parameter must be nonnegative)
  - It also gives the type of the return value and describes how the return value relates to the inputs
  - You've already seen and used specifications on your *programming exercises* and *assignments* in this class
  - In code, the specification consists of ***the method signature*** and ***the comment above it that describes what it does***
- Writing tests first is a good way to understand the specification. The specification can be buggy, too — *incorrect, incomplete, ambiguous, missing corner cases* Trying to write tests can uncover these problems early, before you've wasted time writing an implementation of a buggy spec

Choosing Test Cases by Partitioning

- Creating a good test suite is a challenging and interesting design problem
  - We want to pick a set of test cases that is ***small enough*** to run quickly, yet ***large enough*** to validate the program

- To do this, we divide the input space into **subdomains**, each consisting of a set of inputs
  - Taken together the subdomains form **a partition**: a collection of *disjoint* sets that *completely* covers the input space, so that every input lies in exactly one subdomain
  - Then we choose **one test case from each subdomain**, and that's our test suite



Subdomains

- The idea behind subdomains is to partition the input space into sets of ***similar inputs*** on which the program has ***similar behavior***
  - Then we use ***one*** representative of each set
  - This approach makes the best use of limited testing resources by choosing dissimilar test cases, and
  - forcing the testing to explore parts of the input space that random testing might not reach

- We can also partition the *output space* into subdomains (similar outputs on which the program has similar behavior) if we need to ensure our tests will explore different parts of the output space
  - Most of the time, partitioning the input space is sufficient

Example 1: BigInteger.multiply()

Let's look at an example:

- `BigInteger` is a class built into the Java library that can represent integers of *any size*, unlike the primitive types int and long that have only limited ranges.
  `BigInteger` has a method `multiply` that multiplies two `BigInteger` values together:

```
/**
 * @param val another BigInteger
 * @return a BigInteger whose value is (this * val).
 */
public BigInteger multiply(BigInteger val)
```

its spec

- Here's how it might be used:

```
BigInteger a = new BigInteger("9500000000");   // 9.5 billion
BigInteger b = new BigInteger("2");
BigInteger c = a.multiply(b);          // should be 19 billion
```

BigInteger.multiply() Arguments

- The example shows that even though only one parameter is explicitly shown in the method's declaration, `multiply` is actually a function of *two arguments*:
  - the object you're calling the method on (`a` in the example),
    and the parameter that you're passing in the parentheses (`b` in the example)
- In Java, you don't mention the receiving object in the parameters,
  and it's called **`this`**
- So we should think of multiply as a function taking two inputs, each of type `BigInteger`, and producing one output of type `BigInteger`:
  - `multiply : BigInteger × BigInteger → BigInteger`

Partition the Input Space (1)

So we have a two-dimensional input space, consisting of all the pairs of integers `(a,b)`

Now let's partition it!

Thinking about how multiplication works, we might start with these partitions:

- `a` and `b` are both positive
- `a` and `b` are both negative
- `a` is positive, `b` is negative
- `a` is negative, `b` is positive

There are also some special cases for multiplication that we should check: 0, 1 and -1

- `a` or `b` is 0, 1 or -1

Partition the Input Space (2)

Finally, as a suspicious tester trying to find bugs,
we might suspect that the implementer of `BigInteger` might try to make it faster by using int or long internally when possible,
and only fall back on an expensive general representation (like a list of digits) when the value is too big.

So we should definitely also try integers that are very big, bigger than the biggest `long`:

- `a` or `b` is small
- the absolute value of `a` or `b` is bigger than `Long.MAX_VALUE`
  the biggest possible primitive integer in Java, which is roughly 2^{63}

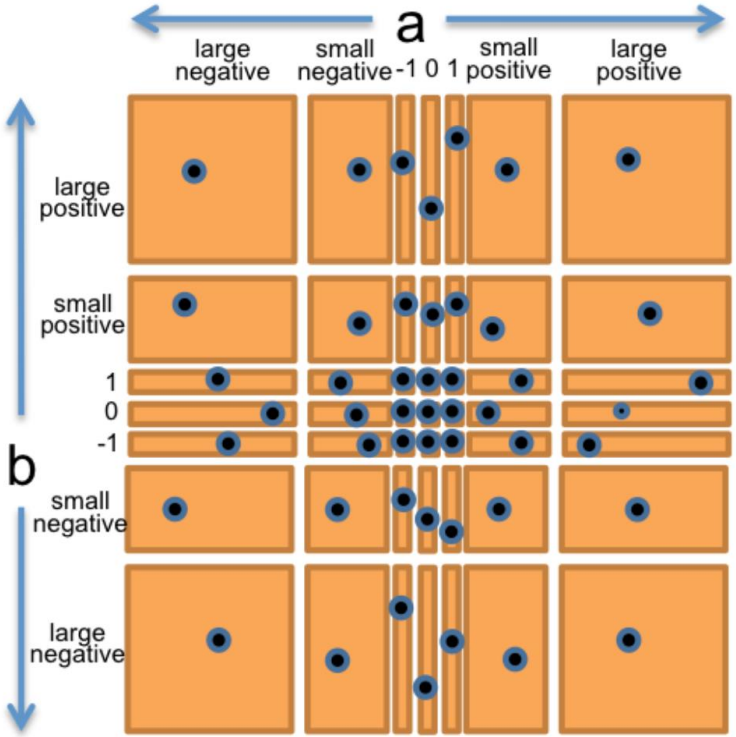
Partition the Input Space (3)

Let's bring all these observations together into a straightforward partition of the whole (a, b) space.

We'll choose a and b independently from:

- 0
- 1
- -1
- small positive integer
- small negative integer
- huge positive integer
- huge negative integer

So this will produce 7 × 7 = 49 partitions that completely cover the space of pairs of integers



Partition the Input Space (4)

To produce the test suite,

we would pick an arbitrary pair (a, b) from each square of the grid, for example:

- (a, b) = (-3, 25) to cover (small negative, small positive)
- (a, b) = (0, 30) to cover (0, small positive)
- (a, b) = (2^{100}, 1) to cover (large positive, 1)
- and so on

Example 2: max()

Let's look at an another example from the Java library:

the integer `max()` function, found in the `Math` class

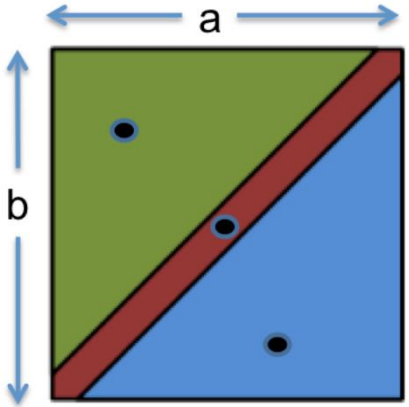
```
/**
 * @param a  an argument
 * @param b  another argument
 * @return the larger of a and b.
 */
public static int max(int a, int b)
```

- Mathematically, this method is a function of the following type:
  - $\texttt{max} : \texttt{int} \times \texttt{int} \to \texttt{int}$

Partition the Input Space

From the specification, it makes sense to partition this function as:

- a < b
- a = b
- a > b



Our test suite might then be:

- (a, b) = (1, 2) to cover a < b
- (a, b) = (9, 9) to cover a = b
- (a, b) = (-5, -6) to cover a > b

In-Class Quiz 1

- You want to partition the input space of this integer square root function:

```
/**
 * @param x is a nonnegative integer
 * @return nearest integer to the square root of x
 */
public static int intSqrt(int x)
```

- Which one is a **good partition**?
  - Partition: `x < 0` and `x >= 0`
  - Partition: `x` is a perfect square and `x` is an integer `> 0` but not a perfect square
  - Partition: `x = 0, x = 1, x = 5, x = 16`
  - Partition: `x` even, `x` odd, `x >= 100`

Include Boundaries in the Partition (1)

Bugs often occur at boundaries between subdomains

Some examples:

- 0 is a boundary between positive numbers and negative numbers
- the maximum and minimum values of numeric types, like `int` and `double`
- emptiness (the empty string, empty list, empty array) for collection types
- the first and last element of a collection

Include Boundaries in the Partition (2)

- Why do bugs often happen at boundaries?
  - One reason is that programmers often make **off-by-one mistakes**
    (like writing `<=` instead of `<`, or initializing a counter to `0` instead of `1`)
  - Another is that some boundaries may need to be handled as special cases in the code
  - Another is that boundaries may be places of discontinuity in the code's behavior
    When an int variable grows beyond its maximum positive value,
    for example, it abruptly becomes a negative number

- It's important to ***include boundaries*** as subdomains in your partition,
  so that you're choosing an input from the boundary

Example 3: max() version 2 (1)

Let's redo `max : int × int → int`.

Partition into:

- relationship between a and b
  - a < b
  - a > b
- value of a
  - a = 0
  - a < 0
  - a > 0
  - a = minimum integer
  - a = maximum integer
- value of b
  - b = 0
  - b < 0
  - b > 0
  - b = minimum integer
  - b = maximum integer

Example 3: max() version 2 (2)

Now let's pick test values that cover all these classes:

- (1, 2) covers a < b, a > 0, b > 0
- (-1, -3) covers a > b, a < 0, b < 0
- (0, 0) covers a = b, a = 0, b = 0
- (`Integer.MIN_VALUE`, `Integer.MAX_VALUE`)
  covers a < b, a = minint, b = maxint
- (`Integer.MAX_VALUE`, `Integer.MIN_VALUE`)
  covers a > b, a = maxint, b = minint

In-Class Quiz 2

- Consider the following function and values:

```
/**
 * @param winsAndLosses is a string of length at most 5 consisting of 'W' or 'L' characters
 * @return the fraction of characters in winsAndLosses that are 'W'
 */
public static double winLossRatio(String winsAndLosses)
```

(i) "" (ii) "LLLLL" (iii) "WLWL" (iv) "WWWWW" (v) "xxxxx"

- Which are appropriate **boundary values** for testing this function ?
  - (i)
  - (ii), (iv), (v)
  - (i), (ii), (iv)
  - (i), (ii), (iv), (v)

Two Extremes for Covering the Partition (1)

After partitioning the input space,
we can choose how exhaustive we want the test suite to be:

1. **Full Cartesian product**
   Every legal combination of the partition dimensions is covered by one test case.
   This is what we did for the *BigInteger multiply* example,
   and it gave us 7 × 7 = 49 test cases.
   For the *max* example that included boundaries, which has three dimensions with 3 parts, 5 parts, and 5 parts respectively, it would mean up to 3 × 5 × 5 = 75 test cases.

   In practice not all of these combinations are possible, however.
   For example, there's no way to cover the combination a < b, a = 0, b = 0,
   because a can't be simultaneously less than zero and equal to zero.

Two Extremes for Covering the Partition (2)

2. **Cover each part**

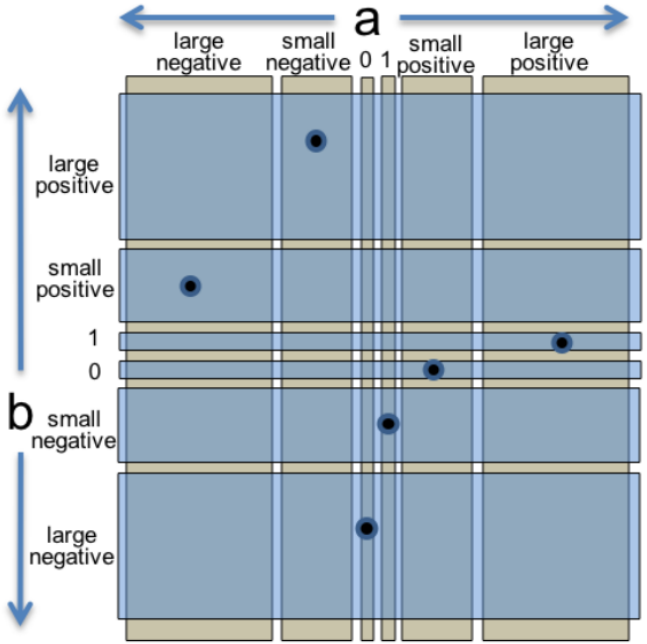
   Every part of each dimension is covered by at least *one* test case,

   but not necessarily every combination.

   With this approach, the test suite for max might be as small as 5 test cases if carefully chosen.

   That's the approach we took in `max()` *version 2*, which allowed us to choose 5 test cases;

   and 6 test cases for the `BigInteger` example, as shown on the right.



Often we strike some compromise *between* these two extremes, based on human judgement and caution.

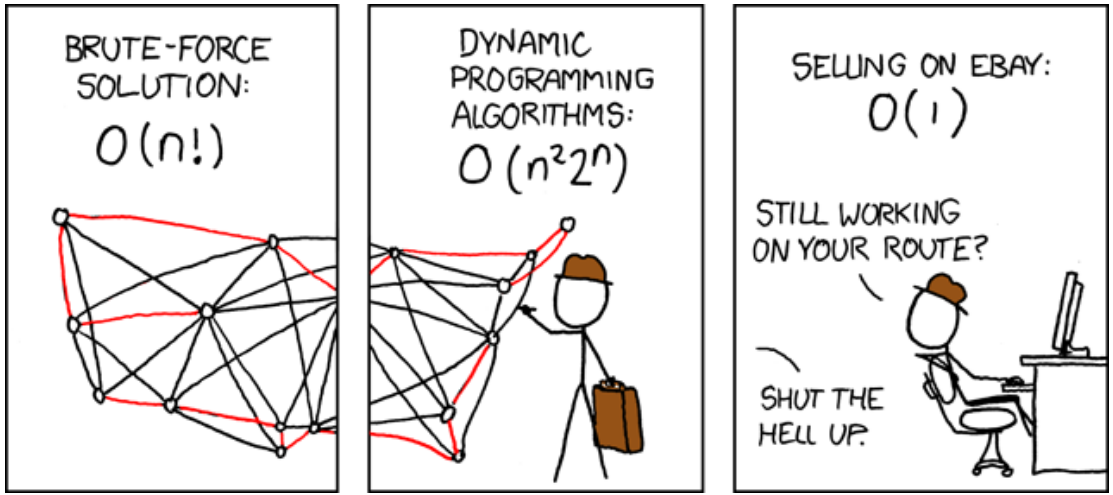
Part 2: Recursion

- In this part of the lecture, we're going to talk about how to implement a method, focussing on one particular technique: ***recursion***
- Recursion is not appropriate for every problem, but it’s an important tool in your software development toolbox (and one that many people scratch their heads over)
- We want you to be comfortable and competent with recursion, because you will encounter it over and over (that’s a joke, but it’s also true)
- Since you've taken **CSE105**, recursion is not completely new to you, and you have seen and written recursive functions like ***factorial*** and ***fibonacci*** before
    - But don't worry if you forget, we will review them again here...

Travelling Salesman Problem

- Asks what is the shortest tour that visits all n cities
- Brute-force solution: try all tours
- How many possible tours are there ? n! possible tours

this number is called "n factorial"



Factorial

- Consider writing a method to compute factorial
- We can define factorial in two different ways
- The first one using **product**: $n! = n \times (n-1) \times \cdots \times 2 \times 1 = \prod_{k=1}^{n} k$
  - which leads to **iterative** implementation:

```
public static int factorial(int n) {
    int fact = 1;
    for (int i = 1; i <= n; i++) {
         fact = fact * i;
    }
    return fact;
}
```

Factorial Quiz 1

What is `factorial(4)` ?

a) `factorial(4) = 4`
b) `factorial(4) = 12`
c) `factorial(4) = 24`

Factorial Quiz 2

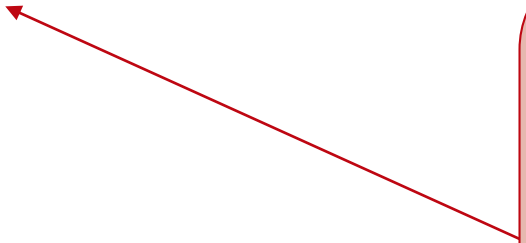
What is `factorial(5)` ?

a) `factorial(5) = 5`
b) `factorial(5) = 24`
c) `factorial(5) = 120`

Factorial Quiz 2

What is `factorial(5)` ?

a) `factorial(5) = 5`
b) `factorial(5) = 24`
c) `factorial(5) = 120`



you can compute again:
5 x 4 x 3 x 2 x 1
but have computed:
4 x 3 x 2 x 1
from the previous quiz !

so, you just need to compute:
5 x `factorial(4)`

this brings us to the second definition on the next slide...

Recursive Factorial

- The second definition using **recurrence relation**:

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ (n-1)! \times n & \text{if } n > 0 \end{cases}$$

  - which leads to recursive implementation:

```
public static int factorial(int n) {
    if (n == 0) {
        return 1;
    }
    else {
        return n * factorial(n-1);
    }
}
```

Recursive Factorial

- The second definition using **recurrence relation**:

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ (n-1)! \times n & \text{if } n > 0 \end{cases}$$

  - which leads to recursive implementation:

```
public static int factorial(int n) {
    if (n == 0) {
        return 1;
    }
    else {
        return n * factorial(n-1);
    }
}
```

But how do we get the answer to this `factorial(n-1)` ?

Recursion God

- A good way to think about this, is that you have an all-powerful entity called **Recursion God**:
  - that if you solve a problem of original size, say size n
  - Recursion God will answer ***correctly*** problems of size ***less than*** n !

```
public static int factorial(int n) {
    if (n == 0) {
        return 1;
    }
    else {
        return n * factorial(n-1);
    }
}
```

Now since this `factorial(n-1)` is a problem of size smaller than the original problem, Recursion God will freely answer it !
For example, for n = 5:

factorial(4) = 24

Recursion God

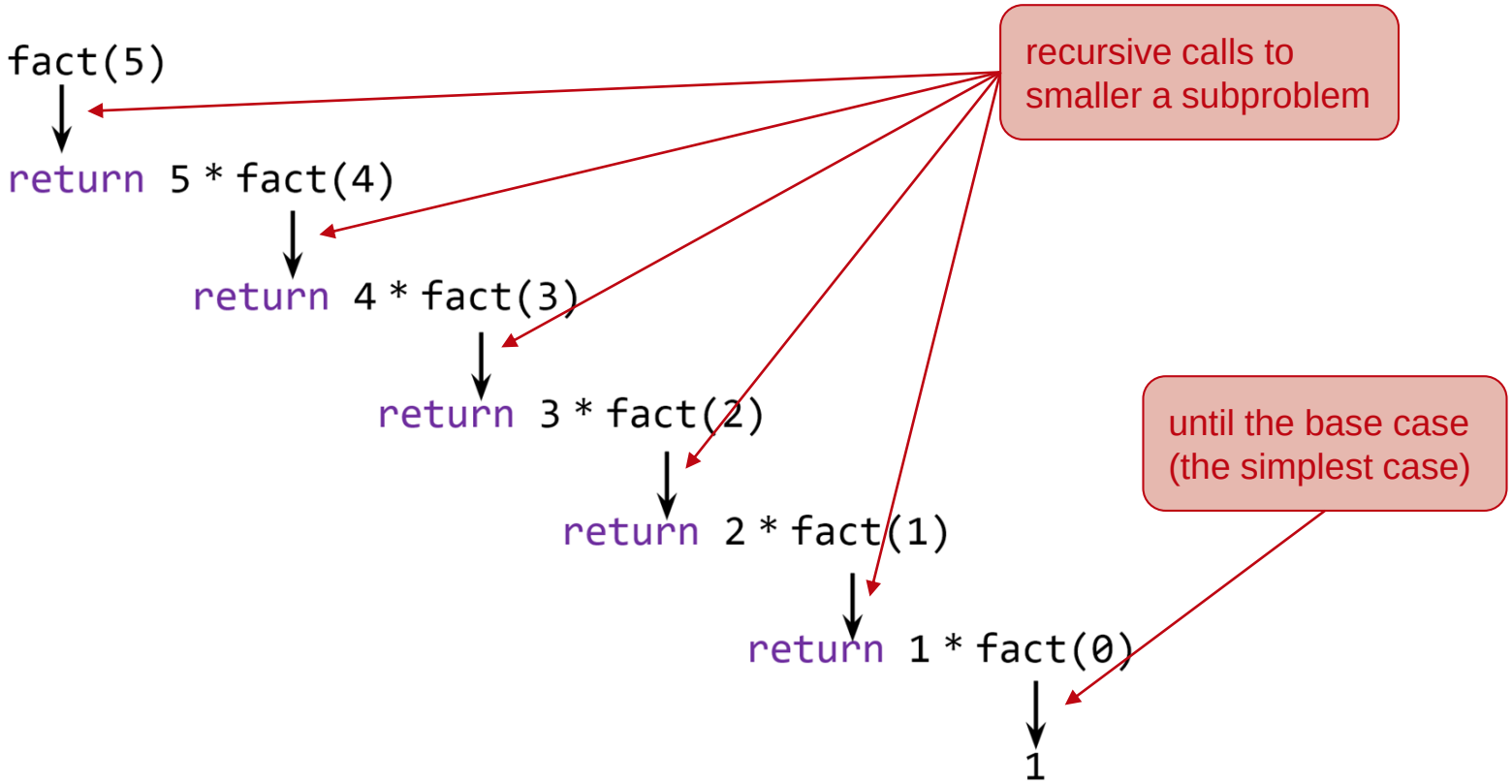
- A good way to think about this, is that you have an all-powerful entity called **Recursion God**:
  - that if you solve a problem of original size, say size n
  - Recursion God will answer ***correctly*** problems of size ***less than*** n !

```
public static int factorial(int n) {
    if (n == 0) {
        return 1;
    }
    else {
        return n * factorial(n-1);
    }
}
```

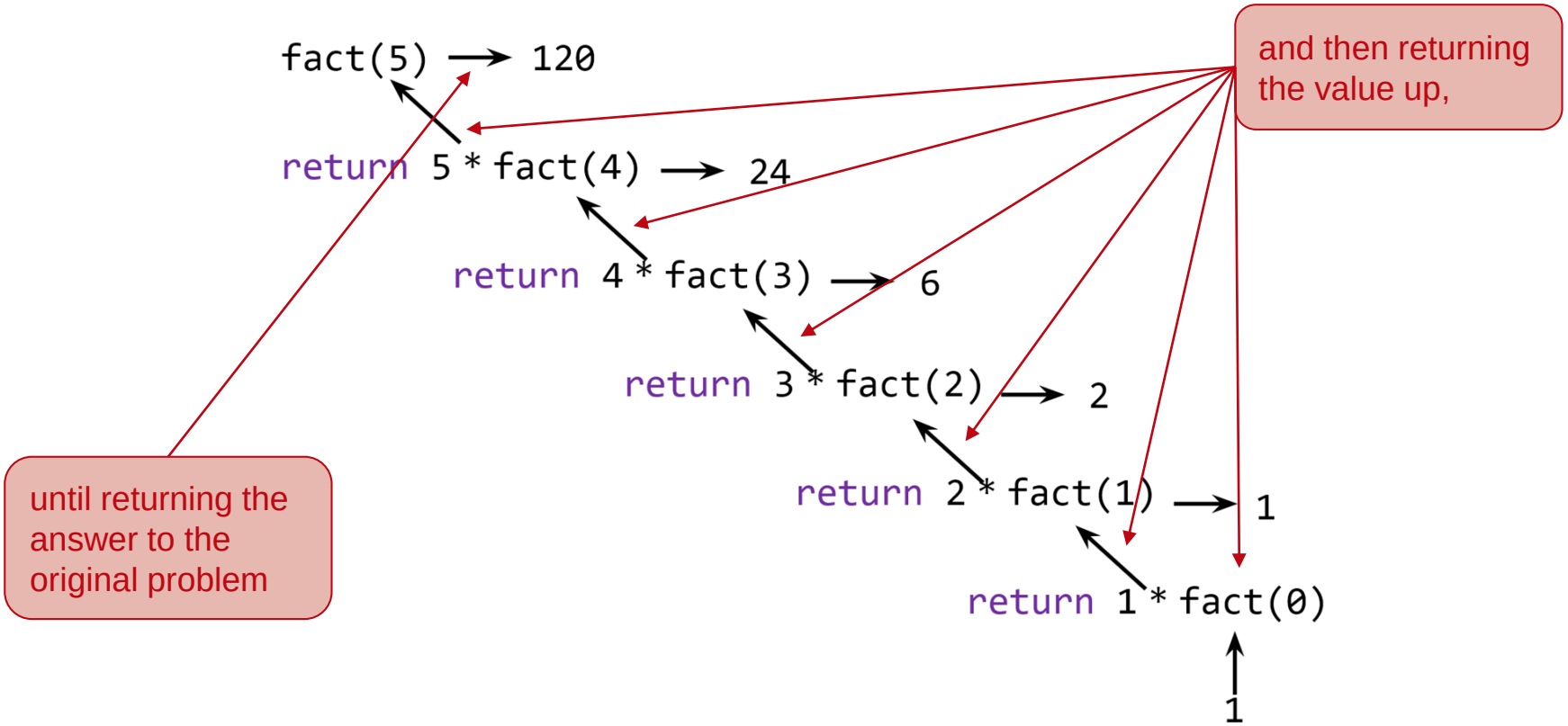
factorial(4) = 24

Then, having the answer from Recursion God, you believe it, and to compute the answer to the original problem, you just multiply his answer with n

What really happened behind the scene...



What really happened behind the scene...



Recipe for Recursion

- First, check that the problem can be reduced into ***smaller*** and ***easier*** subproblem(s)
  - that is, the solution to a problem depends on the solution(s) to smaller instance(s) of the *same* problem
- Next, find the **Base Case** :
  - Find the *simplest instance* that can be directly solved
- Finally, compute the **Recursive Step** :
  - Try a couple of cases to see the *recursive relation* among instances
  - Formulate generally for n, trust Recursion God to solve the smaller instances
  - Make sure to reach the base case

Recipe for Recursion

- First, check that the problem can be reduced into ***smaller*** and ***easier*** subproblem(s)
  - that is, the solution to a problem depends on the instance(s) of the *same* problem
- Next, find the **Base Case** :
  - Find the *simplest instance* that can be directly solved
- Finally, compute the **Recursive Step** :
  - Try a couple of cases to see the *recursive relation* among instances
  - Formulate generally for n, trust Recursion God to solve the smaller instances
  - Make sure to reach the base case

factorial(n) can be computed if we know how to compute factorial(n-1), and n-1 is < n

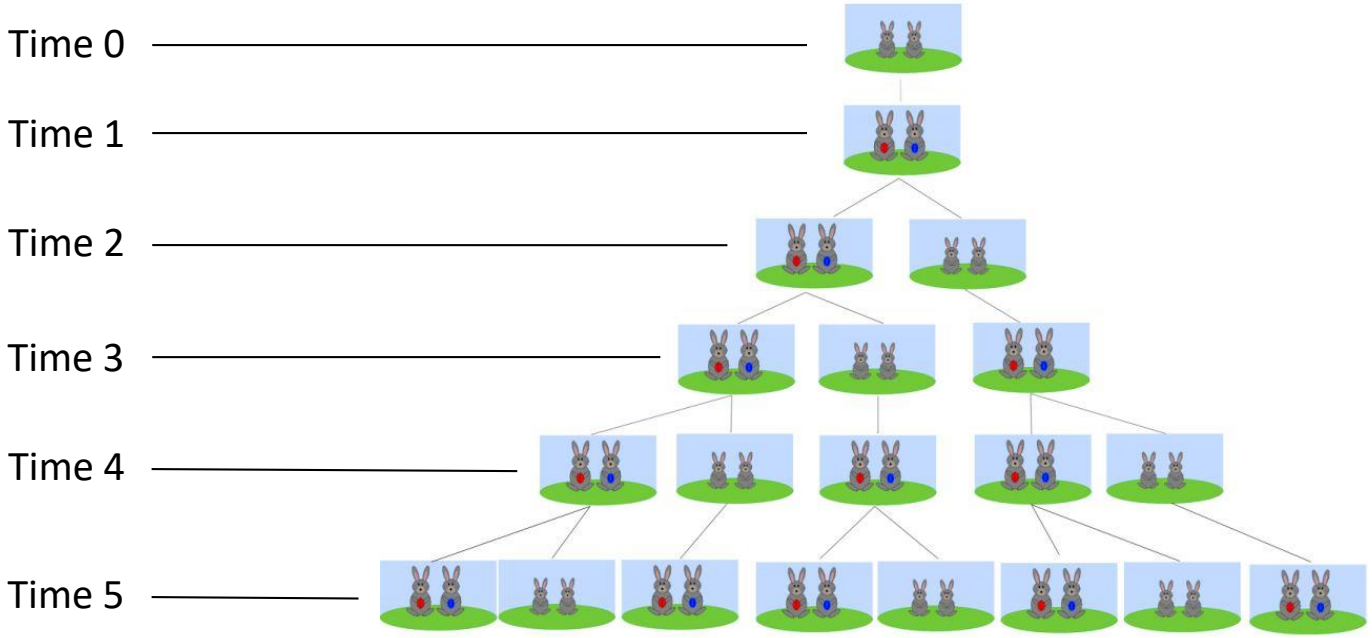
factorial(0) is 1

try factorial of 4, of 5, ...

given the answer of factorial(n-1), I can compute factorial(n) by multiplying it with n

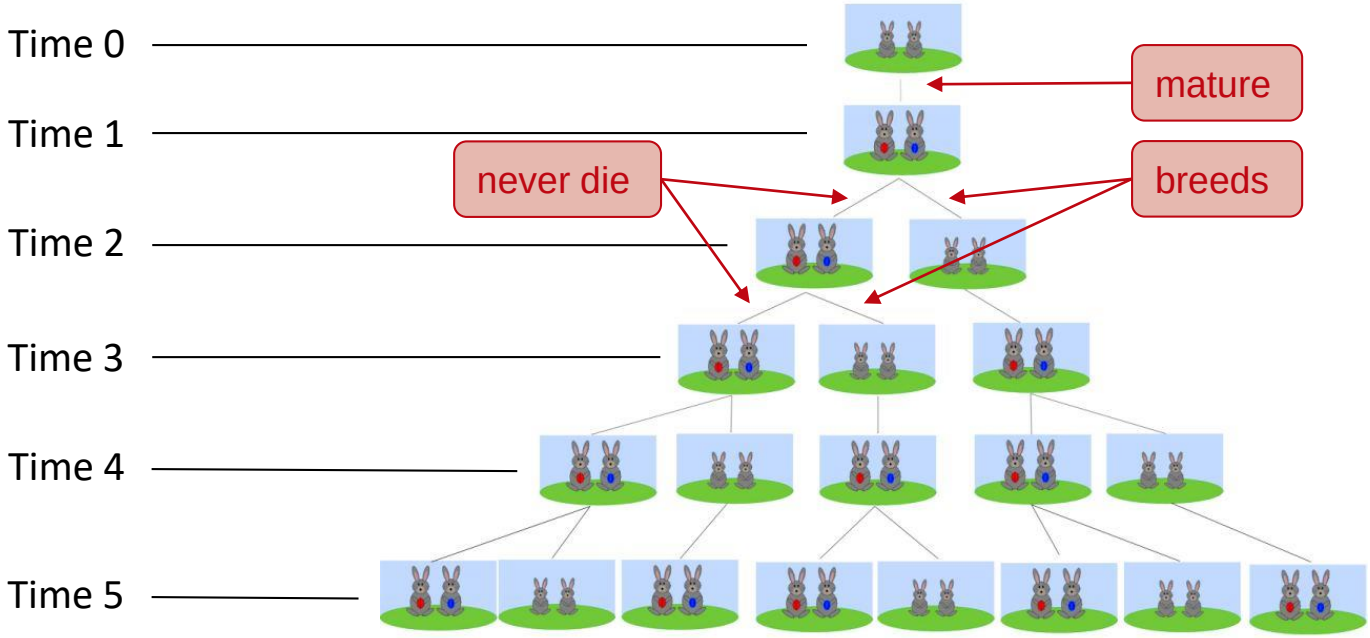
n, n-1, n-2, and so on, will **always** reach base case = 0

Rabbit Breeding Problem



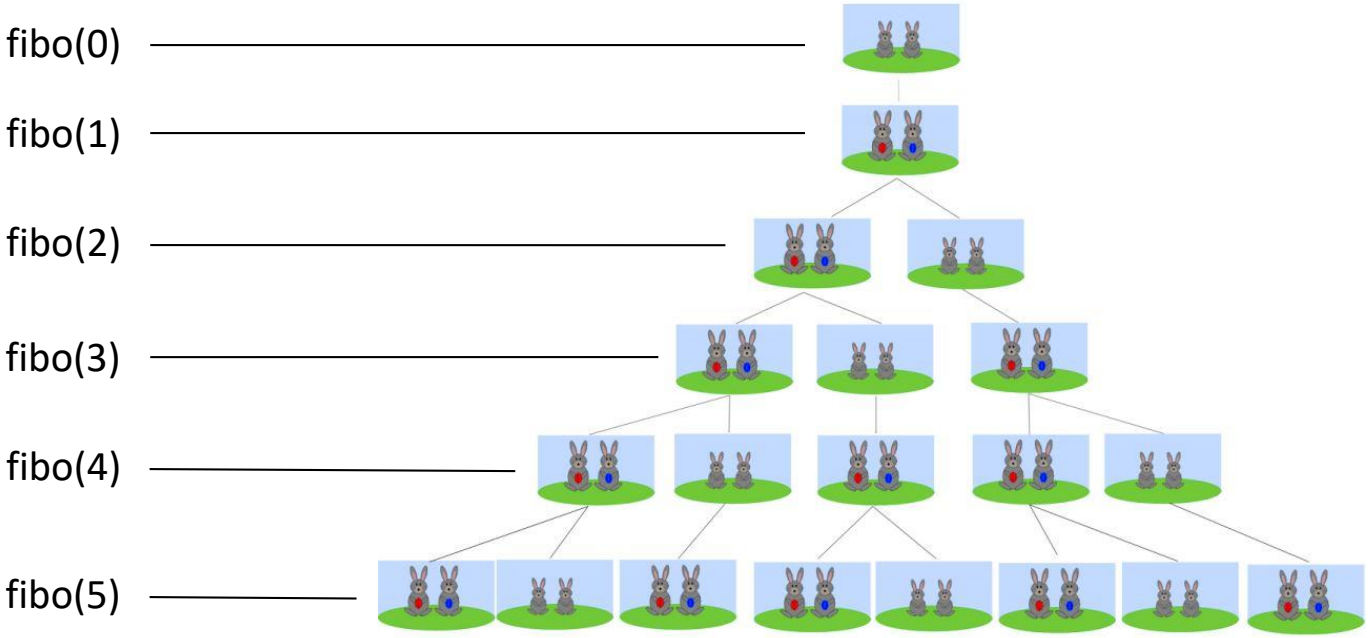
- Assume a pair of rabbits take one time period to mature and breed, always conceive a male female pair, never die
- How many total pairs of rabbits after time n ?

Rabbit Breeding Problem



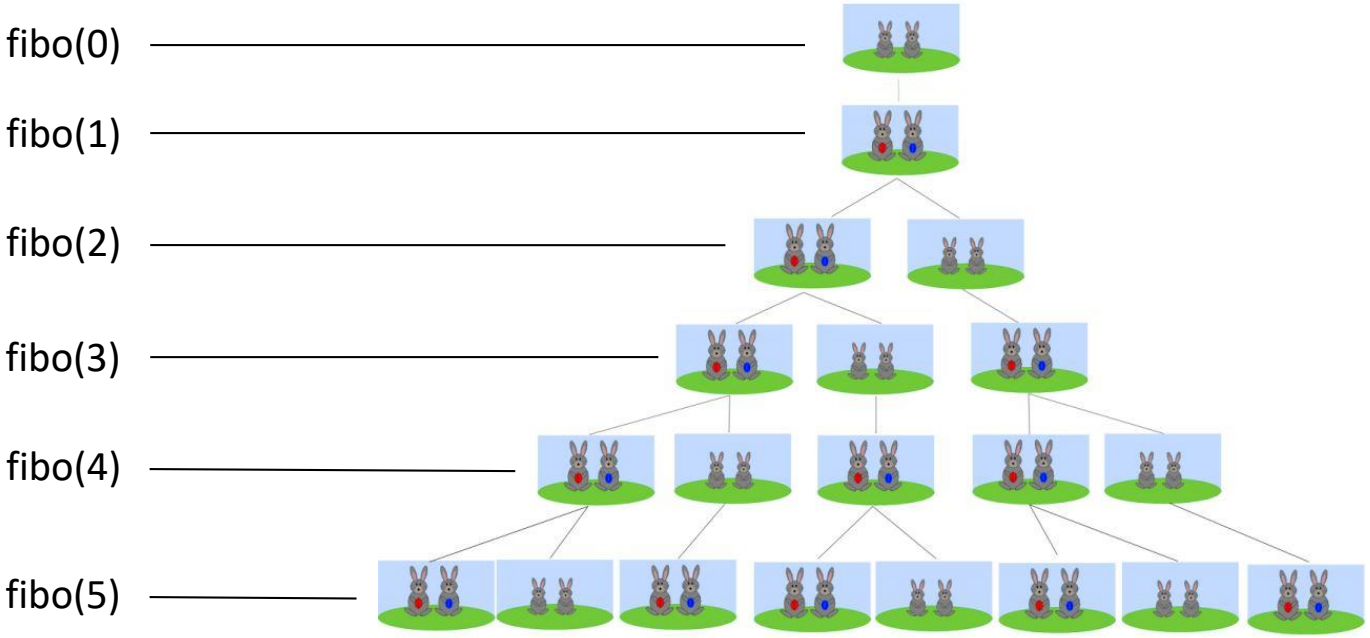
- Assume a pair of rabbits take one time period to mature and breed, always conceive a male female pair, never die
- How many total pairs of rabbits after time n ?

Fibonacci Number



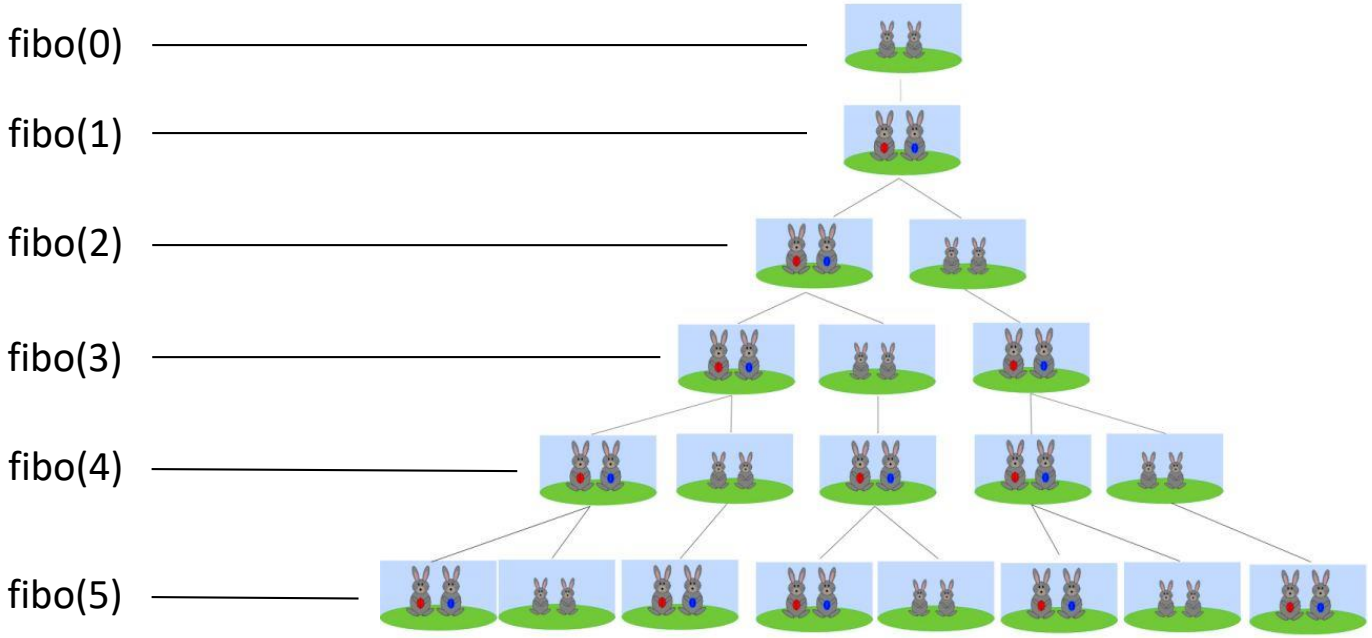
- That total pairs of rabbits at time n is the **fibonacci number** fibo(n)

Fibonacci Number



- That total pairs of rabbits at time n is the **fibonacci number** fibo(n)
  - what is the base case ?
  - what is the recursive step ?

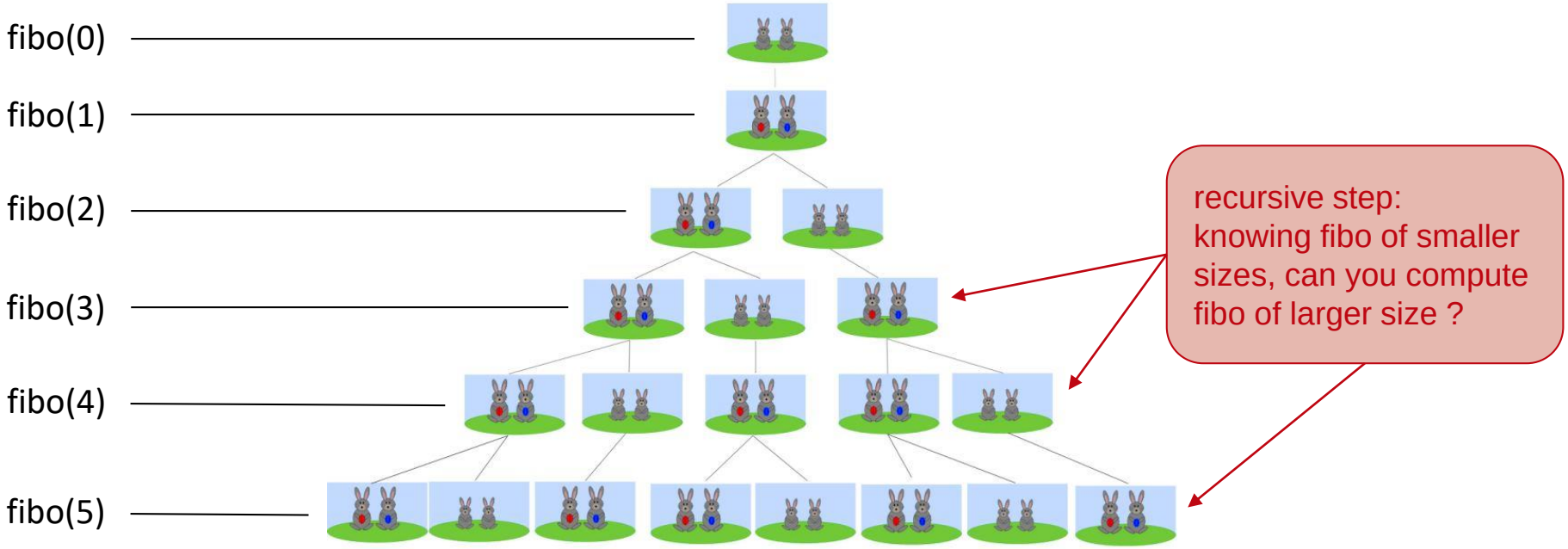
Fibonacci Number



- That total pairs of rabbits at time n is the **fibonacci number** fibo(n)
  - what is the base case ?
  - what is the recursive step ?

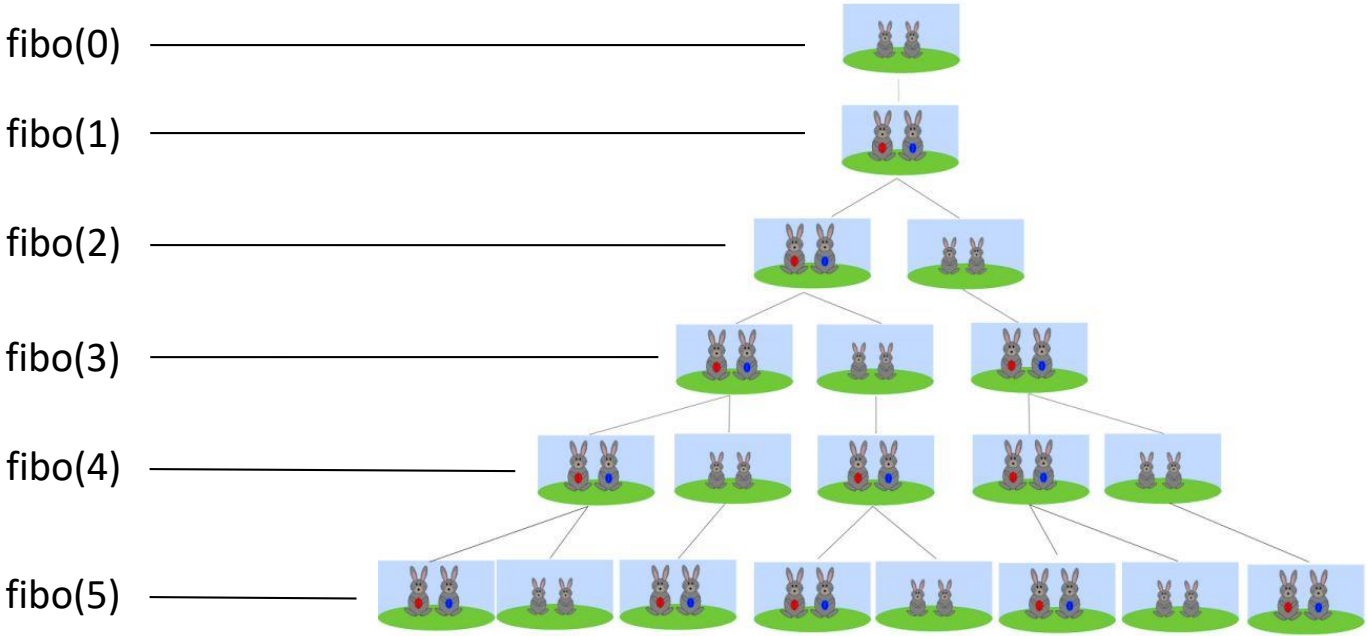
the simplest case(s) where fibo(n) is just some number

Fibonacci Number



- That total pairs of rabbits at time n is the **fibonacci number** fibo(n)
  - what is the base case ?
  - what is the recursive step ?

Fibonacci Number



- That total pairs of rabbits at time n is the **fibonacci number** fibo(n)
  - two bases cases: when n = 0 and n = 1, fibo(n) = 1
  - recursive step: fibo(n) = fibo(n-1) + fibo(n-2)

Recursive Fibonacci Code

```
/**
 * Compute a Fibonacci number
 * @param n >= 0
 * @return the nth Fibonacci number
 */
public static int fibo(int n) {
    // base cases
    if (n == 0 || n == 1) {
        return 1;
    }
    // recursive step
    return fibo(n-1) + fibo(n-2);
}
```

Sum of Digits

- Given a non-negative integer n, write a method that returns the sum of its digits ***recursively***.

  Do **not** use any loops anywhere within your code.

```
/**
 * Compute the sum of digits of a non-negative int
 * @param n an integer >= 0
 * @return the sum of digits
 */
public static int sumDigits(int n) {
    // base case

    // recursive step

}
```

Sum of Digits

- Given a non-negative integer n, write a method that returns the sum of its digits ***recursively***.
  Do **not** use any loops anywhere within your code.

- Test cases:
  - sumDigits(123) → 6
  - sumDigits(99999) → 45
  - sumDigits(0) → 0
  - sumDigits(1) → 1
  - sumDigits(10) → 1
  - sumDigits(101010101) → 5

Sum of Digits

- Given a non-negative integer n, write a method that returns the sum of its digits ***recursively***.
  Do **not** use any loops anywhere within your code.

no `for`, no `while`, will be penalized even if written in comment

- Test cases:
  - sumDigits(123) → 6
  - sumDigits(99999) → 45
  - sumDigits(0) → 0
  - sumDigits(1) → 1
  - sumDigits(10) → 1
  - sumDigits(101010101) → 5

a good set of test cases, includes corner cases

Sum of Digits Recursion Recipe: Base Case

- What is the base case?
  - What is the simplest case that sumDigits can immediately return?

stop! try to think and answer by yourself first!

Sum of Digits Recursion Recipe: Base Case

- What is the base case?
  - What is the simplest case that sumDigits can immediately return?
  - One idea is when n is 0
    - it is easy to check (just n==0)
    - it is the smallest instance, since n is non-negative by the assumption
    - and the return value is just 0

Sum of Digits Recursion Recipe: Recursive Step

- What is the recursive step?
  - Knowing answer to a "smaller instance" of the same problem sumDigits, how do I use it to compute the answer to original problem?
  - And, how do I compute that smaller instance?
  - Will the base case always be reached?

stop! try to think and answer by yourself first!

Sum of Digits Recursion Recipe: Recursive Step

- What is the recursive step?
  - Knowing answer to a "smaller instance" of the same problem sumDigits, how do I use it to compute the answer to original problem?
    - Say n = 123, smaller instance, one digit less, would be 12
    - How to compute 12?
    - Use integer division, 123 / 10 is 12
    - Given sumDigits(12), how do I get sumDigits(123)?
    - By adding 3 to sumDigits(12)
    - How do I get 3 from 123?
    - Use modulo, 123 % 10 is 3
  - Will the base case always be reached?
    - 123 / 10 is 12, 12 / 10 is 1, 1 / 10 is 0 which is the base case

Recursive Sum of Digits Code

- After applying the recipe, we can write (and test, debug) the code

```
/**
 * Compute the sum of digits of a non-negative int
 * @param n an integer >= 0
 * @return the sum of digits
 */
public static int sumDigits(int n) {
    // base case
    if (n == 0) {
        return 0;
    }
    // recursive step
    return (n % 10) + sumDigits(n / 10);
}
```

Number of Substrings

- Given an input string and a non-empty substring subs, write a method to compute **recursively** the number of times that subs appears in the string. The substrings must **not** overlap and **no** loops used.

```
/**
 * Compute the number of substrings in an input string
 * without overlapping
 * @param input the input string
 * @param subs the substring, a non-empty string
 * @return the number of substrings
 */
public static int numSubs(String input, String subs) {

    // base case


    // recursive step

}
```

Number of Substrings

- Given an input string and a non-empty substring subs, write a method to compute **recursively** the number of times that subs appears in the string. The substrings must **not** overlap and **no** loops used.

- Test cases:
  - numSubs("abcxxxxabc", "abc") → 2
  - numSubs("abcdef", "xyz") → 0
  - numSubs("aaaaaaaaaa", "a") → 10
  - numSubs("aaaaaaaaaa", "aa") → 5
  - numSubs("", "a") → 0
  - numSubs("a", "a") → 1

Number of Substrings Recursion Recipe: Base Case

- What is the base case?
  - What is the simplest case that numSubs can immediately return?

stop! try to think and answer by yourself first!

Number of Substrings Recursion Recipe: Base Case

- What is the base case?
  - What is the simplest case that numSubs can immediately return?
  - One idea is when input string is strictly shorter than subs
    - it is easy to check, just by `input.length() < subs.length()`
    - it includes the smallest instance, since the input string can be empty, while the substring subs is not
    - and the return value is just 0, since there is no way subs appear in input

Number of Substrings Recursion Recipe: Recursive Step

- What is the recursive step?
  - Knowing answer to a "smaller instance" of the same problem numSubs, how do I use it to compute the answer to original problem?
  - And, how do I compute that smaller instance?
  - Will the base case always be reached?

stop! try to think and answer by yourself first!

Number of Substrings Recursion Recipe: Recursive Step

- What is the recursive step?
  - Knowing answer to a "smaller instance" of the same problem numSubs, how do I use it to compute the answer to original problem?
    - Say we check whether subs is the prefix of input
    - There are 2 cases:
      - subs is the prefix of input, which means we find 1 occurrence, so we return 1 + numSubs of substring of line that starts *after* subs, because the next occurrence cannot overlap
      - subs is not the prefix of input, which means we don't find it this time, so we return 0 + numSubs of substring of input one char to the right
  - Will the base case always be reached?
    - in both cases, we *always* reduce input length of numSubs, so at some point it will be shorter than subs and reaches the base case

Recursive Number of Substrings Code

- After applying the recipe, we can write (and test, debug) the code

```
/**
 * Compute the number of substrings in an input string
 * without overlapping
 * @param input the input string
 * @param subs the substring, a non-empty string
 * @return the number of substrings
 */
public static int numSubs(String input, String subs) {
    // base case
    if (input.length() < subs.length()) {
        return 0;
    }
    // recursive step
    if (input.substring(0, subs.length()).equals(subs)) {
        return 1 + numSubs(input.substring(subs.length()), subs);
    }
    return numSubs(input.substring(1), subs);
}
```

Recursion + Helper Method

- We have learned about Recursion technique
  - Recursion Recipe: Base Case and Recursive Step

- Next, we are going to learn about additional technique called **helper method** that will make our recursive step more *simple* and *elegant*

Subsequences Problem

- We will use the Subsequences Problem as our running example
- Given a word consisting only of letters A-Z or a-z
  - Return all subsequences of word, separated by commas, where a subsequence is a string of letters found in word in the same order that they appear in word
  - For example:
    `subsequences("abc")` → `"abc,ab,bc,ac,a,b,c,"`
    - Note the trailing comma preceding the empty string "", which is also a valid subsequence

Subsequence Problem : Direct Recursive Solution

- We can solve it directly using recursion as follows:

```
public static String subseq(String word) {
    // base case
    if (word.equals(""))
         return "";
    // recursive step
    char firstLetter = word.charAt(0);
    String restOfWord = word.substring(1);
    String subsequencesOfRest = subseq(restOfWord);
    String result = "";
    for (String subsequence : subsequencesOfRest.split(",", -1)) {
        result += "," + subsequence;
        result += "," + firstLetter + subsequence;
    }
    result = result.substring(1); // remove extra leading comma
    return result;
}
```

However, as you can see, it looks complicated and unclear at the first glance

Is there a better way?

Helper Method

- For the Subsequences Problem, instead of solving it with direct recursive implementation:

```
public static String subseq(String word)
```

one parameter

two parameters

it is easier to create **additional Helper Method**:

```
private static String subseqHelper(String partialSubseq, String word)
```

- Helper method ***introduces another parameter(s)*** to give you more flexibility in doing (more creative) recursion calls

Subsequence Problem with Helper Method

- Idea: we build up a ***partial subsequence*** using the initial letters of the word
  then use recursive calls to complete that partial subsequence using the remaining letters of the word
- For example, word = "abc"
  - First, select "a" to be in the partial subsequence,
    and recursively extend it with all subsequences of "bc"
  - Then, use "" as the partial subsequence,
    and again recursively extend it with all subsequences of "bc"
- In other words, recursively build up subsequences that,
  either use the first letter 'a' or not

Subsequence Problem with Helper Method Code (1)

```
private static String subseqHelper(String partialSubseq, String word) {
    // base case
    if (word.equals(""))
         return partialSubseq;
    // recursive step
    return subseqHelper(partialSubseq + word.charAt(0), word.substring(1))
         + ","
         + subseqHelper(partialSubseq, word.substring(1));
}
```

- *Base case* when all letters in word has been used, partialSubseq is the subsequences
- *Recursive steps* call the ones that either use the first letter of word, or not
- Very simple and elegant!

Subsequence Problem with Helper Method Code (2)

- Don't forget to implement the original problem:

```
public static String subseq(String word) {
    // call the recursive helper method
    return subseqHelper("", word);
}
```

partial subseq starts with empty string

- Hide the helper method from client; you can overload it by naming it subseq as well

```
private static String subseqHelper(String partialSubseq, String word) {
    // base case
    ...
}
```

Reverse String

- Given a string, return the reverse of that string ***recursively***

- Test cases:
  - reverseStr("abcde") → "edcba"
  - reverseStr("") → ""
  - reverseStr("a") → "a"
  - reverseStr("ab") → "ba"

Reverse String

- Given a string, return the reverse of that string ***recursively***
- Let's solve this by using a recursive helper method that also has a second parameter to store the current reversal result (partial result)

```
/**
 * Reverse the input string.
 * @param str is the input string.
 * @return the reverse of str.
 */
public static String reverseStr(String str) {

    // call your recursive helper method
    return reverseStrHelper(str, "");

}
```

in the beginning, the partial result is nothing = empty string

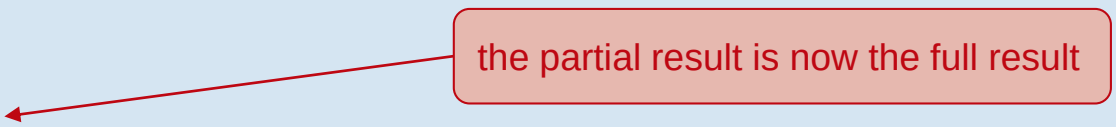
Reverse String Recursive Helper Method: Base Case

- We are going to remove and reverse the letter of str one by one, so the base case is when there is finally no letter left
  - in that case, we are done, and we just return the reversal result

```
private static String reverseStrHelper(String str, String reverse) {
    // base case
    if (str.equals(""))
        return reverse;

    // recursive step
    return reverseStrHelper(str.substring(1), str.charAt(0) + reverse);

}
```



Reverse String Recursive Helper Method: Recursive Step

- We call recursively the reverseStrHelper on str except the first letter, and append first letter on the partial result
  - e.g. ("abc", "") → ("bc", "a") → ("c", "ba") → ("", "cba") → "cba"

```
private static String reverseStrHelper(String str, String reverse) {

    // base case
    if (str.equals(""))
        return reverse;

    // recursive step
    return reverseStrHelper(str.substring(1), str.charAt(0) + reverse);

}
```

the partial result is now the full result

Subset Sum

- Given a list of integers, we want to know whether it is possible to choose a subset of some of the integers, such that the integers in the subset adds up to the given sum ***recursively***

- Test cases:
  - subsetSum([2, 5, 8], 10) → true
  - subsetSum([2, 5, 8], 15) → true
  - subsetSum([2, 5, 8], 12) → false
  - subsetSum([], 0) → true
  - subsetSum([], 1) → false
  - subsetSum([1], 1) → true

Subset Sum

- Given a list of integers, we want to know whether it is possible to choose a subset of some of the integers, such that the integers in the subset adds up to the given sum ***recursively***

```
/**
 * Decide whether there is a subset in the input list
 * that adds up to the target sum.
 * For example, subsetSum([2, 5, 8], 10) → true.
 * @param list is the input list.
 * @param sum is the target sum.
 * @return true iff there is a subset of list that adds to sum.
 */
public static boolean subsetSum(List<Integer> list, int sum) {

    // call your recursive helper method

}
```

Subset Sum with Helper Method

- Rather than looking at the whole list, we consider the part of the list starting at index `start` and continuing to the end of the list
  - subsetSum can call subsetSumHelper by passing the target sum and the whole list starting at 0

```
private static boolean subsetSumHelper(List<Integer> list, int start, int sum) {

    // base case


    // recursive step


}
```

Subset Sum Recursion Recipe: Base Case

- What is the base case?
  - What is the simplest case that subsetSum can immediately return?

stop! try to think and answer by yourself first!

Subset Sum Recursion Recipe: Base Case

- What is the base case?
  - What is the simplest case that subsetSum can immediately return?
  - Since we are incrementing start, the base case is when start exceeds the last index of the list, which means the only subset is the empty set
  - It is easy to check, by comparing start with the size of the list
  - There are two cases:
    - when sum is zero
      the empty set sums up to zero, so return true
    - when sum is not zero
      no way we can get that sum, so return false

Subset Sum Recursion Recipe: Recursive Step

- What is the recursive step?
  - How do I construct the "smaller instance"?
  - Knowing answer to a "smaller instance" of the same problem subsetSum , how do I use it to compute the answer to original problem?
  - Will the base case always be reached?

stop! try to think and answer by yourself first!

Subset Sum Recursion Recipe: Recursive Step (1)

- What is the recursive step?
  - How do I construct the "smaller instance"?
  - There are two cases
    - We include element at start in our subset
      smaller instance = list starting at start+1,
      and target sum becomes sum - element at start
    - We don't include element at start in our subset
      smaller instance = list starting at start+1,
      and target sum remains sum

Subset Sum Recursion Recipe: Recursive Step (2)

- What is the recursive step?
  - Knowing answer to a "smaller instance" of the same problem subsetSum, how do I use it to compute the answer to original problem?
    - If either of those two cases return true,
      it means at least one of the subsets adds up to sum,
      so we return true
    - If neither, no subsets add up to sum, so we return false

Subset Sum Recursion Recipe: Recursive Step (3)

- What is the recursive step?
  - Will the base case always be reached?
    - Initial value of start is always 0
    - In both cases, we always increment start,
      at some point it will be the same as the size of the list

Recursive Subset Sum Code

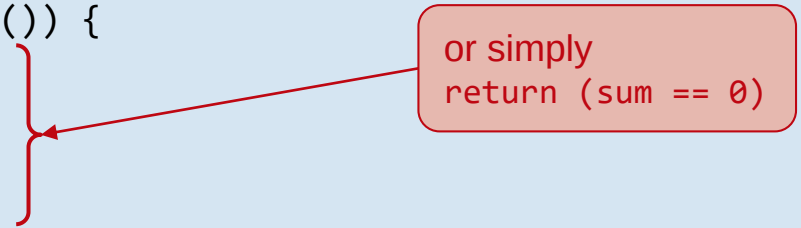
- After applying the recipe, we can write (and test, debug) the code

```
private static boolean subsetSumHelper(List<Integer> list, int start, int sum) {
    // base case
    if (start == list.size()) {
        if (sum == 0) {
            return true;
        }
        return false;
    }
    // recursive step
    if (subsetSumHelper(list, start+1, sum-list.get(start)))
        return true;
    if (subsetSumHelper(list, start+1, sum))
        return true;
    return false;
}
```

Recursive Subset Sum Code

- After applying the recipe, we can write (and test, debug) the code

```
private static boolean subsetSumHelper(List<Integer> list, int start, int sum) {
    // base case
    if (start == list.size()) {
        if (sum == 0) {
            return true;
        }
        return false;
    }
    // recursive step
    if (subsetSumHelper(list, start+1, sum-list.get(start)))
        return true;
    if (subsetSumHelper(list, start+1, sum))
        return true;
    return false;
}
```



Recursive Subset Sum Code

- After applying the recipe, we can write (and test, debug) the code

```
private static boolean subsetSumHelper(List<Integer> list, int start, int sum) {
    // base case
    if (start == list.size()) {
        if (sum == 0) {
            return true;
    // recursive step
    if (subsetSumHelper(list, start+1, sum-list.get(start)))
        return true;
    if (subsetSumHelper(list, start+1, sum))
        return true;
    return false;
}
```

or simply

```
return (subsetSum(list, start+1, sum-list.get(start)) || subsetSum(list, start+1, sum))
```

Thank you for your attention !

- In this lecture, you have learned:
  - Useful Coding Rules
  - Testing 3
    - Partitioning Input Space, Boundaries
    - Covering Partitions, Choosing Test Suite
  - Recursion
    - Factorial, Fibonacci, Sum of Digits, Number of Substrings
    - Helper Method: Subsequence Problem, Reverse String, Subset Sum

- Please continue to Lecture Quiz 3 and Lab 3:
  - to practice recursion, helper method, and testing with JUnit,
  - to do Lab Exercise 3.1, 3.2, and
  - to do Exercise 3.1 - 3.5