Lecture 22 - Nov 28

Inheritance, Recursion

Type-Checking Rules
Solving Problems Recursively: Fac vs. Fib
Recursions on Strings: Palindrome

Announcements

- **Lab5** to be released on Wednesday

Static Types and Anticipated Expectations

```
class A {
  void m1() { ... }
}
class B extends A { }


class C extends A {}
```

At the moment, no new methods have been introduced ⇒ B & C & A have identical exp.

In the future, new methods are anticipated.

```
① B obj1 = new A();

  A = obj2 = new A();
② B obj3 = (B) obj2;
```

① not compile ∵ DT A not a descendant of the ST of obj1 (B)

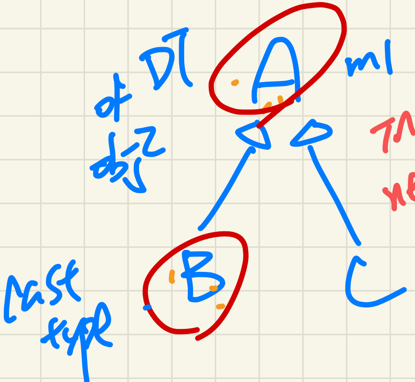
② A cannot fulfil the ① exp. of B

CCE ∵ DT A can't fulfill exp. of cast type

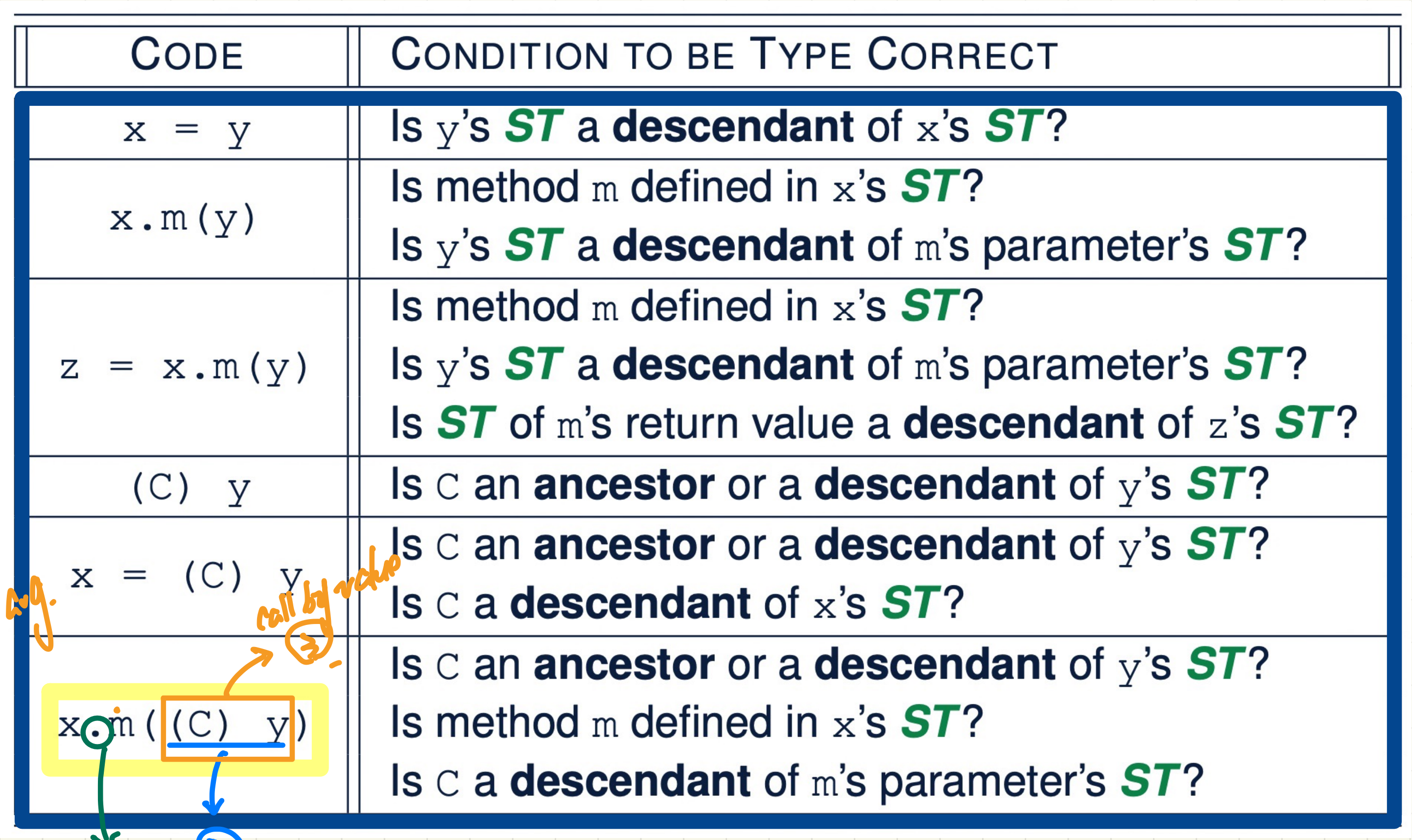
ST: A

↳ ∴ downward cast

2. CCE ?



Summary: Type Checking Rules



CODE	CONDITION TO BE TYPE CORRECT
`x = y`	Is `y`'s ***ST*** a **descendant** of `x`'s ***ST***?
`x.m(y)`	Is method `m` defined in `x`'s ***ST***? Is `y`'s ***ST*** a **descendant** of `m`'s parameter's ***ST***?
`z = x.m(y)`	Is method `m` defined in `x`'s ***ST***? Is `y`'s ***ST*** a **descendant** of `m`'s parameter's ***ST***? Is ***ST*** of `m`'s return value a **descendant** of `z`'s ***ST***?
`(C) y`	Is `C` an **ancestor** or a **descendant** of `y`'s ***ST***?
`x = (C) y`	Is `C` an **ancestor** or a **descendant** of `y`'s ***ST***? Is `C` a **descendant** of `x`'s ***ST***?
`x.m((C) y)`	Is `C` an **ancestor** or a **descendant** of `y`'s ***ST***? Is method `m` defined in `x`'s ***ST***? Is `C` a **descendant** of `m`'s parameter's ***ST***?

param = arg.

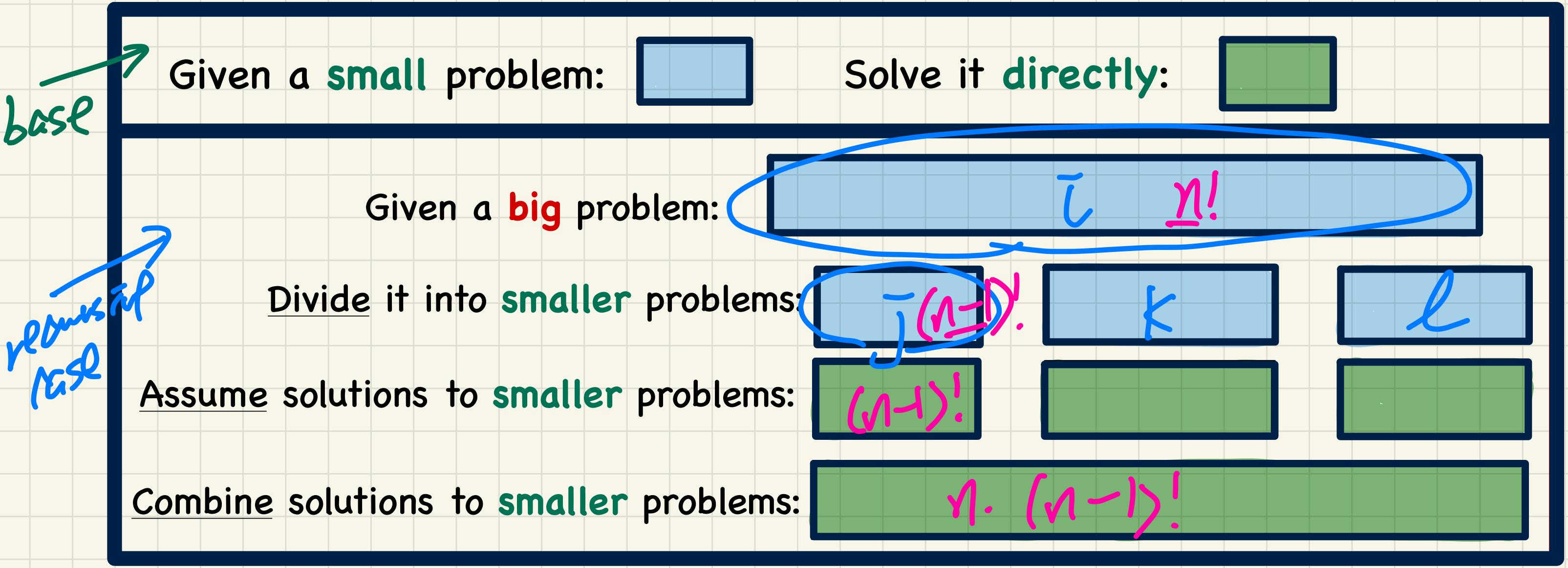
call by value

③

①

②

Solving a Problem **Recursively**



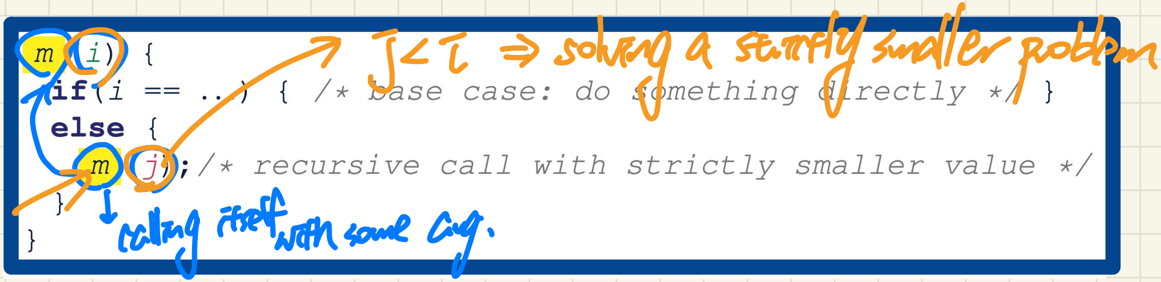
Given a **small** problem: 🟦 Solve it **directly**: 🟩

Given a **big** problem: 🟦

Divide it into **smaller** problems: 🟦 🟦 🟦

Assume solutions to **smaller** problems: 🟩 🟩 🟩

Combine solutions to **smaller** problems: 🟩



```
m(i) {
  if(i == ...) { /* base case: do something directly */ }
  else {
    m(j); /* recursive call with strictly smaller value */
  }
}
```

j < i ⇒ solving a strictly smaller problem
calling itself with some arg.

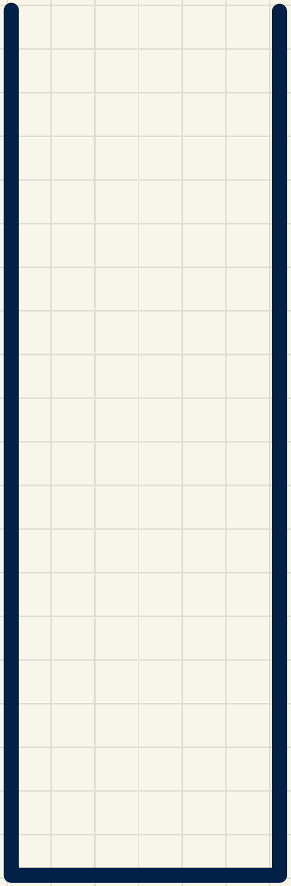
Tracing Recursion via a Stack

- When a method is called, it is ***activated*** (and becomes *active*) and *pushed* onto the stack.
- When the body of a method makes a (helper) method call, that (helper) method is ***activated*** (and becomes *active*) and *pushed* onto the stack.

  ⇒ The stack contains activation records of all *active* methods.

  - *Top* of stack denotes the current point of execution.
  - Remaining parts of stack are (temporarily) ***suspended***.
- When entire body of a method is executed, stack is *popped*.

  ⇒ The current point of execution is returned to the new *top* of stack (which was ***suspended*** and just became ***active***).
- Execution terminates when the stack becomes *empty*.



Runtime Stack

Recursive Solution: factorial

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot \overbrace{(n-1)\cdot(n-2)\cdot\cdots\cdot 3\cdot 2\cdot 1}^{(n-1)!} & \text{if } n \geq 1 \end{cases}$$

→ base case

problem

is this recursive?

↳ No!

∵ the problem is **not** reduced into smaller problem(s) in the def (!)

Recursive Solution

① Base Cases : $0! = 1$

② Recursive Cases : $n! = \boxed{(n-1)!} \cdot n$

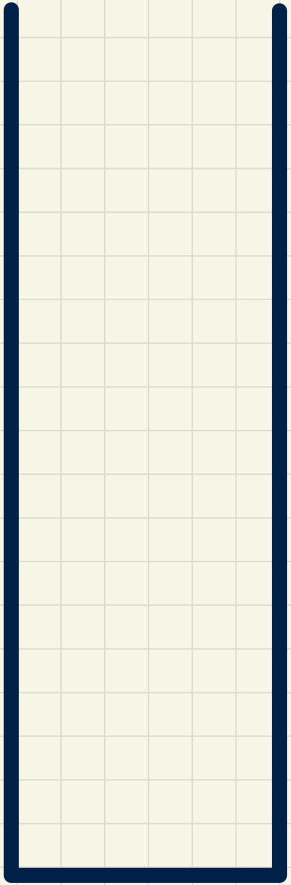
→ solution to a strictly smaller problem

Recursive Solution in Java: factorial

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

```java
int factorial (int n) {
  int result;
  if(n == 0) { /* base case */ result = 1; }
  else { /* recursive case */
    result = n * factorial (n - 1);
  }
  return result;
}
```

Example: factorial(3)



Runtime Stack

Recursive Solution in Java: factorial

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

3 2 1 0

```
int  factorial (int n) {
  int result;
  if(n == 0) { /* base case */ result = 1; }
  else { /* recursive case */
    result = n * factorial (n - 1);
  }
  return result;
}
```

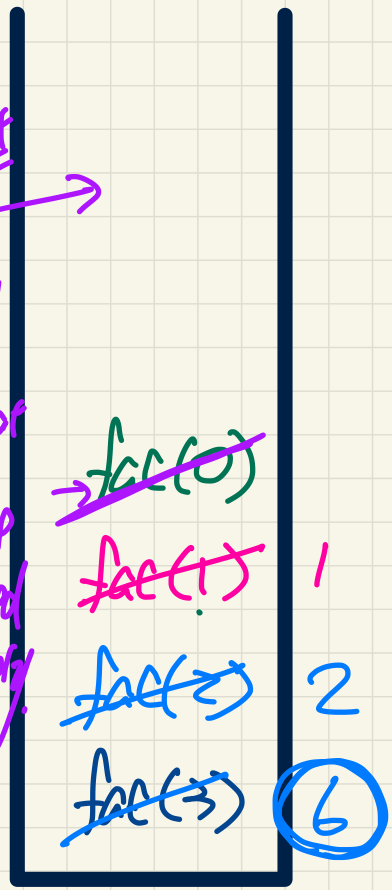
3 * fac(2) 2 · ~~6~~ 6

2 * fac(1) 1 2

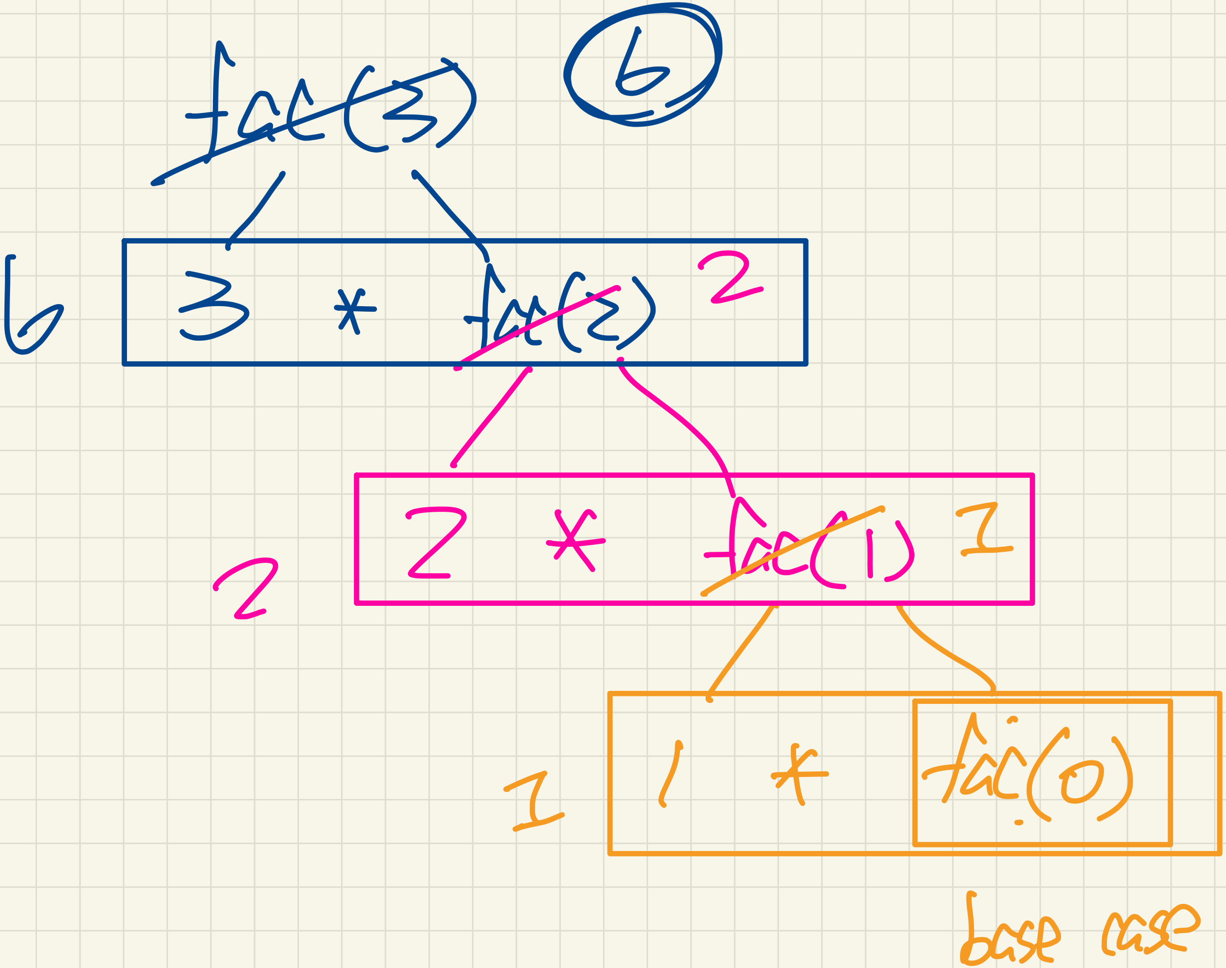
1 * ~~fac(0)~~ 1 = fac(1)

In order for the call stack not to grow indefinitely, we need to make sure the base case is reached ultimately.

Example: factorial(3)

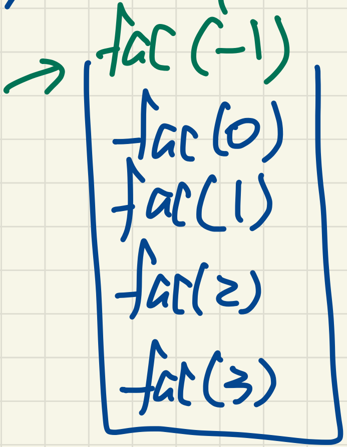


Runtime Stack


~~fac(3)~~
6
6
3 * ~~fac(2)~~
2
2
2 * ~~fac(1)~~
1
1
1 * fac(0)
base case

Common Errors of Recursion (1)

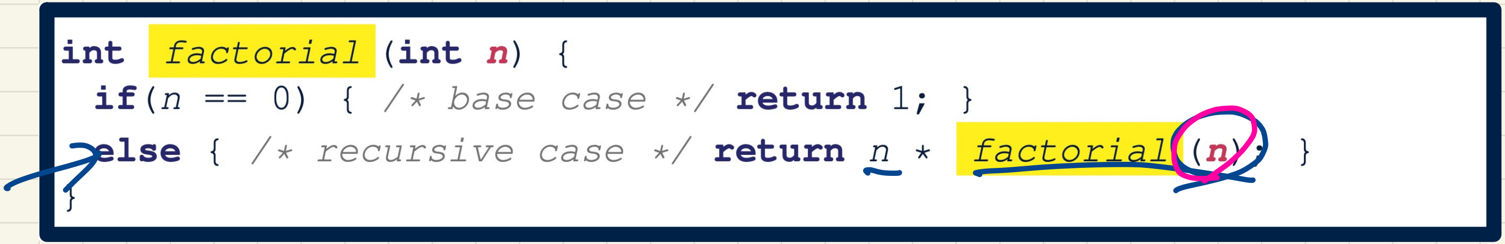
```
int factorial (int n) {
return n * factorial (n - 1);
}
```



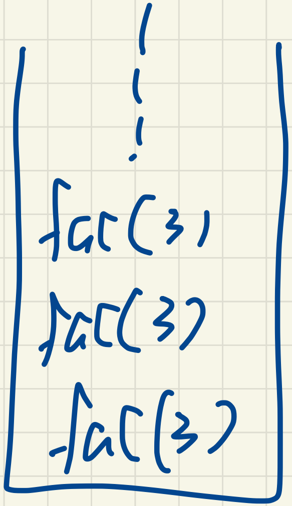
StackOverflowException

→ always put at least one base case

Common Errors of Recursion (2)



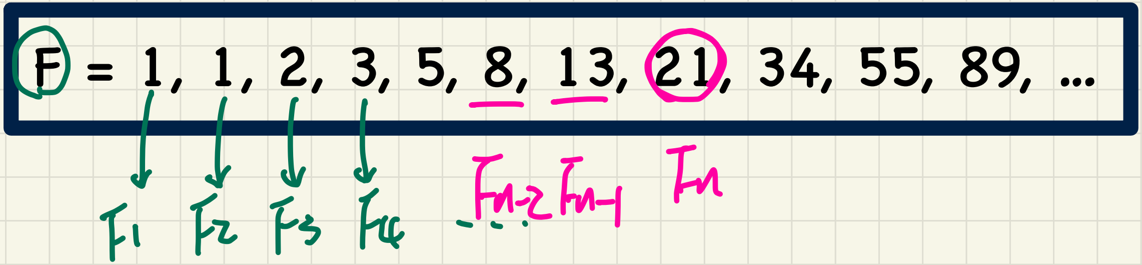
```
int  factorial (int n) {
  if(n == 0) { /* base case */ return 1; }
  else { /* recursive case */ return n * factorial(n); }
}
```



StackOverflow Excp.

→ when making a recursive call, make sure to call the method on a strictly smaller input.

Recursive Solution: Fibonacci Numbers



F = 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

F_n $\geq$

$1..2$

$F_1 = 1$

$F_2 = 1$

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

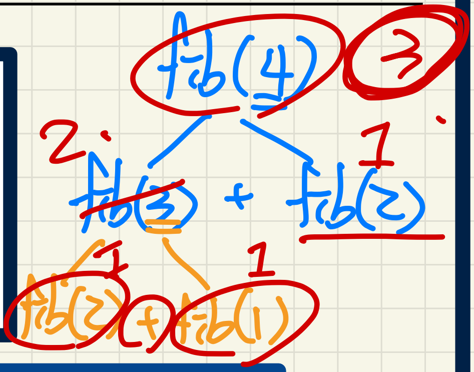
Recursive Solution in Java: Fibonacci Numbers

$$F_n = \begin{cases} 1 & \text{if } n = 1 \\ 1 & \text{if } n = 2 \\ F_{n-1} + F_{n-2} & \text{if } n > 2 \end{cases}$$

```
int fib (int n) {
  int result;
  if(n == 1) { /* base case */ result = 1; }
  else if(n == 2) { /* base case */ result = 1; }
  else { /* recursive case */
    result = fib (n - 1) + fib (n - 2);
  }
  return result;
}
```

solution to a smaller problem

solution to another strictly smaller problem



Example: fib(4)

Runtime Stack

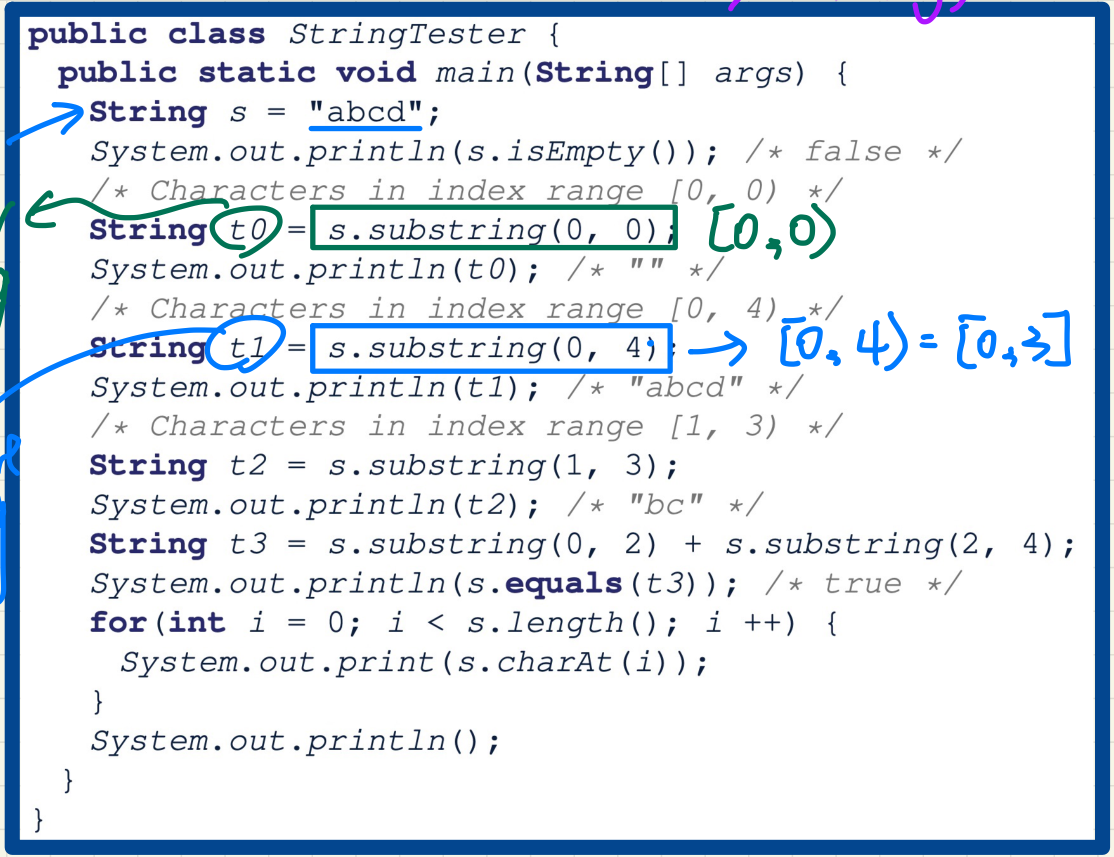
Use of String

substring (i, j) [2, 5) = 2, 3, 4

↳ [i, j)

0 1 2 3

s → "a b c d"



```
public class StringTester {
  public static void main(String[] args) {
    String s = "abcd";
    System.out.println(s.isEmpty()); /* false */
    /* Characters in index range [0, 0) */
    String t0 = s.substring(0, 0);
    System.out.println(t0); /* "" */
    /* Characters in index range [0, 4) */
    String t1 = s.substring(0, 4);
    System.out.println(t1); /* "abcd" */
    /* Characters in index range [1, 3) */
    String t2 = s.substring(1, 3);
    System.out.println(t2); /* "bc" */
    String t3 = s.substring(0, 2) + s.substring(2, 4);
    System.out.println(s.equals(t3)); /* true */
    for(int i = 0; i < s.length(); i ++) {
      System.out.print(s.charAt(i));
    }
    System.out.println();
  }
}
```

empty string

[0, 0)

→ [0, 4) = [0, 3]

entire string

Recursions on Strings

palin("aracecars")

= 'a' == 's' &&

palin("racecar")

strictly smaller problem.

Palindrome

"racecar"

"aracecars"

"raceacar"

compare the 1st and last characters

C1: Same

↳ ______

C2: Diff

↳ not palindrome

Reversal

"abcd"

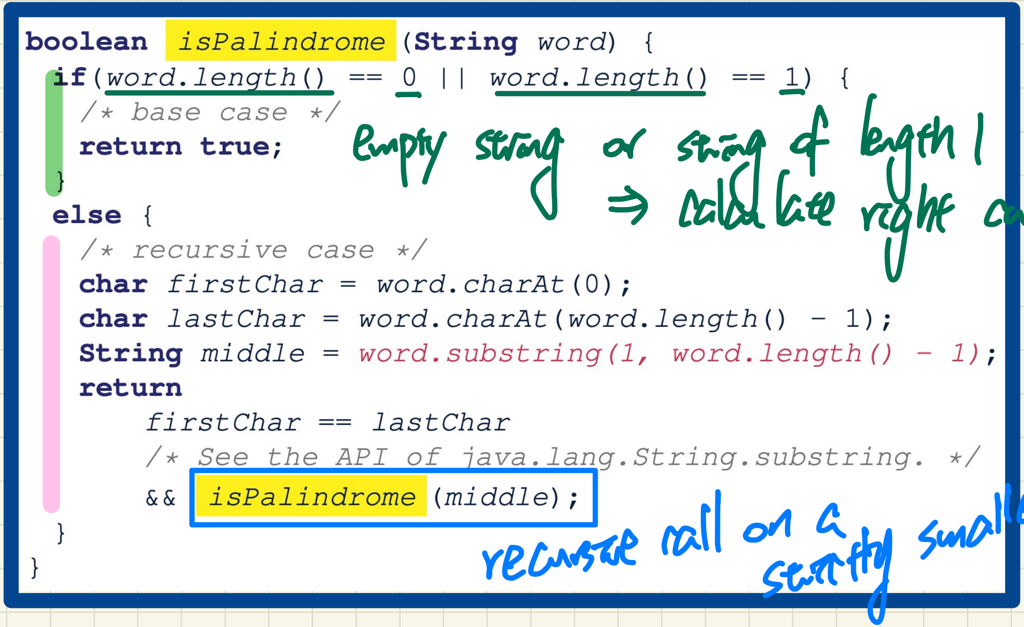
Number of Occurrences

"abca"

'a'

'b'

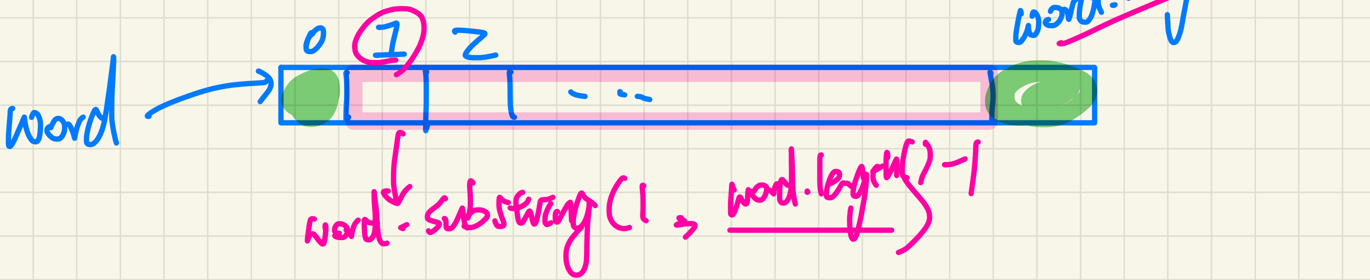
Problem: Palindrome



```
boolean isPalindrome (String word) {
  if(word.length() == 0 || word.length() == 1) {
    /* base case */
    return true;
  }
  else {
    /* recursive case */
    char firstChar = word.charAt(0);
    char lastChar = word.charAt(word.length() - 1);
    String middle = word.substring(1, word.length() - 1);
    return
        firstChar == lastChar
        /* See the API of java.lang.String.substring. */
        && isPalindrome (middle);
  }
}
```

Empty string or string of length 1
⇒ calculate right away

recursive call on a strictly smaller problem.



word
0
1
2
...
word.length()-1
word.substring(1, word.length()-1)