LECTURE 13

WEDNESDAY OCTOBER 23

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:

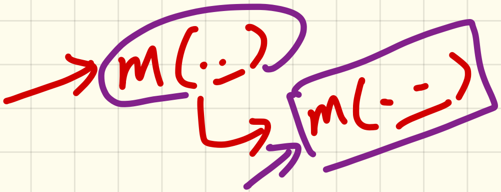
```
m(i) {
  if(i == ...) { /* base case: do something directly */ }
  else {
    m(j);/* recursive call with strictly smaller value */
  }
}
```

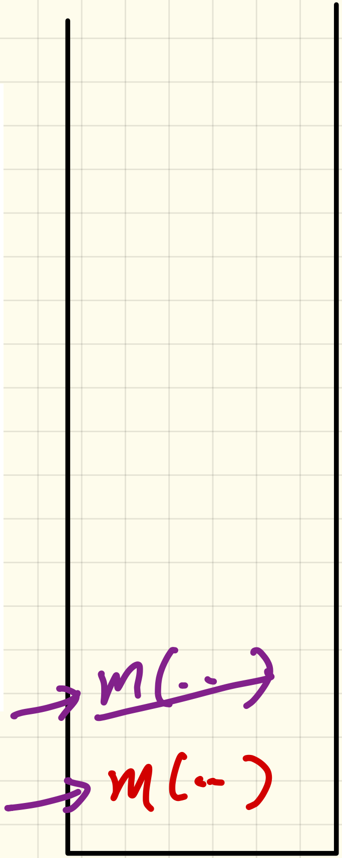
Tracing Recursion via a Stack

- 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

Problem

4! = 4 x 3!

→ size

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

n → size of original prob.

(n−1)! → size of a strictly small problem prob.

4 x 3 x 2 x 1

4 → size of

3 x 2 x 1 = 3!

3! → solution to a strictly smaller problem.

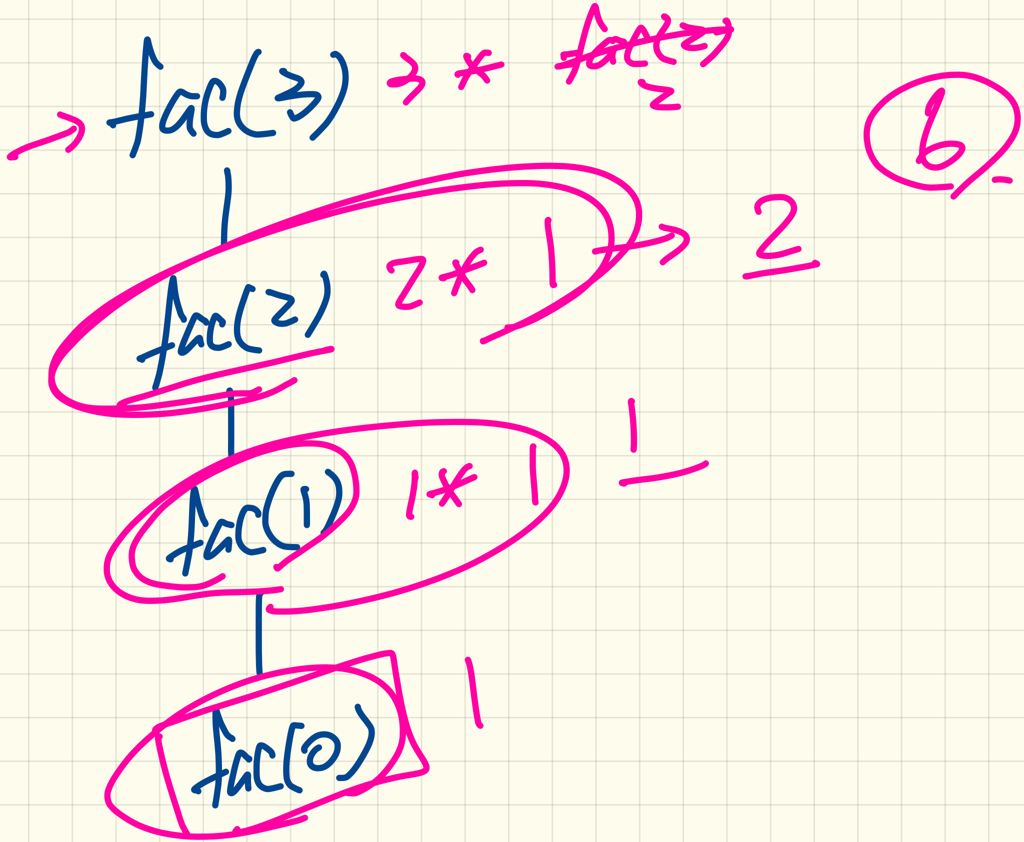
Recursive Solution: factorial

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

```
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)

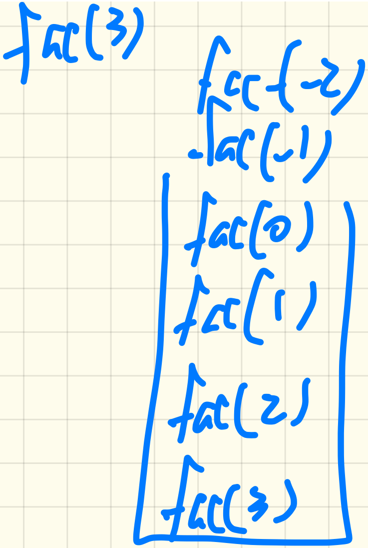
Runtime Stack



fac(3) 3 * fac(2)
2
6
fac(2) 2 * 1
2
fac(1) 1 * 1
1
fac(0)
1

Common Errors of Recursion (1)

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

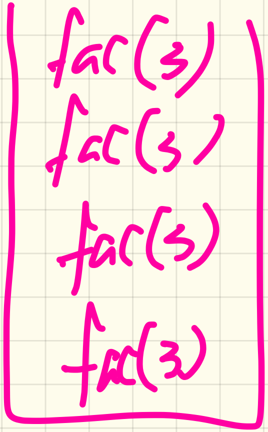


missing base case → no termination

Common Errors of Recursion (2)

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

fac(3)



1.	1	2 ✓	3 ✓	5
fib(1)	fib(2)	fib(3)	fib(4)	fib(5)

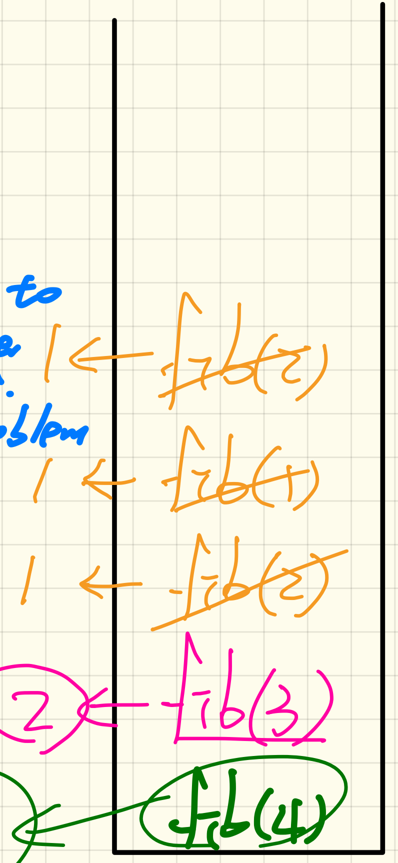
fib(4) = fib(3) + fib(2)

Recursive Solution: Fibonacci Number

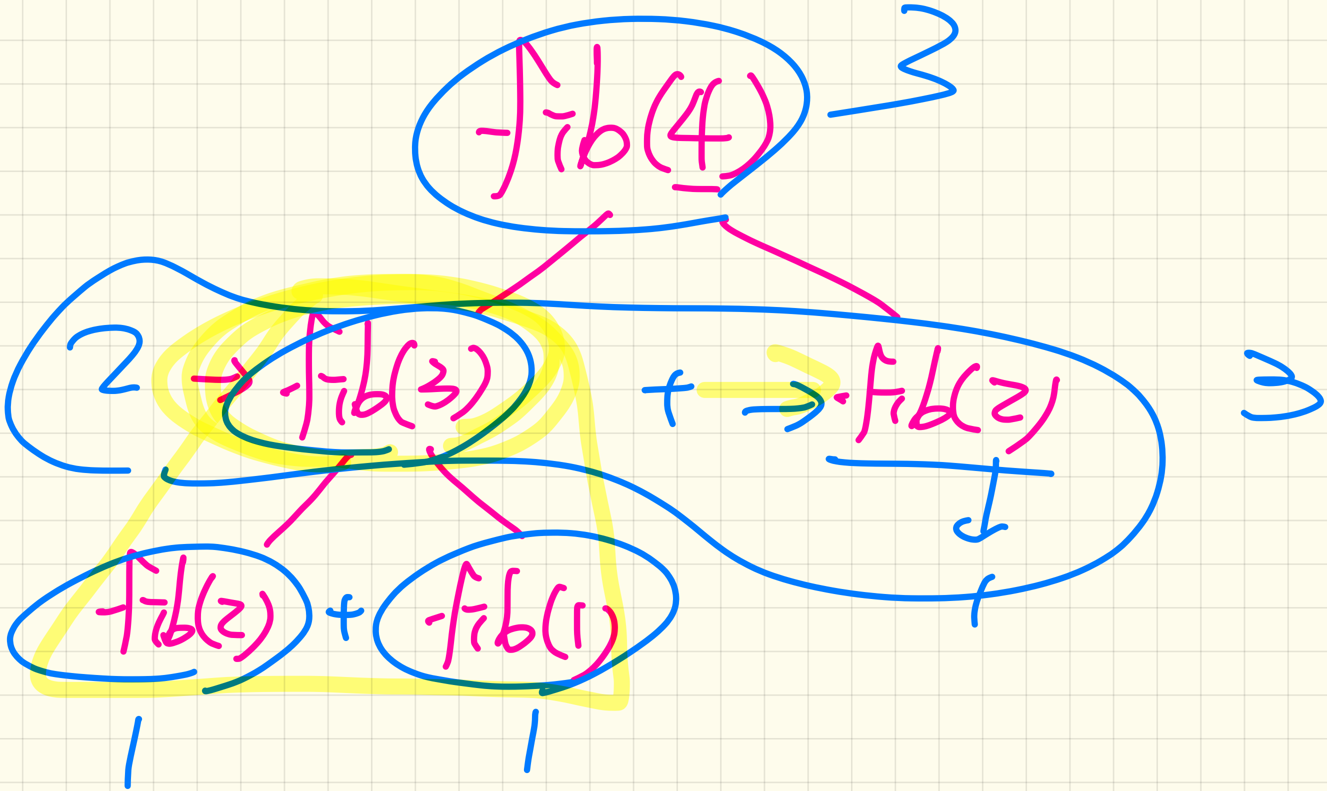
$$F_n = \begin{cases} 1 & \text{if } n = 1 \\ 1 & \text{if } n = 2 \\ F_{n-1} + F_{n-2} & \text{if } n > 2 \end{cases}$$

```
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;
}
```

Example: fib(4)



Runtime Stack



fib(4)
3
2
fib(3)
+
fib(2)
3
1
fib(2)
+
fib(1)
1
1

Use of String

```java
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();
  }
}
```

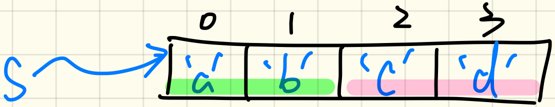
0, -1

0 1 2 3
'a' 'b' 'c' 'd'

→ "" [0, 0) inclusive exclusive

→ get from S[0] ~ S[3]

get from S[1] ~ S[2]



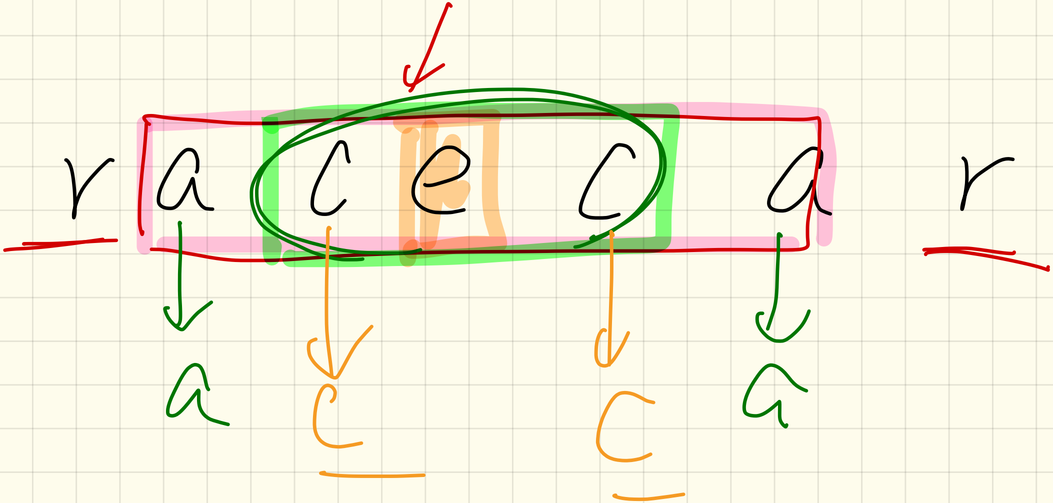
String s

```
s.substring(0, 2)
+
s.substring(2, s.length())
```

4

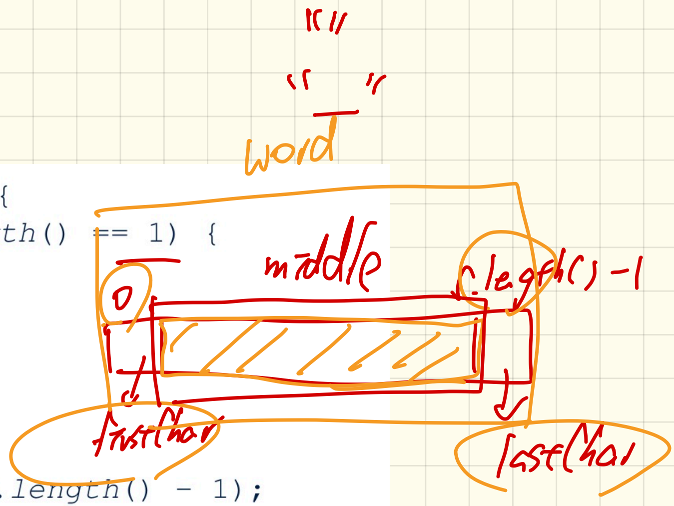
i is a valid index

```
s.substring(0, i)
+
s.substring(i, s.length())
==
s
```


r a c e c a r
a c c a

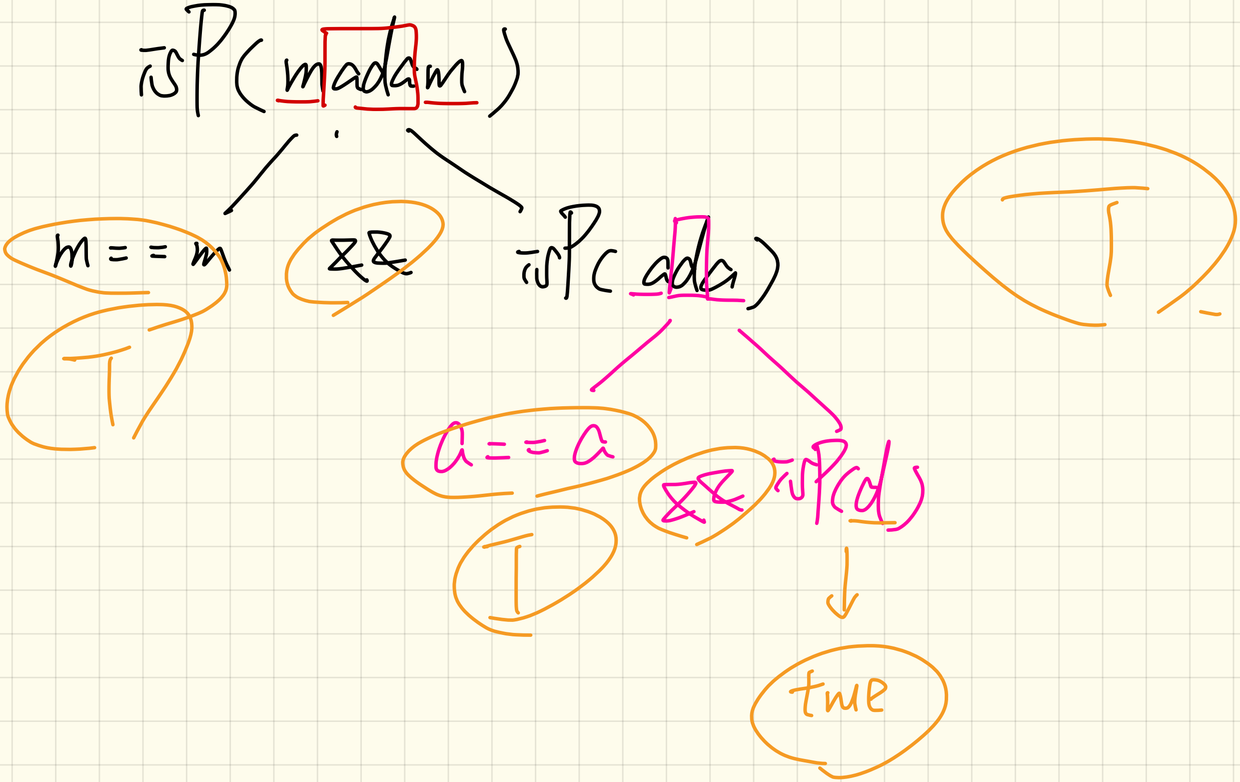
Problem: Palindrome

```java
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);
  }
}
```

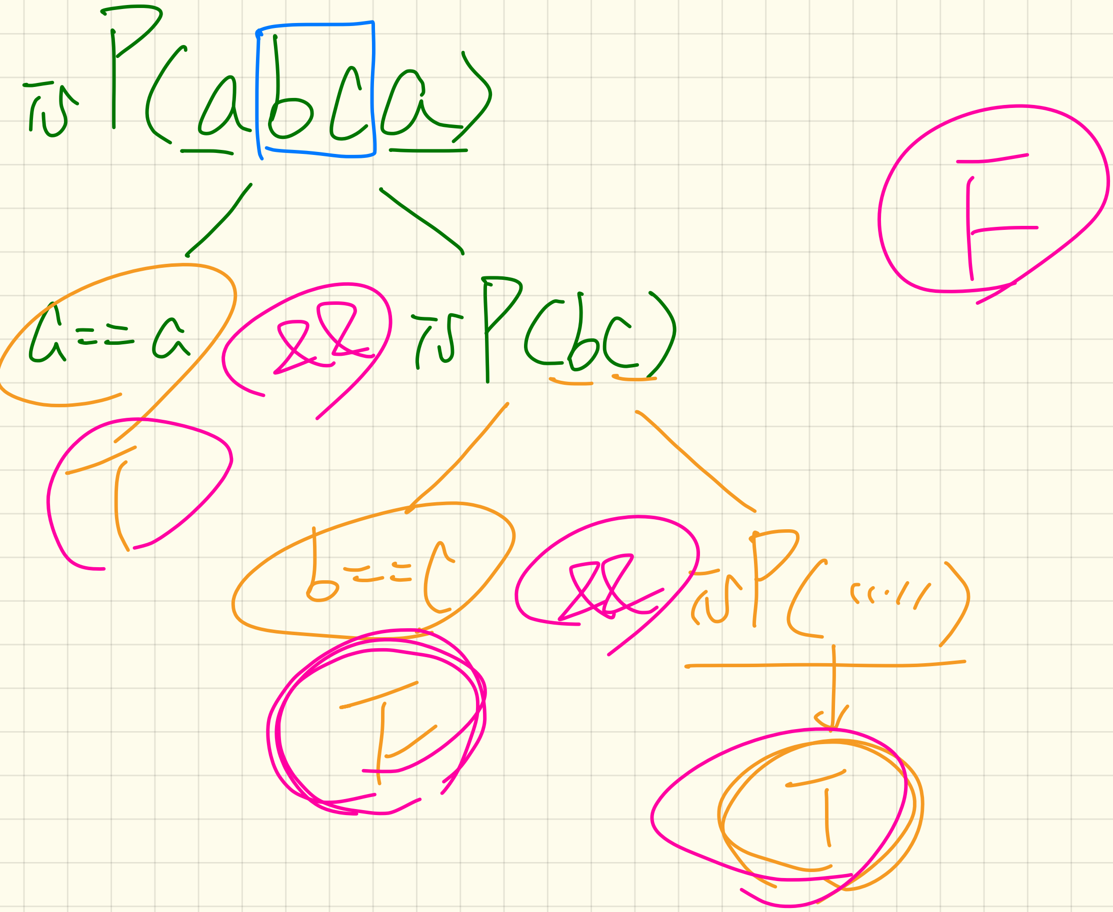


middle vs. word

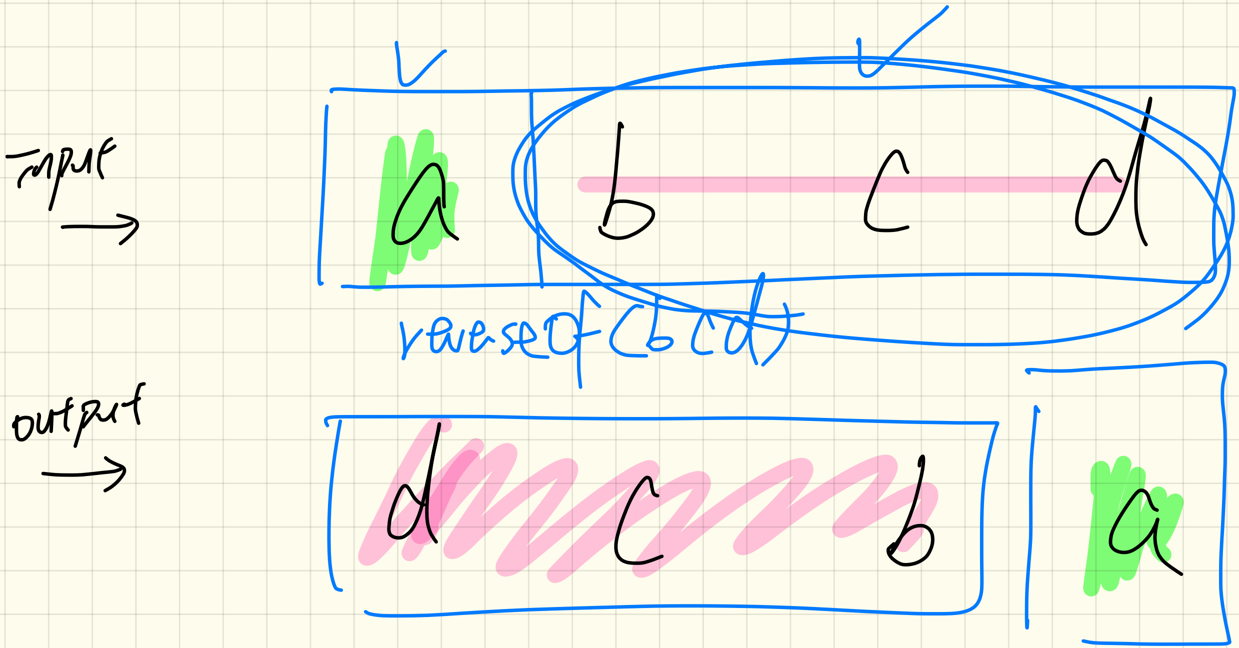
<

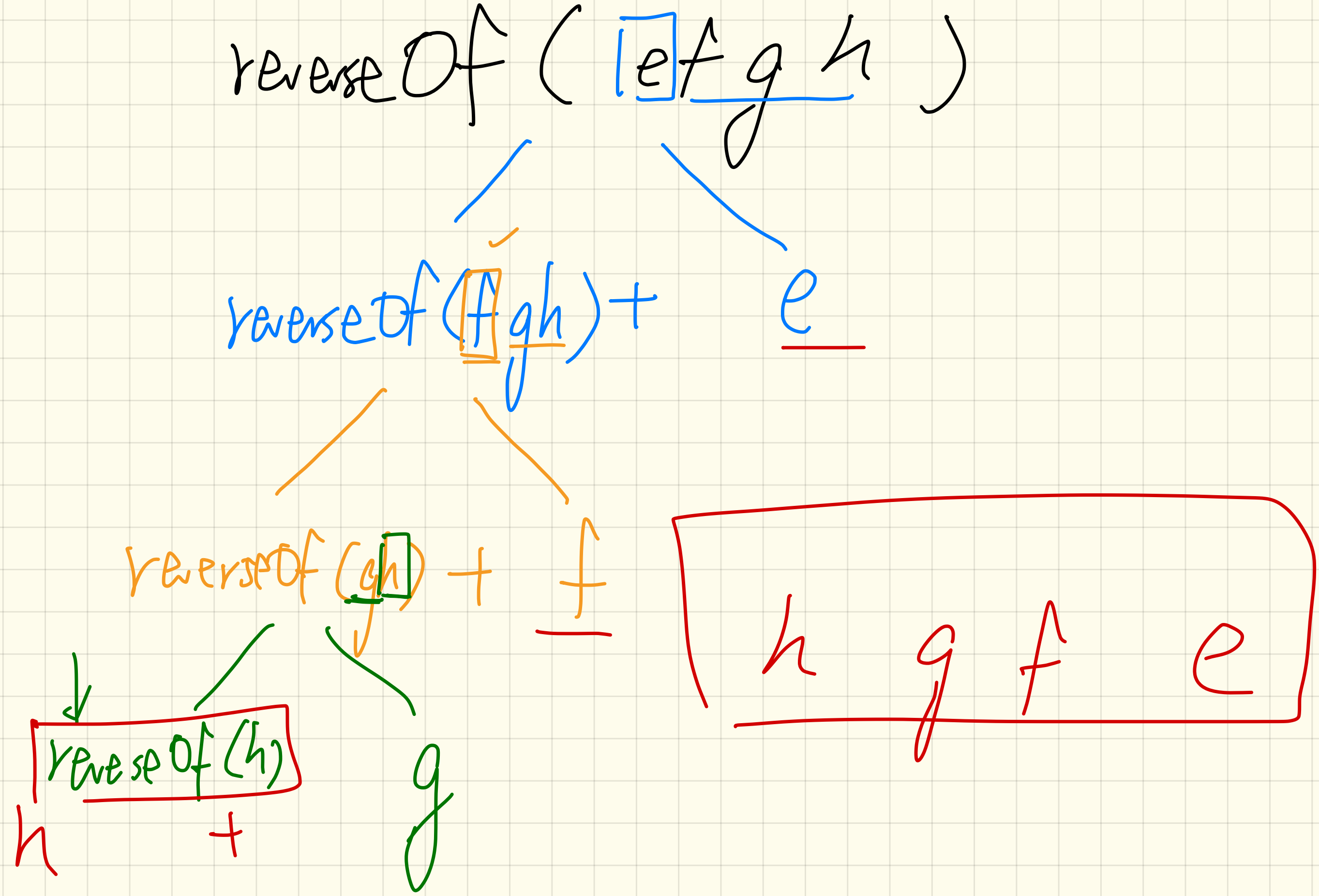


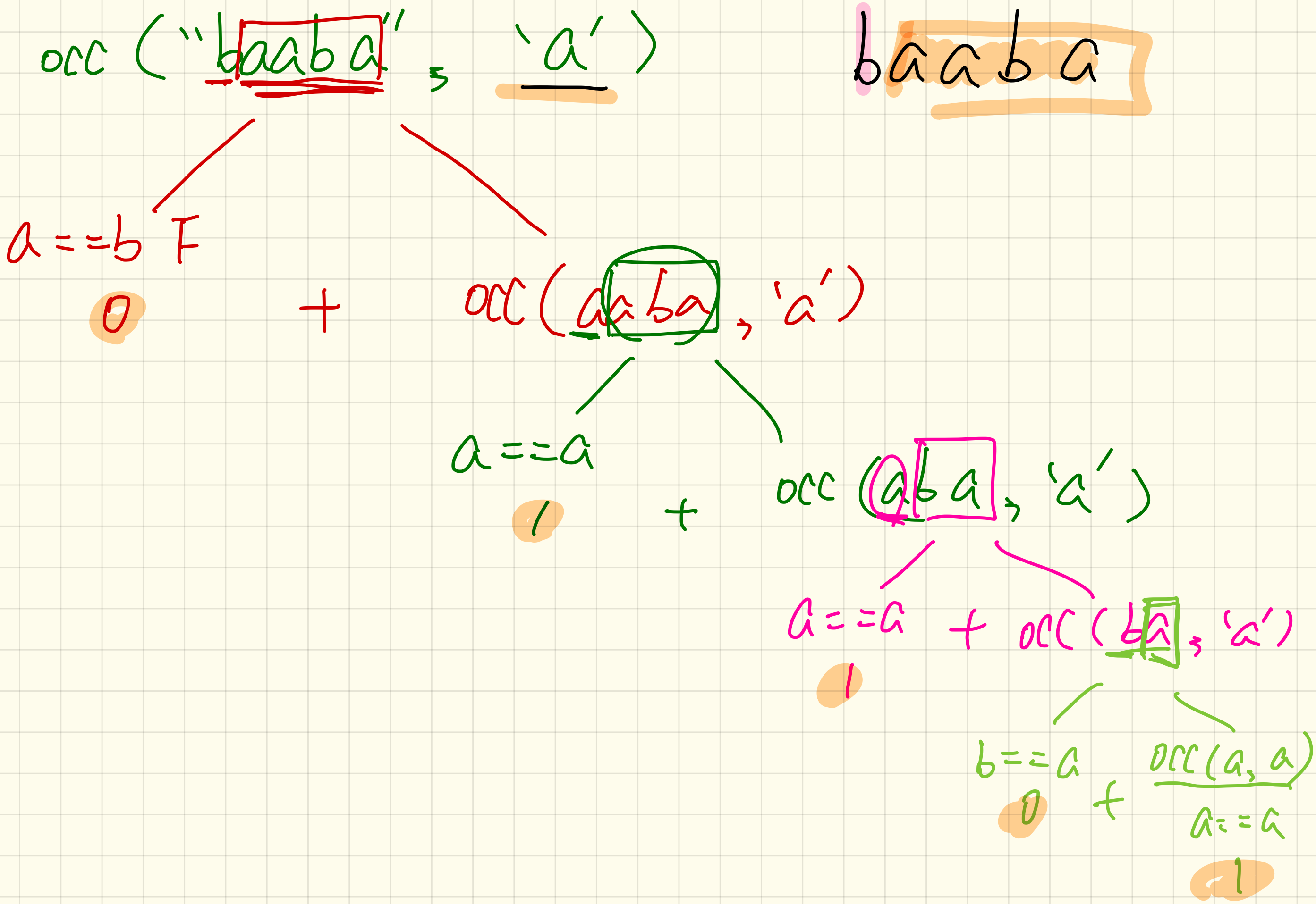
isP(madam)
m == m
&&
isP(ada)
T
a == a
&&
isP(d)
T
true
T



is P(abca)
a == a
&&
is P(bc)
T
b == c
&&
is P(...)
F
F
T


input
a
b
c
d
reverseOf(bcd)
output
d
c
b
a


reverseOf (e f g h)
reverseOf (f g h) + e
reverseOf (g h) + f
reverseOf (h)
g
h +
h g f e

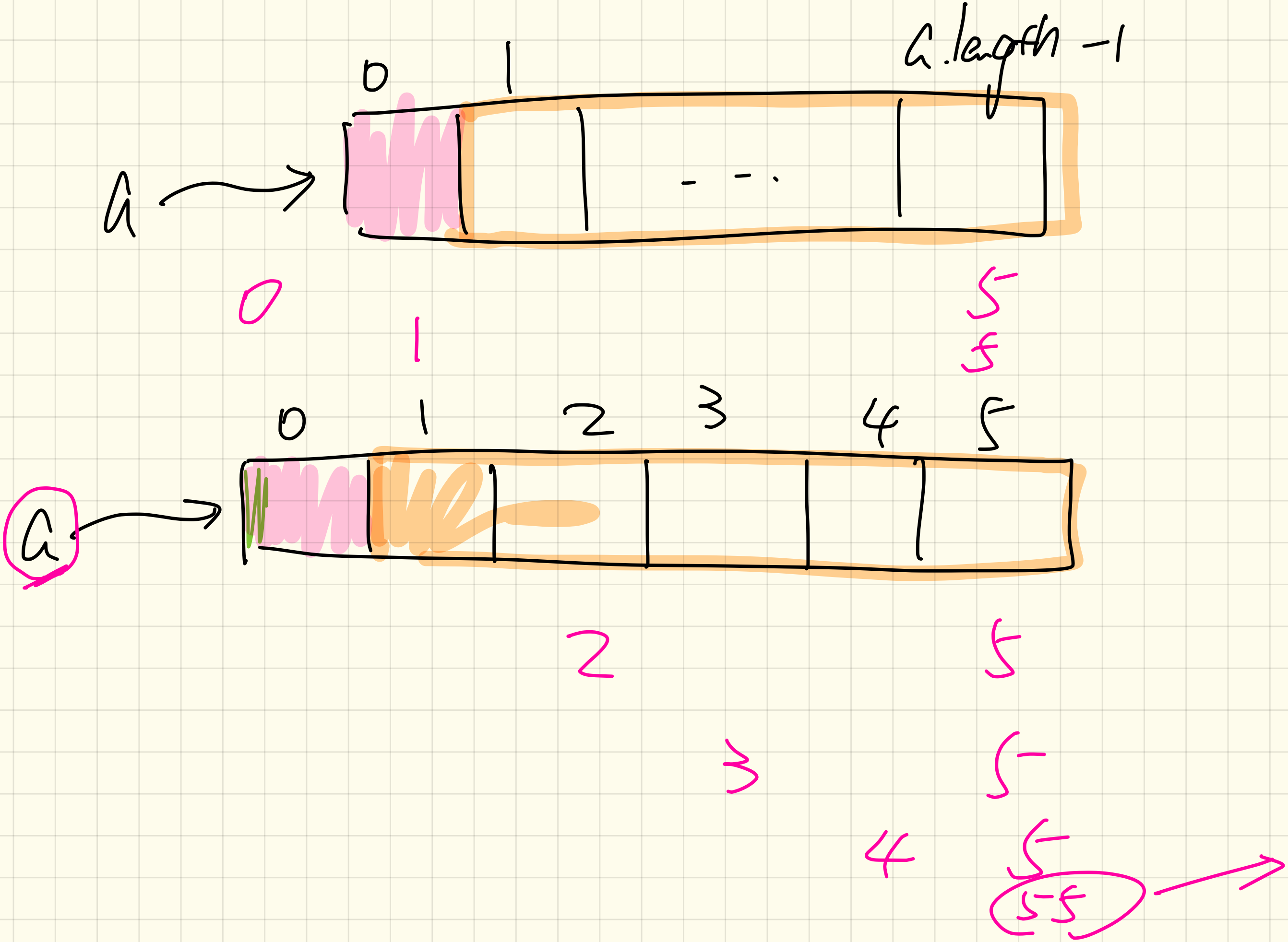

occ("baaba", 'a')
baaba
a==b F
0
+
occ(aaba, 'a')
a==a
1
+
occ(aba, 'a')
a==a
1
+ occ(ba, 'a')
b==a
0
+
occ(a, a)
a==a
1

Problem: Reverse of a String

```java
String reverseOf (String s) {
  if(s.isEmpty()) { /* base case 1 */
    return "";
  }
  else if(s.length() == 1) { /* base case 2 */
    return s;
  }
  else { /* recursive case */
    String tail = s.substring(1, s.length());
    String reverseOfTail = reverseOf (tail);
    char head = s.charAt(0);
    return reverseOfTail + head;
  }
}
```

Problem: Number of Occurrences

```
int occurrencesOf (String s, char c) {
  if(s.isEmpty()) {
    /* Base Case */
    return 0;
  }
  else {
    /* Recursive Case */
    char head = s.charAt(0);
    String tail = s.substring(1, s.length());
    if(head == c) {
      return 1 + occurrencesOf (tail, c);
    }
    else {
      return 0 + occurrencesOf (tail, c);
    }
  }
}
```



a.length -1
0
1
a
0
1
5
5
0
1
2
3
4
5
a
2
5
3
5
4
5
55