Attack of the Clones

ESC101: Fundamentals of Computing

Purushottam Kar

Recursion

Recursion

Process of solving a problem using solutions to “smaller” versions of the same problem!

Recursion

Process of solving a problem using solutions to "smaller" versions of the same problem!

You have already encountered recursion in mathematics

Recursion

Process of solving a problem using solutions to "smaller" versions of the same problem!

You have already encountered recursion in mathematics

Factorial function is defined in terms of factorial itself!

Recursion

Process of solving a problem using solutions to "smaller" versions of the same problem!

You have already encountered recursion in mathematics

Factorial function is defined in terms of factorial itself!

$$\text{fac}(0) = 1 \text{ and } \text{fac}(n) = n \cdot \text{fac}(n-1),\ \text{ for } n > 0$$

Recursion

Process of solving a problem using solutions to "smaller" versions of the same problem!

You have already encountered recursion in mathematics

Factorial function is defined in terms of factorial itself!

$$\mathrm{fac}(0) = 1 \text{ and } \mathrm{fac}(n) = n \cdot \mathrm{fac}(n-1), \text{ for } n > 0$$

Proof by induction is basically a *recursive proof*

Recursion

Process of solving a problem using solutions to "smaller" versions of the same problem!

You have already encountered recursion in mathematics

Factorial function is defined in terms of factorial itself!

$$\mathrm{fac}(0) = 1 \text{ and } \mathrm{fac}(n) = n \cdot \mathrm{fac}(n-1), \text{ for } n > 0$$

Proof by induction is basically a *recursive proof*

Claim: 1 + 2 + 3 + … + n = n(n+1)/2

Recursion

Process of solving a problem using solutions to "smaller" versions of the same problem!

You have already encountered recursion in mathematics

Factorial function is defined in terms of factorial itself!

$$\mathrm{fac}(0) = 1 \text{ and } \mathrm{fac}(n) = n \cdot \mathrm{fac}(n-1), \text{ for } n > 0$$

Proof by induction is basically a *recursive proof*

Claim: 1 + 2 + 3 + … + n = n(n+1)/2

Proof: *Base case*: for n = 1 true by inspection

Recursion

Process of solving a problem using solutions to "smaller" versions of the same problem!

You have already encountered recursion in mathematics

Factorial function is defined in terms of factorial itself!

$$\text{fac}(0) = 1 \text{ and } \text{fac}(n) = n \cdot \text{fac}(n-1), \text{ for } n > 0$$

Proof by induction is basically a *recursive proof*

Claim: 1 + 2 + 3 + … + n = n(n+1)/2

Proof: *Base case*: for n = 1 true by inspection

Inductive case: (1 + … + n) = (1 + … + n-1) + n = (n-1)n/2 + n = n(n+1)/2 ☺

Recursion

Process of solving a problem using solutions to "smaller" versions of the same problem!

You have already encountered recursion in mathematics

Factorial function is defined in terms of factorial itself!

$$\mathrm{fac}(0) = 1 \text{ and } \mathrm{fac}(n) = n \cdot \mathrm{fac}(n-1), \text{ for } n > 0$$

Proof by induction is basically a *recursive proof*

Claim: 1 + 2 + 3 + … + n = n(n+1)/2

Proof: *Base case*: for n = 1 true by inspection

Inductive case: (1 + … + n) = (1 + … + n-1) + n = (n-1)n/2 + n = n(n+1)/2 ☺

Notice that we need a base case and recursive case

Recursion

Process of solving a problem using solutions to "smaller" versions of the same problem!

You have already encountered recursion in mathematics

Factorial function is defined in terms of factorial itself!

$$\text{fac}(0) = 1 \text{ and } \text{fac}(n) = n \cdot \text{fac}(n-1), \text{ for } n > 0$$

Proof by induction is basically a *recursive proof*

Claim: 1 + 2 + 3 + … + n = n(n+1)/2

Proof: *Base case*: for n = 1 true by inspection

Inductive case: (1 + … + n) = (1 + … + n-1) + n = (n-1)n/2 + n = n(n+1)/2 ☺

Notice that we need a base case and recursive case

In case of factorial, fac(0) was the *base case*.

Recursion

Process of solving a problem using solutions to "smaller" versions of the same problem!

You have already encountered recursion in mathematics

Factorial function is defined in terms of factorial itself!

$$\mathrm{fac}(0) = 1 \text{ and } \mathrm{fac}(n) = n \cdot \mathrm{fac}(n-1), \text{ for } n > 0$$

Proof by induction is basically a *recursive proof*

Claim: 1 + 2 + 3 + … + n = n(n+1)/2

Proof: *Base case*: for n = 1 true by inspection

Inductive case: (1 + … + n) = (1 + … + n-1) + n = (n-1)n/2 + n = n(n+1)/2 ☺

Notice that we need a base case and recursive case

In case of factorial, fac(0) was the *base case*.

This is true when writing recursive functions in C language as well

Recursion

Process of solving a problem using solutions to "smaller" versions of the same problem!

You have already encountered recursion in mathematics

Factorial function is defined in terms of factorial itself!

$$\text{fac}(0) = 1 \text{ and } \text{fac}(n) = n \cdot \text{fac}(n-1), \text{ for } n > 0$$

Proof by induction is basically a *recurs*

We used the proof for the case n-1 to prove the case n

- **Claim**: 1 + 2 + 3 + … + n = n(n+1)/2
- **Proof**: *Base case*: for n = 1 true by inspection
- *Inductive case*: (1 + … + n) = (1 + … + n-1) + n = (n-1)n/2 + n = n(n+1)/2 ☺

Notice that we need a base case and recursive case

- In case of factorial, fac(0) was the *base case*.
- This is true when writing recursive functions in C language as well

About Recursion

About Recursion

Recursion can allow us to write very elegant code

About Recursion

Recursion can allow us to write very elegant code

- Very easy to understand what is going on by just reading function definition

About Recursion

Recursion can allow us to write very elegant code

- Very easy to understand what is going on by just reading function definition
- Sometimes you can just copy the function definition into code ☺

About Recursion

- Recursion can allow us to write very elegant code
  - Very easy to understand what is going on by just reading function definition
  - Sometimes you can just copy the function definition into code ☺
- Careful: do not forget to write down the base case

About Recursion

- Recursion can allow us to write very elegant code
  - Very easy to understand what is going on by just reading function definition
  - Sometimes you can just copy the function definition into code ☺
- Careful: do not forget to write down the base case
  - Will go into something like an infinite loop if you forget the base case

About Recursion

- Recursion can allow us to write very elegant code
  - Very easy to understand what is going on by just reading function definition
  - Sometimes you can just copy the function definition into code ☺
- Careful: do not forget to write down the base case
  - Will go into something like an infinite loop if you forget the base case
  - May end up exceeding time and memory limits on Prutor

About Recursion

- Recursion can allow us to write very elegant code
  - Very easy to understand what is going on by just reading function definition
  - Sometimes you can just copy the function definition into code ☺
- Careful: do not forget to write down the base case
  - Will go into something like an infinite loop if you forget the base case
  - May end up exceeding time and memory limits on Prutor
  - Will get a TLE/runtime error message on Prutor

About Recursion

Recursion can allow us to write very elegant code

- Very easy to understand what is going on by just reading function definition
- Sometimes you can just copy the function definition into code ☺

Careful: do not forget to write down the base case

- Will go into something like an infinite loop if you forget the base case
- May end up exceeding time and memory limits on Prutor
- Will get a TLE/runtime error message on Prutor

Careful: problems that can be solved using recursion can always be solved using loops too

About Recursion

- Recursion can allow us to write very elegant code
  - Very easy to understand what is going on by just reading function definition
  - Sometimes you can just copy the function definition into code ☺
- Careful: do not forget to write down the base case
  - Will go into something like an infinite loop if you forget the base case
  - May end up exceeding time and memory limits on Prutor
  - Will get a TLE/runtime error message on Prutor
- Careful: problems that can be solved using recursion can always be solved using loops too
  - Fundamental result in computer science: Church-Turing thesis

About Recursion

Recursion can allow us to write very elegant code

- Very easy to understand what is going on by just reading function definition
- Sometimes you can just copy the function definition into code ☺

Careful: do not forget to write down the base case

- Will go into something like an infinite loop if you forget the base case
- May end up exceeding time and memory limits on Prutor
- Will get a TLE/runtime error message on Prutor

Careful: problems that can be solved using recursion can always be solved using loops too

- Fundamental result in computer science: Church-Turing thesis
- Disadvantage: loop solutions sometimes very difficult to write and read

About Recursion

Recursion can allow us to write very elegant code

- Very easy to understand what is going on by just reading function definition
- Sometimes you can just copy the function definition into code ☺

Careful: do not forget to write down the base case

- Will go into something like an infinite loop if you forget the base case
- May end up exceeding time and memory limits on Prutor
- Will get a TLE/runtime error message on Prutor

Careful: problems that can be solved using recursion can always be solved using loops too

- Fundamental result in computer science: Church-Turing thesis
- <u>Disadvantage</u>: loop solutions sometimes very difficult to write and read
- <u>Advantage</u>: loop solutions can be much faster than recursion solution

Recognizing Recursion

Recognizing Recursion

Sometimes it is very easy to see that the problem can be solved using recursion – example factorial, Fibonacci

Recognizing Recursion

Sometimes it is very easy to see that the problem can be solved using recursion – example factorial, Fibonacci

Sometimes it is harder to see that recursion can be used to solve the problem – example gcd, partition

Recognizing Recursion

Sometimes it is very easy to see that the problem can be solved using recursion – example factorial, Fibonacci

Sometimes it is harder to see that recursion can be used to solve the problem – example gcd, partition

No small set of golden rules on how to find out when and if a problem can be solved using recursion ☹

Recognizing Recursion

Sometimes it is very easy to see that the problem can be solved using recursion – example factorial, Fibonacci

Sometimes it is harder to see that recursion can be used to solve the problem – example gcd, partition

No small set of golden rules on how to find out when and if a problem can be solved using recursion ☹

Need to look at the problem carefully and see if it can be solved using smaller versions of the same problem

Recognizing Recursion

Sometimes it is very easy to see that the problem can be solved using recursion – example factorial, Fibonacci

Sometimes it is harder to see that recursion can be used to solve the problem – example gcd, partition

No small set of golden rules on how to find out when and if a problem can be solved using recursion ☹

Need to look at the problem carefully and see if it can be solved using smaller versions of the same problem

Will see several examples of this in ESC101. More examples in advanced courses e.g. ESO207, CS345

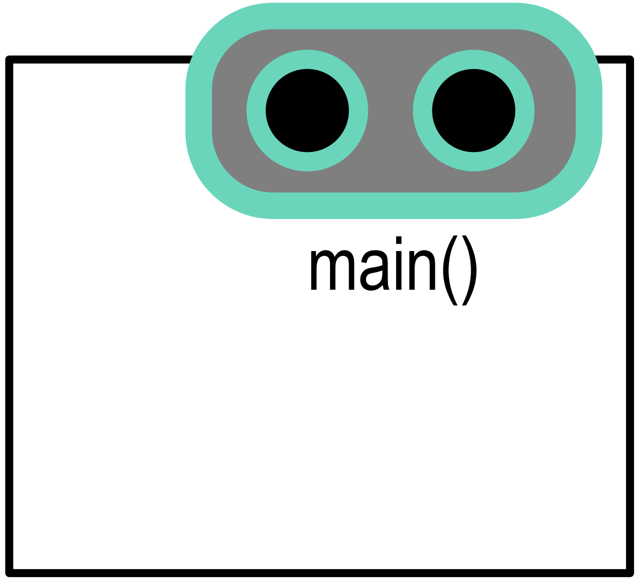
Example 1: Factorial

Example 1: Factorial


```
int fact(int a){
    if(a == 0) return 1;
    return a * fact(a - 1);
}
int main(){
    printf("%d", fact(1+1));
}
```

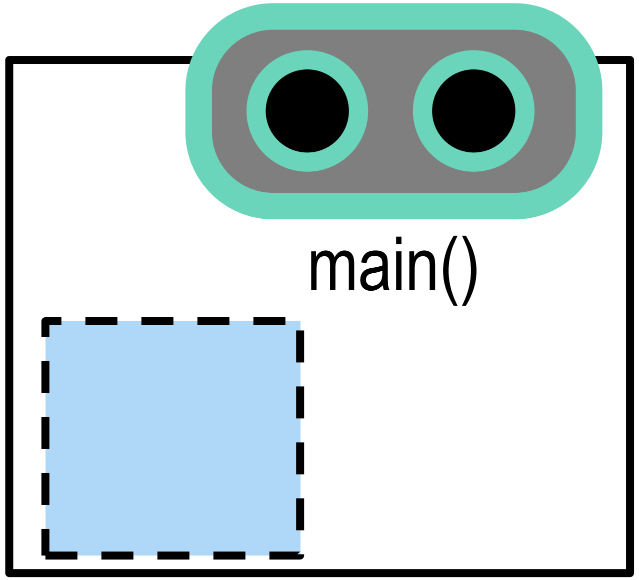
Example 1: Factorial

```
int fact(int a){
    if(a == 0) return 1;
    return a * fact(a - 1);
}
int main(){
    printf("%d", fact(1+1));
}
```



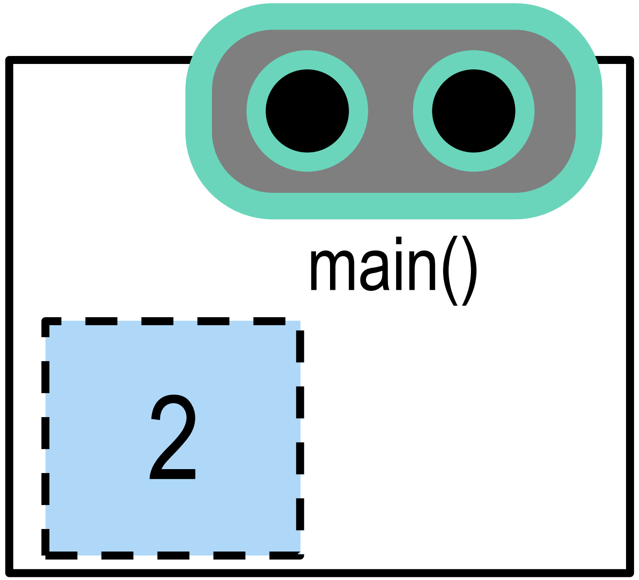
Example 1: Factorial

```
int fact(int a){
    if(a == 0) return 1;
    return a * fact(a - 1);
}
int main(){
    printf("%d", fact(1+1));
}
```



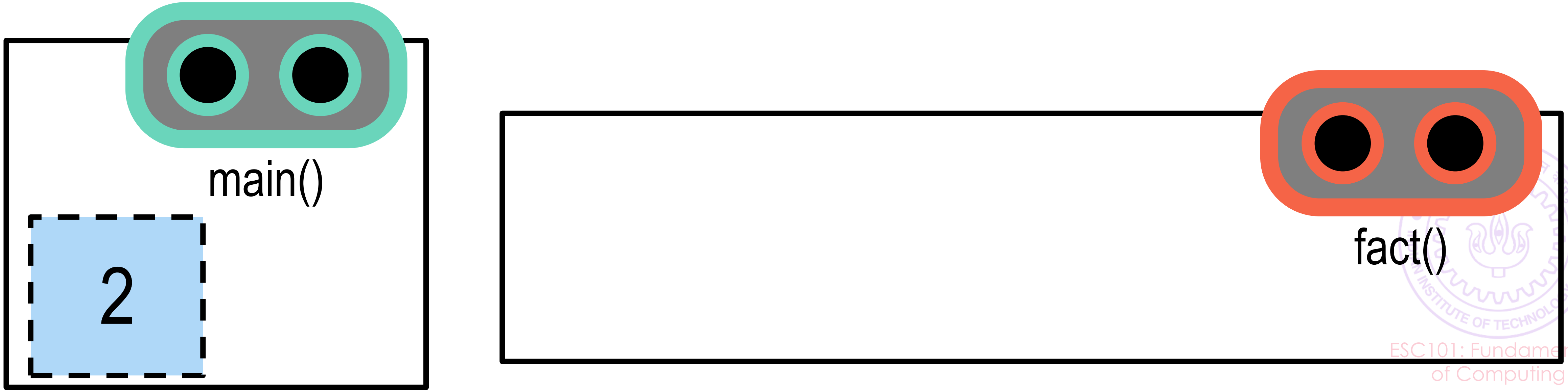
Example 1: Factorial


```
int fact(int a){
    if(a == 0) return 1;
    return a * fact(a - 1);
}
int main(){
    printf("%d", fact(1+1));
}
```



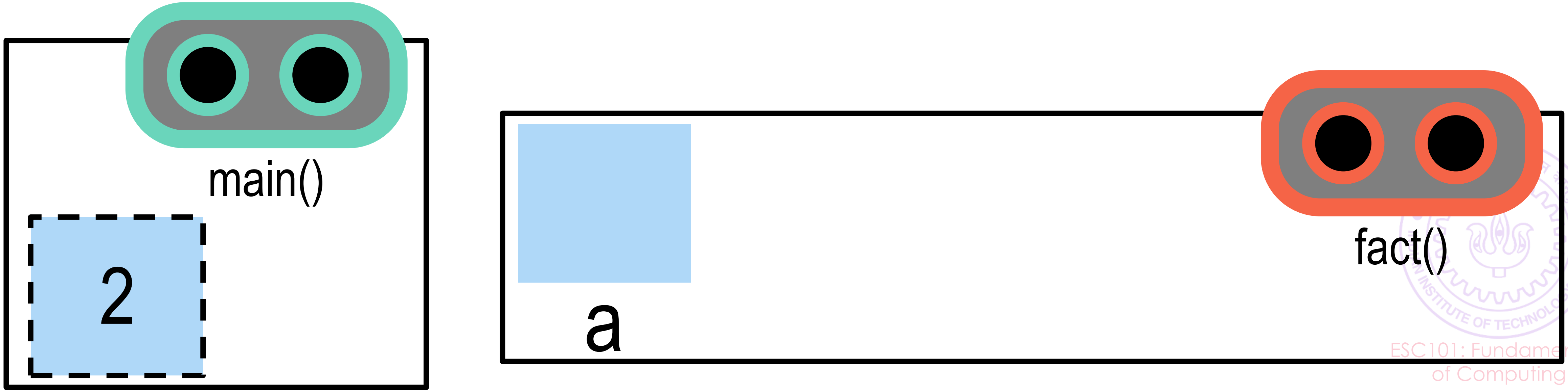
Example 1: Factorial

```
int fact(int a){
    if(a == 0) return 1;
    return a * fact(a - 1);
}
int main(){
    printf("%d", fact(1+1));
}
```



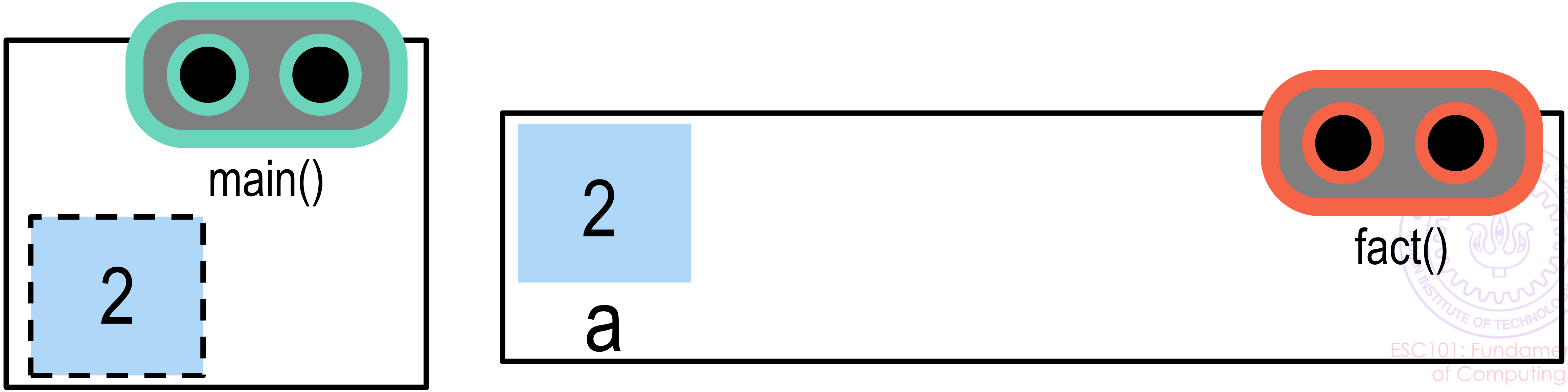
Example 1: Factorial


```
int fact(int a){
    if(a == 0) return 1;
    return a * fact(a - 1);
}
int main(){
    printf("%d", fact(1+1));
}
```



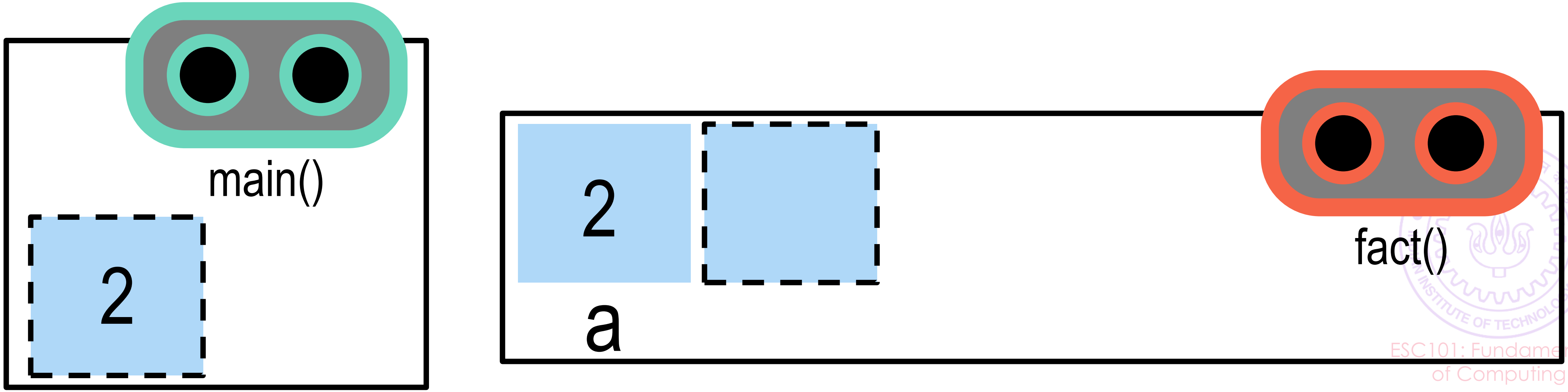
Example 1: Factorial


```
int fact(int a){
    if(a == 0) return 1;
    return a * fact(a - 1);
}
int main(){
    printf("%d", fact(1+1));
}
```



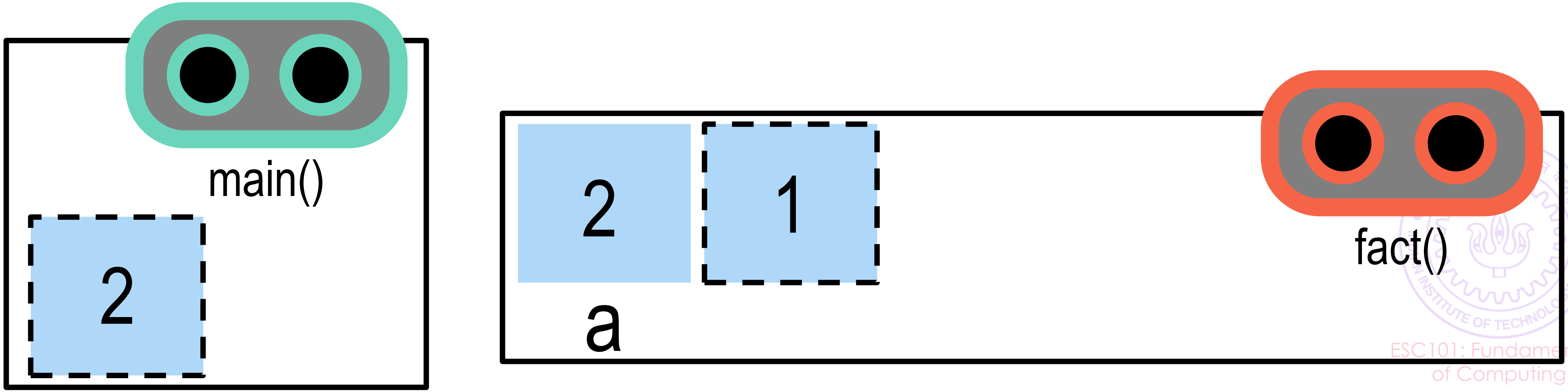
Example 1: Factorial


```
int fact(int a){
    if(a == 0) return 1;
    return a * fact(a - 1);
}
int main(){
    printf("%d", fact(1+1));
}
```



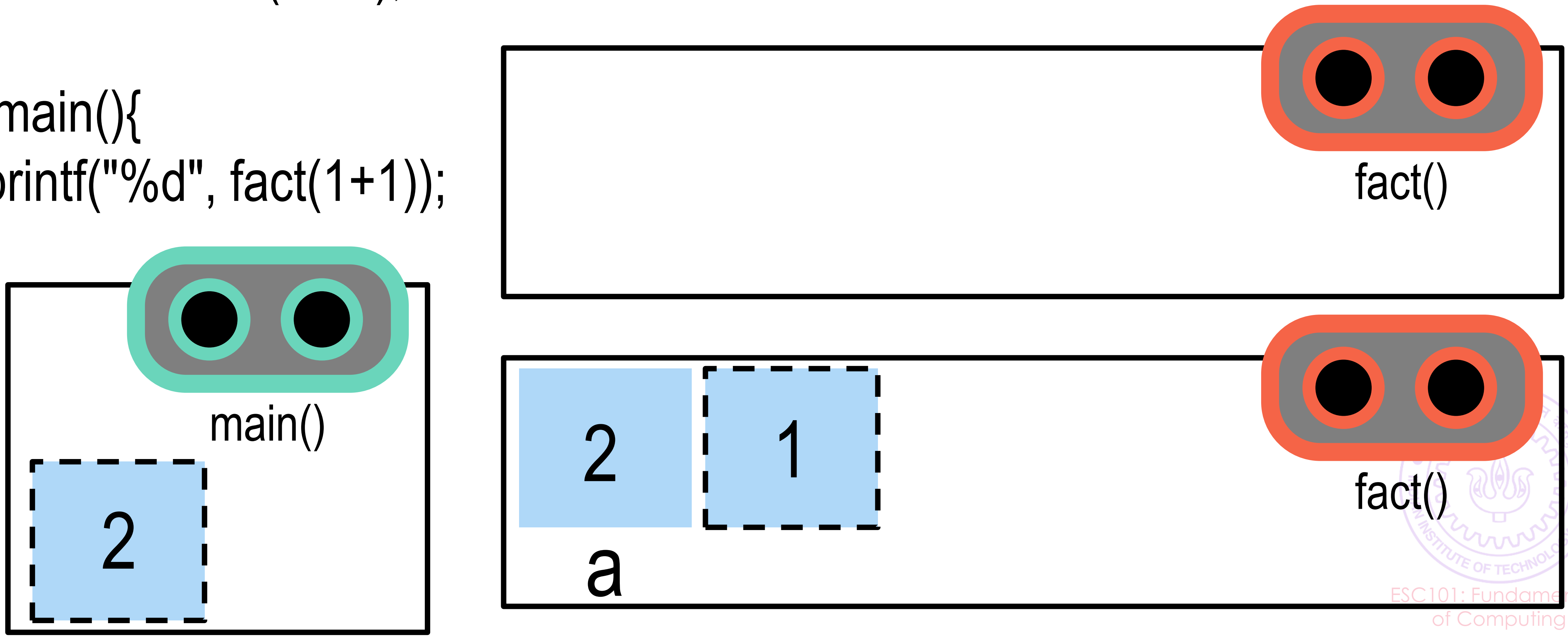
Example 1: Factorial

```
int fact(int a){
    if(a == 0) return 1;
    return a * fact(a - 1);
}
int main(){
    printf("%d", fact(1+1));
}
```



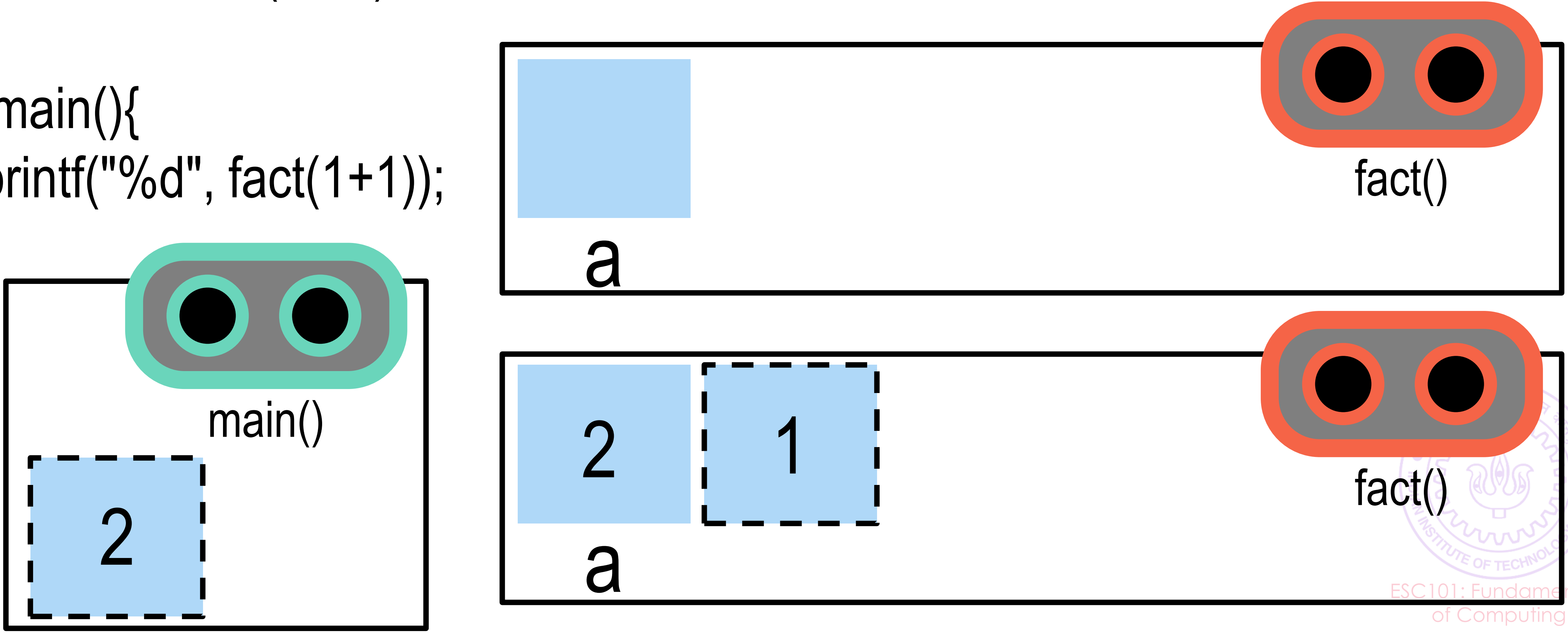
Example 1: Factorial


```
int fact(int a){
    if(a == 0) return 1;
    return a * fact(a - 1);
}
int main(){
    printf("%d", fact(1+1));
}
```



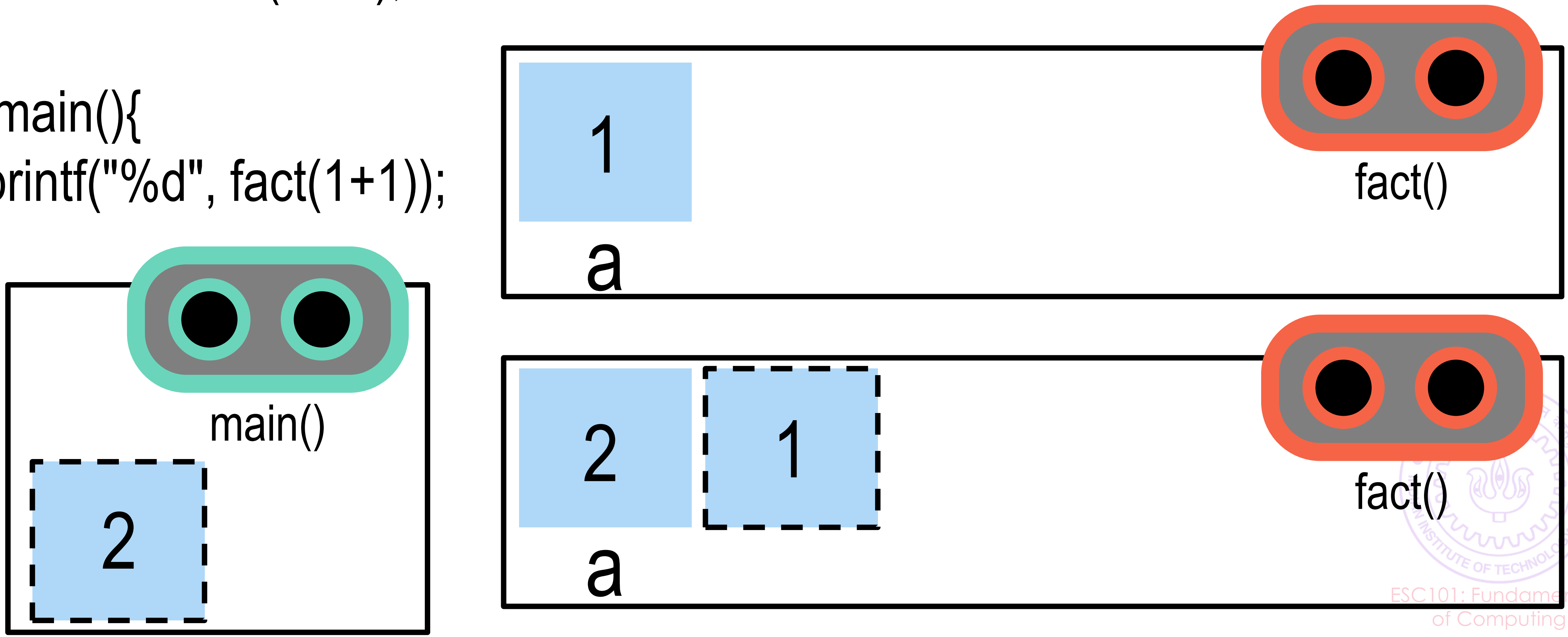
Example 1: Factorial


```
int fact(int a){
    if(a == 0) return 1;
    return a * fact(a - 1);
}
int main(){
    printf("%d", fact(1+1));
}
```



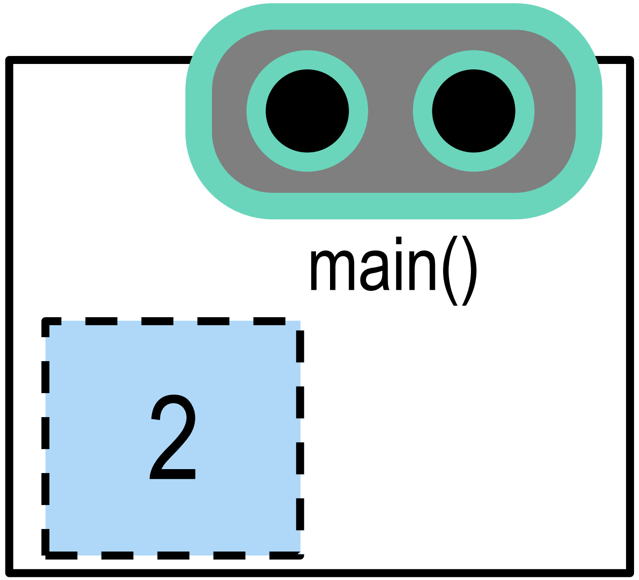
Example 1: Factorial

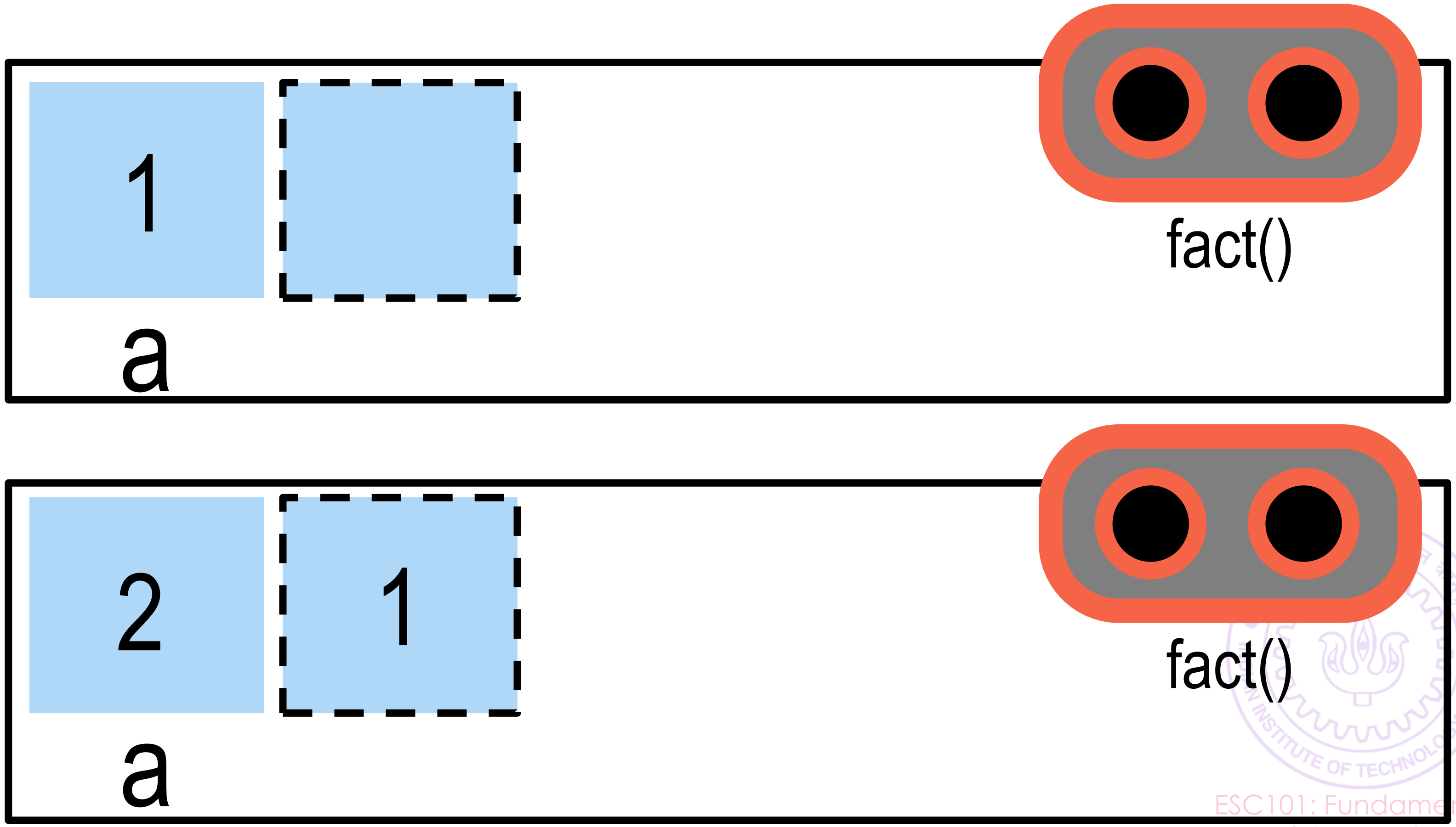

```
int fact(int a){
    if(a == 0) return 1;
    return a * fact(a - 1);
}
int main(){
    printf("%d", fact(1+1));
}
```



Example 1: Factorial

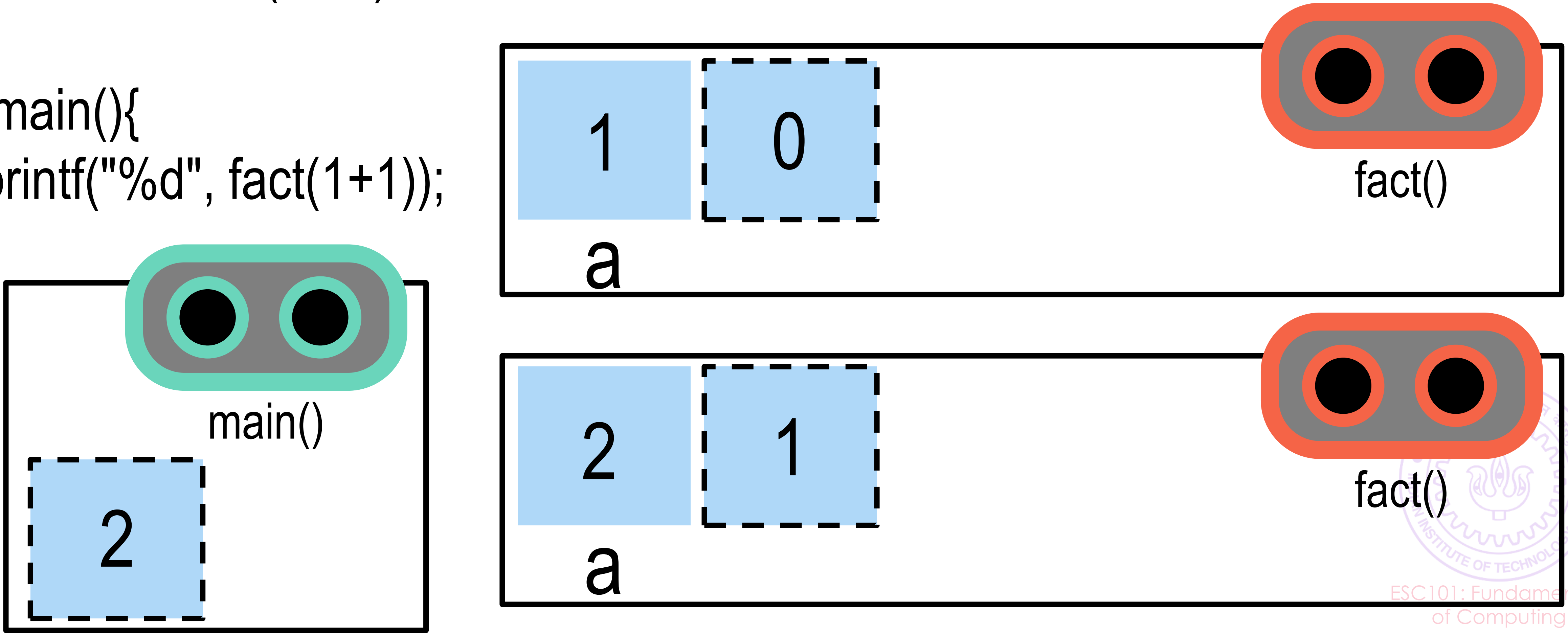
```
int fact(int a){
    if(a == 0) return 1;
    return a * fact(a - 1);
}
int main(){
    printf("%d", fact(1+1));
}
```





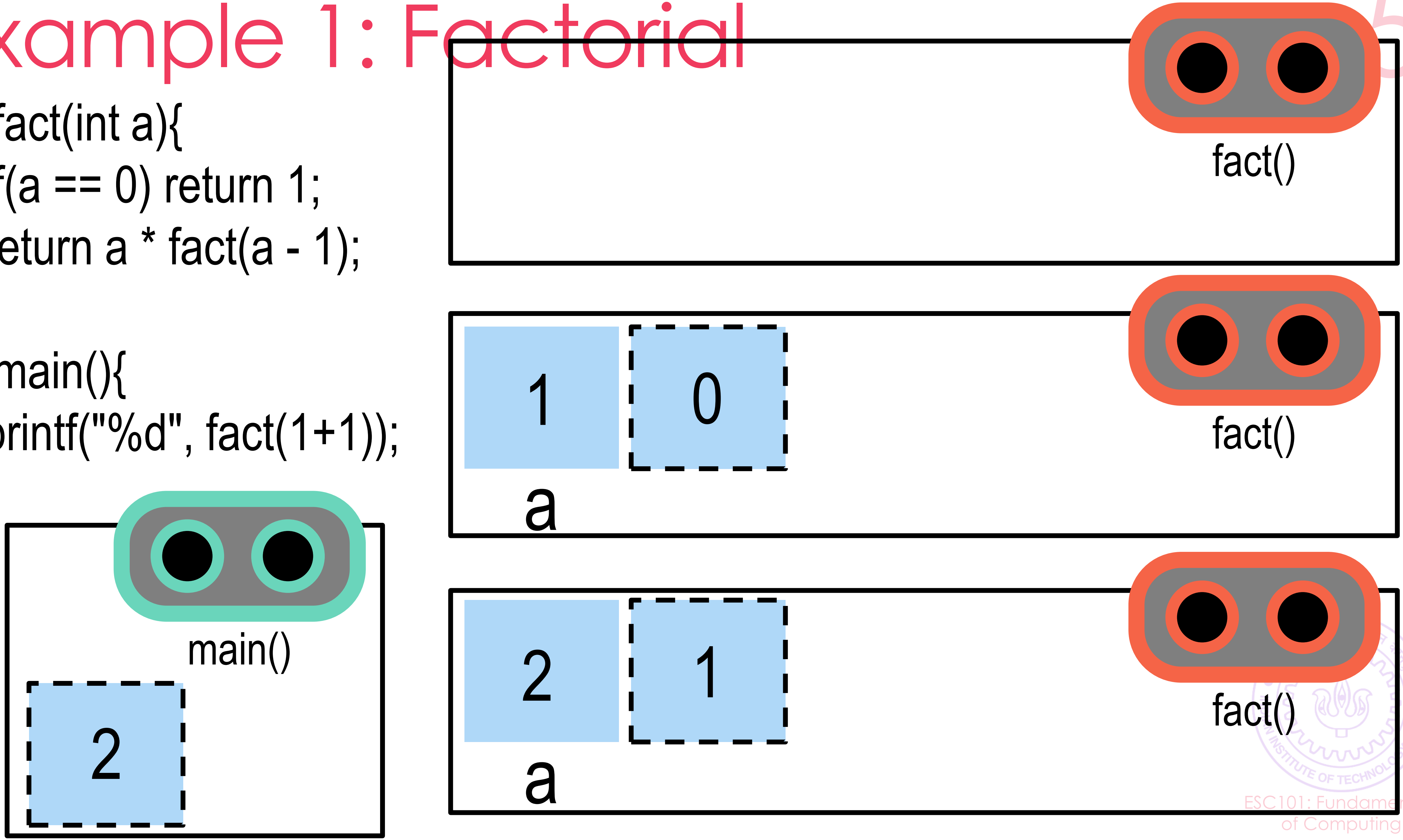
Example 1: Factorial


```
int fact(int a){
    if(a == 0) return 1;
    return a * fact(a - 1);
}
int main(){
    printf("%d", fact(1+1));
}
```



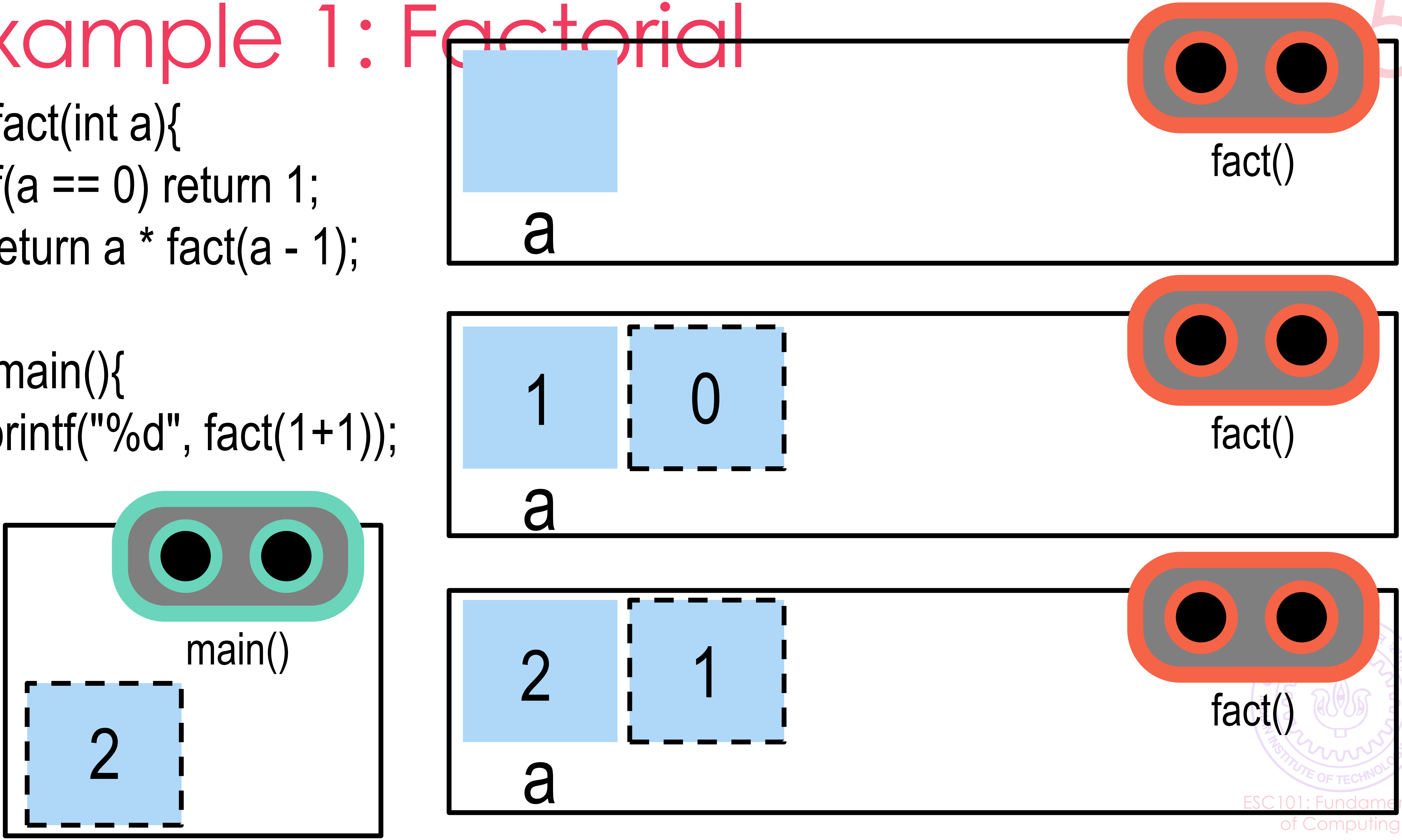
Example 1: Factorial


```
int fact(int a){
    if(a == 0) return 1;
    return a * fact(a - 1);
}
int main(){
    printf("%d", fact(1+1));
}
```



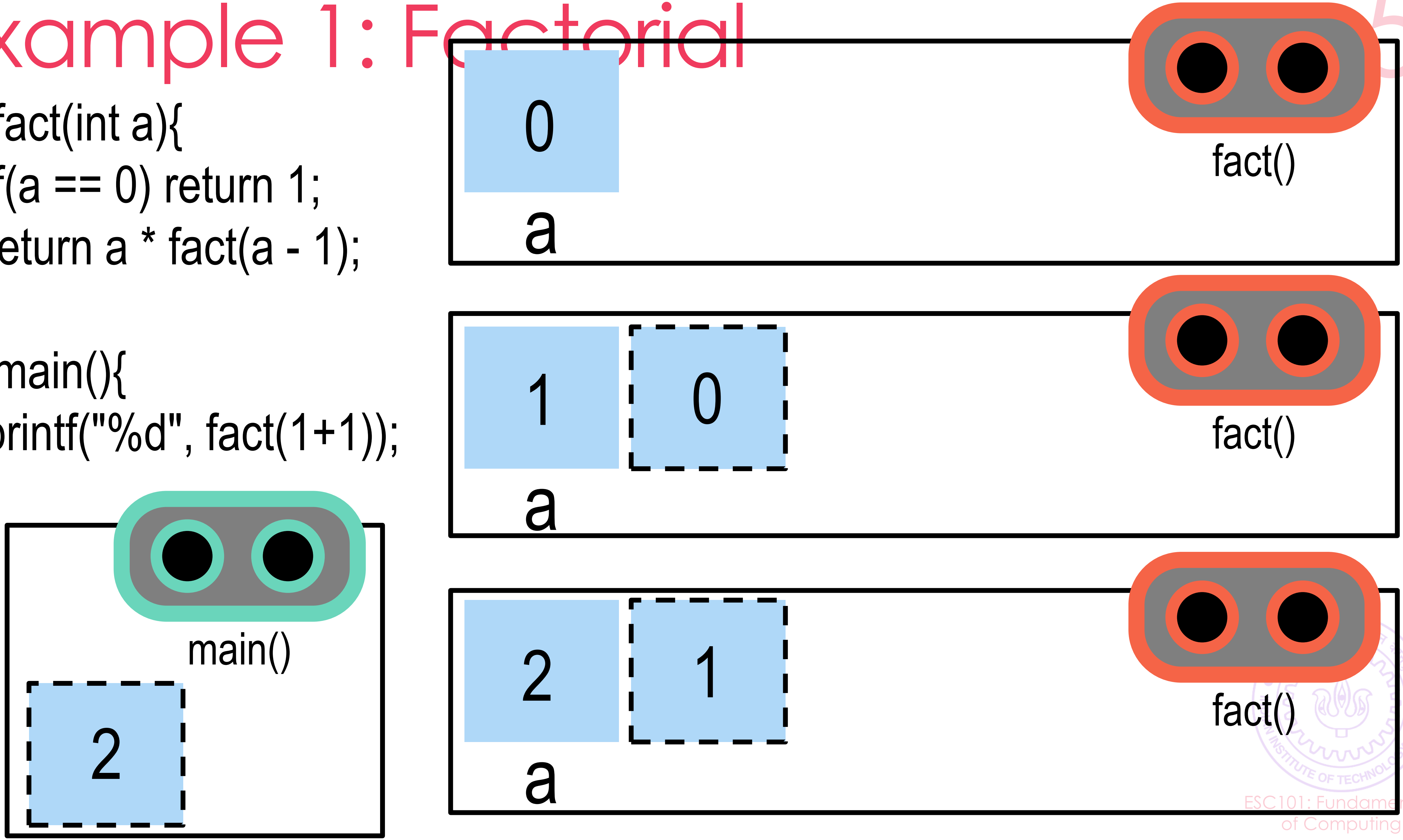
Example 1: Factorial


```
int fact(int a){
    if(a == 0) return 1;
    return a * fact(a - 1);
}
int main(){
    printf("%d", fact(1+1));
}
```



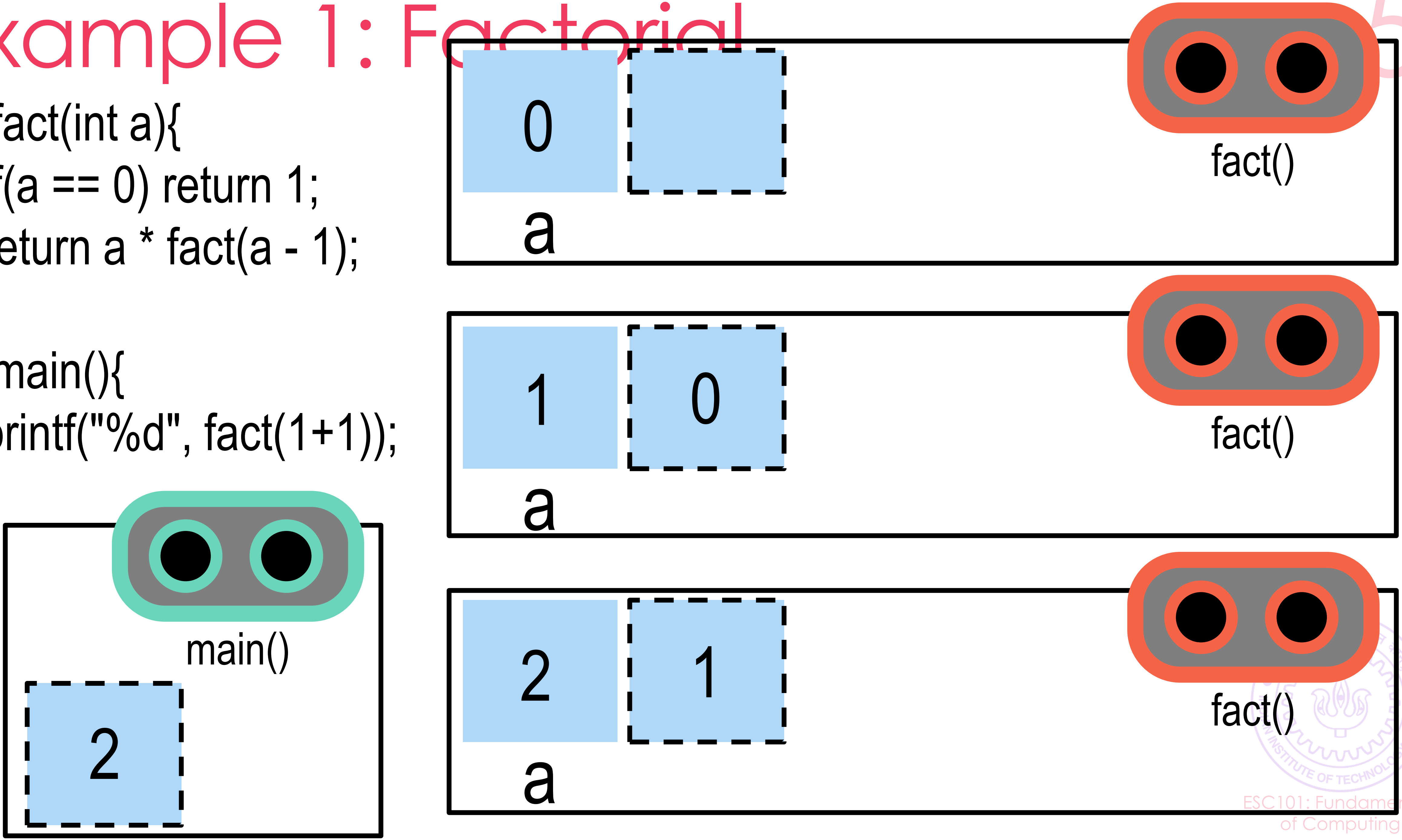
Example 1: Factorial


```
int fact(int a){
    if(a == 0) return 1;
    return a * fact(a - 1);
}
int main(){
    printf("%d", fact(1+1));
}
```



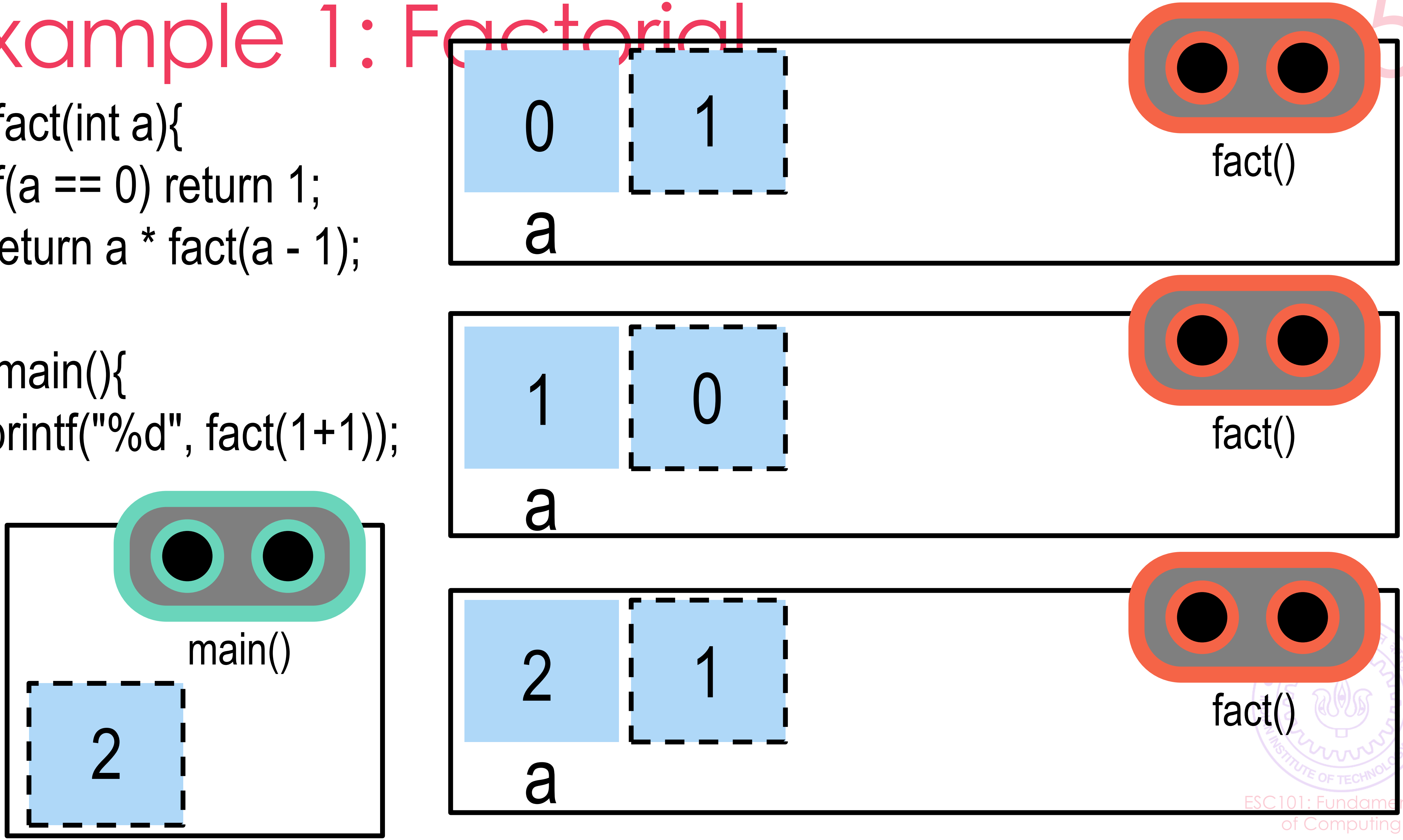
Example 1: Factorial


```
int fact(int a){
    if(a == 0) return 1;
    return a * fact(a - 1);
}
int main(){
    printf("%d", fact(1+1));
}
```



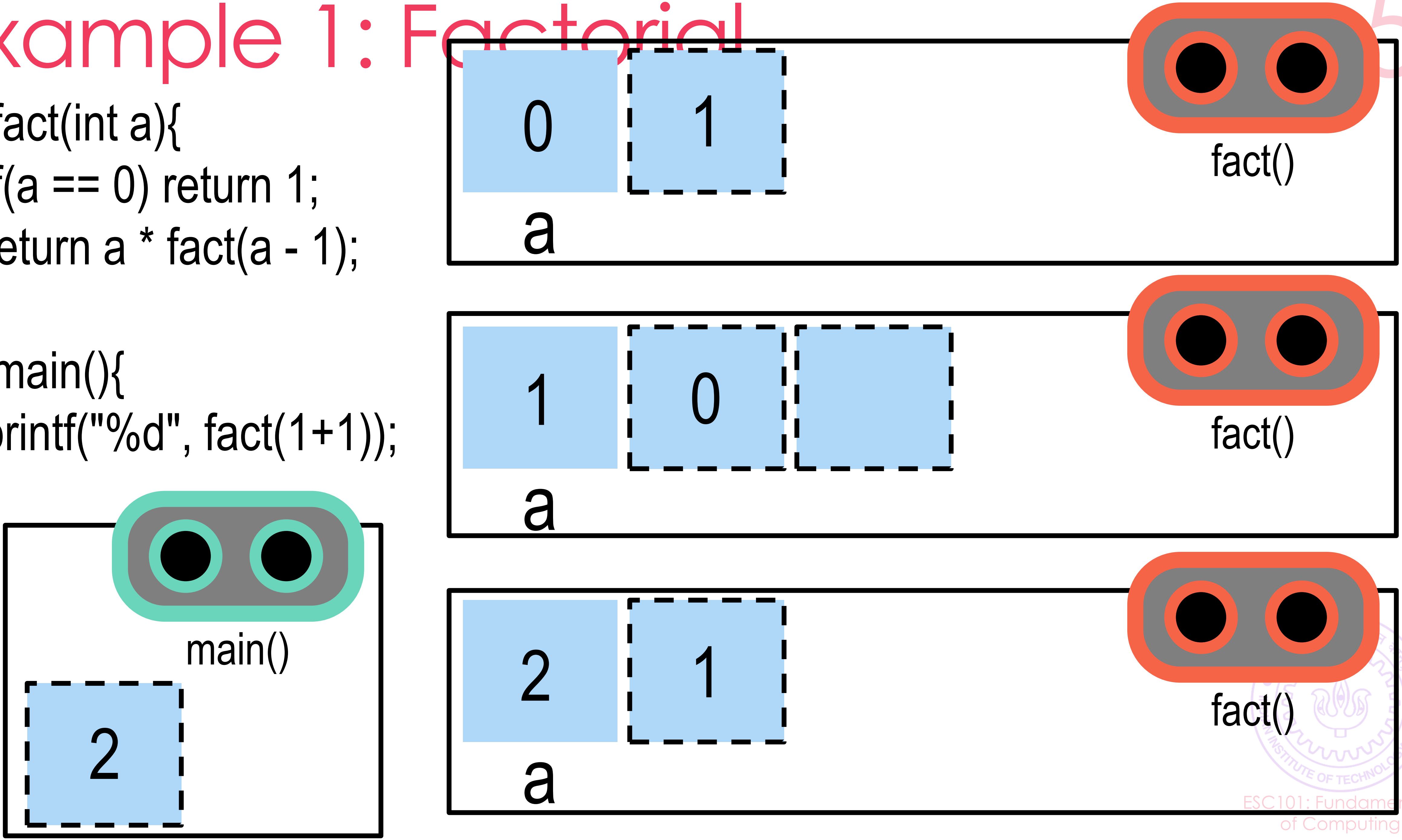
Example 1: Factorial


```
int fact(int a){
    if(a == 0) return 1;
    return a * fact(a - 1);
}
int main(){
    printf("%d", fact(1+1));
}
```



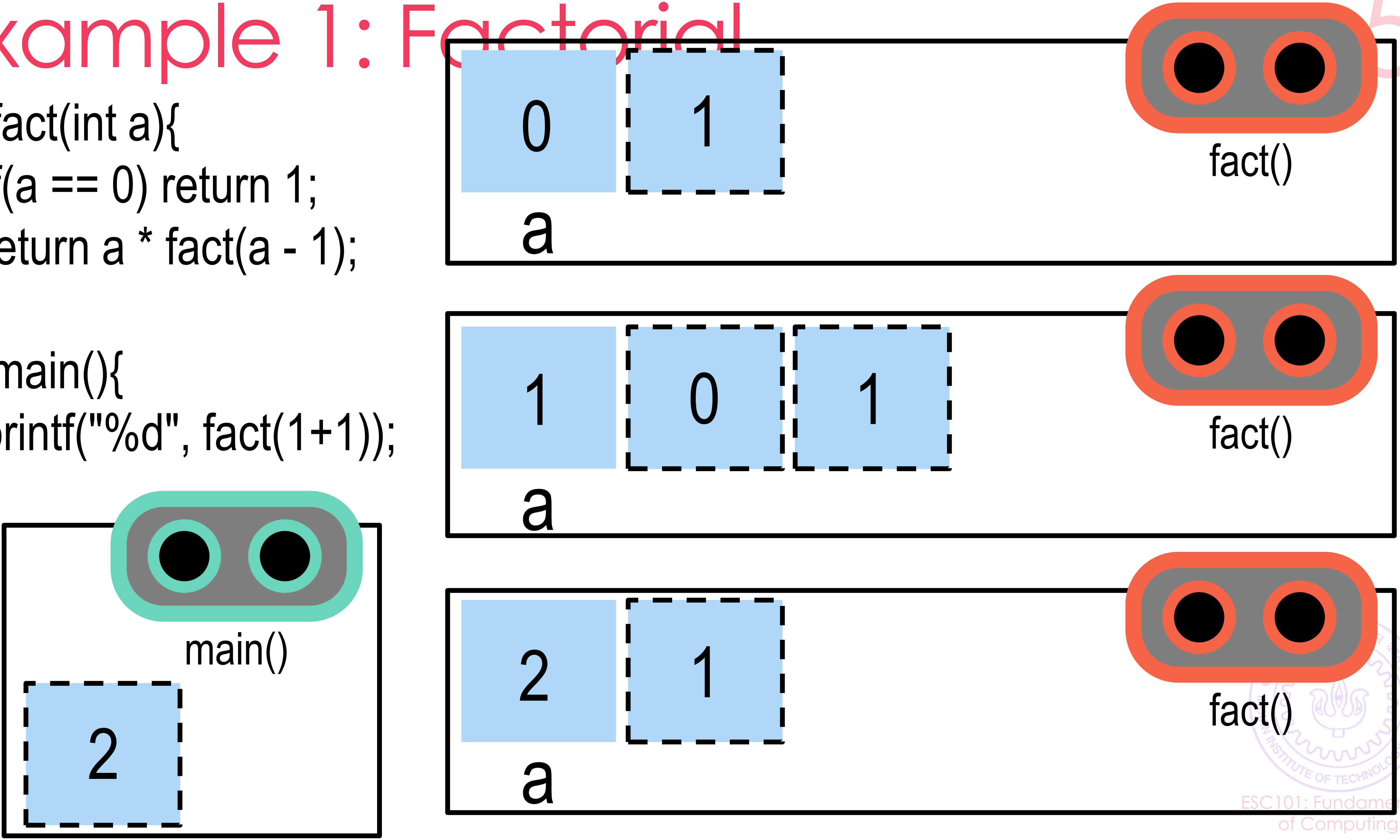
Example 1: Factorial


```
int fact(int a){
    if(a == 0) return 1;
    return a * fact(a - 1);
}
int main(){
    printf("%d", fact(1+1));
}
```



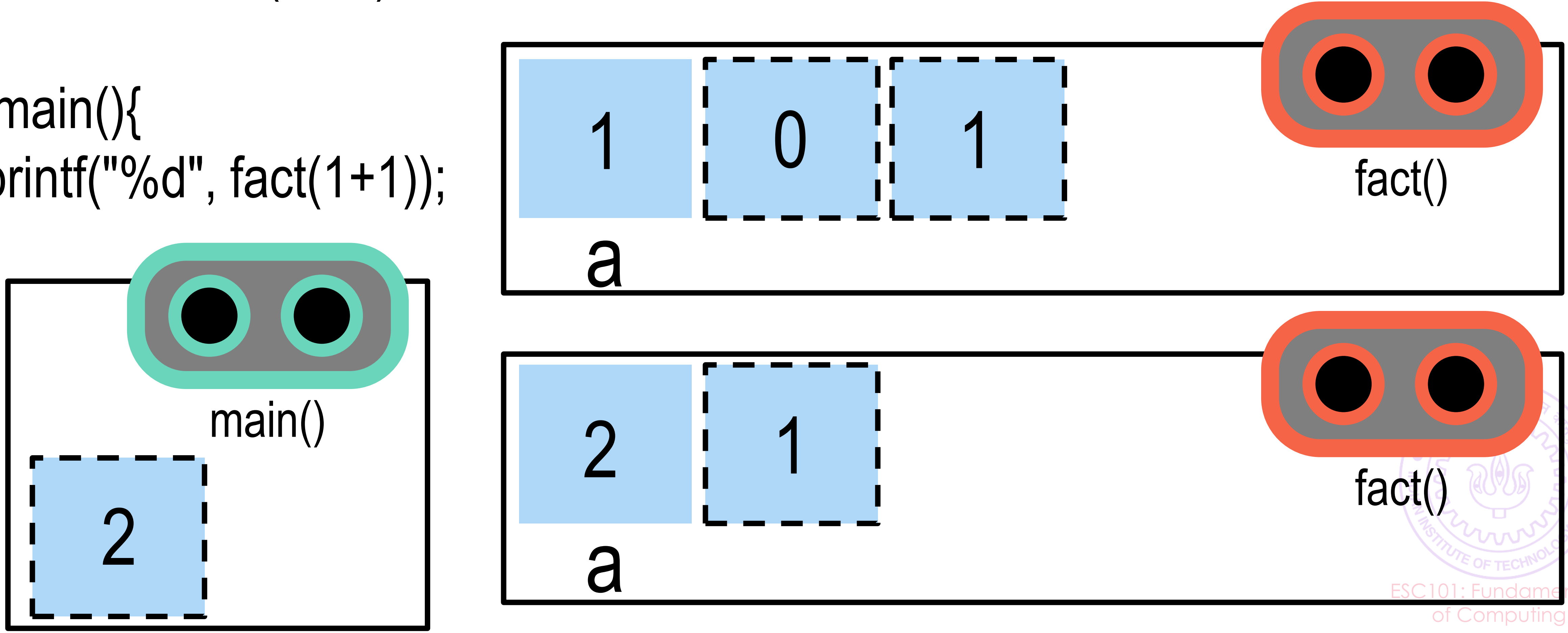
Example 1: Factorial


```
int fact(int a){
    if(a == 0) return 1;
    return a * fact(a - 1);
}
int main(){
    printf("%d", fact(1+1));
}
```



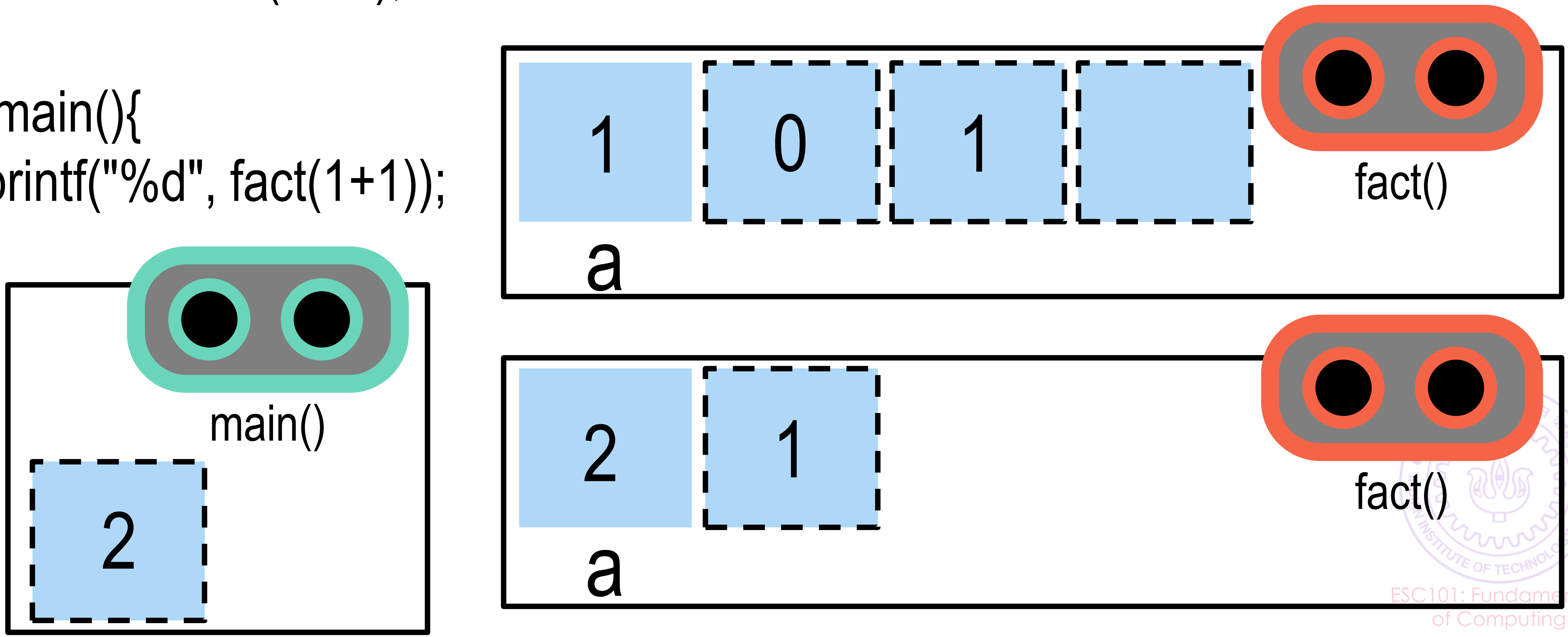
Example 1: Factorial


```
int fact(int a){
    if(a == 0) return 1;
    return a * fact(a - 1);
}
int main(){
    printf("%d", fact(1+1));
}
```



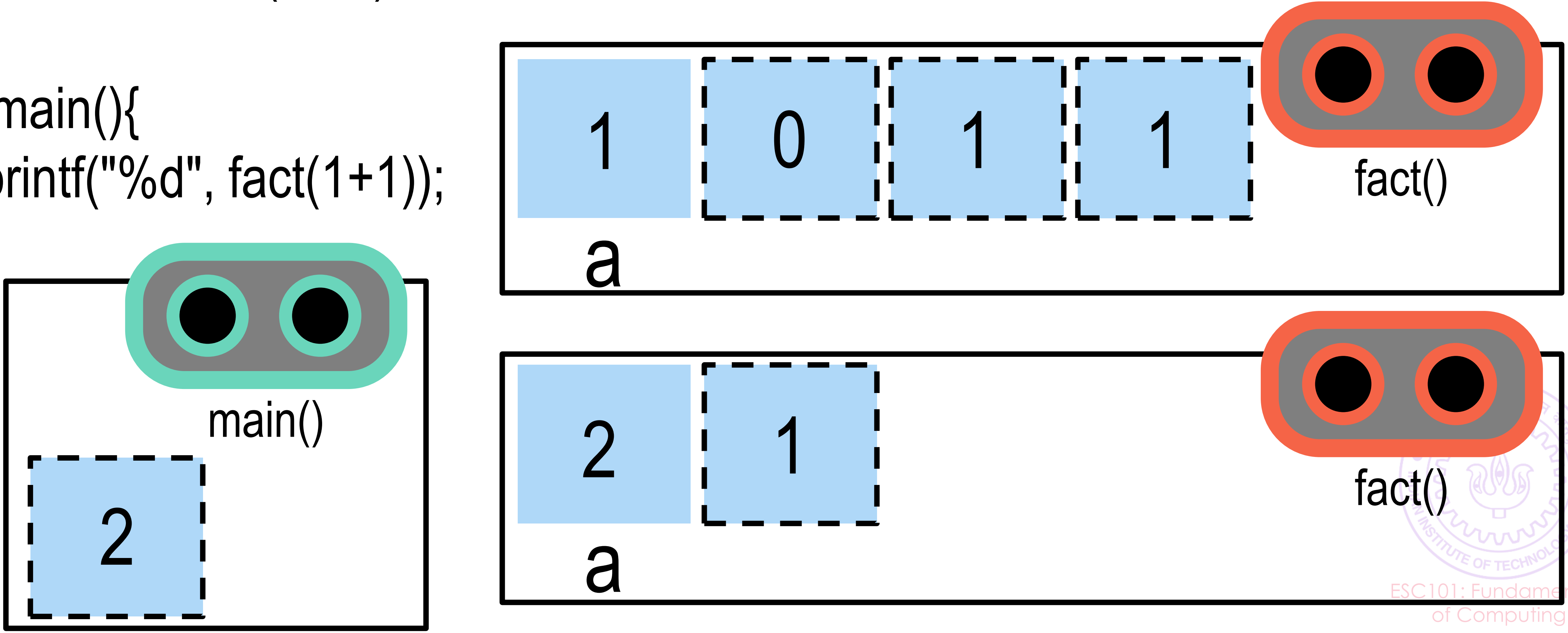
Example 1: Factorial


```
int fact(int a){
    if(a == 0) return 1;
    return a * fact(a - 1);
}
int main(){
    printf("%d", fact(1+1));
}
```



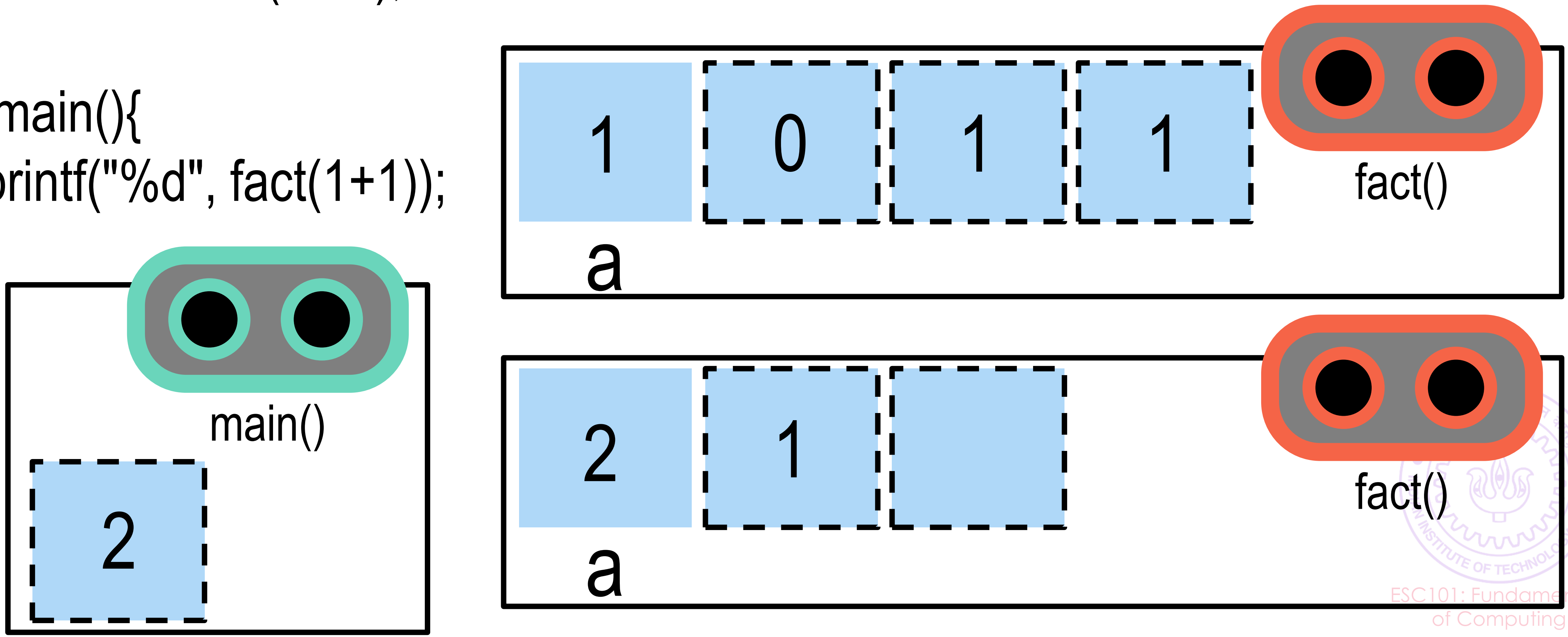
Example 1: Factorial


```
int fact(int a){
    if(a == 0) return 1;
    return a * fact(a - 1);
}
int main(){
    printf("%d", fact(1+1));
}
```



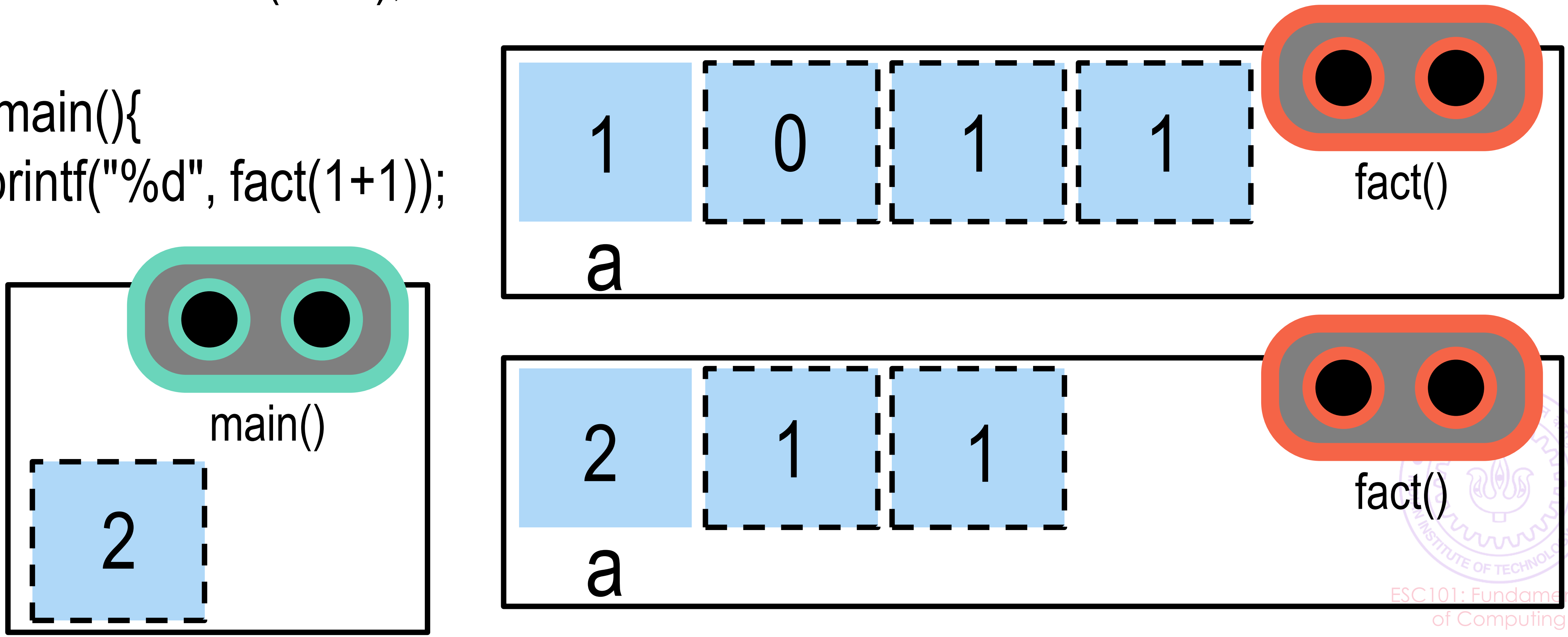
Example 1: Factorial


```
int fact(int a){
    if(a == 0) return 1;
    return a * fact(a - 1);
}
int main(){
    printf("%d", fact(1+1));
}
```



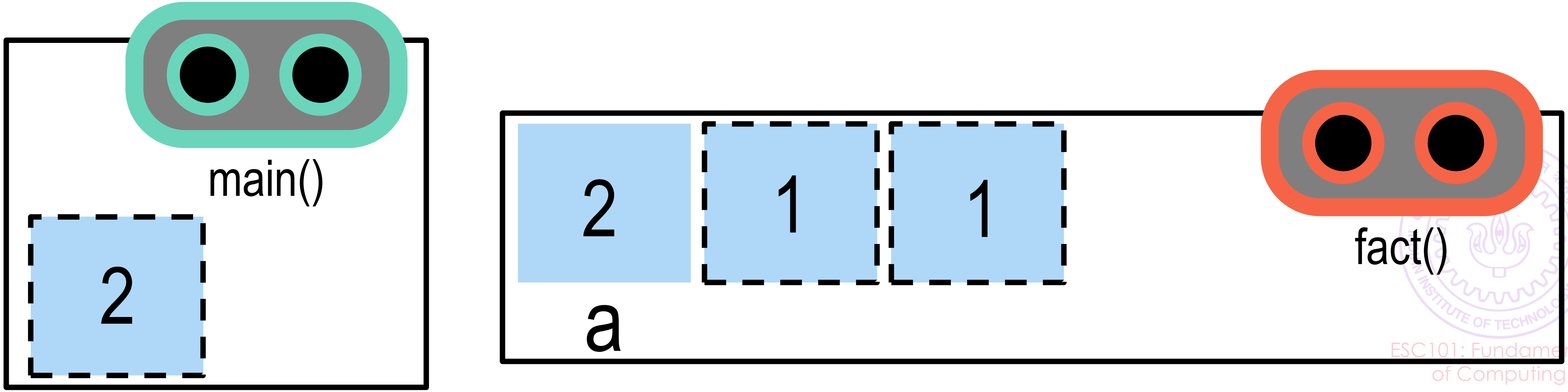
Example 1: Factorial


```
int fact(int a){
    if(a == 0) return 1;
    return a * fact(a - 1);
}
int main(){
    printf("%d", fact(1+1));
}
```



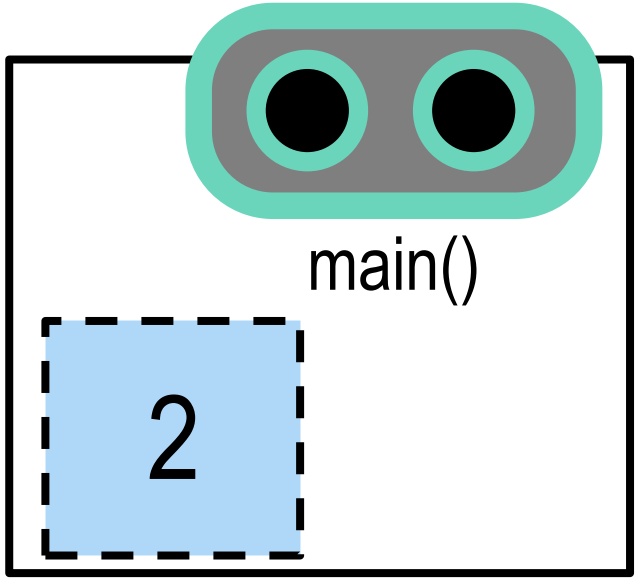
Example 1: Factorial

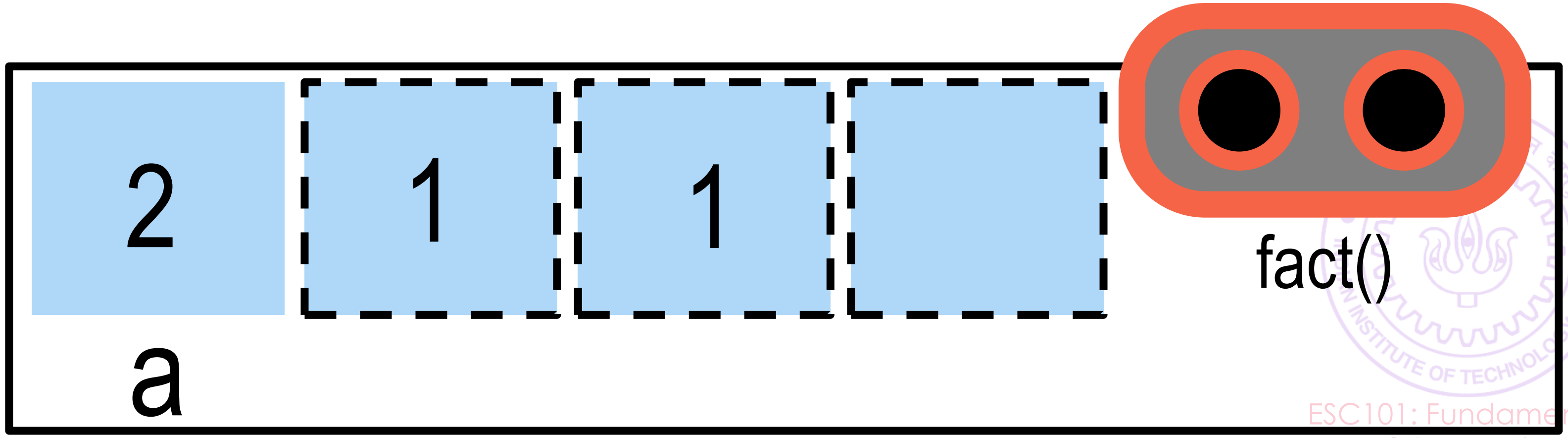

```
int fact(int a){
    if(a == 0) return 1;
    return a * fact(a - 1);
}
int main(){
    printf("%d", fact(1+1));
}
```



Example 1: Factorial

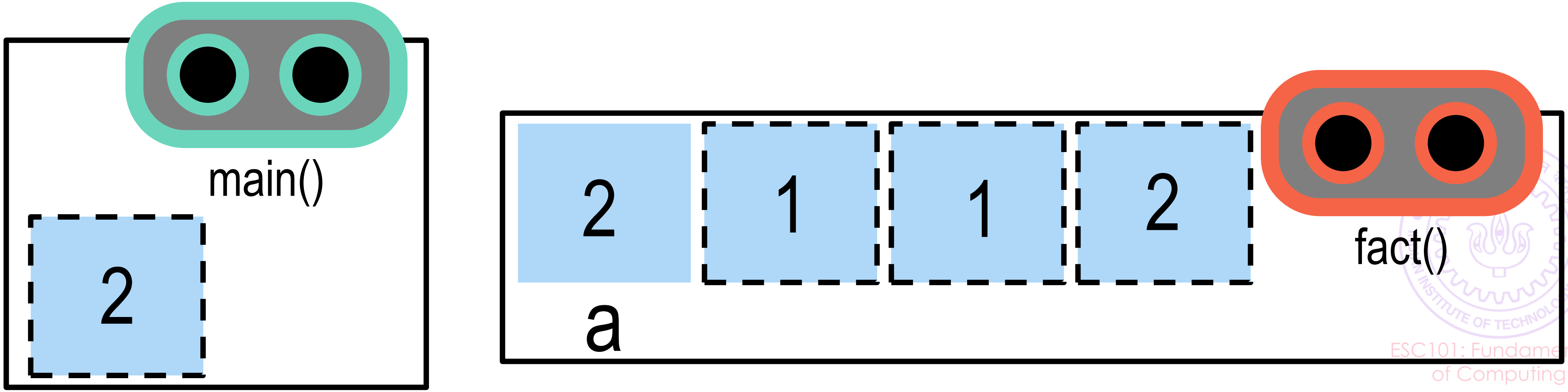

```
int fact(int a){
    if(a == 0) return 1;
    return a * fact(a - 1);
}
int main(){
    printf("%d", fact(1+1));
}
```





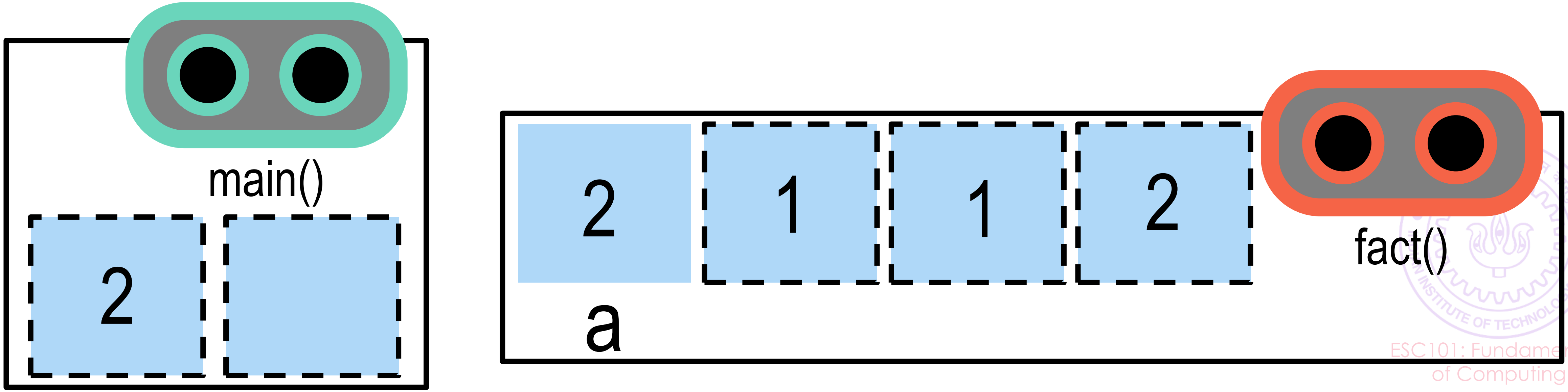
Example 1: Factorial

```
int fact(int a){
    if(a == 0) return 1;
    return a * fact(a - 1);
}
int main(){
    printf("%d", fact(1+1));
}
```



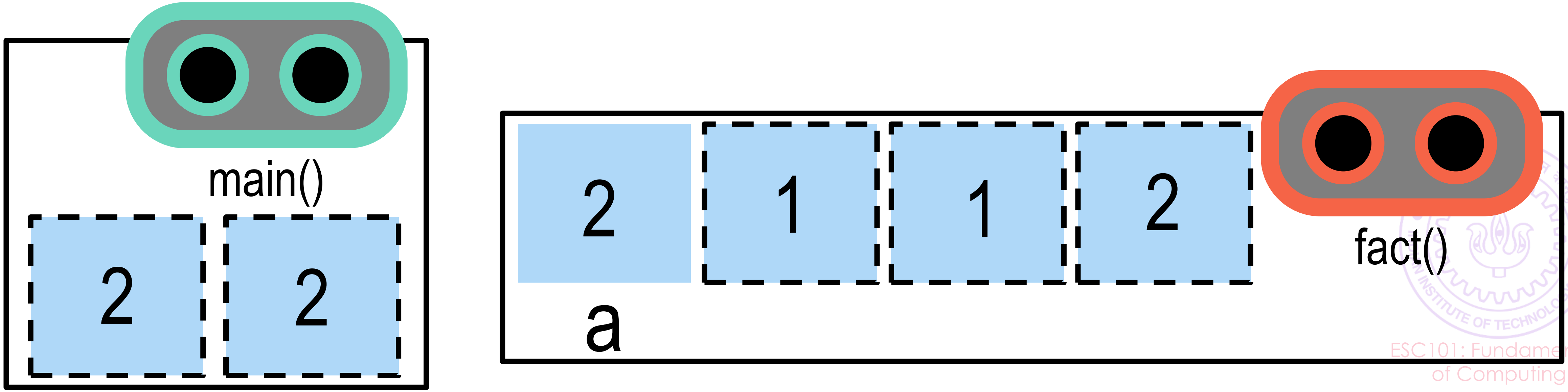
Example 1: Factorial


```
int fact(int a){
    if(a == 0) return 1;
    return a * fact(a - 1);
}
int main(){
    printf("%d", fact(1+1));
}
```



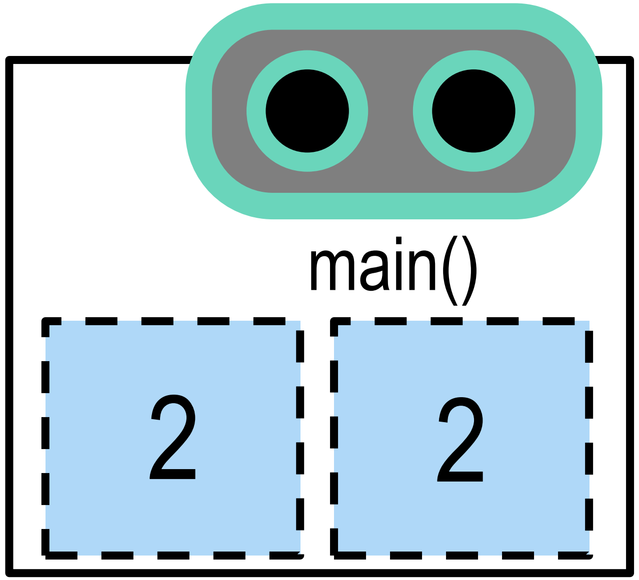
Example 1: Factorial


```
int fact(int a){
    if(a == 0) return 1;
    return a * fact(a - 1);
}
int main(){
    printf("%d", fact(1+1));
}
```



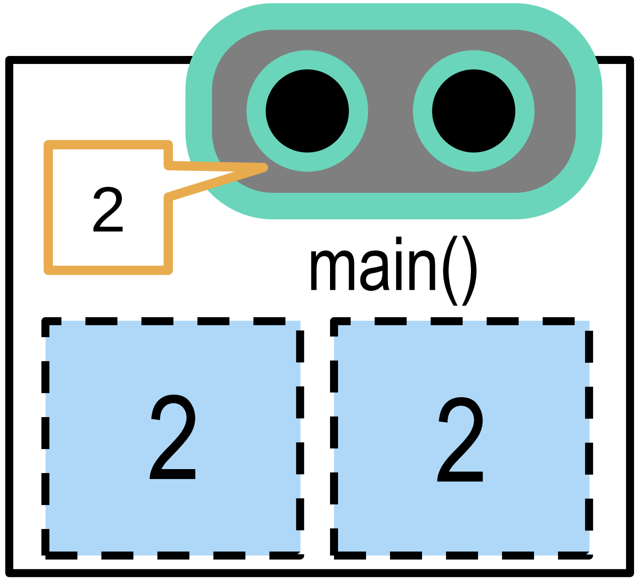
Example 1: Factorial


```
int fact(int a){
    if(a == 0) return 1;
    return a * fact(a - 1);
}
int main(){
    printf("%d", fact(1+1));
}
```



Example 1: Factorial


```
int fact(int a){
    if(a == 0) return 1;
    return a * fact(a - 1);
}
int main(){
    printf("%d", fact(1+1));
}
```



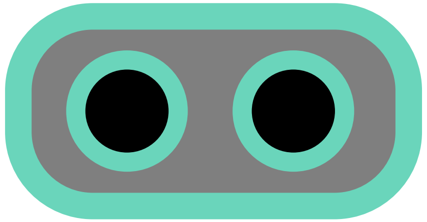
Example 1: Factorial

Example 1: Factorial

```
int fact(int a){
    if(a == 0) return 1;
    return a * fact(a - 1);
}
int main(){
    printf("%d", fact(2*3));
}
```

Example 1: Factorial

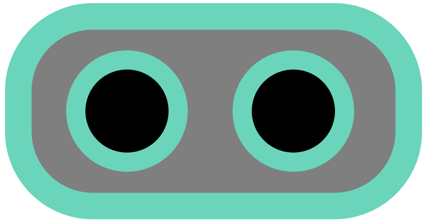
```
int fact(int a){
    if(a == 0) return 1;
    return a * fact(a - 1);
}
int main(){
    printf("%d", fact(2*3));
}
```



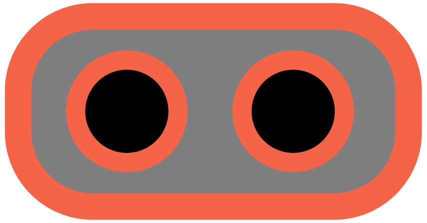
main()

Example 1: Factorial

```
int fact(int a){
    if(a == 0) return 1;
    return a * fact(a - 1);
}
int main(){
    printf("%d", fact(2*3));
}
```



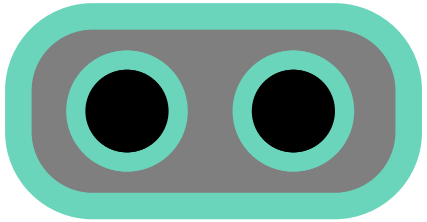
main()



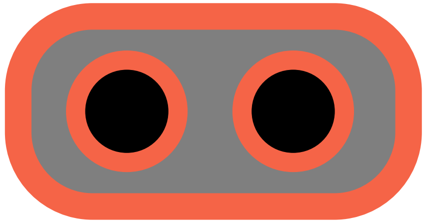
fact(6)

Example 1: Factorial

```
int fact(int a){
    if(a == 0) return 1;
    return a * fact(a - 1);
}
int main(){
    printf("%d", fact(2*3));
}
```



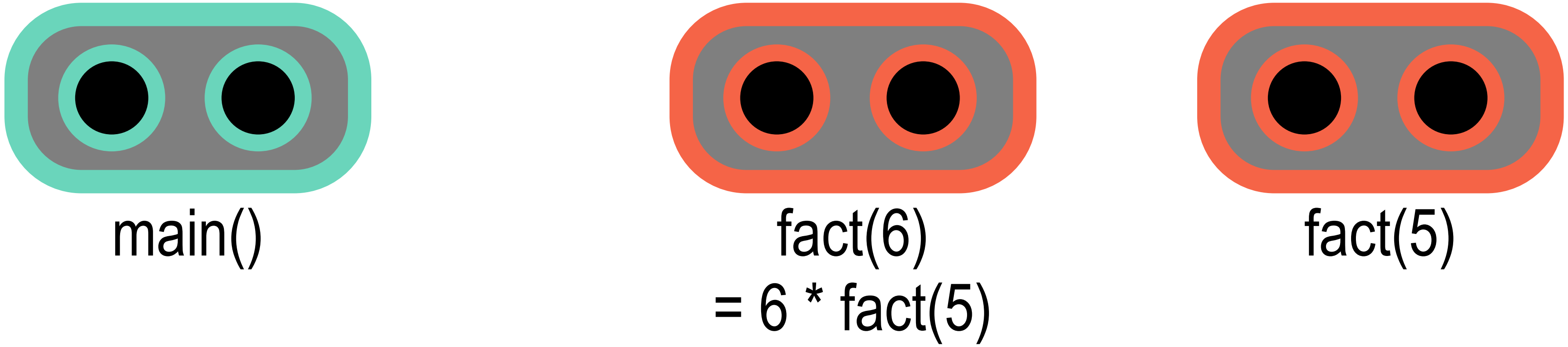
main()



fact(6)
= 6 * fact(5)

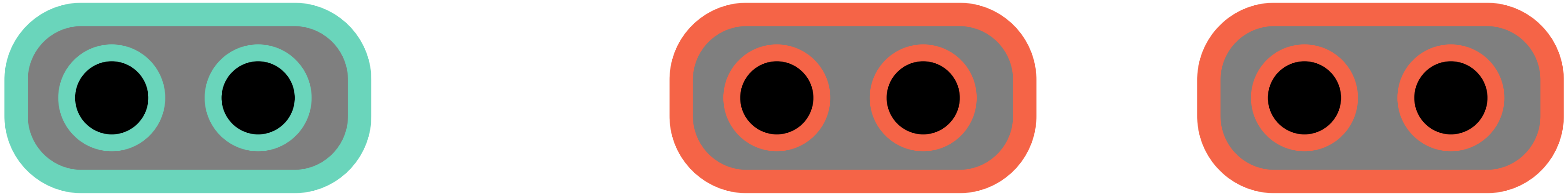
Example 1: Factorial

```
int fact(int a){
    if(a == 0) return 1;
    return a * fact(a - 1);
}
int main(){
    printf("%d", fact(2*3));
}
```



Example 1: Factorial

```
int fact(int a){
    if(a == 0) return 1;
    return a * fact(a - 1);
}
int main(){
    printf("%d", fact(2*3));
}
```



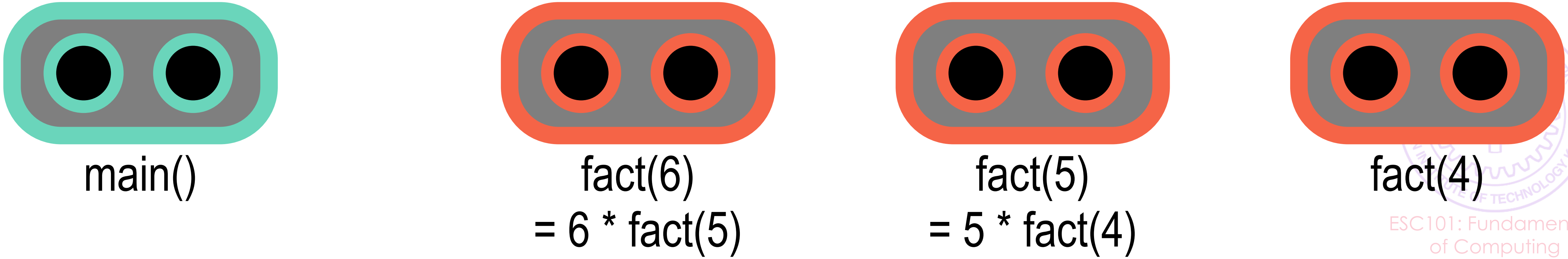
main()

fact(6)
= 6 * fact(5)

fact(5)
= 5 * fact(4)

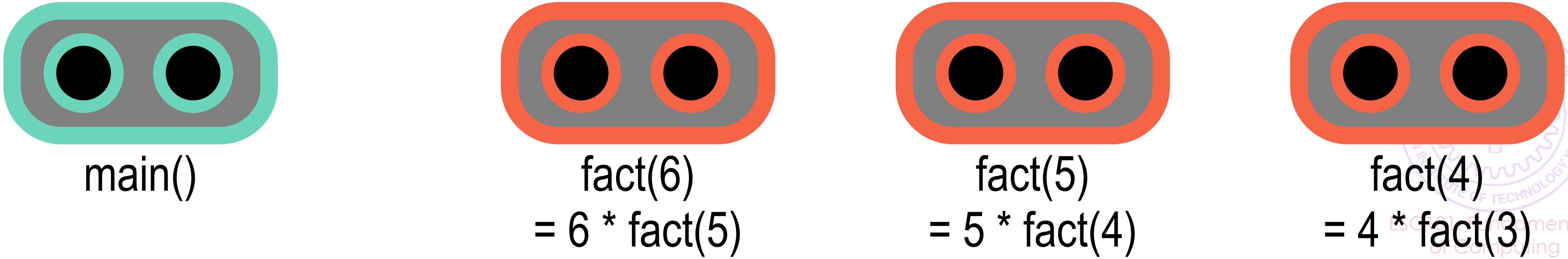
Example 1: Factorial

```
int fact(int a){
    if(a == 0) return 1;
    return a * fact(a - 1);
}
int main(){
    printf("%d", fact(2*3));
}
```



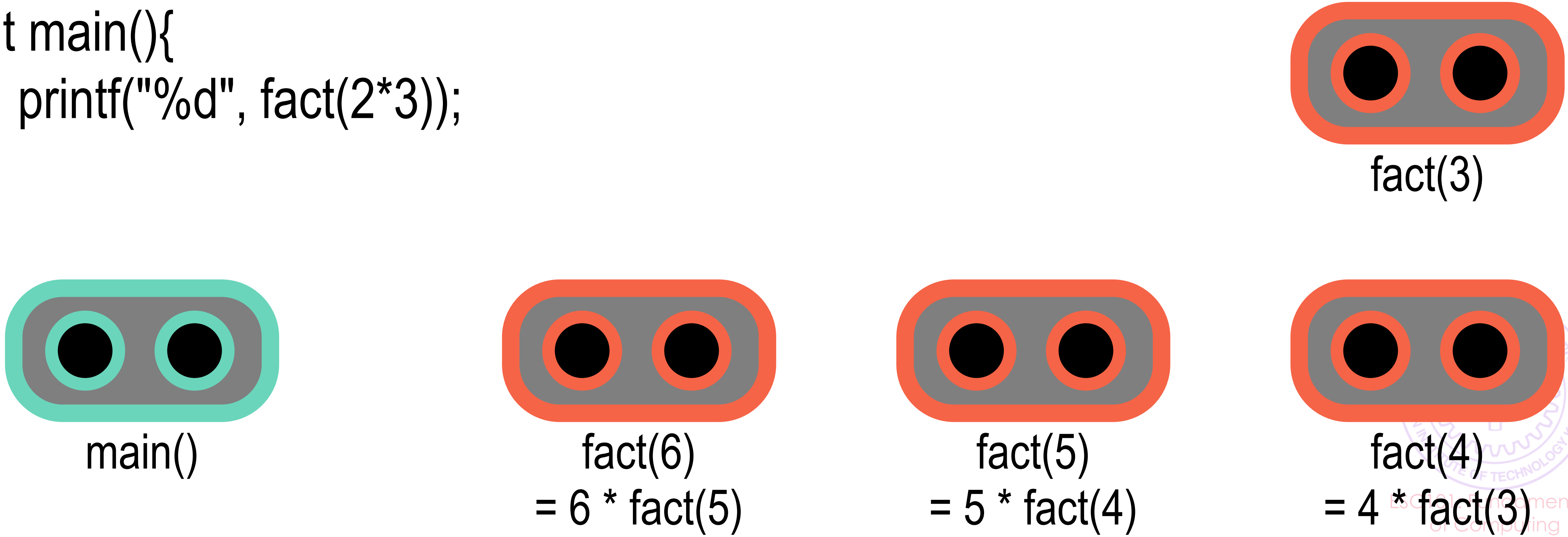
Example 1: Factorial

```
int fact(int a){
    if(a == 0) return 1;
    return a * fact(a - 1);
}
int main(){
    printf("%d", fact(2*3));
}
```



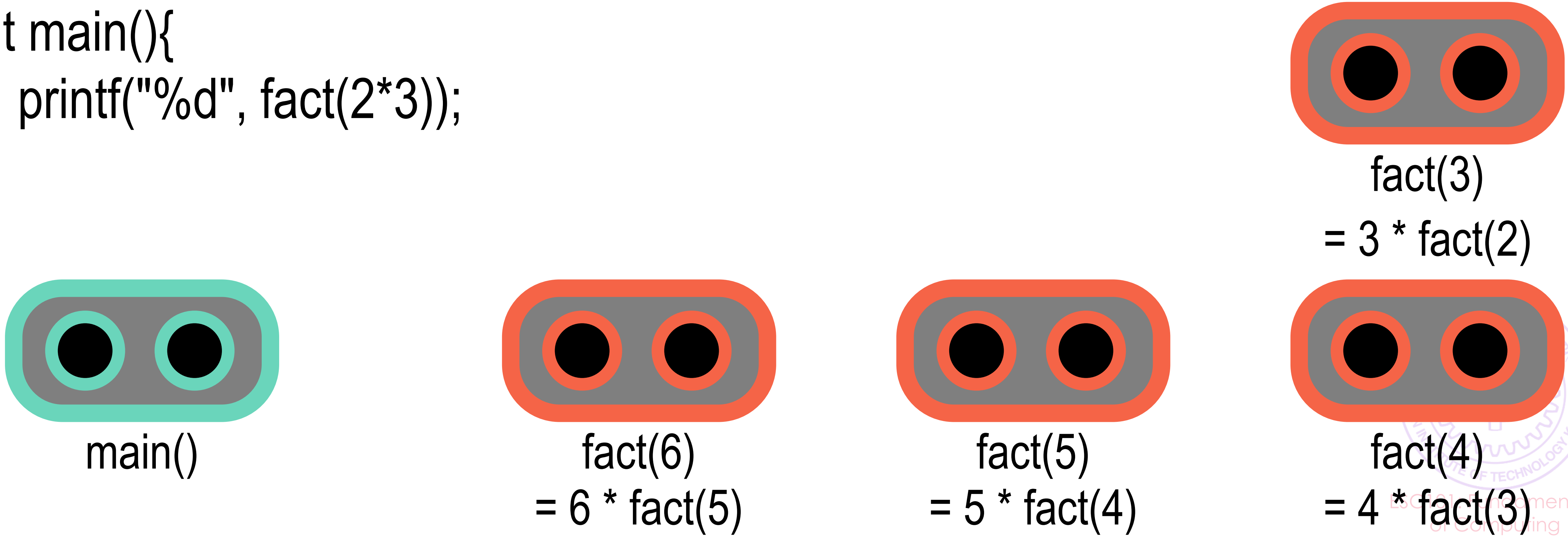
Example 1: Factorial

```
int fact(int a){
    if(a == 0) return 1;
    return a * fact(a - 1);
}
int main(){
    printf("%d", fact(2*3));
}
```



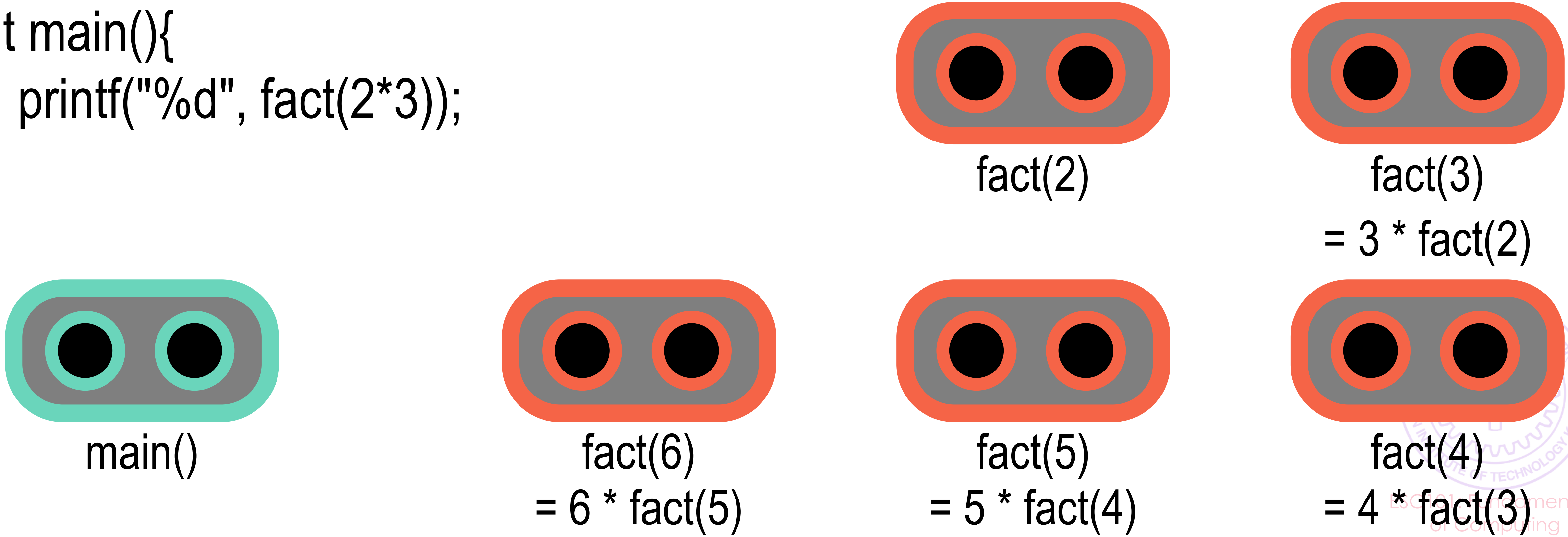
Example 1: Factorial

```
int fact(int a){
    if(a == 0) return 1;
    return a * fact(a - 1);
}
int main(){
    printf("%d", fact(2*3));
}
```



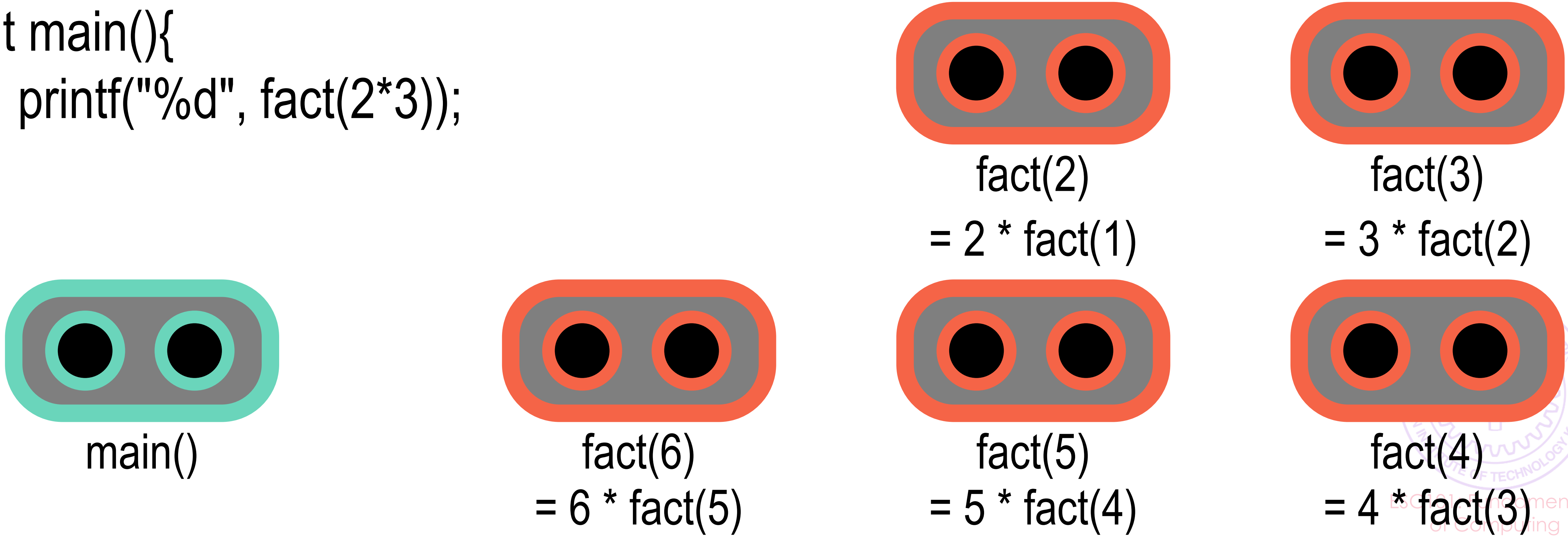
Example 1: Factorial

```
int fact(int a){
    if(a == 0) return 1;
    return a * fact(a - 1);
}
int main(){
    printf("%d", fact(2*3));
}
```



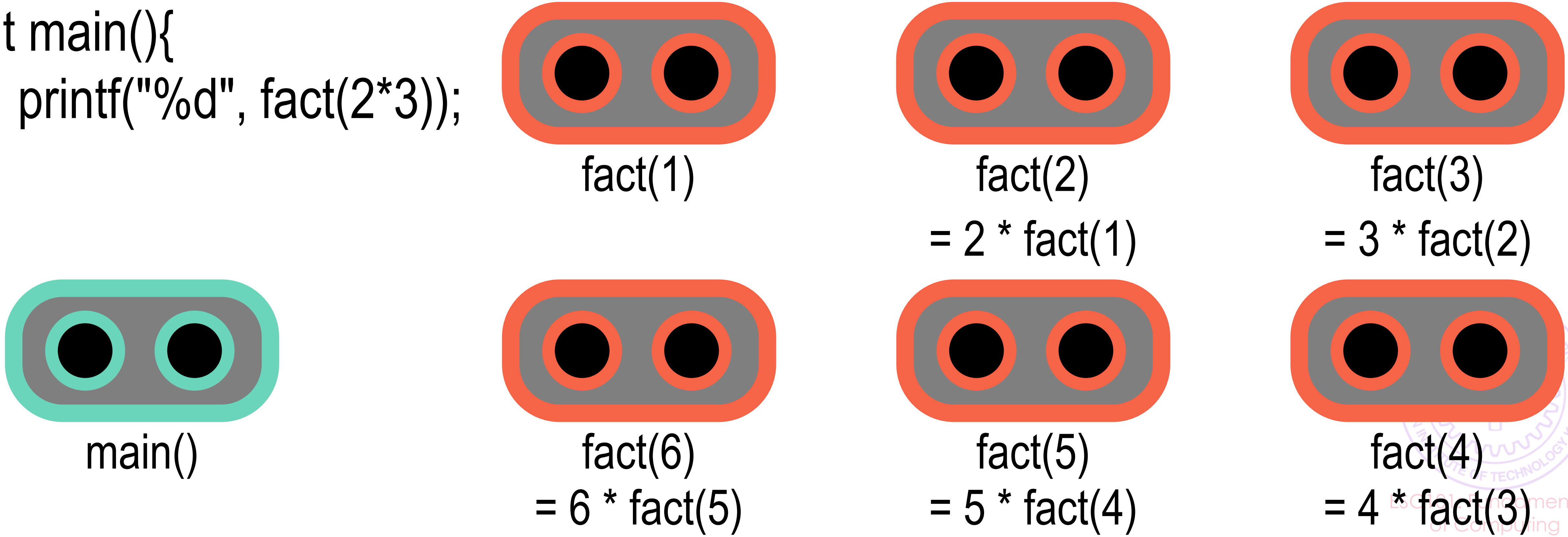
Example 1: Factorial

```
int fact(int a){
    if(a == 0) return 1;
    return a * fact(a - 1);
}
int main(){
    printf("%d", fact(2*3));
}
```



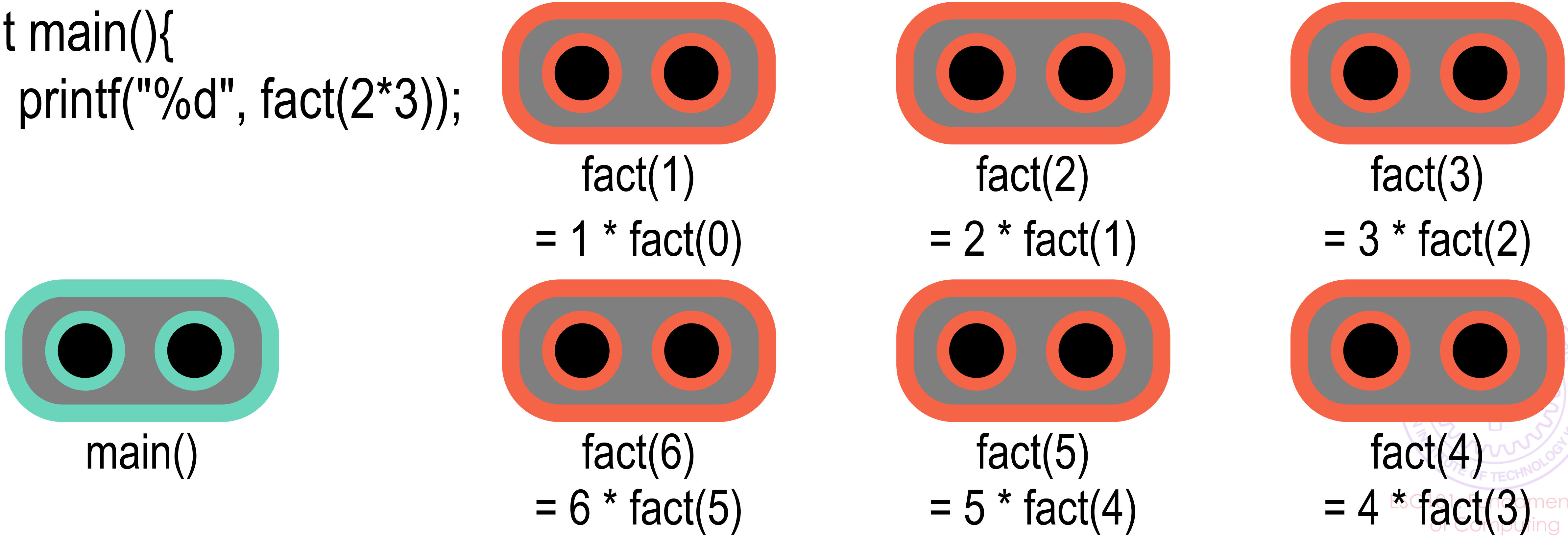
Example 1: Factorial

```
int fact(int a){
    if(a == 0) return 1;
    return a * fact(a - 1);
}
int main(){
    printf("%d", fact(2*3));
}
```



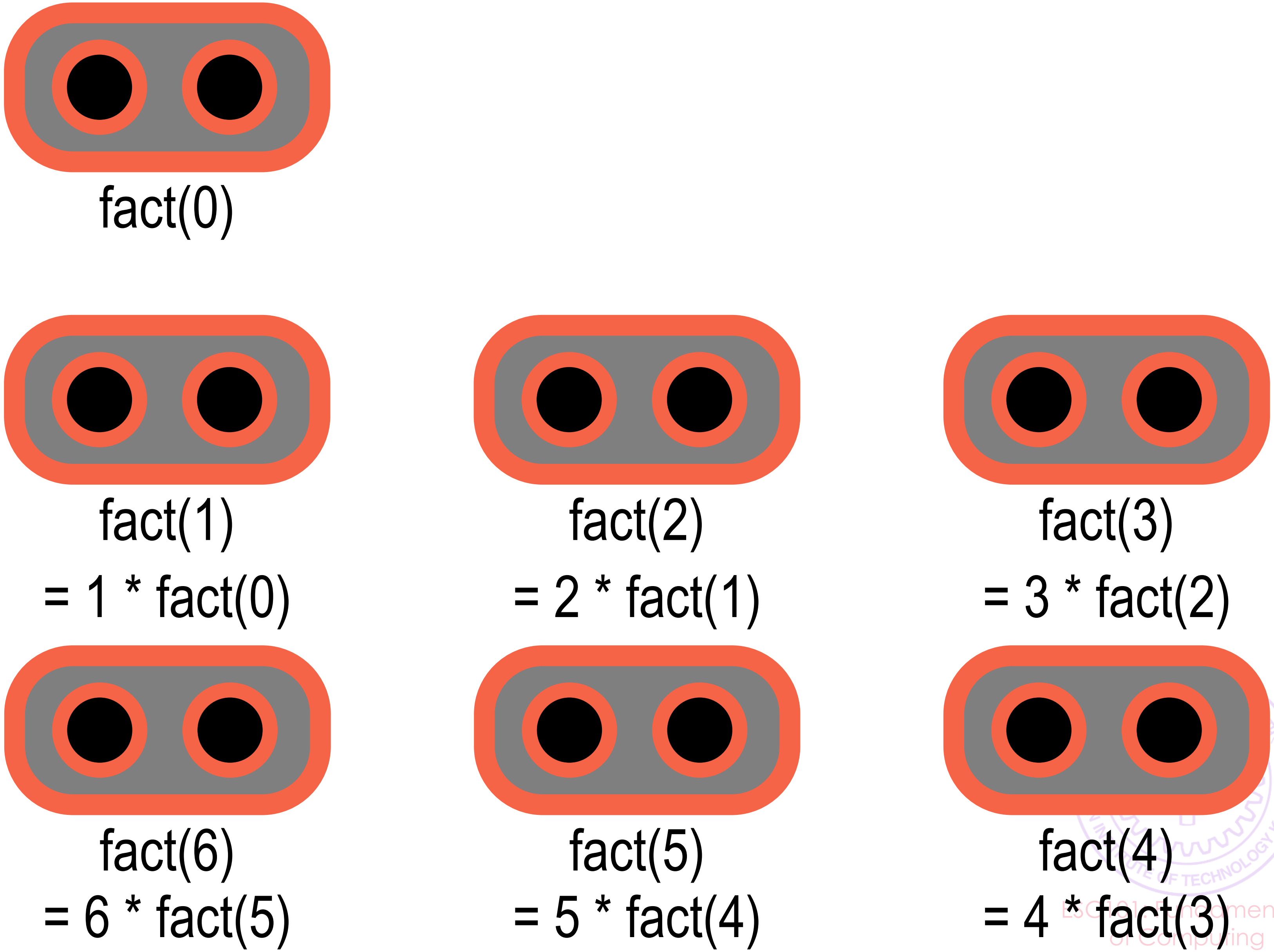
Example 1: Factorial

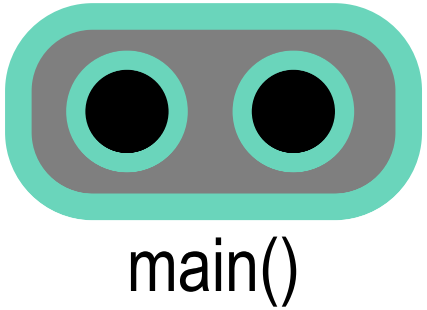
```
int fact(int a){
    if(a == 0) return 1;
    return a * fact(a - 1);
}
int main(){
    printf("%d", fact(2*3));
}
```



Example 1: Factorial

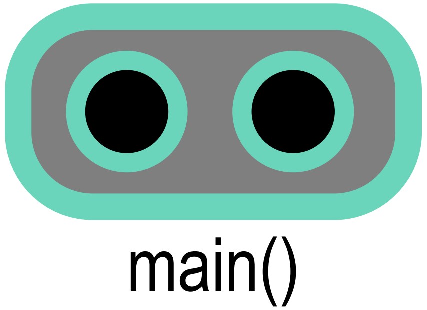
```
int fact(int a){
    if(a == 0) return 1;
    return a * fact(a - 1);
}
int main(){
    printf("%d", fact(2*3));
}
```

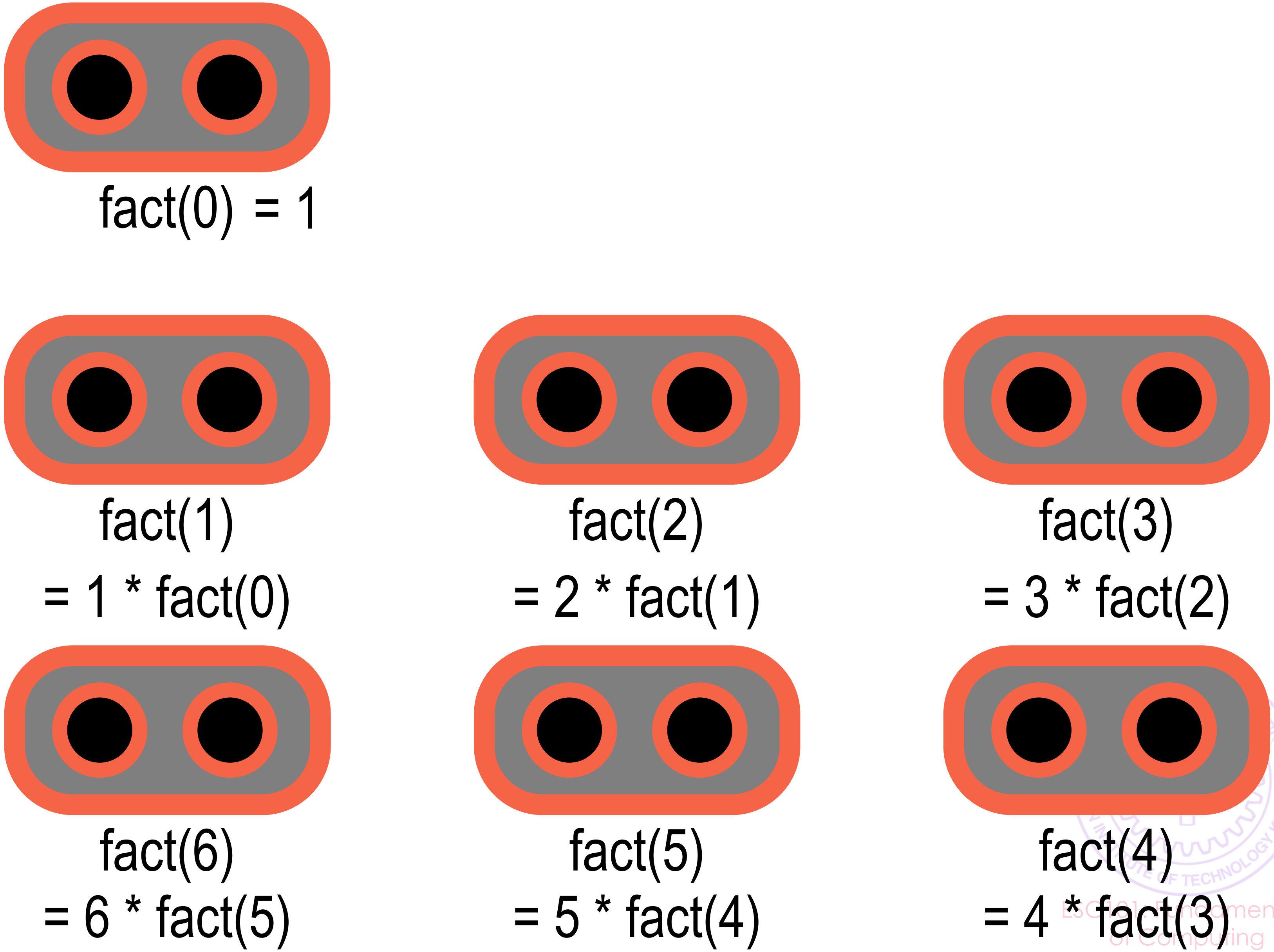




Example 1: Factorial

```
int fact(int a){
    if(a == 0) return 1;
    return a * fact(a - 1);
}
int main(){
    printf("%d", fact(2*3));
}
```





Example 1: Factorial

```
int fact(int a){
    if(a == 0) return 1;
    return a * fact(a - 1);
}
int main(){
    printf("%d", fact(2*3));
}
```

fact(0) = 1

fact(1)
= 1 * fact(0)

fact(2)
= 2 * fact(1)

fact(3)
= 3 * fact(2)

main()

fact(6)
= 6 * fact(5)

fact(5)
= 5 * fact(4)

fact(4)
= 4 * fact(3)

Example 1: Factorial

See how many clones got created!

```
int fact(int a){
    if(a == 0) return 1;
    return a * fact(a - 1);
}
int main(){
    printf("%d", fact(2*3));
}
```

fact(0) = 1

fact(1)
= 1 * fact(0)

fact(2)
= 2 * fact(1)

fact(3)
= 3 * fact(2)

main()

fact(6)
= 6 * fact(5)

fact(5)
= 5 * fact(4)

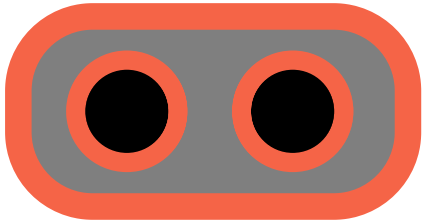
fact(4)
= 4 * fact(3)

Example 1: Factorial

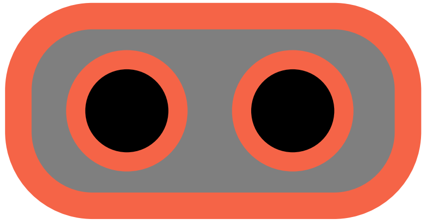
See how many clones got created!

Creating each clone takes time and memory

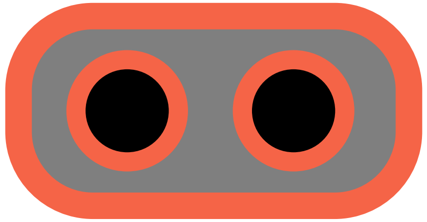
```
int fact(int a){
    if(a == 0) return 1;
    return a * fact(a - 1);
}
int main(){
    printf("%d", fact(2*3));
}
```



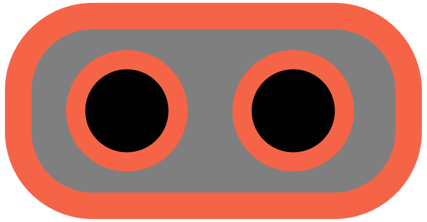
fact(0) = 1



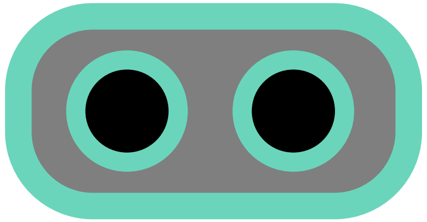
fact(1)
= 1 * fact(0)



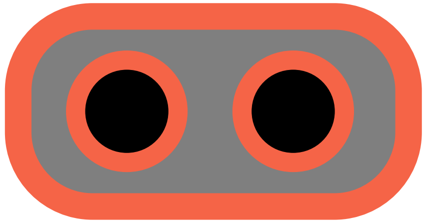
fact(2)
= 2 * fact(1)



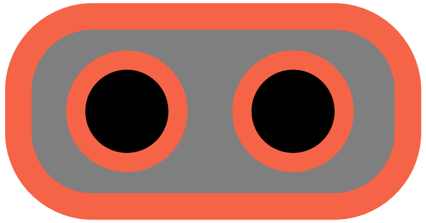
fact(3)
= 3 * fact(2)



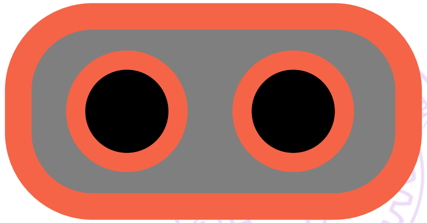
main()



fact(6)
= 6 * fact(5)



fact(5)
= 5 * fact(4)



fact(4)
= 4 * fact(3)

Example 1: Factorial

```
int fact(int a){
    if(a == 0) return 1;
    return a * fact(a - 1);
}
int main(){
    printf("%d", fact(2*3));
}
```

See how many clones got created!

Creating each clone takes time and memory

This is why recursive programs can be a bit slower – be careful

fact(0) = 1

fact(1) = 1 * fact(0)

fact(2) = 2 * fact(1)

fact(3) = 3 * fact(2)

main()

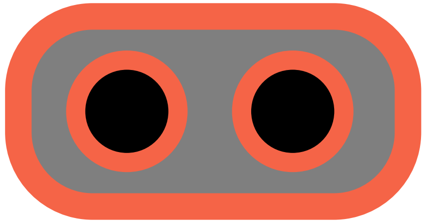
fact(6) = 6 * fact(5)

fact(5) = 5 * fact(4)

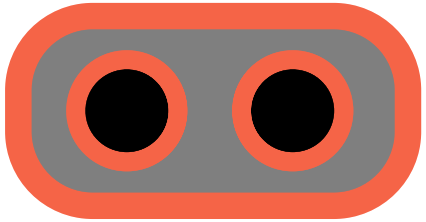
fact(4) = 4 * fact(3)

Example 1: Factorial

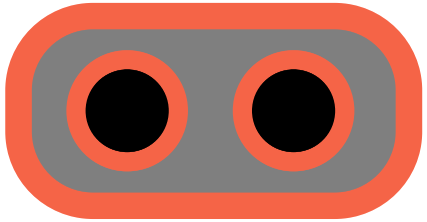
```
int fact(int a){
    if(a == 0) return 1;
    return a * fact(a - 1);
}
int main(){
    printf("%d", fact(2*3));
}
```



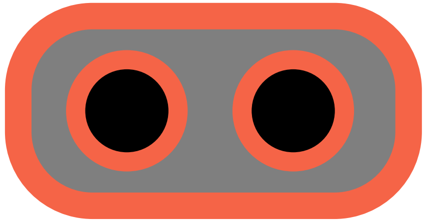
fact(0) = 1



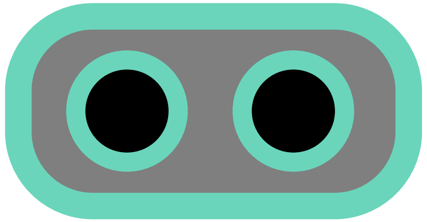
fact(1)
= 1 * 1 = 1



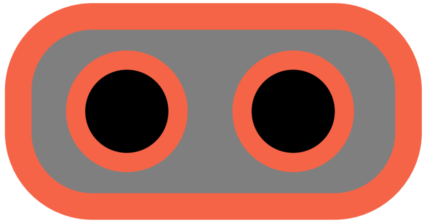
fact(2)
= 2 * fact(1)



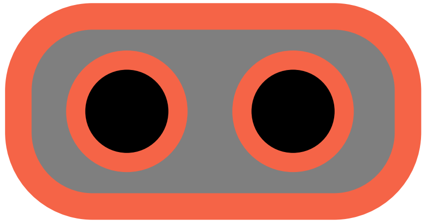
fact(3)
= 3 * fact(2)



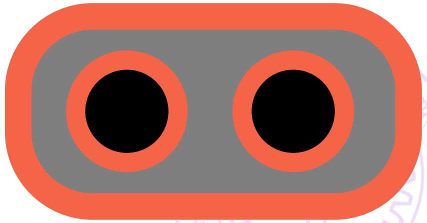
main()



fact(6)
= 6 * fact(5)



fact(5)
= 5 * fact(4)



fact(4)
= 4 * fact(3)

See how many clones got created!

Creating each clone takes time and memory

This is why recursive programs can be a bit slower – be careful

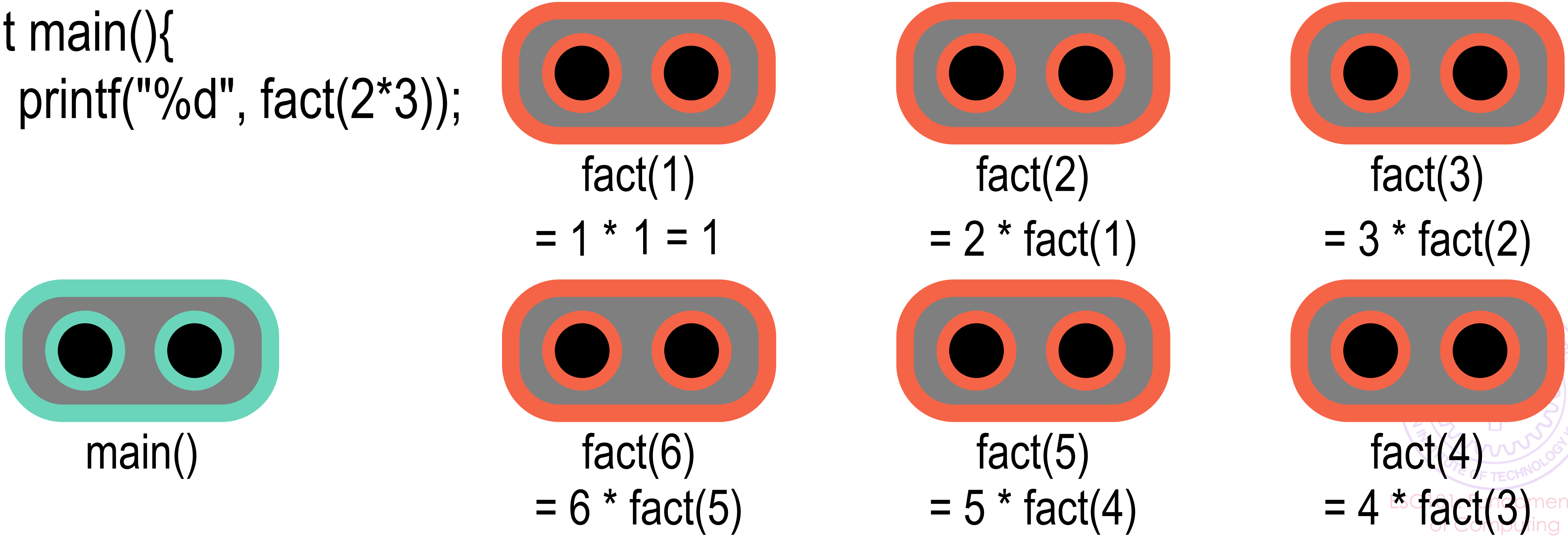
Example 1: Factorial

```
int fact(int a){
    if(a == 0) return 1;
    return a * fact(a - 1);
}
int main(){
    printf("%d", fact(2*3));
}
```

See how many clones got created!

Creating each clone takes time and memory

This is why recursive programs can be a bit slower – be careful



Example 1: Factorial

```
int fact(int a){
    if(a == 0) return 1;
    return a * fact(a - 1);
}
int main(){
    printf("%d", fact(2*3));
}
```

See how many clones got created!

Creating each clone takes time and memory

This is why recursive programs can be a bit slower – be careful

fact(1) = 1 * 1 = 1

fact(2) = 2 * 1 = 2

fact(3) = 3 * fact(2)

main()

fact(6) = 6 * fact(5)

fact(5) = 5 * fact(4)

fact(4) = 4 * fact(3)

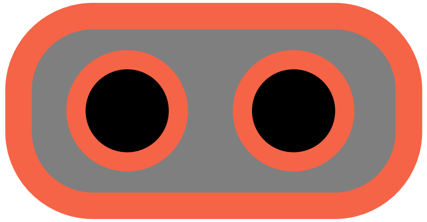
Example 1: Factorial

```
int fact(int a){
    if(a == 0) return 1;
    return a * fact(a - 1);
}
int main(){
    printf("%d", fact(2*3));
}
```

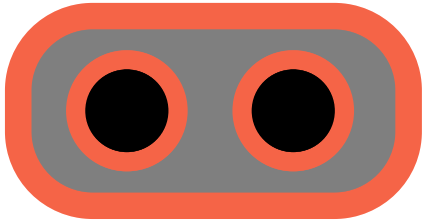
See how many clones got created!

Creating each clone takes time and memory

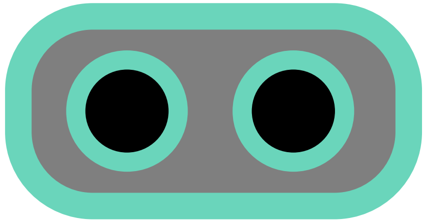
This is why recursive programs can be a bit slower – be careful



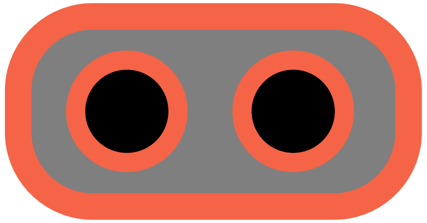
fact(2)
= 2 * 1 = 2



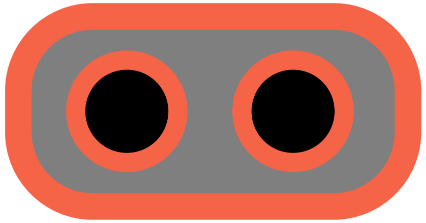
fact(3)
= 3 * fact(2)



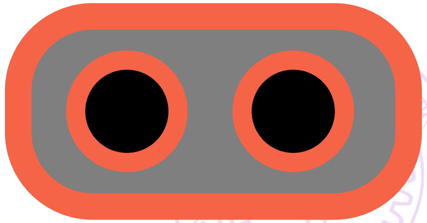
main()



fact(6)
= 6 * fact(5)



fact(5)
= 5 * fact(4)



fact(4)
= 4 * fact(3)

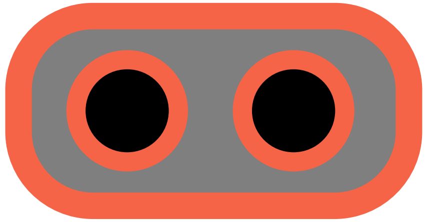
Example 1: Factorial

```
int fact(int a){
    if(a == 0) return 1;
    return a * fact(a - 1);
}
int main(){
    printf("%d", fact(2*3));
}
```

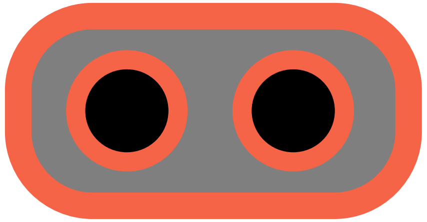
See how many clones got created!

Creating each clone takes time and memory

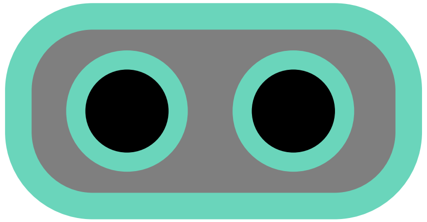
This is why recursive programs can be a bit slower – be careful



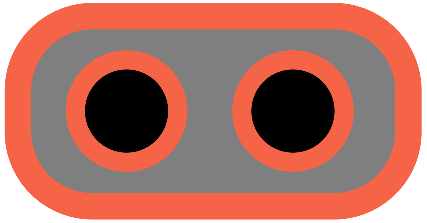
fact(2)
= 2 * 1 = 2



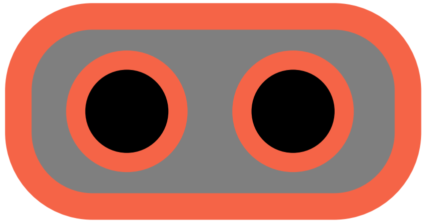
fact(3)
= 3 * 2 = 6



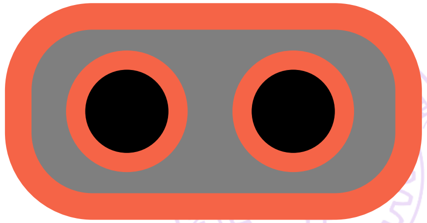
main()



fact(6)
= 6 * fact(5)



fact(5)
= 5 * fact(4)



fact(4)
= 4 * fact(3)

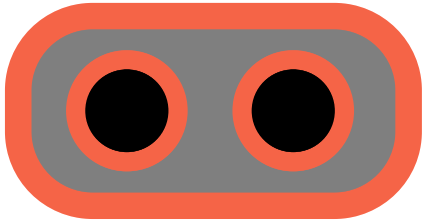
Example 1: Factorial

```
int fact(int a){
    if(a == 0) return 1;
    return a * fact(a - 1);
}
int main(){
    printf("%d", fact(2*3));
}
```

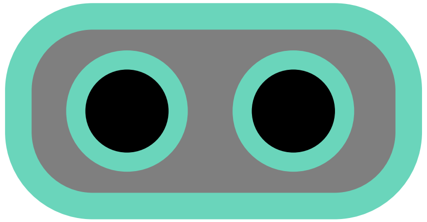
See how many clones got created!

Creating each clone takes time and memory

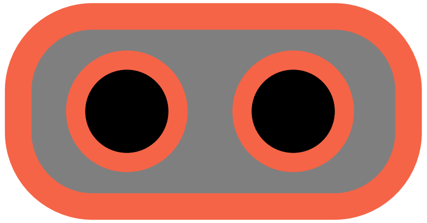
This is why recursive programs can be a bit slower – be careful



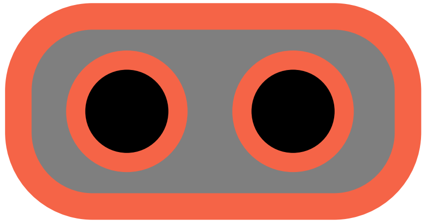
fact(3)
= 3 * 2 = 6



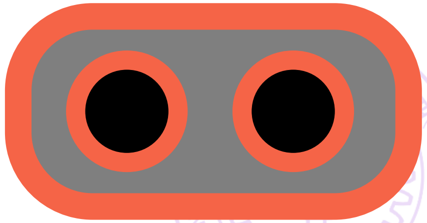
main()



fact(6)
= 6 * fact(5)



fact(5)
= 5 * fact(4)



fact(4)
= 4 * fact(3)

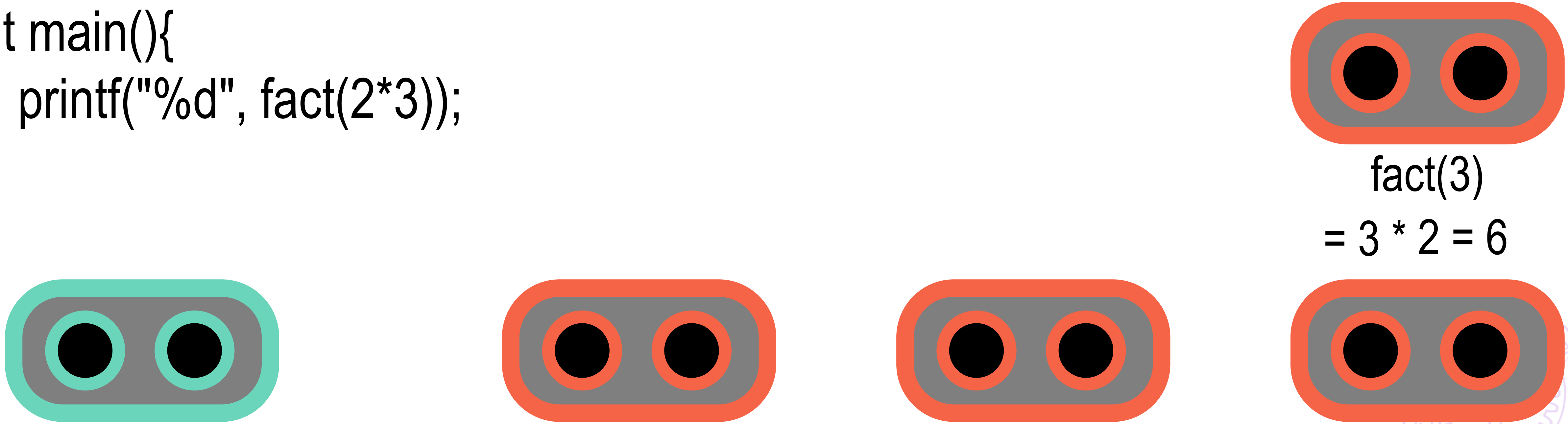
Example 1: Factorial

```
int fact(int a){
    if(a == 0) return 1;
    return a * fact(a - 1);
}
int main(){
    printf("%d", fact(2*3));
}
```

See how many clones got created!

Creating each clone takes time and memory

This is why recursive programs can be a bit slower – be careful



fact(3)
= 3 * 2 = 6

main()

fact(6)
= 6 * fact(5)

fact(5)
= 5 * fact(4)

fact(4)
= 4 * 6 = 24

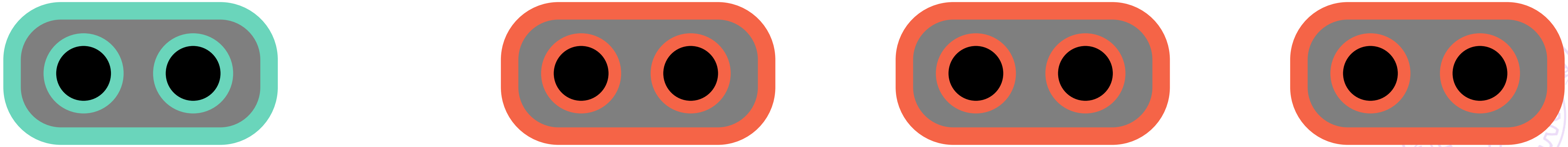
Example 1: Factorial

```
int fact(int a){
    if(a == 0) return 1;
    return a * fact(a - 1);
}
int main(){
    printf("%d", fact(2*3));
}
```

See how many clones got created!

Creating each clone takes time and memory

This is why recursive programs can be a bit slower – be careful



main()

fact(6)
= 6 * fact(5)

fact(5)
= 5 * fact(4)

fact(4)
= 4 * 6 = 24

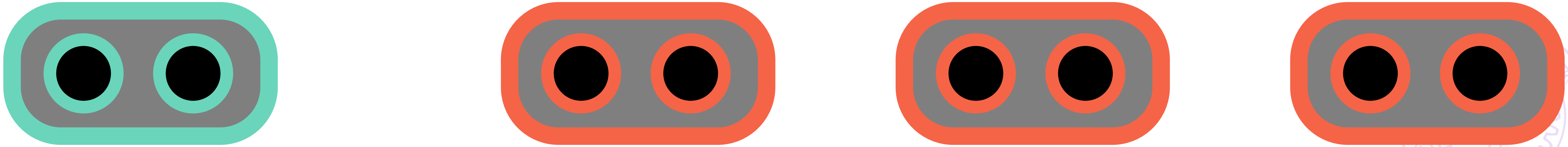
Example 1: Factorial

```
int fact(int a){
    if(a == 0) return 1;
    return a * fact(a - 1);
}
int main(){
    printf("%d", fact(2*3));
}
```

See how many clones got created!

Creating each clone takes time and memory

This is why recursive programs can be a bit slower – be careful



main()

fact(6)
= 6 * fact(5)

fact(5)
= 5 * 24 = 120

fact(4)
= 4 * 6 = 24

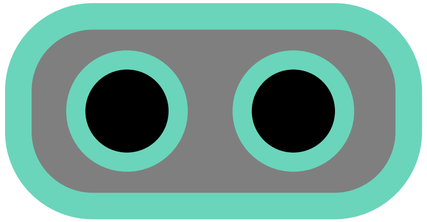
Example 1: Factorial

```
int fact(int a){
    if(a == 0) return 1;
    return a * fact(a - 1);
}
int main(){
    printf("%d", fact(2*3));
}
```

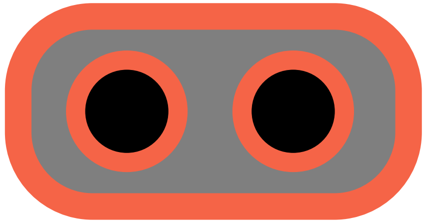
See how many clones got created!

Creating each clone takes time and memory

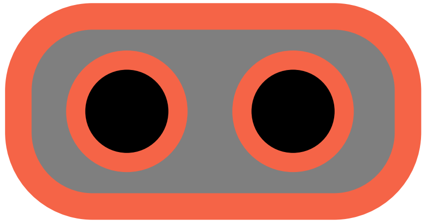
This is why recursive programs can be a bit slower – be careful



main()



fact(6)
= 6 * fact(5)



fact(5)
= 5 * 24 = 120

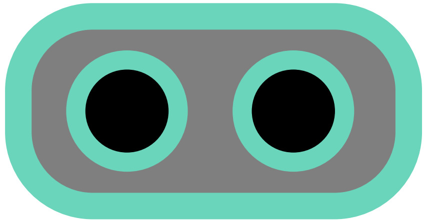
Example 1: Factorial

```
int fact(int a){
    if(a == 0) return 1;
    return a * fact(a - 1);
}
int main(){
    printf("%d", fact(2*3));
}
```

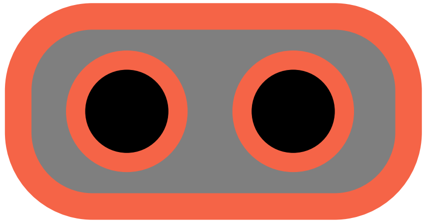
See how many clones got created!

Creating each clone takes time and memory

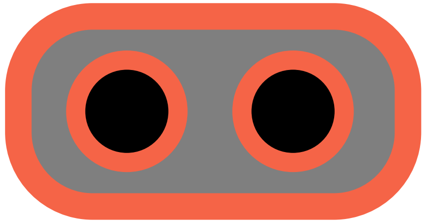
This is why recursive programs can be a bit slower – be careful



main()



fact(6)
= 6 * 120 = 720



fact(5)
= 5 * 24 = 120

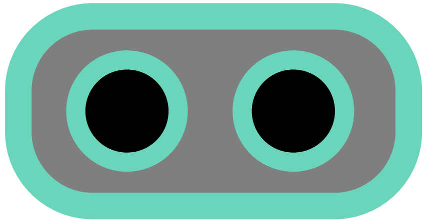
Example 1: Factorial

```
int fact(int a){
    if(a == 0) return 1;
    return a * fact(a - 1);
}
int main(){
    printf("%d", fact(2*3));
}
```

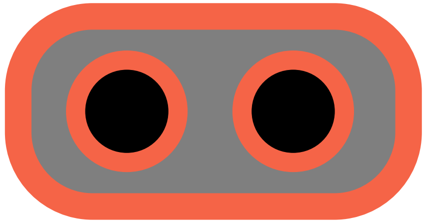
See how many clones got created!

Creating each clone takes time and memory

This is why recursive programs can be a bit slower – be careful



main()



fact(6)
= 6 * 120 = 720

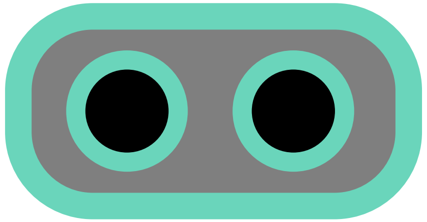
Example 1: Factorial

```
int fact(int a){
    if(a == 0) return 1;
    return a * fact(a - 1);
}
int main(){
    printf("%d", fact(2*3));
}
```

See how many clones got created!

Creating each clone takes time and memory

This is why recursive programs can be a bit slower – be careful



main()

Example 1: Factorial

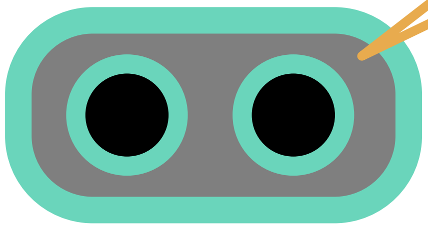
```
int fact(int a){
    if(a == 0) return 1;
    return a * fact(a - 1);
}
int main(){
    printf("%d", fact(2*3));
}
```

See how many clones got created!

Creating each clone takes time and memory

This is why recursive programs can be a bit slower – be careful

720



main()

Example 2: Fibonacci Numbers

Example 2: Fibonacci Numbers

There are two base cases for Fibonacci numbers

Example 2: Fibonacci Numbers

There are two base cases for Fibonacci numbers

- The first Fibonacci number is defined to be 0

Example 2: Fibonacci Numbers

There are two base cases for Fibonacci numbers

- The first Fibonacci number is defined to be 0
- The second Fibonacci number is defined to be 1

Example 2: Fibonacci Numbers

There are two base cases for Fibonacci numbers

- The first Fibonacci number is defined to be 0
- The second Fibonacci number is defined to be 1

Recursive case: for n > 2, n-th Fibonacci number is the sum of the previous two Fibonacci numbers

Example 2: Fibonacci Numbers

There are two base cases for Fibonacci numbers

- The first Fibonacci number is defined to be 0
- The second Fibonacci number is defined to be 1

Recursive case: for n > 2, n-th Fibonacci number is the sum of the previous two Fibonacci numbers

```
int fib(int n){
    if(n == 1) return 0;
    if(n == 2) return 1;
    return fib(n-1) + fib(n-2);
}
int main(){
    printf("%d", fib(5));
}
```

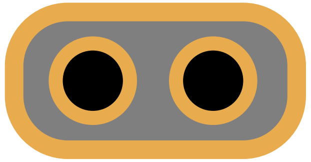
Example 2: Fibonacci Numbers

There are two base cases for Fibonacci numbers

- The first Fibonacci number is defined to be 0
- The second Fibonacci number is defined to be 1

Recursive case: for n > 2, n-th Fibonacci number is the sum of the previous two Fibonacci numbers

```
int fib(int n){
    if(n == 1) return 0;
    if(n == 2) return 1;
    return fib(n-1) + fib(n-2);
}
int main(){
    printf("%d", fib(5));
}
```



fib(5)

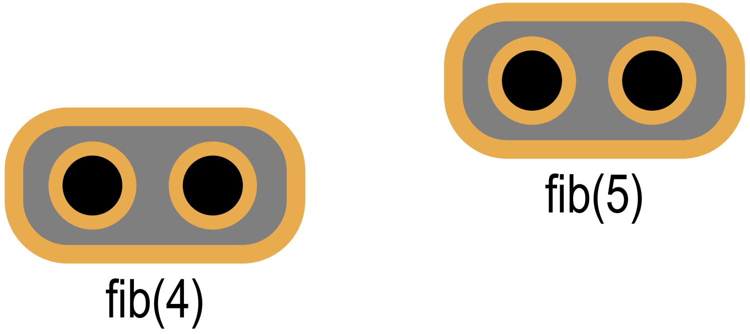
Example 2: Fibonacci Numbers

There are two base cases for Fibonacci numbers

- The first Fibonacci number is defined to be 0
- The second Fibonacci number is defined to be 1

Recursive case: for n > 2, n-th Fibonacci number is the sum of the previous two Fibonacci numbers

```
int fib(int n){
    if(n == 1) return 0;
    if(n == 2) return 1;
    return fib(n-1) + fib(n-2);
}
int main(){
    printf("%d", fib(5));
}
```



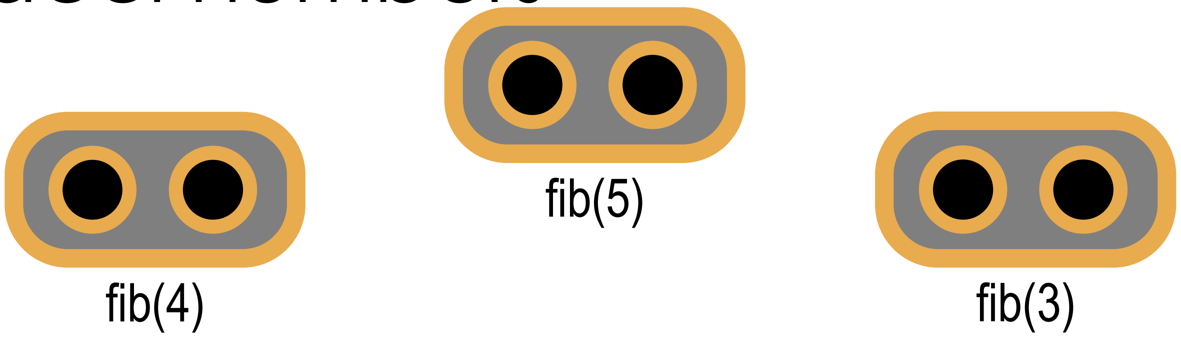
Example 2: Fibonacci Numbers

There are two base cases for Fibonacci numbers

- The first Fibonacci number is defined to be 0
- The second Fibonacci number is defined to be 1

Recursive case: for n > 2, n-th Fibonacci number is the sum of the previous two Fibonacci numbers

```
int fib(int n){
    if(n == 1) return 0;
    if(n == 2) return 1;
    return fib(n-1) + fib(n-2);
}
int main(){
    printf("%d", fib(5));
}
```



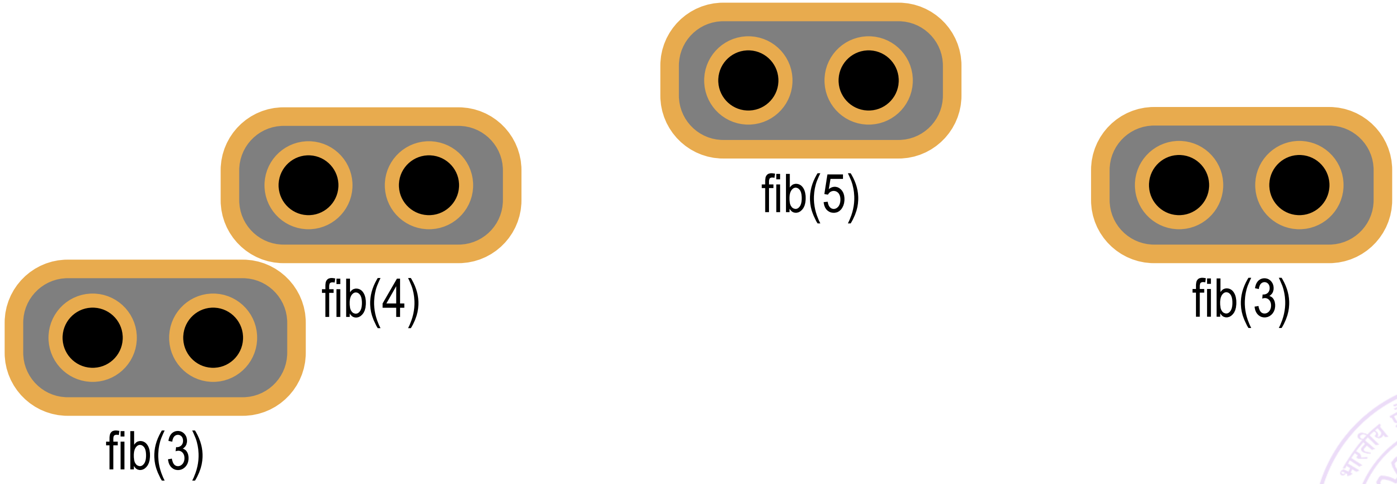
Example 2: Fibonacci Numbers

There are two base cases for Fibonacci numbers

- The first Fibonacci number is defined to be 0
- The second Fibonacci number is defined to be 1

Recursive case: for n > 2, n-th Fibonacci number is the sum of the previous two Fibonacci numbers

```
int fib(int n){
    if(n == 1) return 0;
    if(n == 2) return 1;
    return fib(n-1) + fib(n-2);
}
int main(){
    printf("%d", fib(5));
}
```



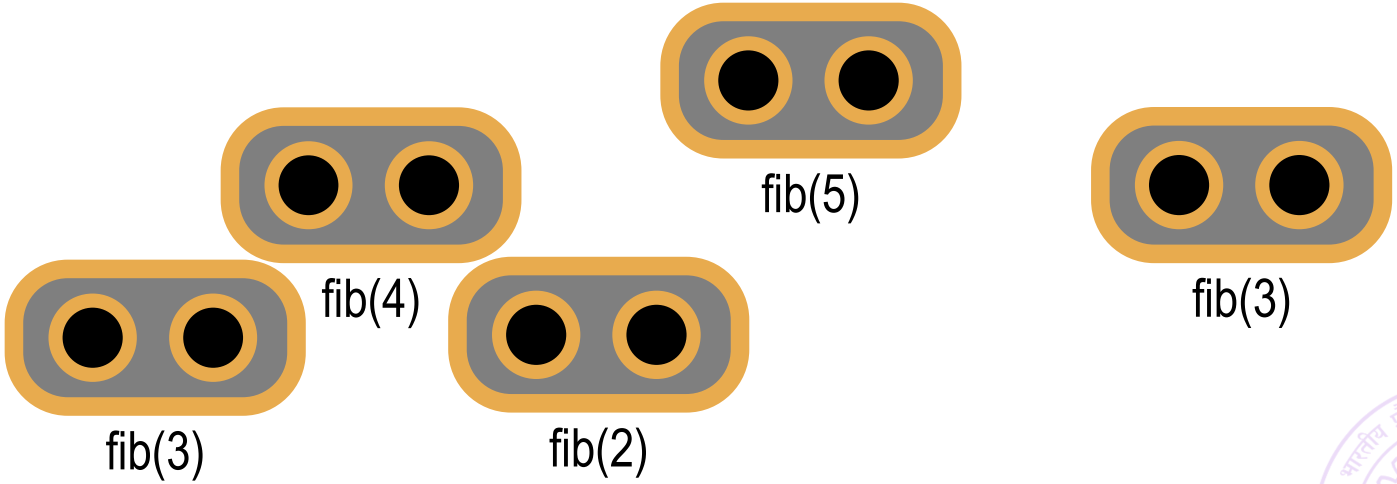
Example 2: Fibonacci Numbers

There are two base cases for Fibonacci numbers

- The first Fibonacci number is defined to be 0
- The second Fibonacci number is defined to be 1

Recursive case: for n > 2, n-th Fibonacci number is the sum of the previous two Fibonacci numbers

```
int fib(int n){
    if(n == 1) return 0;
    if(n == 2) return 1;
    return fib(n-1) + fib(n-2);
}
int main(){
    printf("%d", fib(5));
}
```



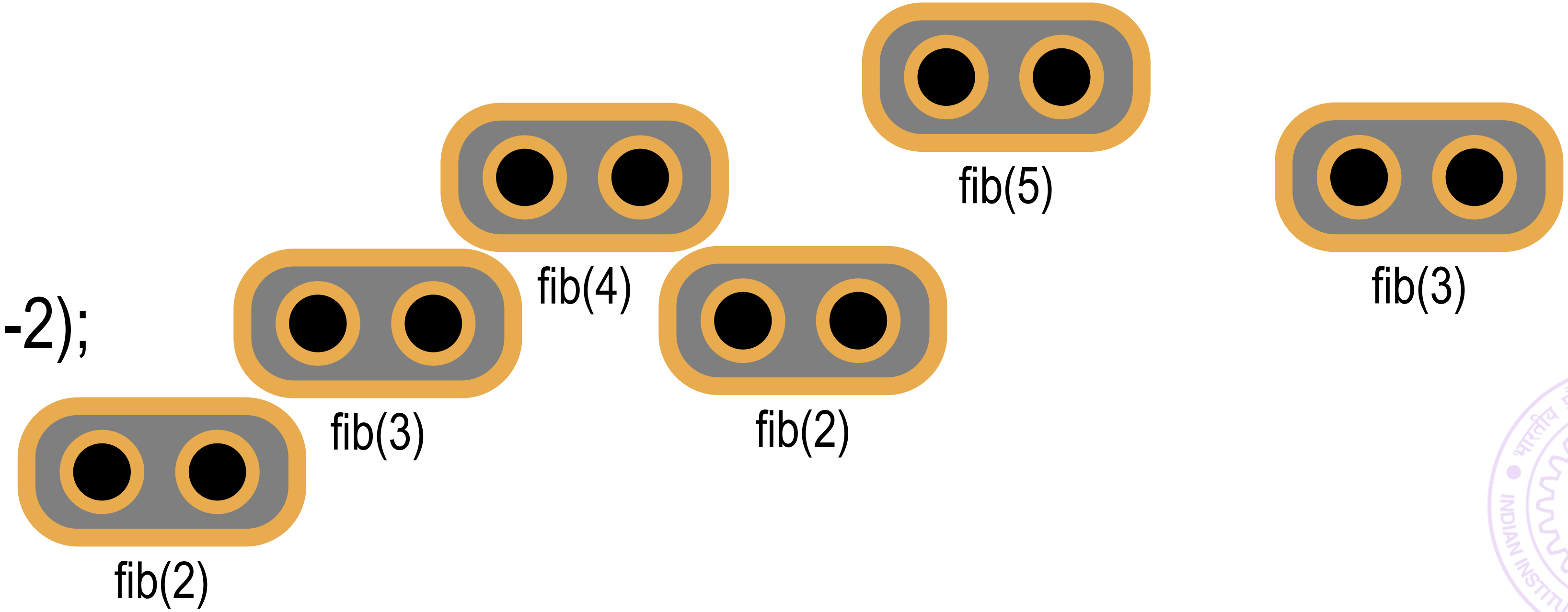
Example 2: Fibonacci Numbers

There are two base cases for Fibonacci numbers

- The first Fibonacci number is defined to be 0
- The second Fibonacci number is defined to be 1

Recursive case: for n > 2, n-th Fibonacci number is the sum of the previous two Fibonacci numbers

```
int fib(int n){
    if(n == 1) return 0;
    if(n == 2) return 1;
    return fib(n-1) + fib(n-2);
}
int main(){
    printf("%d", fib(5));
}
```



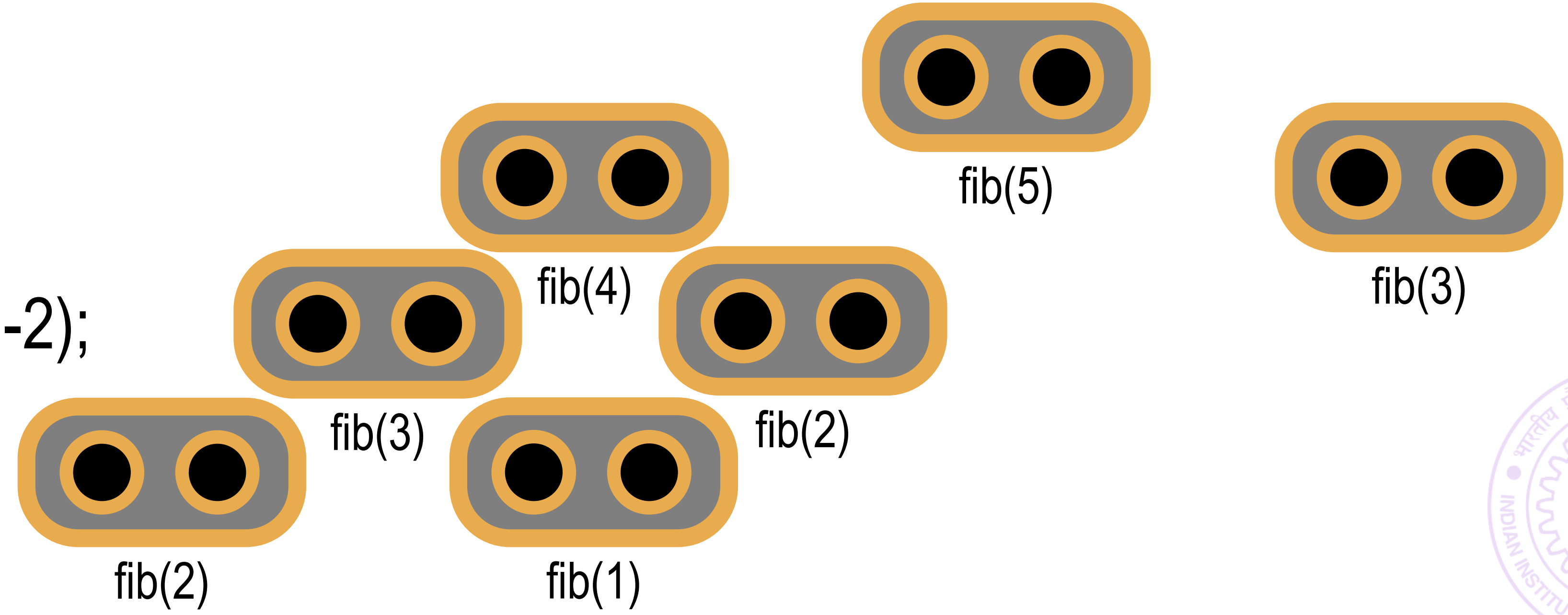
Example 2: Fibonacci Numbers

There are two base cases for Fibonacci numbers

- The first Fibonacci number is defined to be 0
- The second Fibonacci number is defined to be 1

Recursive case: for n > 2, n-th Fibonacci number is the sum of the previous two Fibonacci numbers

```
int fib(int n){
    if(n == 1) return 0;
    if(n == 2) return 1;
    return fib(n-1) + fib(n-2);
}
int main(){
    printf("%d", fib(5));
}
```



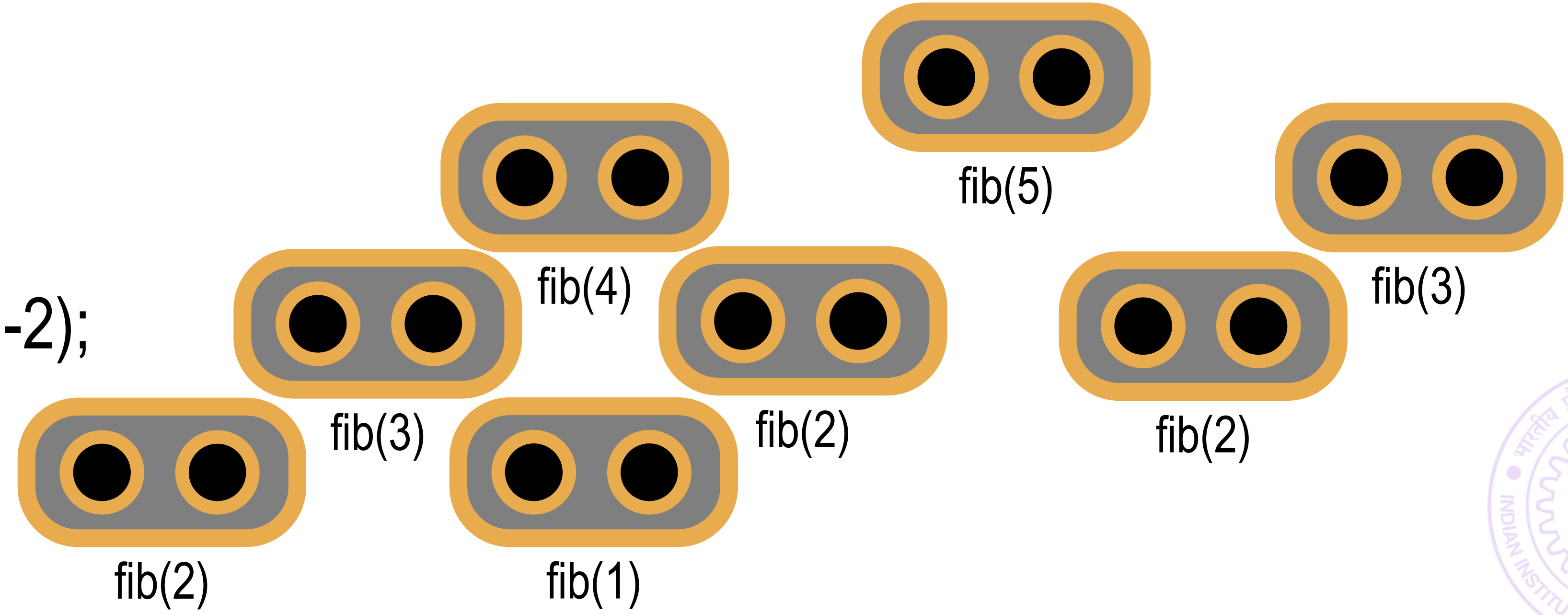
Example 2: Fibonacci Numbers

There are two base cases for Fibonacci numbers

- The first Fibonacci number is defined to be 0
- The second Fibonacci number is defined to be 1

Recursive case: for n > 2, n-th Fibonacci number is the sum of the previous two Fibonacci numbers

```
int fib(int n){
    if(n == 1) return 0;
    if(n == 2) return 1;
    return fib(n-1) + fib(n-2);
}
int main(){
    printf("%d", fib(5));
}
```



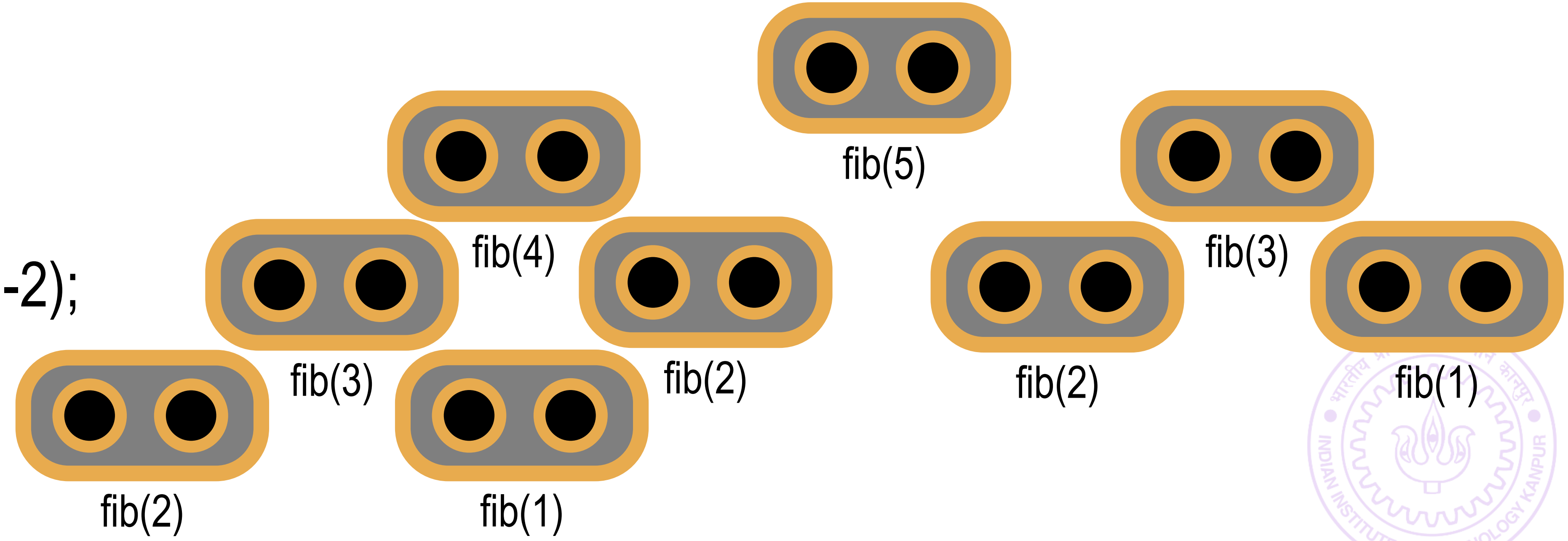
Example 2: Fibonacci Numbers

There are two base cases for Fibonacci numbers

- The first Fibonacci number is defined to be 0
- The second Fibonacci number is defined to be 1

Recursive case: for n > 2, n-th Fibonacci number is the sum of the previous two Fibonacci numbers

```
int fib(int n){
    if(n == 1) return 0;
    if(n == 2) return 1;
    return fib(n-1) + fib(n-2);
}
int main(){
    printf("%d", fib(5));
}
```



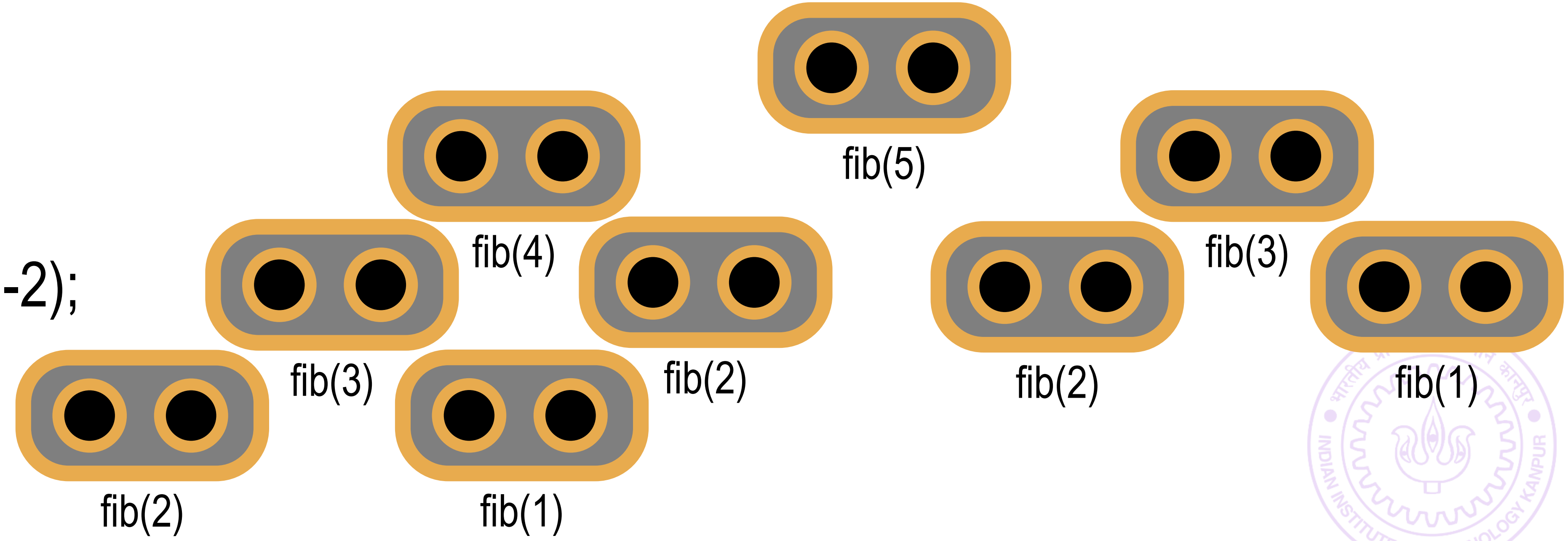
Example 2: Fibonacci Numbers

There are two base cases for Fibonacci numbers

- The first Fibonacci number is defined to be 0
- The second Fibonacci number is defined to be 1

Recursive case: for n > 2, n-th Fibonacci number is the sum of the previous two Fibonacci numbers

```
int fib(int n){
    if(n == 1) return 0;
    if(n == 2) return 1;
    return fib(n-1) + fib(n-2);
}
int main(){
    printf("%d", fib(5));
}
```



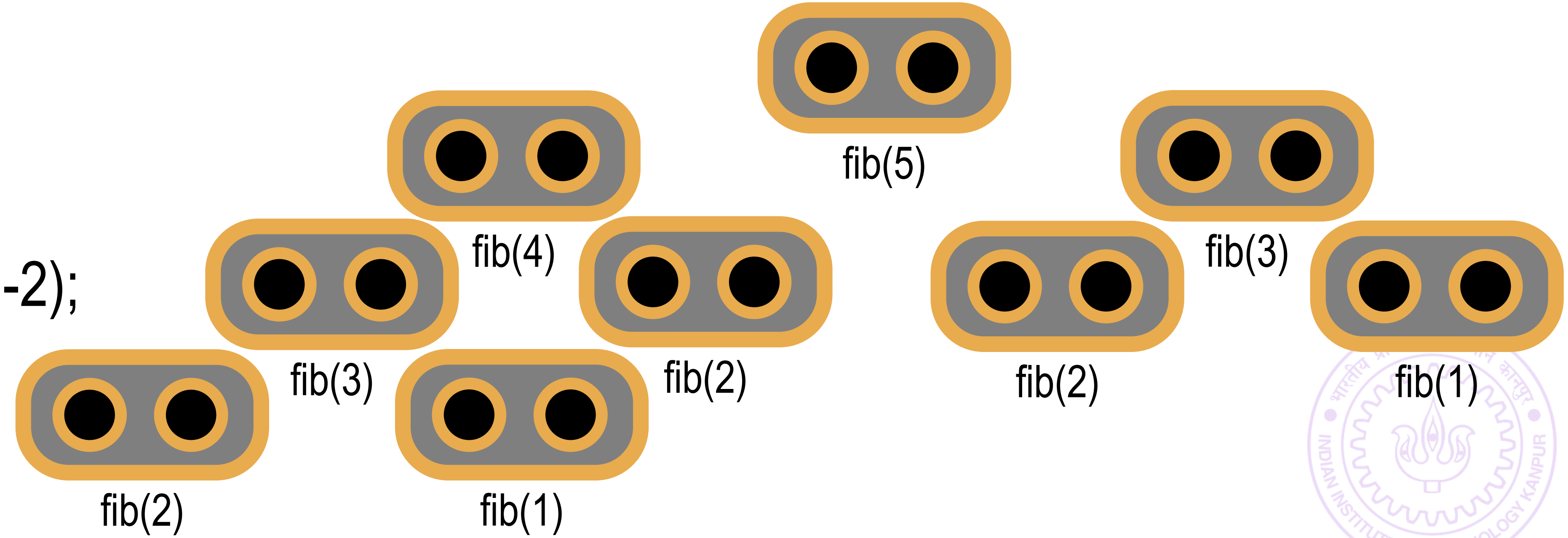
Example 2: Fibonacci Numb

There are two base cases for Fibonacci numbers

- The first Fibonacci number is defined to be 0
- The second Fibonacci number is defined to be 1

Recursive case: for n > 2, n-th Fibonacci number is the sum of the previous two Fibonacci numbers

```
int fib(int n){
    if(n == 1) return 0;
    if(n == 2) return 1;
    return fib(n-1) + fib(n-2);
}
int main(){
    printf("%d", fib(5));
}
```



Example 2: Fibo

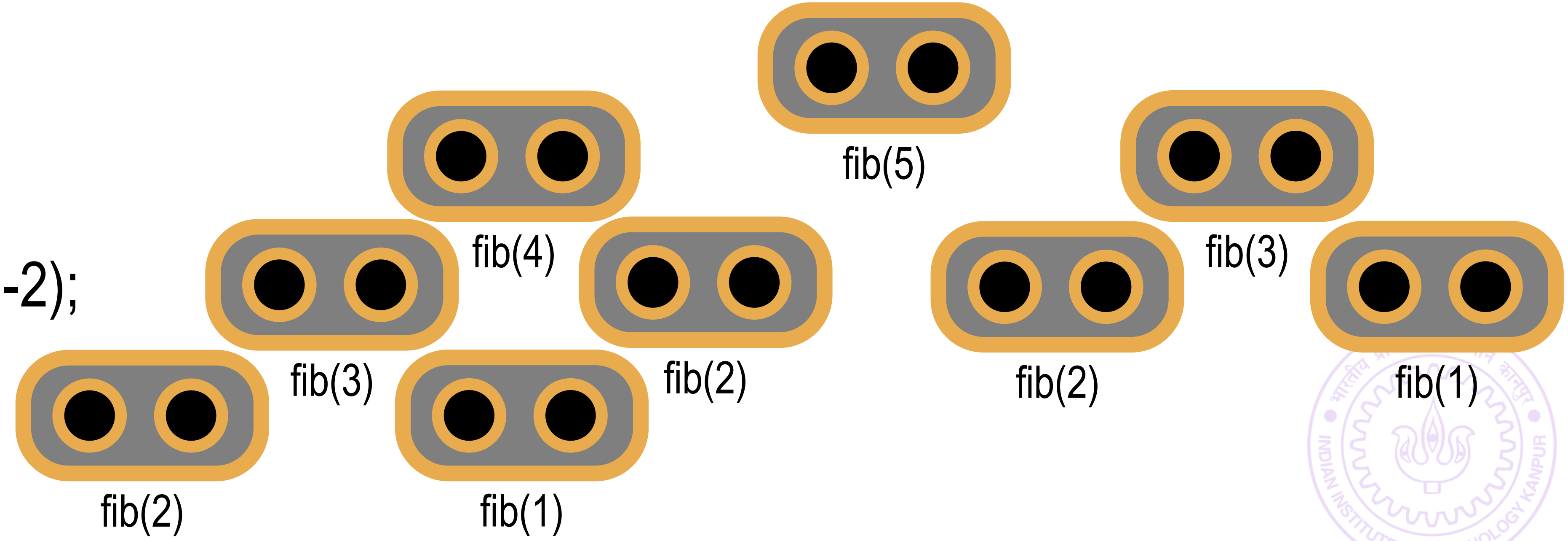
Wasted effort too!
fib(1) calculated twice
fib(2) calculated 3 times
fib(3) calculated twice

Explosion of clones!

There are two base cas umbers
- The first Fibonacci number is defined to be 0
- The second Fibonacci number is defined to be 1

Recursive case: for n > 2, n-th Fibonacci number is the sum of the previous two Fibonacci numbers

```
int fib(int n){
    if(n == 1) return 0;
    if(n == 2) return 1;
    return fib(n-1) + fib(n-2);
}
int main(){
    printf("%d", fib(5));
}
```



Imagine the number of clones to calculate fib(100). **Challenge**: don't imagine – find out ☺

Wasted effort too!
fib(1) calculated twice
fib(2) calculated 3 times
fib(3) calculated twice

Explosion of clones!

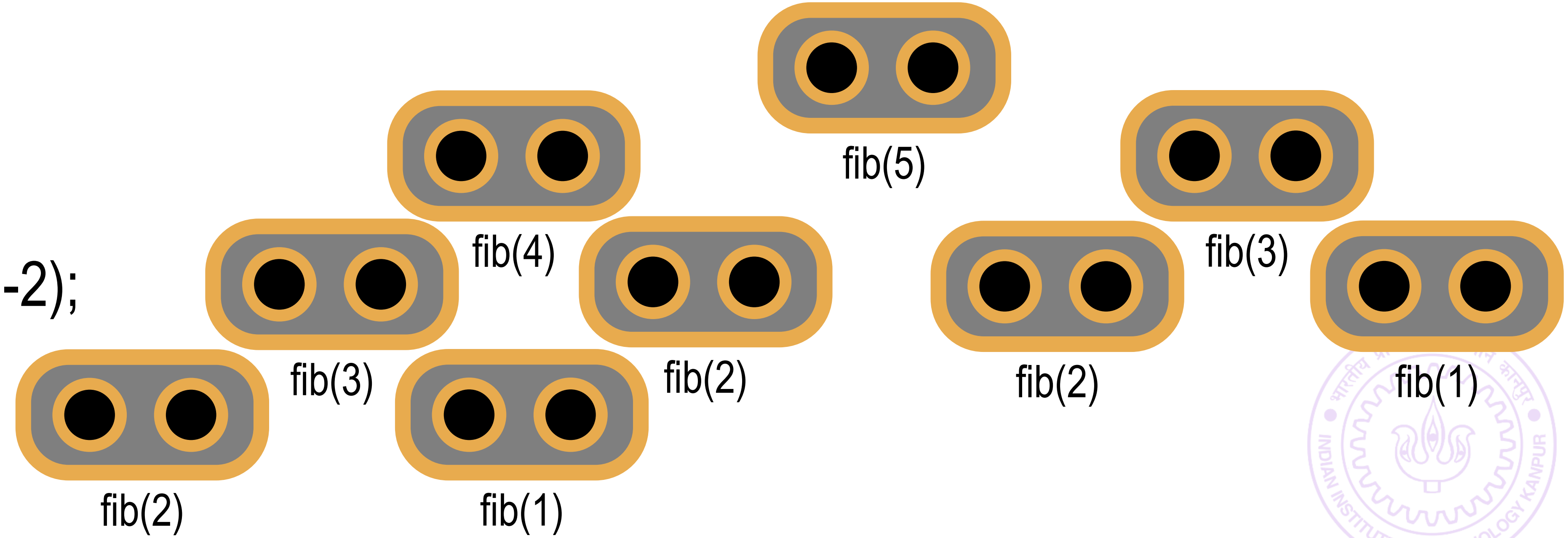
There are two base cas… umbers

The first Fibonacci number is defined to be 0

The second Fibonacci number is defined to be 1

Recursive case: for n > 2, n-th Fibonacci number is the sum of the previous two Fibonacci numbers

```
int fib(int n){
    if(n == 1) return 0;
    if(n == 2) return 1;
    return fib(n-1) + fib(n-2);
}
int main(){
    printf("%d", fib(5));
}
```



Imagine the number of clones to calculate fib(100). **Challenge**: don't imagine – find out ☺

Wasted effort too!
fib(1) calculated twice
fib(2) calculated 3 times
fib(3) calculated twice

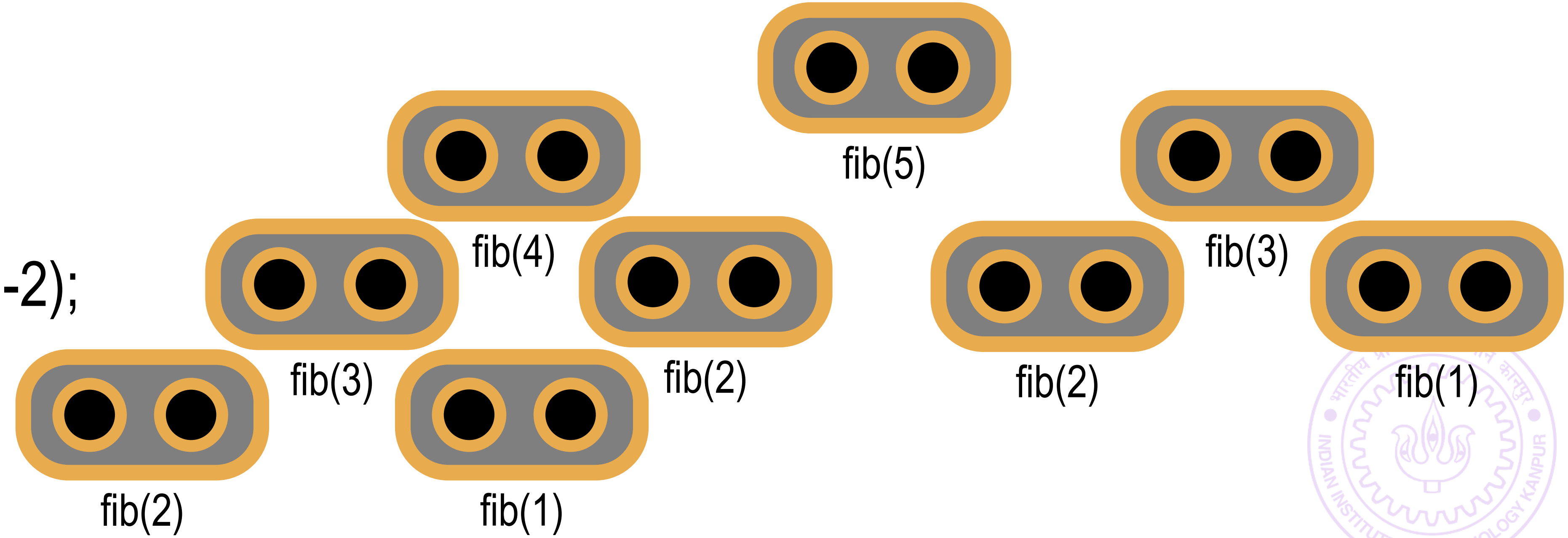
Explosion of clones!

There are two base cas ... umbers

- ... first Fibonacci number is defined to be 0
- ...econd Fibonacci number is defined to be 1

...ursive case: for n > 2, n-th Fibonacci number is the sum ...e previous two Fibonacci numbers

```
... n){
    if(n == 1) return 0;
    if(n == 2) return 1;
    return fib(n-1) + fib(n-2);
}
int main(){
    printf("%d", fib(5));
}
```



Imagine the number of clones to calculate fib(100). **Challenge**: don't imagine – find out ☺

Wasted effort too!
fib(1) calculated twice
fib(2) calculated 3 times
fib(3) calculated twice

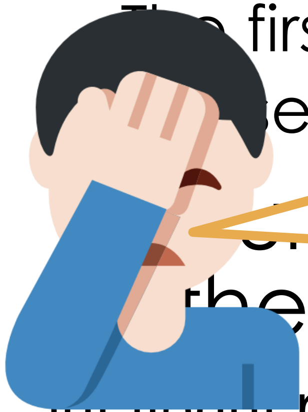
Explosion of clones!

There are two base cas... umbers

The first Fibonacci number is defined to be 0

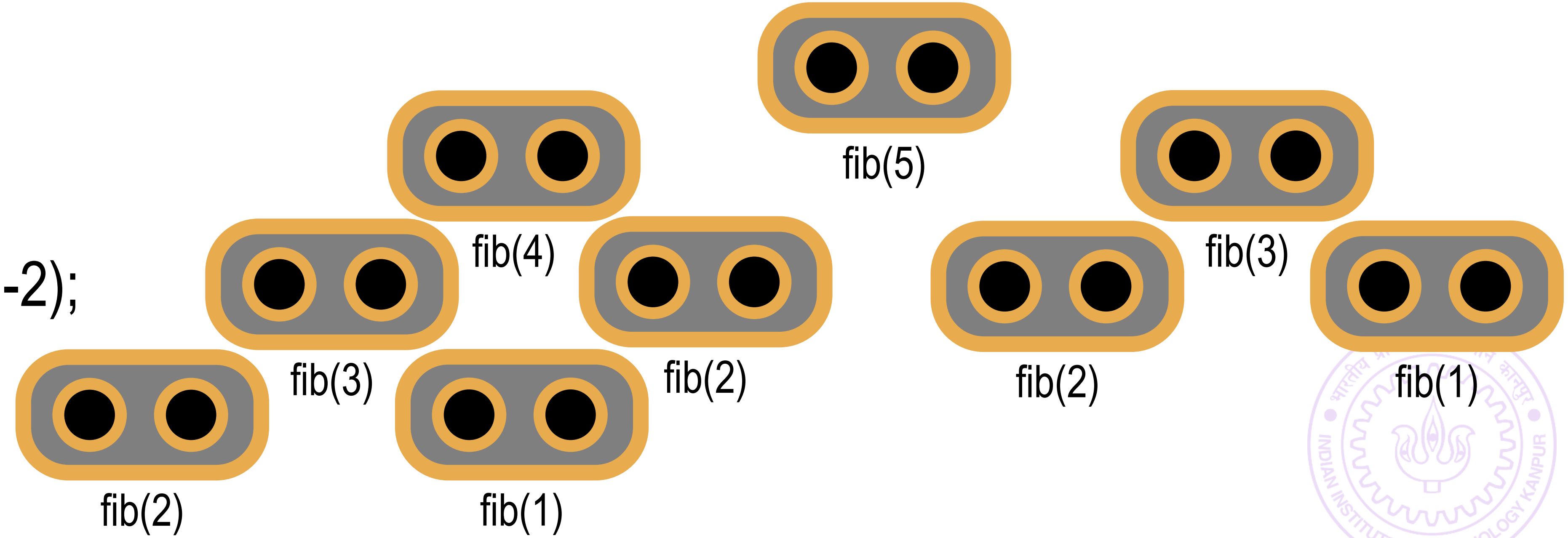
...seco... fined to be 1

...h Fibonacci number is the sum ...the previous two Fibonacci numbers



I could have easily solved this problem using a for loop – much faster and no clones

```
int fib(int n){
    if(n == 1) return 0;
    if(n == 2) return 1;
    return fib(n-1) + fib(n-2);
}
int main(){
    printf("%d", fib(5));
}
```



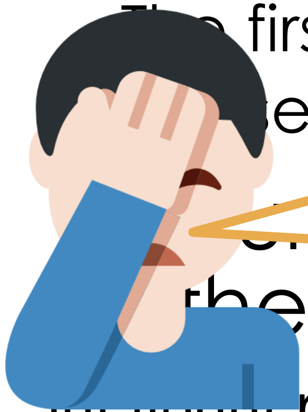
Imagine the number of clones to calculate fib(100). **Challenge**: don't imagine – find out ☺

Wasted effort too!
fib(1) calculated twice
fib(2) calculated 3 times
fib(3) calculated twice

Explosion of clones!

There are two base cas... umbers

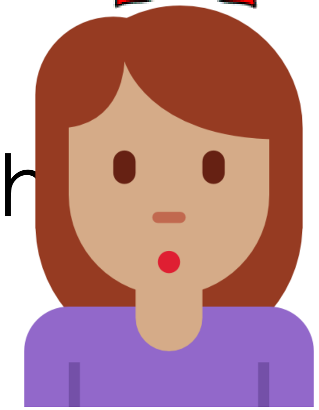
The first Fibonacci number is defined to be 0



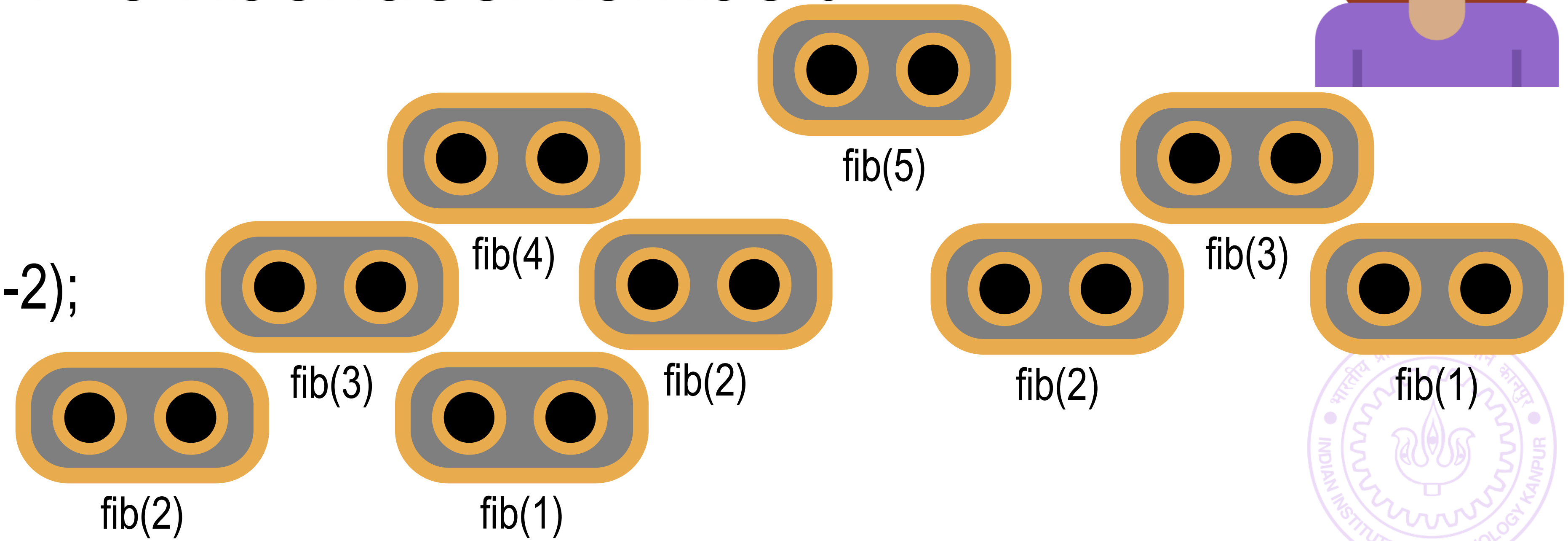
I could have easily solved this problem using a for loop – much faster and no clones

...seco... fined to be 1

...h Fibonacci number is th... the previous two Fibonacci numbers



```
int fib(int n){
    if(n == 1) return 0;
    if(n == 2) return 1;
    return fib(n-1) + fib(n-2);
}
int main(){
    printf("%d", fib(5));
}
```

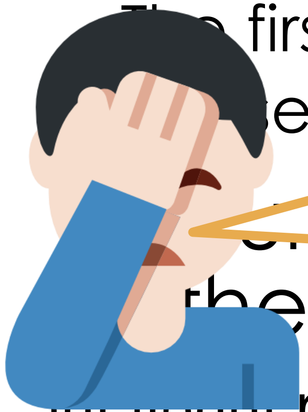


Imagine the number of clones to calculate fib(100). **Challenge**: don't imagine – find out ☺

Wasted effort too!
fib(1) calculated twice
fib(2) calculated 3 times
fib(3) calculated twice

Explosion of clones!

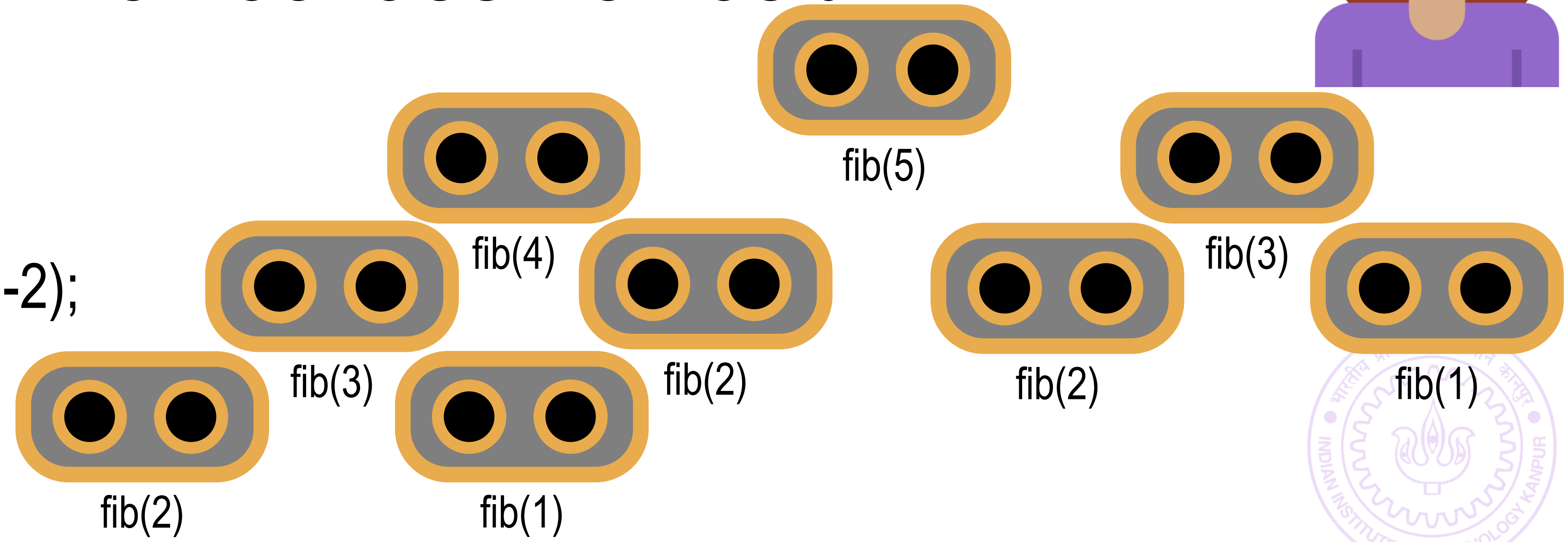
the previous two Fibonacci numbers



I could have easily solved this problem using a for loop – much faster and no clones

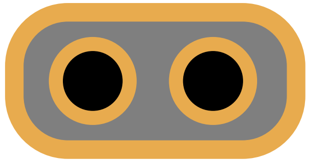
That's why we were warned. Recursion allows neat code but sometimes can be slower

```
int fib(int n){
    if(n == 1) return 0;
    if(n == 2) return 1;
    return fib(n-1) + fib(n-2);
}
int main(){
    printf("%d", fib(5));
}
```



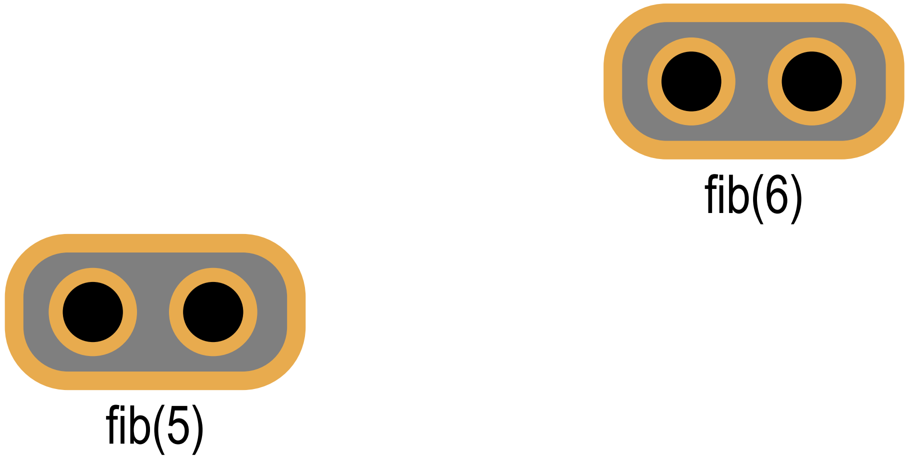
Attack of the Clones

Attack of the Clones

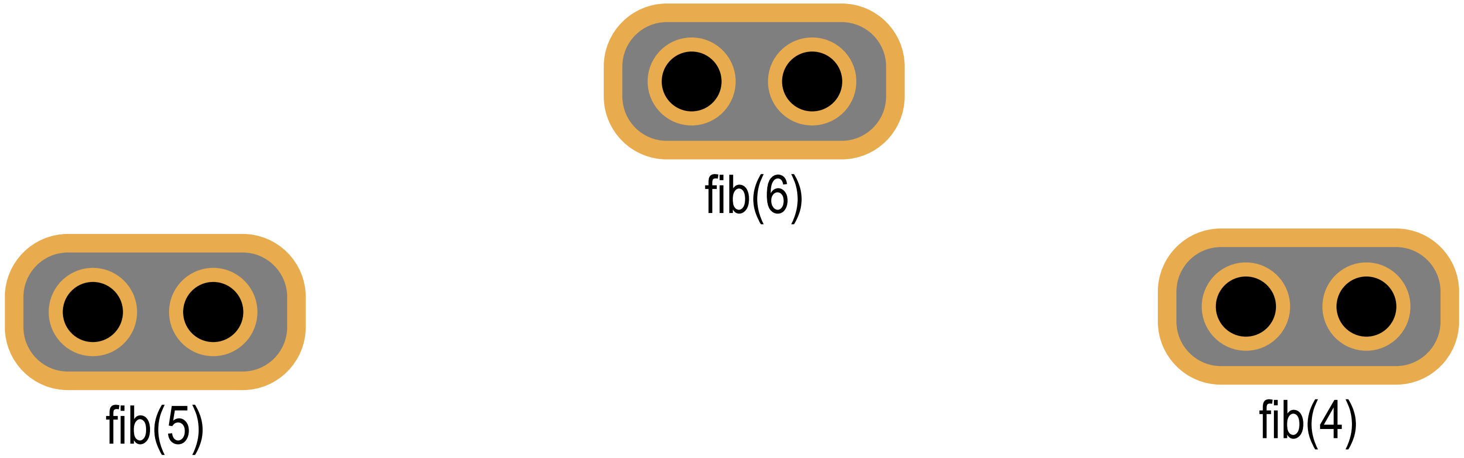


fib(6)

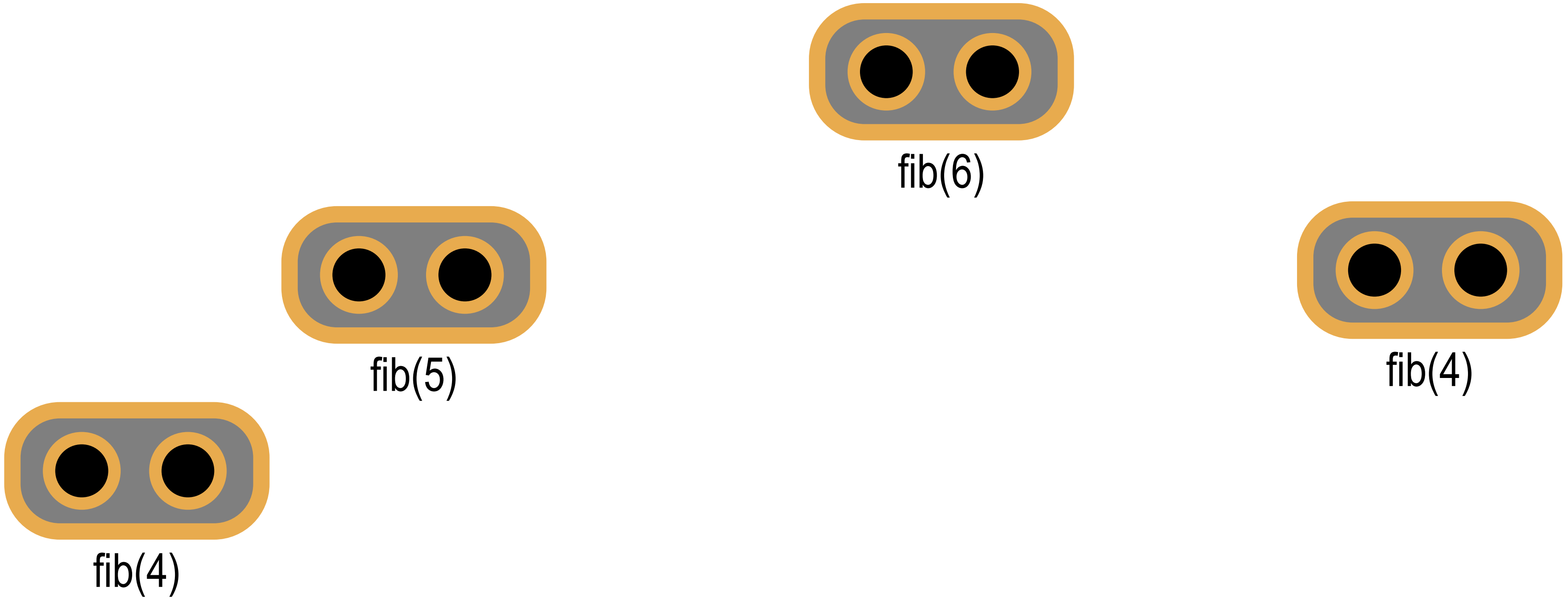
Attack of the Clones



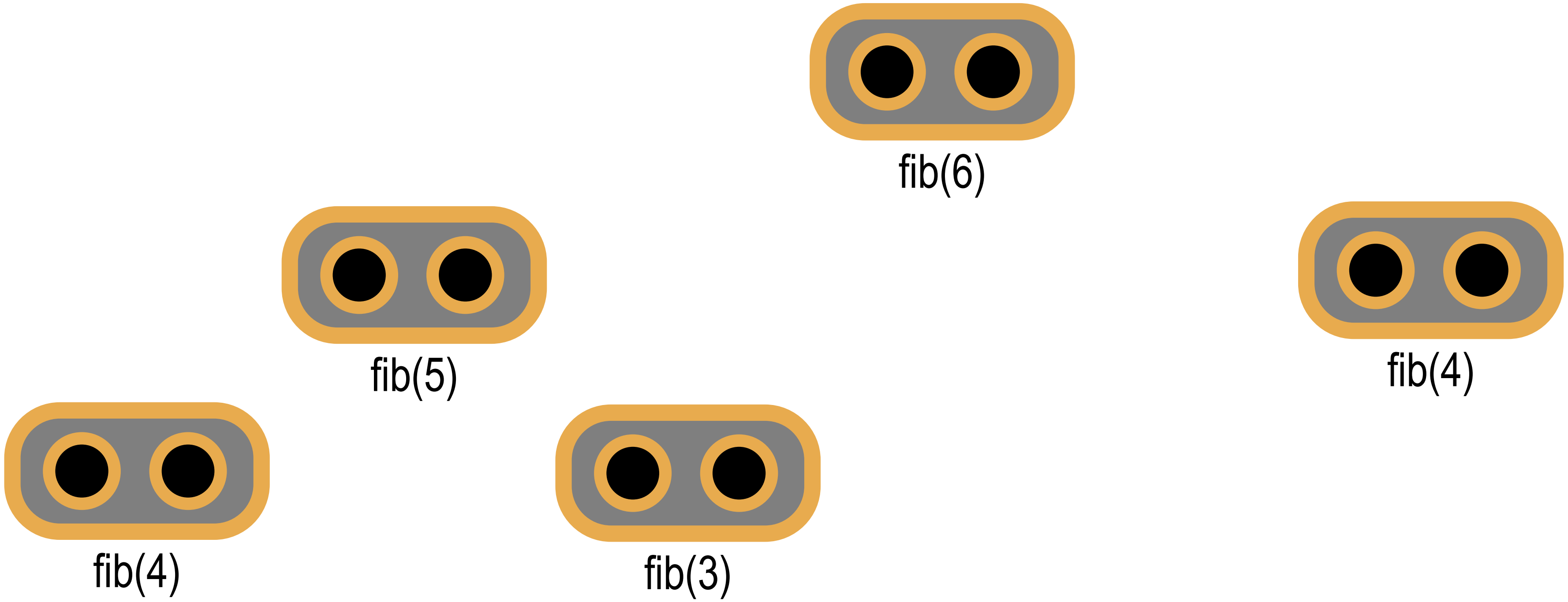
Attack of the Clones



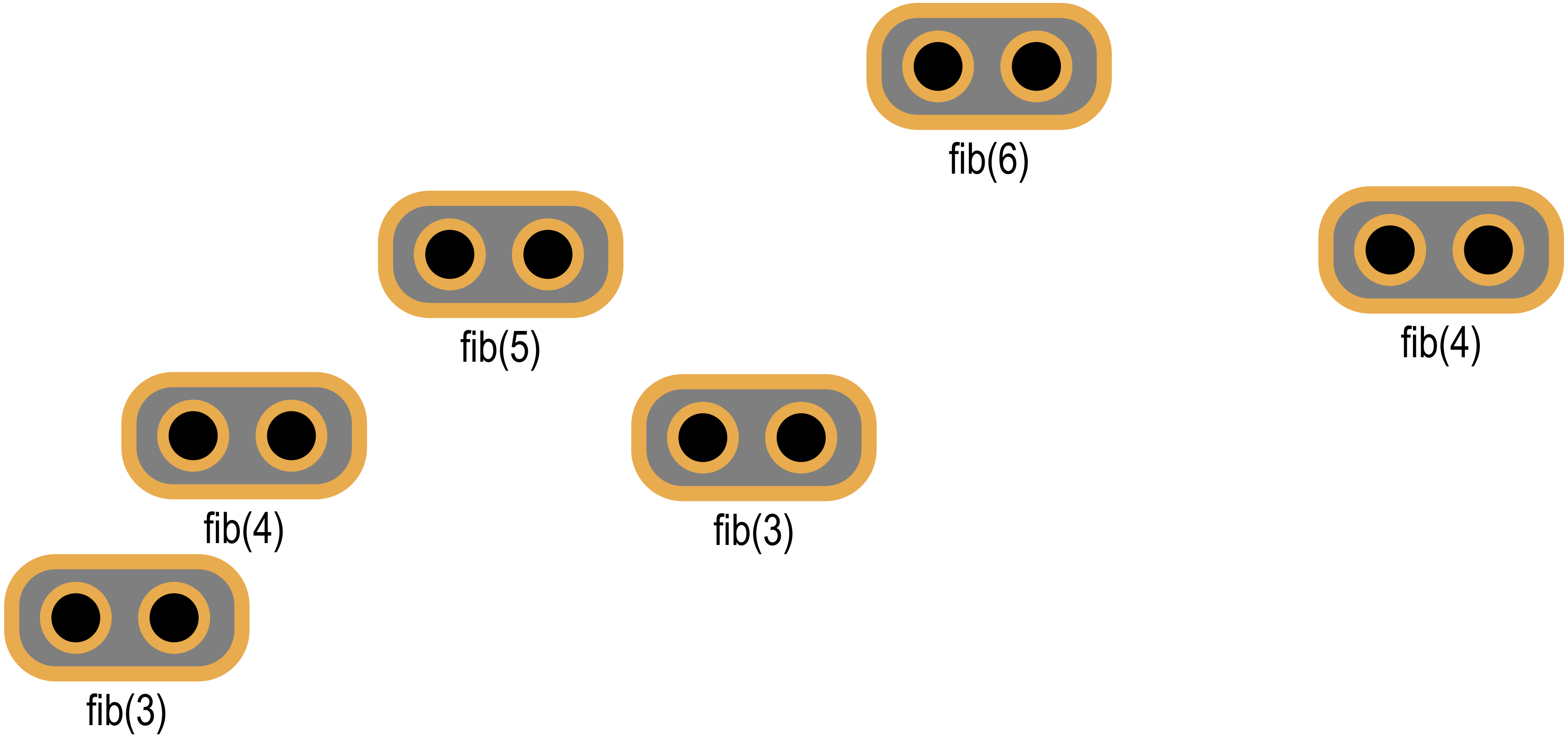
Attack of the Clones



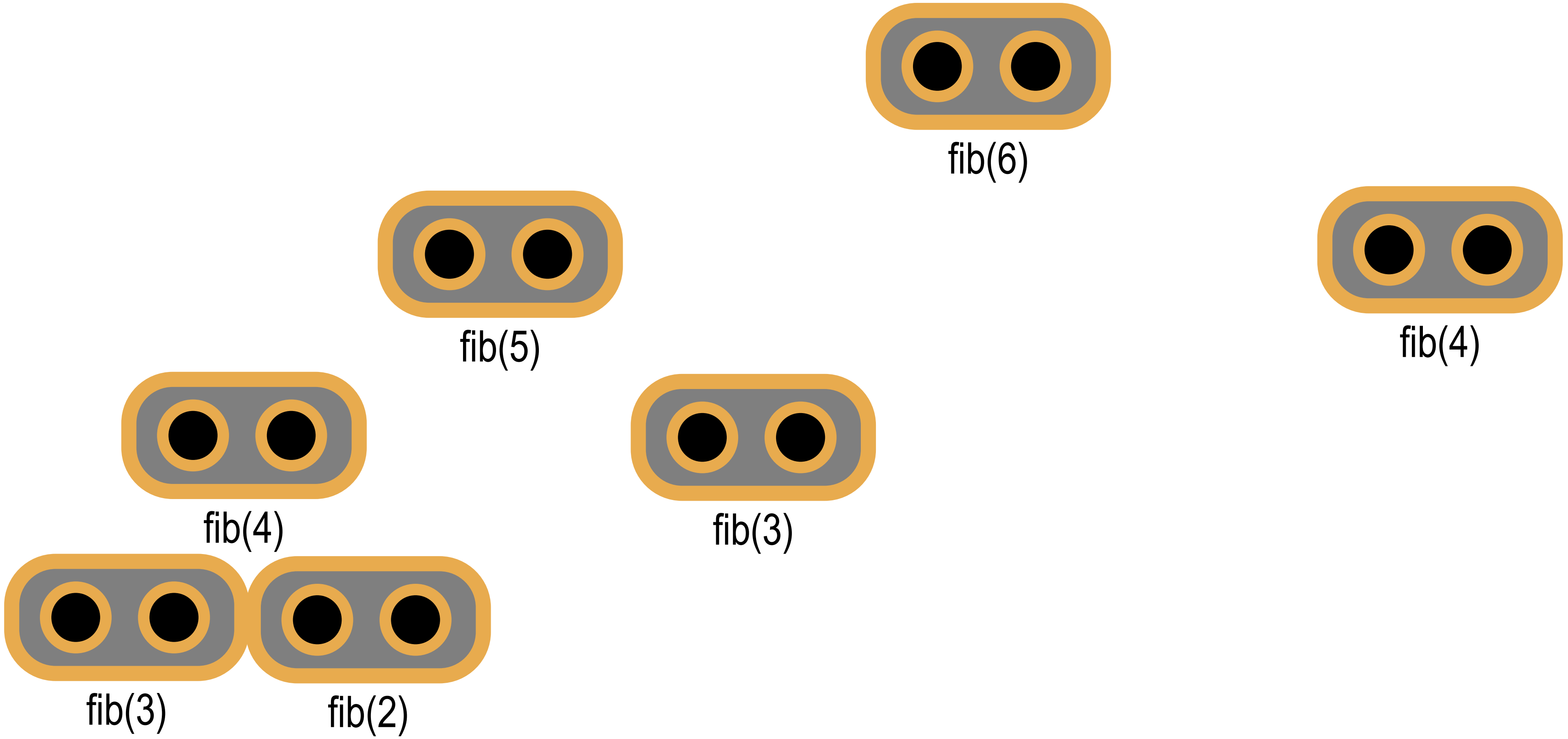
Attack of the Clones



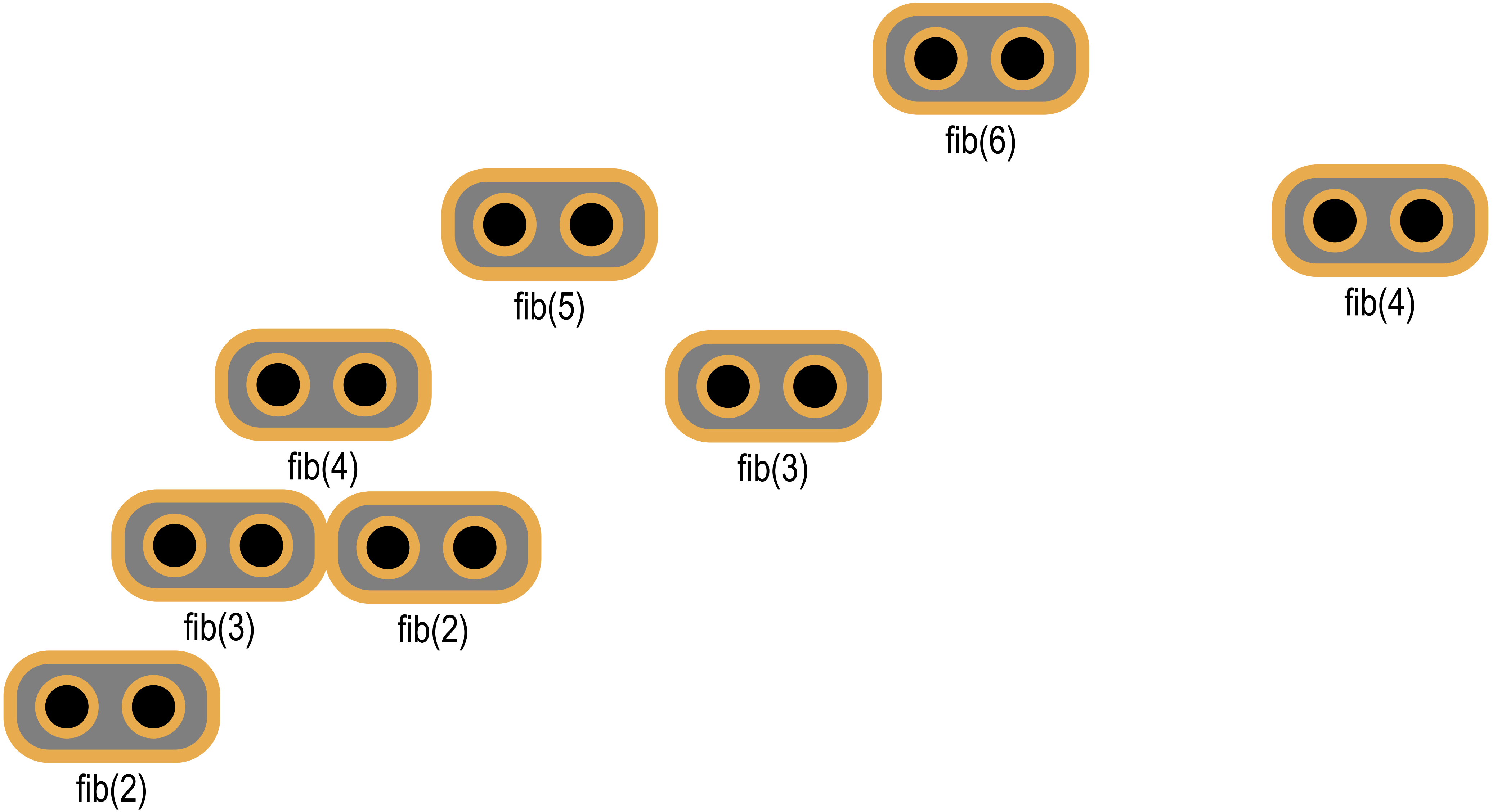
Attack of the Clones



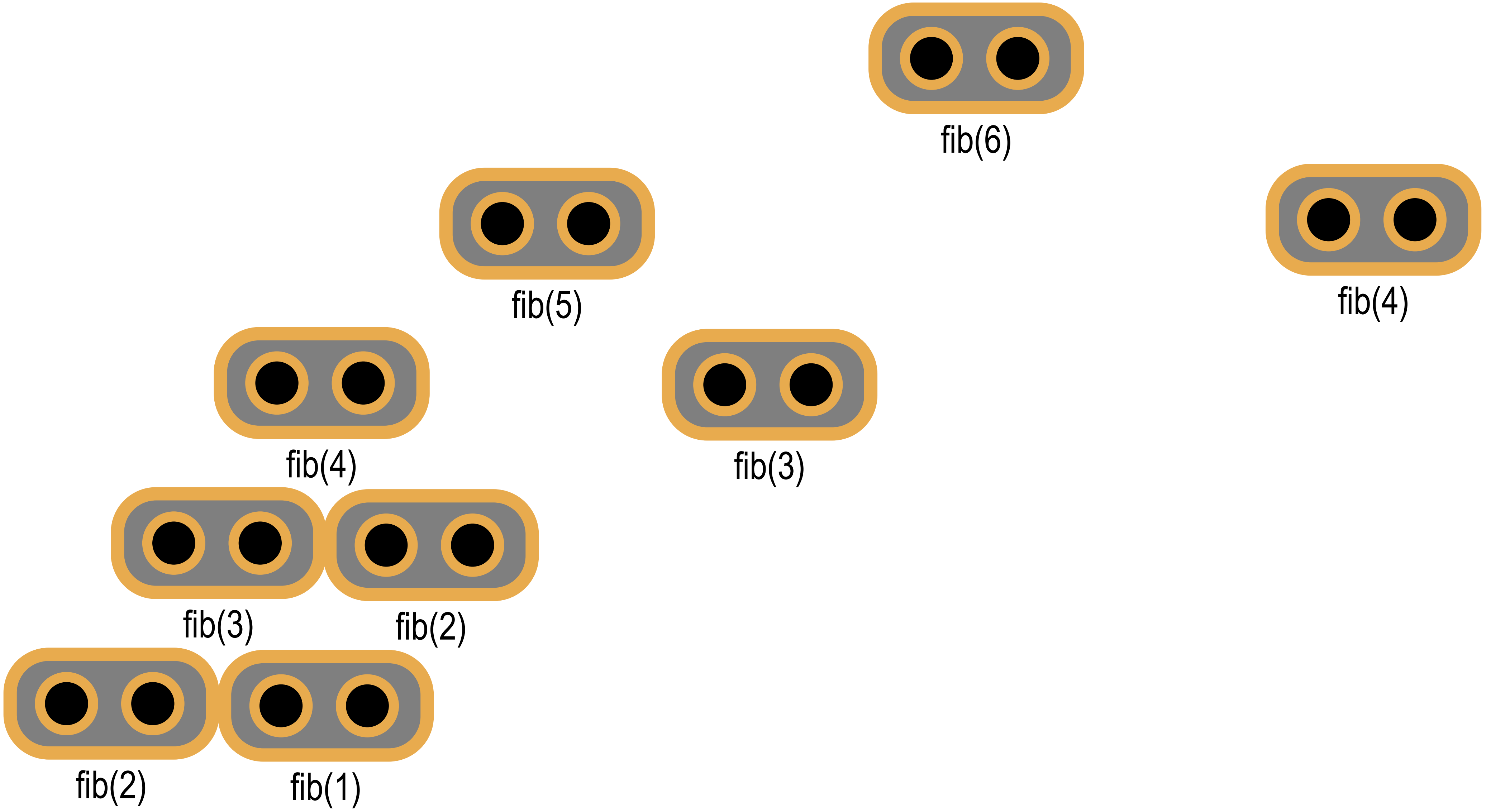
Attack of the Clones



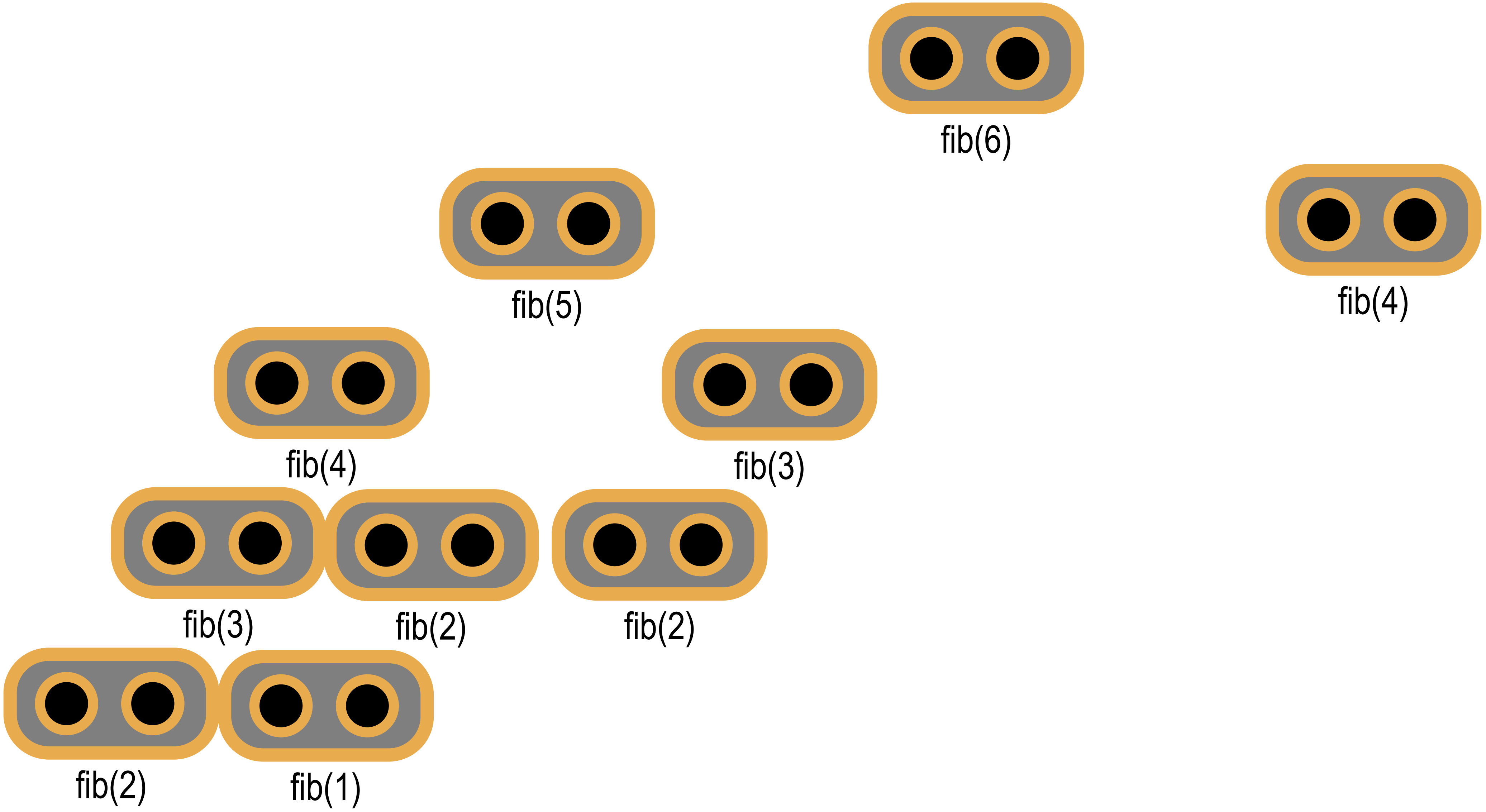
Attack of the Clones



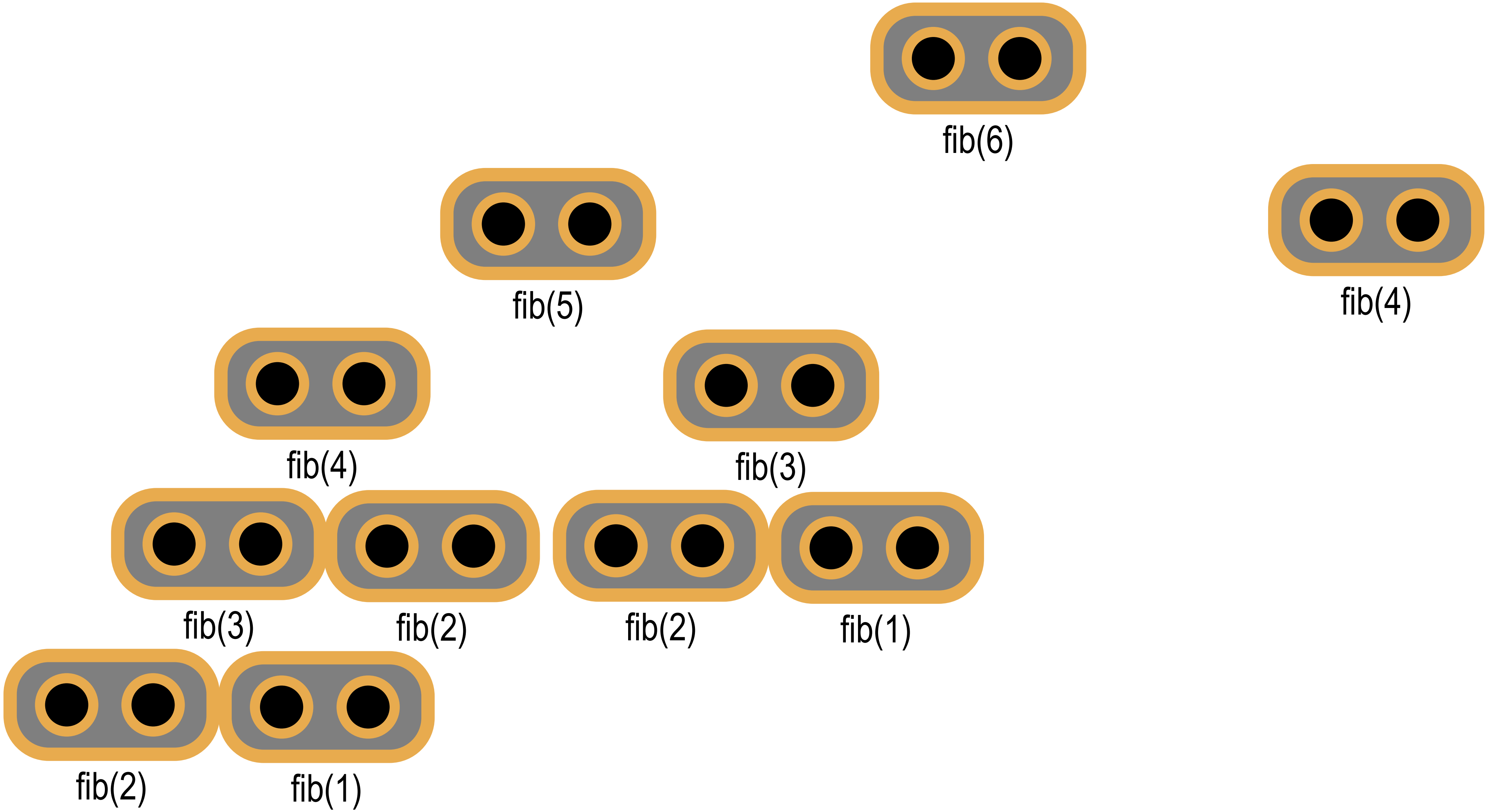
Attack of the Clones



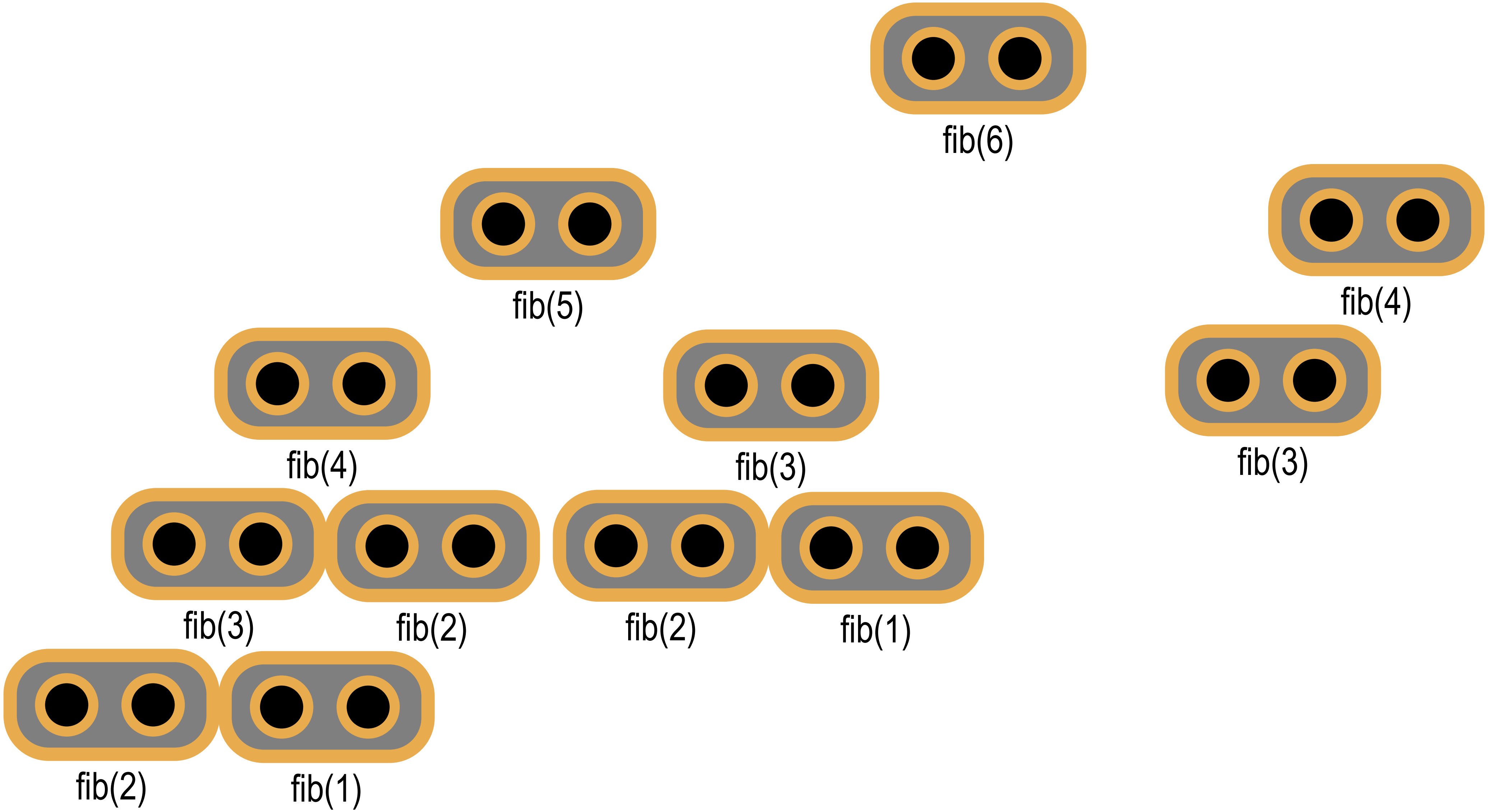
Attack of the Clones



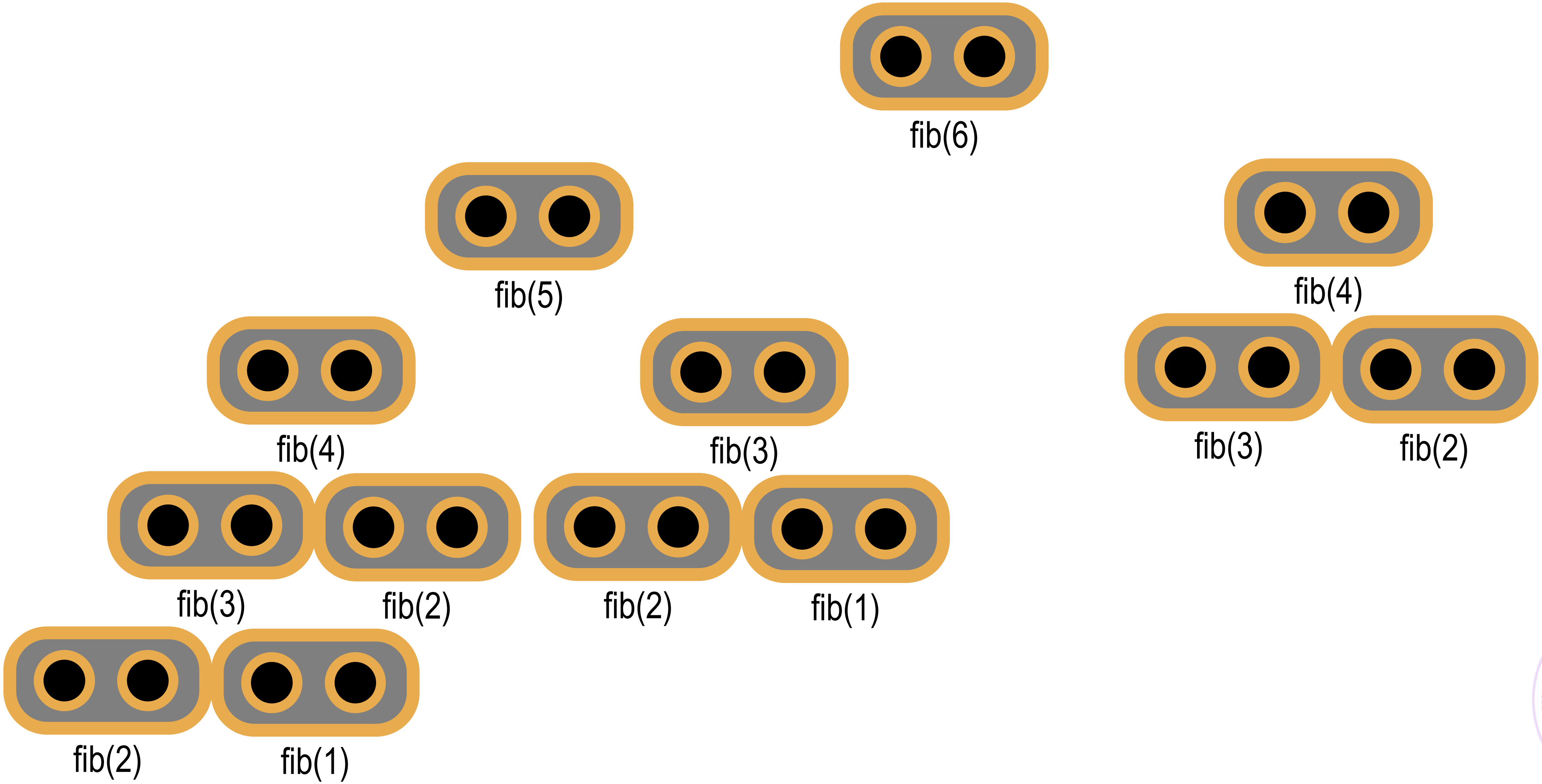
Attack of the Clones



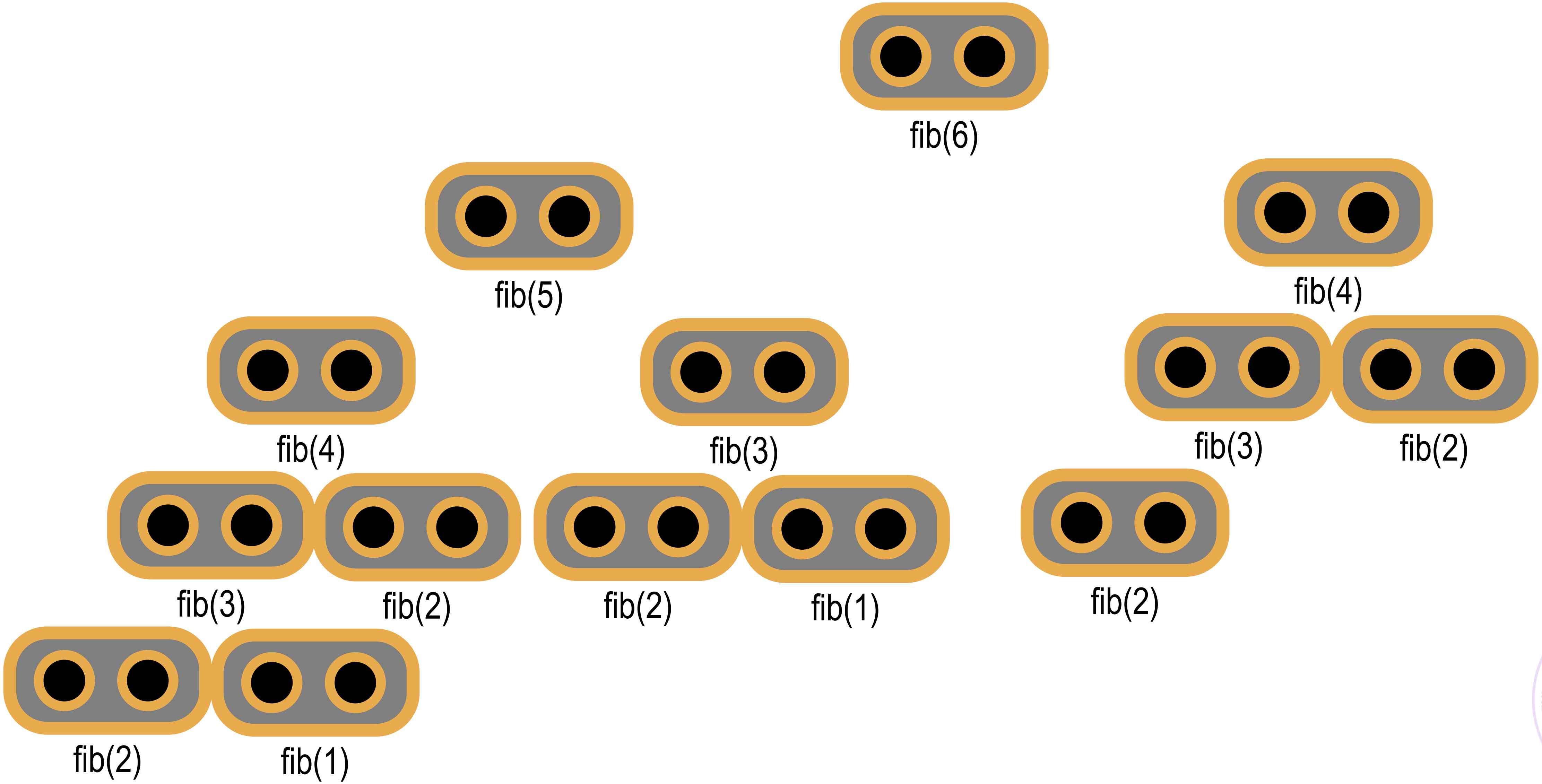
Attack of the Clones



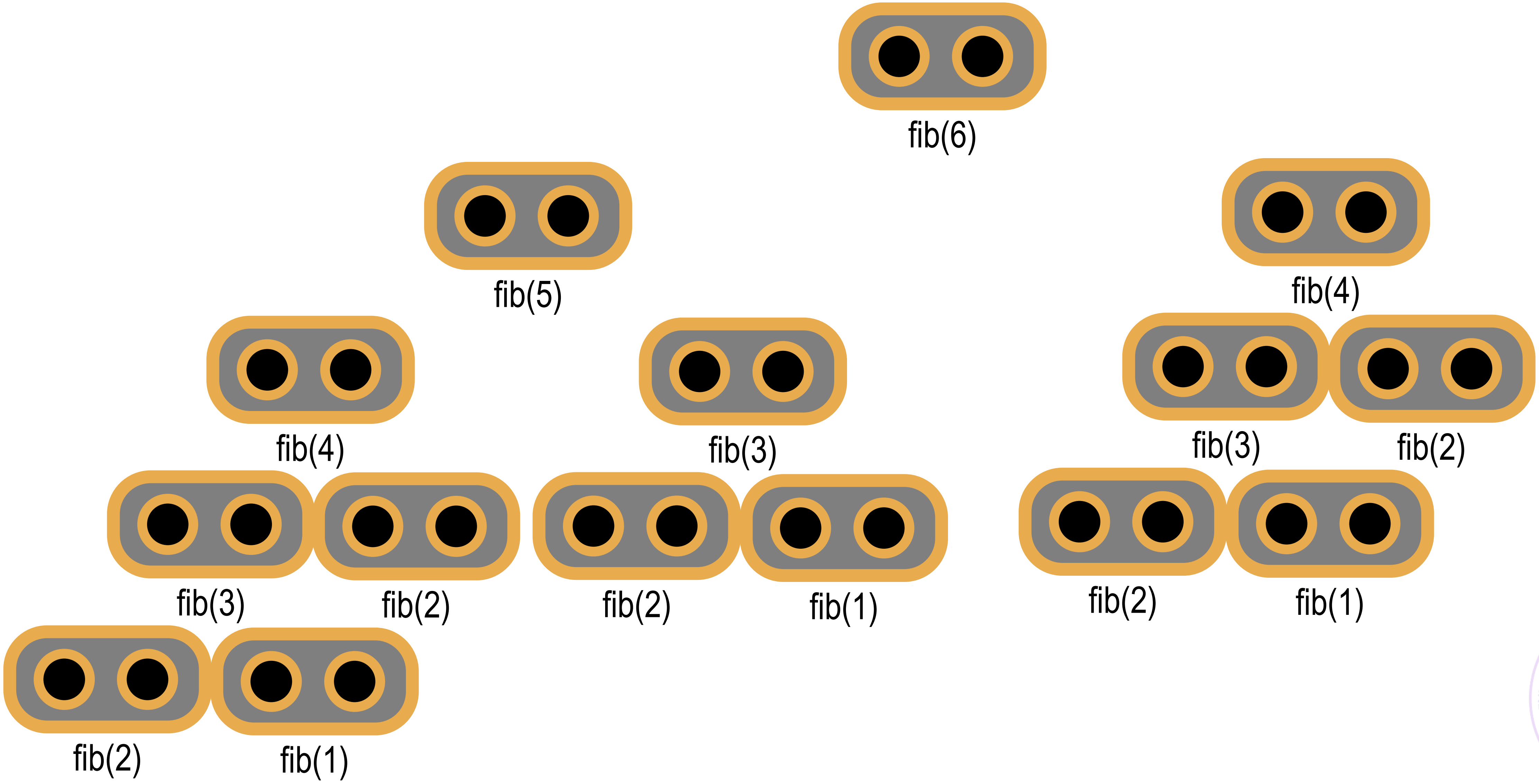
Attack of the Clones



Attack of the Clones

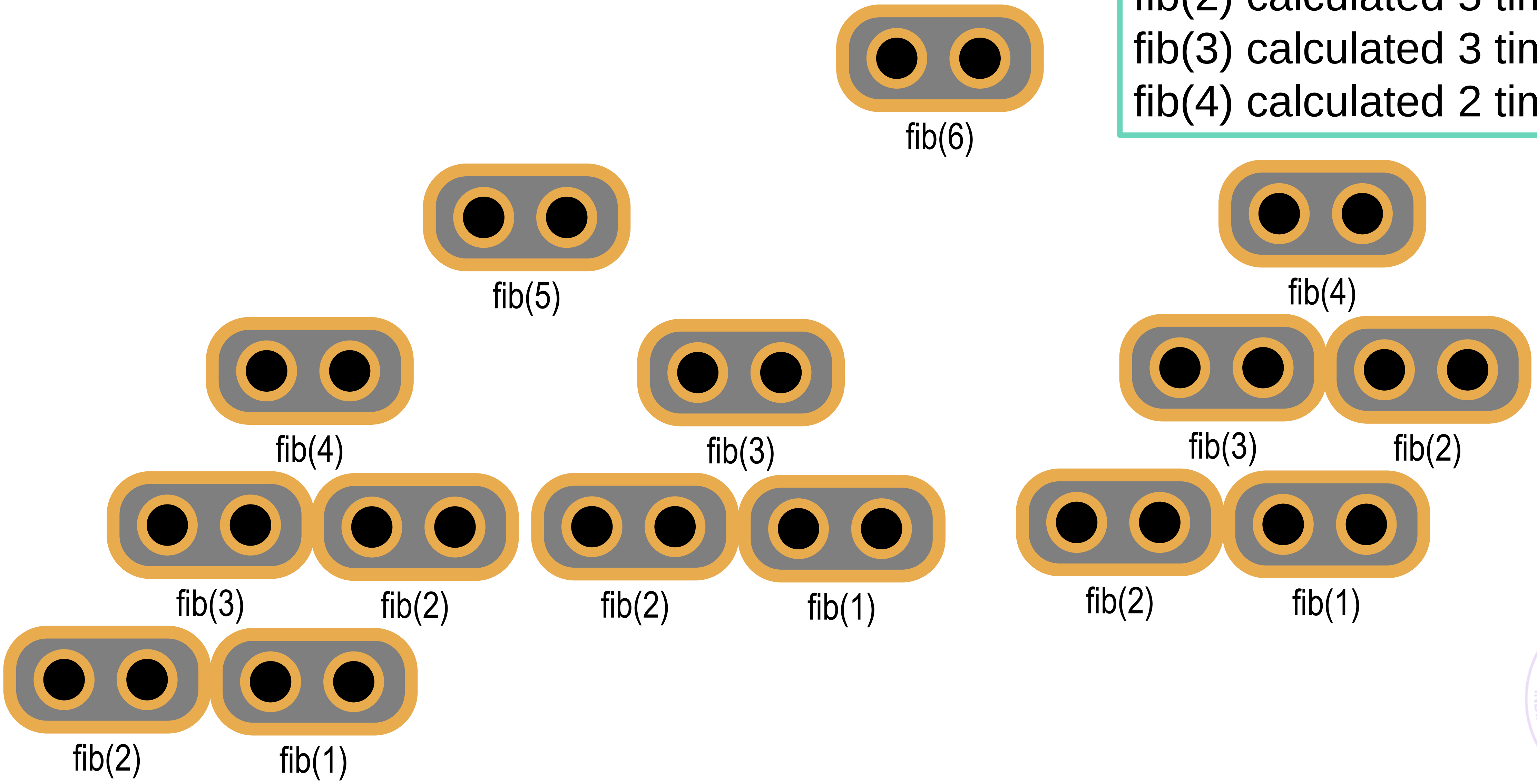


Attack of the Clones



Attack of the Clones

fib(1) calculated 3 times
fib(2) calculated 5 times
fib(3) calculated 3 times
fib(4) calculated 2 times



Greatest common divisor

Greatest common divisor

Sometimes recursion can make code faster too ☺

Greatest common divisor

Sometimes recursion can make code faster too ☺

One of the fastest algorithms for computing gcd is called the Euclid's algorithm and it defines gcd recursively!

Greatest common divisor

Sometimes recursion can make code faster too ☺

One of the fastest algorithms for computing gcd is called the Euclid's algorithm and it defines gcd recursively!

Theorem 1. Suppose $a \geq b > 0$ are two numbers. Then we always have $\gcd(a, b) = \gcd(b, a \;\%\; b)$.

Greatest common divisor

Sometimes recursion can make code faster too ☺

One of the fastest algorithms for computing gcd is called the Euclid's algorithm and it defines gcd recursively!

Theorem 1. Suppose $a \geq b > 0$ are two numbers. Then we always have $\gcd(a, b) = \gcd(b, a \,\%\, b)$.

Note that since a % b is always less than b, this indeed defines gcd in terms of gcd on "smaller" inputs

Greatest common divisor

Sometimes recursion can make code faster too ☺

One of the fastest algorithms for computing gcd is called the Euclid's algorithm and it defines gcd recursively!

Theorem 1. Suppose $a \geq b > 0$ are two numbers. Then we always have $\gcd(a, b) = \gcd(b, a \;\%\; b)$.

Note that since a % b is always less than b, this indeed defines gcd in terms of gcd on "smaller" inputs

What is the base case here?

Greatest common divisor

Sometimes recursion can make code faster too ☺

One of the fastest algorithms for computing gcd is called the Euclid's algorithm and it defines gcd recursively!

Theorem 1. Suppose $a \geq b > 0$ are two numbers. Then we always have $\gcd(a, b) = \gcd(b, a \ \% \ b)$.

Note that since a % b is always less than b, this indeed defines gcd in terms of gcd on "smaller" inputs

What is the base case here?

When b divides a i.e. when a % b = 0, then we have gcd(a, b) = b ☺

Greatest common divisor

Sometimes recursion can make code faster too ☺

One of the fastest algorithms for computing gcd is called the Euclid's algorithm and it defines gcd recursively!

Theorem 1. Suppose $a \geq b > 0$ are two numbers. Then we always have $\gcd(a, b) = \gcd(b, a \;\%\; b)$.

Note that since a % b is always less than b, this indeed defines gcd in terms of gcd on "smaller" inputs

What is the base case here?

When b divides a i.e. when a % b = 0, then we have gcd(a, b) = b ☺

We will prove this theorem in the next slide

The GCD Theorem

The GCD Theorem

Theorem 1. Suppose $a \geq b > 0$ are two numbers. Then we always have $\gcd(a, b) = \gcd(b, a \% b)$.

The GCD Theorem

Theorem 1. Suppose $a \geq b > 0$ are two numbers. Then we always have $\gcd(a, b) = \gcd(b, a \,\%\, b)$.

The theorem follows from the following two lemmata

The GCD Theorem

Theorem 1. Suppose $a \geq b > 0$ are two numbers. Then we always have $\gcd(a, b) = \gcd(b, a \% b)$.

The theorem follows from the following two lemmata

Lemma 1. Suppose $a \geq b > 0$ are two numbers. Then we always have $\gcd(a, b) = \gcd(a - b, b)$.

Lemma 2. Suppose $p, q > 0$ are two numbers. Then we always have $\gcd(p, q) = \gcd(q, p)$.

The GCD Theorem

Theorem 1. Suppose $a \geq b > 0$ are two numbers. Then we always have $\gcd(a, b) = \gcd(b, a \;\%\; b)$.

The theorem follows from the following two lemmata

Lemma 1. Suppose $a \geq b > 0$ are two numbers. Then we always have $\gcd(a, b) = \gcd(a - b, b)$.

Lemma 2. Suppose $p, q > 0$ are two numbers. Then we always have $\gcd(p, q) = \gcd(q, p)$.

We will first assume these lemmata and prove Theorem 1 and then prove Lemma 1.

The GCD Theorem

Theorem 1. Suppose $a \geq b > 0$ are two numbers. Then we always have $\gcd(a, b) = \gcd(b, a \,\%\, b)$.

The theorem follows from the following two lemmata

Lemma 1. Suppose $a \geq b > 0$ are two numbers. Then we always have $\gcd(a, b) = \gcd(a - b, b)$.

Lemma 2. Suppose $p, q > 0$ are two numbers. Then we always have $\gcd(p, q) = \gcd(q, p)$.

We will first assume these lemmata and prove Theorem 1 and then prove Lemma 1.

The proof of Lemma 2 is left as an easy exercise.

Proof of Theorem 1

Proof of Theorem 1

Theorem 1. Suppose $a \geq b > 0$ are two numbers. Then we always have $\gcd(a, b) = \gcd(b, a \; \% \; b)$.

Proof of Theorem 1

Theorem 1. Suppose $a \geq b > 0$ are two numbers. Then we always have $\gcd(a, b) = \gcd(b, a \,\%\, b)$.

Suppose k is the largest positive integer s.t. $a - k \cdot b \geq 0$

Proof of Theorem 1

Theorem 1. Suppose $a \geq b > 0$ are two numbers. Then we always have $\gcd(a, b) = \gcd(b, a \;\%\; b)$.

Suppose k is the largest positive integer s.t. $a - k \cdot b \geq 0$

Then repeatedly applying lemma 1 gives us

Proof of Theorem 1

Theorem 1. Suppose $a \geq b > 0$ are two numbers. Then we always have $\gcd(a, b) = \gcd(b, a \ \% \ b)$.

Suppose k is the largest positive integer s.t. $a - k \cdot b \geq 0$

Then repeatedly applying lemma 1 gives us

gcd(a,b) = gcd(a – b, b) = gcd(a - 2b, b) = gcd(a – 3b, b) = … = gcd(a – k * b, b)

Proof of Theorem 1

Theorem 1. Suppose $a \geq b > 0$ are two numbers. Then we always have $\gcd(a, b) = \gcd(b, a \ \% \ b)$.

Suppose k is the largest positive integer s.t. $a - k \cdot b \geq 0$

Then repeatedly applying lemma 1 gives us

gcd(a,b) = gcd(a – b, b) = gcd(a - 2b, b) = gcd(a – 3b, b) = … = gcd(a – k * b, b)

However, by the definition of k, a – k * b = a % b ☺

Proof of Theorem 1

Theorem 1. Suppose $a \geq b > 0$ are two numbers. Then we always have $\gcd(a, b) = \gcd(b, a \mathbin{\%} b)$.

Suppose k is the largest positive integer s.t. $a - k \cdot b \geq 0$

Then repeatedly applying lemma 1 gives us

gcd(a,b) = gcd(a – b, b) = gcd(a - 2b, b) = gcd(a – 3b, b) = … = gcd(a – k * b, b)

However, by the definition of k, a – k * b = a % b ☺

Now applying Lemma 2 gives us

Proof of Theorem 1

Theorem 1. Suppose $a \geq b > 0$ are two numbers. Then we always have $\gcd(a, b) = \gcd(b, a \ \% \ b)$.

Suppose k is the largest positive integer s.t. $a - k \cdot b \geq 0$

Then repeatedly applying lemma 1 gives us

gcd(a,b) = gcd(a – b, b) = gcd(a - 2b, b) = gcd(a – 3b, b) = … = gcd(a – k * b, b)

However, by the definition of k, a – k * b = a % b ☺

Now applying Lemma 2 gives us

gcd(a,b) = gcd(a % b, b) = gcd(b, a % b)

Proof of Theorem 1

Theorem 1. Suppose $a \geq b > 0$ are two numbers. Then we always have $\gcd(a, b) = \gcd(b, a \ \% \ b)$.

Suppose k is the largest positive integer s.t. $a - k \cdot b \geq 0$

Then repeatedly applying lemma 1 gives us

gcd(a,b) = gcd(a – b, b) = gcd(a - 2b, b) = gcd(a – 3b, b) = … = gcd(a – k * b, b)

However, by the definition of k, a – k * b = a % b ☺

Now applying Lemma 2 gives us

gcd(a,b) = gcd(a % b, b) = gcd(b, a % b)

This proves Theorem 1. QED (quod erat demonstrandum)

Proof of Lemma 1

Proof of Lemma 1

Lemma 1. Suppose $a \geq b > 0$ are two numbers. Then we always have $\gcd(a, b) = \gcd(a - b, b)$.

Proof of Lemma 1

Lemma 1. Suppose $a \geq b > 0$ are two numbers. Then we always have $\gcd(a, b) = \gcd(a - b, b)$.

Suppose gcd(a, b) = g > 0 and say, a = g * m, b = g * n

Proof of Lemma 1

Lemma 1. Suppose $a \geq b > 0$ are two numbers. Then we always have $\gcd(a, b) = \gcd(a - b, b)$.

Suppose gcd(a, b) = g > 0 and say, a = g * m, b = g * n

Then m and n must be coprime. If m, n have a common factor other than 1 then g will not be the gcd of a and b

Proof of Lemma 1

Lemma 1. Suppose $a \geq b > 0$ are two numbers. Then we always have $\gcd(a, b) = \gcd(a - b, b)$.

Suppose gcd(a, b) = g > 0 and say, a = g * m, b = g * n

Then m and n must be coprime. If m, n have a common factor other than 1 then g will not be the gcd of a and b

E.g., if m and n have 2 as common factor i.e. m = 2 * r and n = 2 * s, then this means that a = 2g * r and b = 2g * s i.e. 2g is a larger common factor of a, b

Proof of Lemma 1

Lemma 1. Suppose $a \geq b > 0$ are two numbers. Then we always have $\gcd(a, b) = \gcd(a - b, b)$.

Suppose gcd(a, b) = g > 0 and say, a = g * m, b = g * n

Then m and n must be coprime. If m, n have a common factor other than 1 then g will not be the gcd of a and b

E.g., if m and n have 2 as common factor i.e. m = 2 * r and n = 2 * s, then this means that a = 2g * r and b = 2g * s i.e. 2g is a larger common factor of a, b

The proof is compete after noticing that a – b = (m – n) * g and that (m – n) and n must be coprime which means g must be gcd of (m – n)* g and n * g i.e. gcd of (a-b) and b

Partitions

Partitions

Partitions of a number are the different ways in which we can write the number as a sum of smaller numbers

Partitions

Partitions of a number are the different ways in which we can write the number as a sum of smaller numbers

For example, the partitions of 4 are

Partitions

Partitions of a number are the different ways in which we can write the number as a sum of smaller numbers

For example, the partitions of 4 are

1 + 1 + 1 + 1

Partitions

Partitions of a number are the different ways in which we can write the number as a sum of smaller numbers

For example, the partitions of 4 are

1 + 1 + 1 + 1

1 + 1 + 2

Partitions

Partitions of a number are the different ways in which we can write the number as a sum of smaller numbers

For example, the partitions of 4 are

1 + 1 + 1 + 1

1 + 1 + 2

1 + 3

Partitions

Partitions of a number are the different ways in which we can write the number as a sum of smaller numbers

For example, the partitions of 4 are

1 + 1 + 1 + 1

1 + 1 + 2

1 + 3

2 + 2

Partitions

Partitions of a number are the different ways in which we can write the number as a sum of smaller numbers

For example, the partitions of 4 are

1 + 1 + 1 + 1

1 + 1 + 2

1 + 3

2 + 2

4

Partitions

Partitions of a number are the different ways in which we can write the number as a sum of smaller numbers

For example, the partitions of 4 are

1 + 1 + 1 + 1

1 + 1 + 2

1 + 3

2 + 2

4

Partitions

Partitions of a number are the different ways in which we can write the number as a sum of smaller numbers

For example, the partitions of 4 are

1 + 1 + 1 + 1

1 + 1 + 2

1 + 3

2 + 2

4

Note that these are all the partitions of 3

Partitions

Partitions of a number are the different ways in which we can write the number as a sum of smaller numbers

For example, the partitions of 4 are

1 + 1 + 1 + 1

1 + 1 + 2

1 + 3

2 + 2

4

Note that these are all the partitions of 3

We can generate partitions of n using partitions of n-1 ☺

Need to be a bit careful to ensure that we do not repeat partitions i.e. we do not write both 1 + 3 and 3 + 1 since they are the same partition

Partitions

Partitions of a number are the different ways in which we can write the number as a sum of smaller numbers

For example, the partitions of 4 are

1 + 1 + 1 + 1

1 + 1 + 2

1 + 3

2 + 2

4

Note that these are all the partitions of 3

Easy way of ensuring this – make sure that numbers are writing in increasing order so that 3 + 1 is disqualified

We can generate partitions of n using partitions of n-1 ☺

- Need to be a bit careful to ensure that we do not repeat partitions i.e. we do not write both 1 + 3 and 3 + 1 since they are the same partition