Recursion

ESC101: Fundamentals of Computing

Nisheeth

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 *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

Mutual Recursion

Two functions calling each other in a recursive fashion

```
Even(n) = (n == 0) || Odd(n-1)
Odd(n)  = (n != 0) && Even(n-1)
```

Note: The above example is not the most efficient way to check if a number of even or add but just an illustration of mutual recursion

About Recursion

fac(n) = n*fac(n-1);

- 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
  - Advantage: loop solutions can be much faster (at least in compiled languages like C) than the recursion solution

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

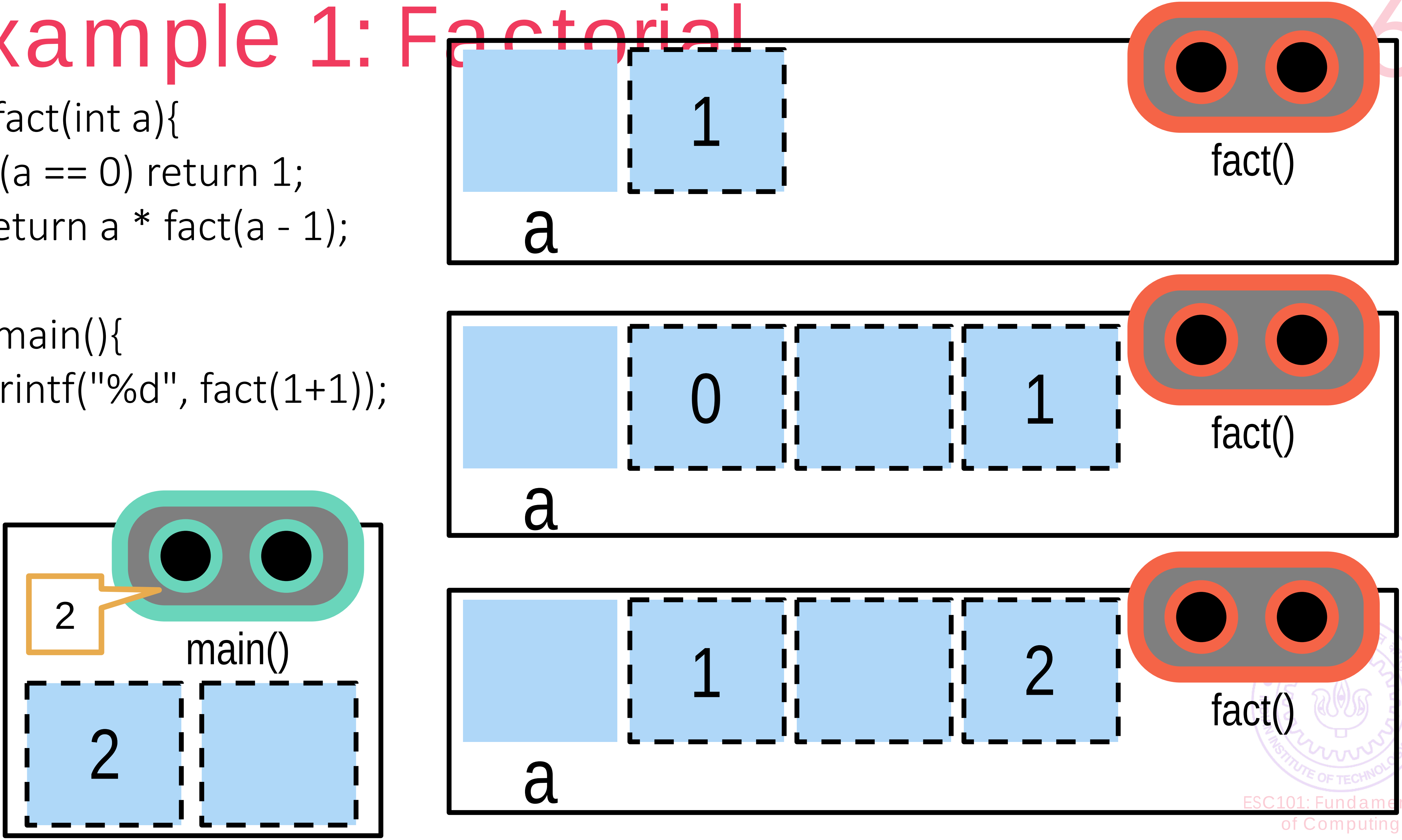
No small set of golden rules on how to find out when and if a problem can be solved using recursion

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

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

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(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 * 1 = 1

fact(2) = 2 * 1 = 2

fact(3) = 3 * 2 = 6

main()

720

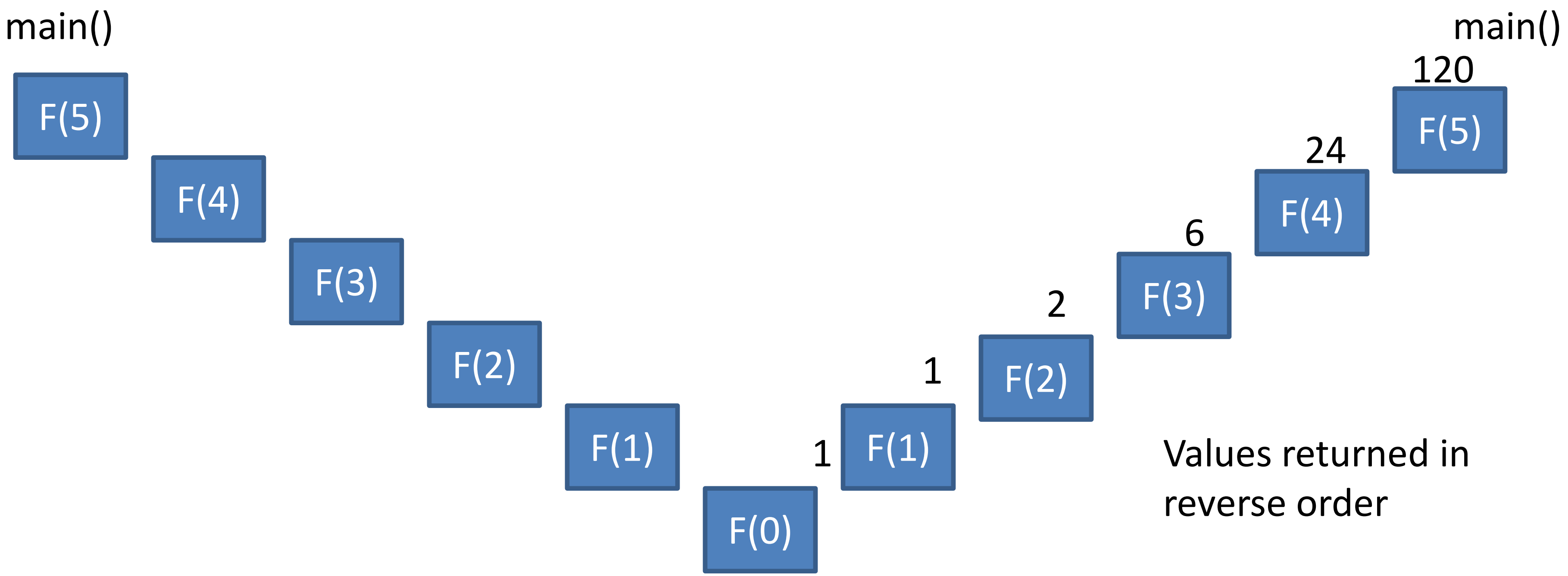
fact(6) = 6 * 120 = 720

fact(5) = 5 * 24 = 120

fact(4) = 4 * 6 = 24

Factorial: The Flow

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



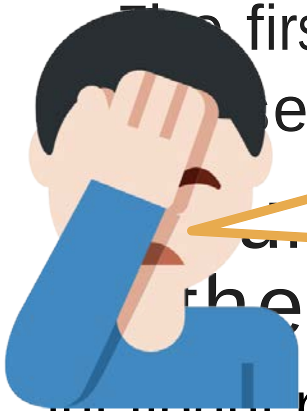
Imagine the number of clones to calculate fib(100). **Challenge**: don't imagine – find out

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

Explosion of clones!

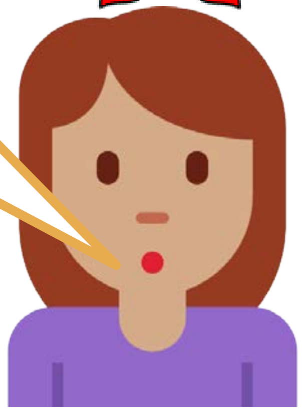
here are two base cas ... umbers

first Fibonacci number is defined



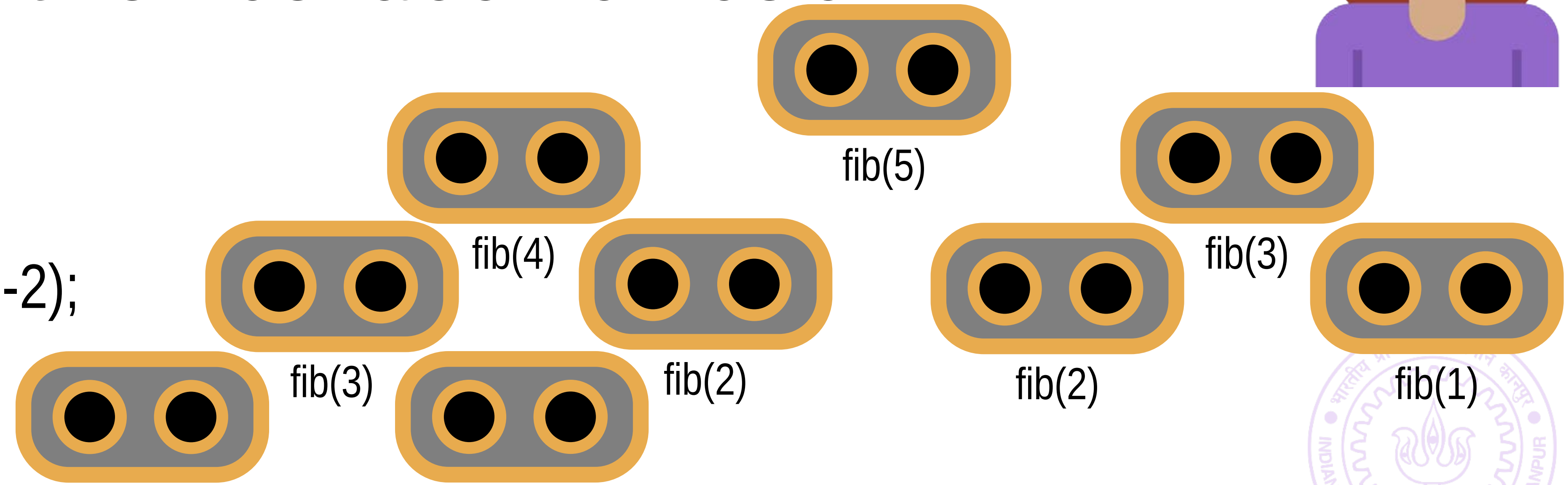
I could have easily solved this problem using a for loop – much faster and no clones

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



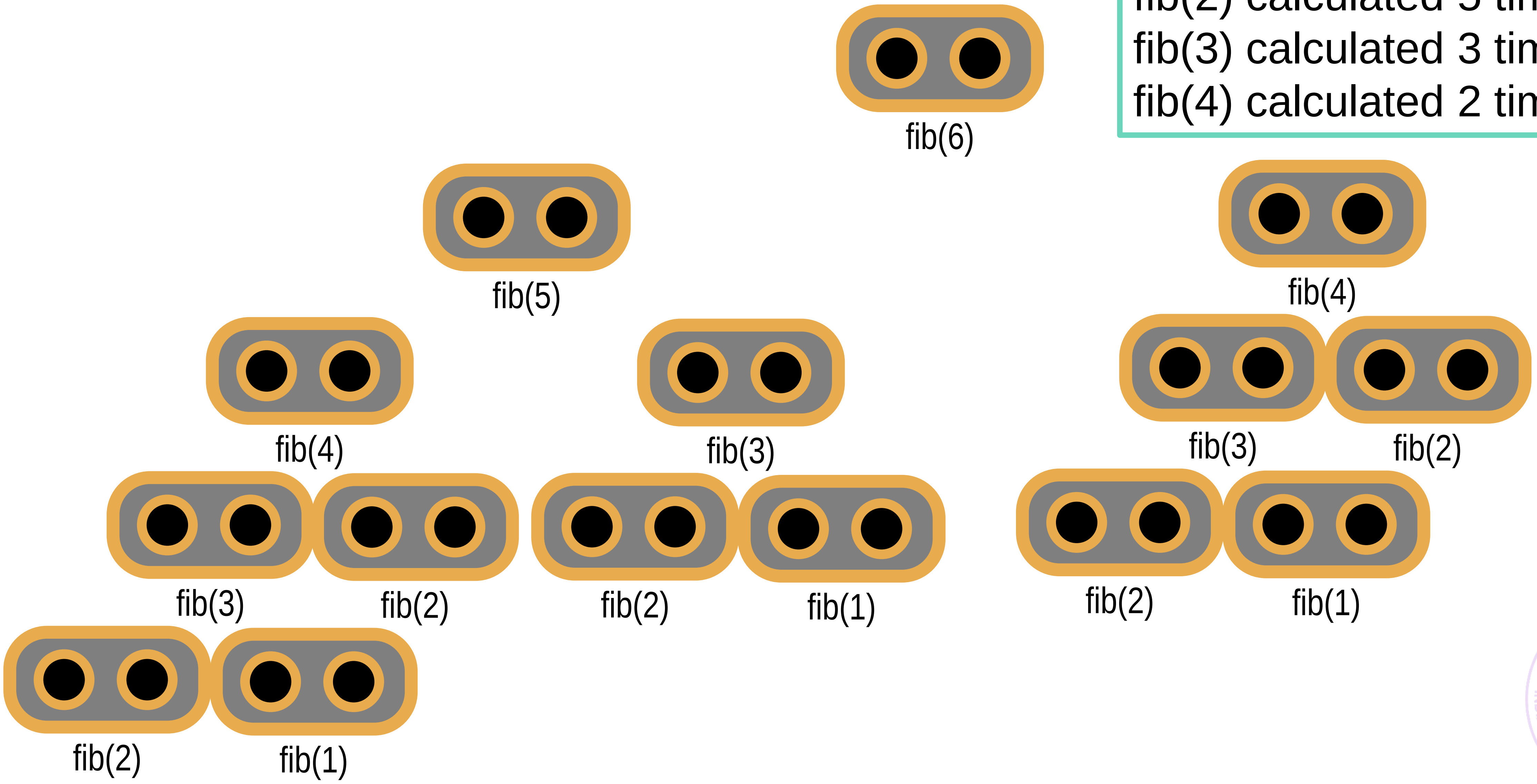
the previous two Fibonacci numbers

```
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));
}
```



Attack of the Clones

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



Recursion vs Iteration

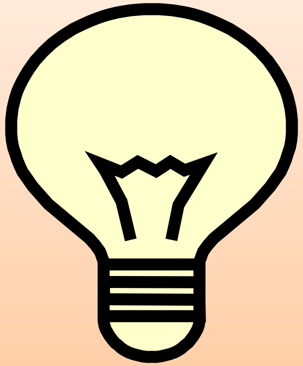
Write a function to compute
the n-th Fibonacci number

```
int fib(int n)
{
   int first = 0, second = 1;
   int next, c;
   if (n <= 1)
      return n;
   for ( c = 1; c < n ; c++ ) {
      next = first + second;
      first = second;
      second = next;
   }
   return next;
}
```

The recursive program is closer to the definition and easier to read.

But very very inefficient

```
int fib(int n)
{
   if ( n <= 1 )
      return n;
   else
      return fib(n-1) + fib(n-2);
}
```



Space complexity of recursion

- Every time a recursive function makes a call to itself
  - Another call to the function is placed in program memory
  - This memory space can no longer be allocated elsewhere
- The amount of memory needed to execute a program is called its space complexity
- Iterative programs' space complexity is relatively easy to analyse
  - It does not change as a function of the inputs (mostly)
- Not so for recursive programs
  - Space complexity is a function of the maximum depth of the recursion that will be needed
  - Is input-dependent
  - Have to be careful about memory limitations when using recursive algorithms

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 ☺

GCD using Recursion


```
int gcd(int a, int b){
    if(a < b)
        return gcd(b, a);
    if(a % b == 0)
        return b;
    return gcd(b, a % b);
}
```

The base case

Convince yourself that this will work by calculating some of the outputs by hand.

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

Code for partitioning


```
void partition(char *str, int n, int next, int min){
        if(n == 0){
            str[next] = '\0';
            printf("%s\n", str);
            return;
        }
        int i;
        if(next)
            str[next++] = '+';
        for(i = min; i <= n; i++){
            str[next] = '0' + i;
            partition(str, n - i, next + 1, i);
        }
}
```

Base case is to terminate the string of numbers and print it

Base case returns

Print ‘+’ after each number

In increasing order, to avoid repeats

Dig down into the recursion well until n-i becomes 0, at which point the base case will return

partition(str, 4, 0, 1)
Output:
1+1+1+1
1+1+2
1+3
2+2
4