Databases and Algorithms

Recursion (basics)

Instructor: Rossano Schifanella

@SDS

Recursion (Basics)

- A recursive function contains a call to itself:

```
void countDown(int num) {
    if (num == 0)
     print("Blastoff!")
    else {
        print(num, "...")
        countDown(num-1) // recursive call
       }
}
```

Why Recursion?

- Recursive functions are used to reduce a complex problem to a simpler-to-solve problem.
- The simplest-to-solve problem is known as the **base case**
- Recursive calls stop when the base case is reached

Recursion vs. Iteration

- Benefits (+), disadvantages(-) for recursion:
  - \+ Models certain algorithms most accurately
  - \+ Results in shorter, simpler functions
  - \- May not execute very efficiently
- Benefits (+), disadvantages(-) for iteration:
  - \+ Executes more efficiently than recursion
  - \- Often is harder to code or understand

Stopping the Recursion

- A recursive function must always include a test to determine if another recursive call should be made, or if the recursion should stop with this call
- In the sample program, the test is:

  if (num == 0)

Stopping the Recursion

```
void countDown(int num) {
    if (num == 0)
        print("Blastoff!")
    else {
        print(num, "...")
        countDown(num-1);
    }
}
```

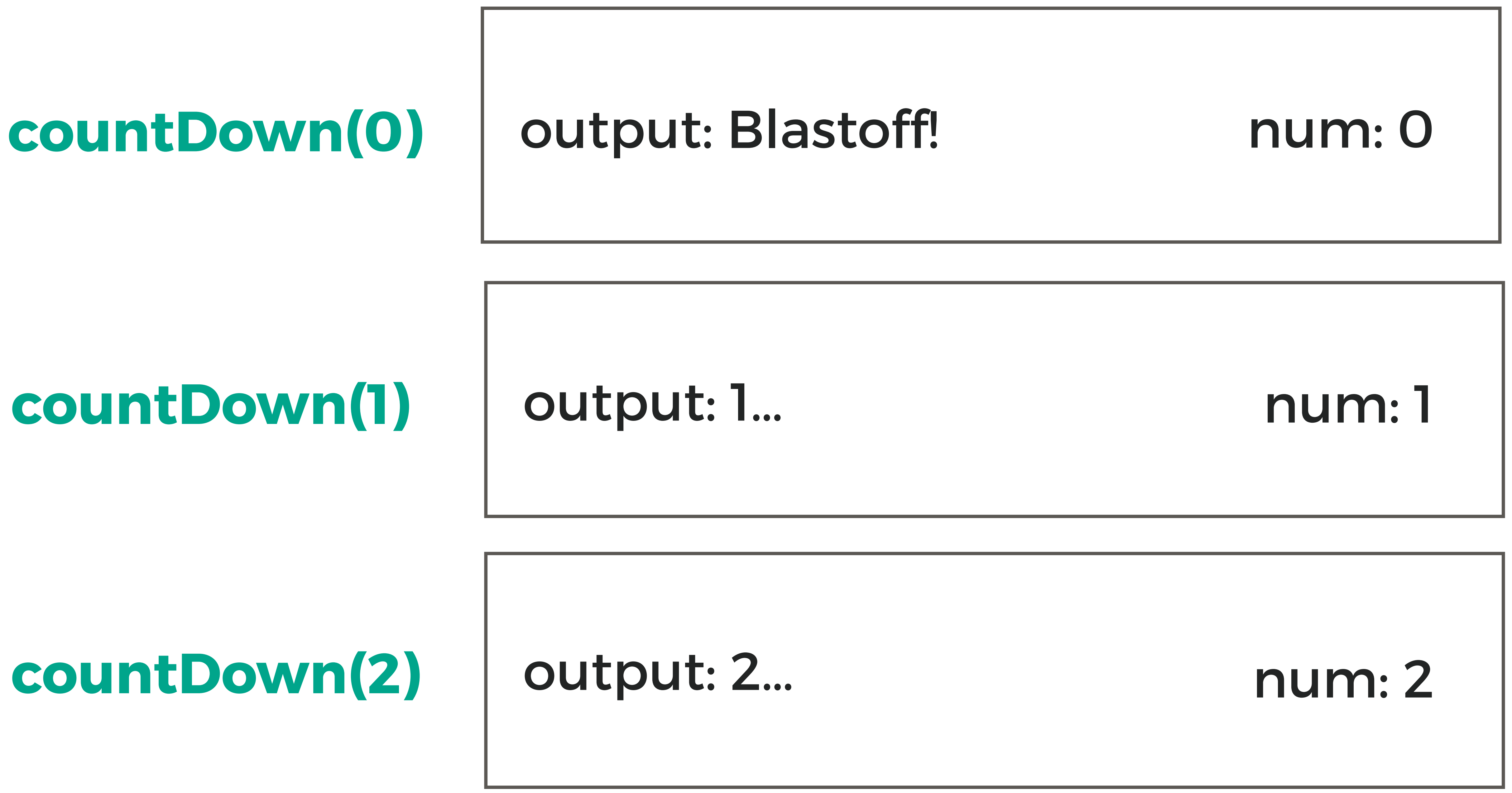
Stopping the Recursion

- Recursion uses a process of breaking a problem down into smaller problems until the problem can be solved
- In the `countDown` function, a different value is passed to the function each time it is called
- Eventually, the parameter reaches the value in the test, and the recursion stops

How It Works

- Each time a recursive function is called, a new copy of the function runs, with new instances of parameters and local variables created
- That is, a new **stack frame** is created
- As each copy finishes executing, it returns to the copy of the function that called it
- When the initial copy finishes executing, it returns to the part of the program that made the initial call to the function

Allocation stack



How It Works

- Remember, a new stack frame with new instances of local variables (and parameters) is created
- Thus the variable **num** in the “countdown” example is not the same memory location in each of the calls

Types of Recursion

- **Direct**
  - a function calls itself
- **Indirect**
  - function A calls function B, and function B calls function A
  - function A calls function B, which calls …, which calls function A

The Recursive Factorial Function

- The factorial function:
  - $n! = n*(n-1)*(n-2)*...*3*2*1$ if $n > 0$
  - $n! = 1$ if $n = 0$
- Can compute factorial of n if the factorial of (n-1) is known:
  - $n! = n * (n-1)!$
- n = 0 is the base case
  - Can you think of something different?

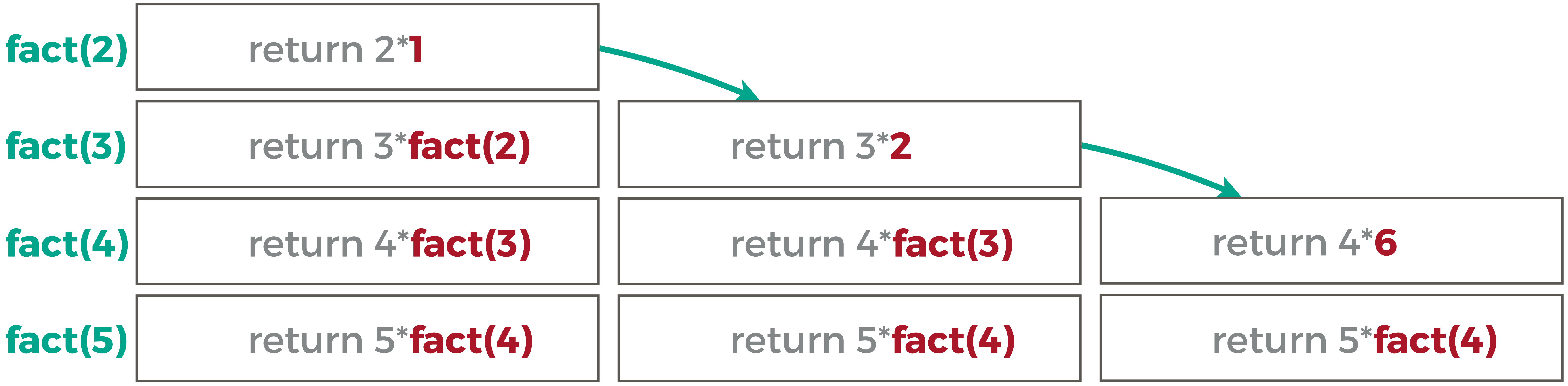
The Recursive Factorial Function

```
int factorial (int num) {
    if (num==1)
        return 1
    else
        return num * factorial(num - 1);
}
```

Allocation stack

fact(1)	return **1**
fact(2)	return 2***fact(1)**
fact(3)	return 3***fact(2)**
fact(4)	return 4***fact(3)**
fact(5)	return 5***fact(4)**

Allocation stack



Allocation stack

fact(4) return 4***6**

fact(5) return 5***fact(4)**

return 5***24 = 120**

Solving Recursively Defined Problems

- The natural definition of some problems leads to a recursive solution
- Example: Fibonacci numbers:
  - 0, 1, 1, 2, 3, 5, 8, 13, 21, ...
- After the starting 0, 1, each number is the sum of the two preceding numbers
- Recursive solution:
  - fib(n) = fib(n – 1) + fib(n – 2);
- Base cases: n <= 0, n == 1

Solving Recursively Defined Problems

```
int fib(int n) {
    if (n <= 0)
        return 0;
    else if (n == 1)
        return 1;
    else
        return fib(n – 1) + fib(n – 2);
}
```

Recursive Binary Search

- The binary search algorithm can easily be written to use recursion
- Base cases: desired value is found, or no more array elements to search
- Algorithm (array in ascending order):
  - If middle element of array segment is desired value, then done
  - Else, if the middle element is too large, repeat binary search in first half of array segment
  - Else, if the middle element is too small, repeat binary search on the second half of array segment

Recursive Binary Search

```
// initially called with low = 0, high = N-1
BinarySearch(A[0..N-1], value, low, high) {
  if (high < low)
      return not_found // value would be inserted at index "low"
  mid = (low + high) / 2
  if (A[mid] > value)
      return BinarySearch(A, value, low, mid-1)
  else if (A[mid] < value)
      return BinarySearch(A, value, mid+1, high)
  else
      return mid
}
```

Questions?

@rschifan

schifane@di.unito.it

http://www.di.unito.it/~schifane