Design and Analysis of Algorithms

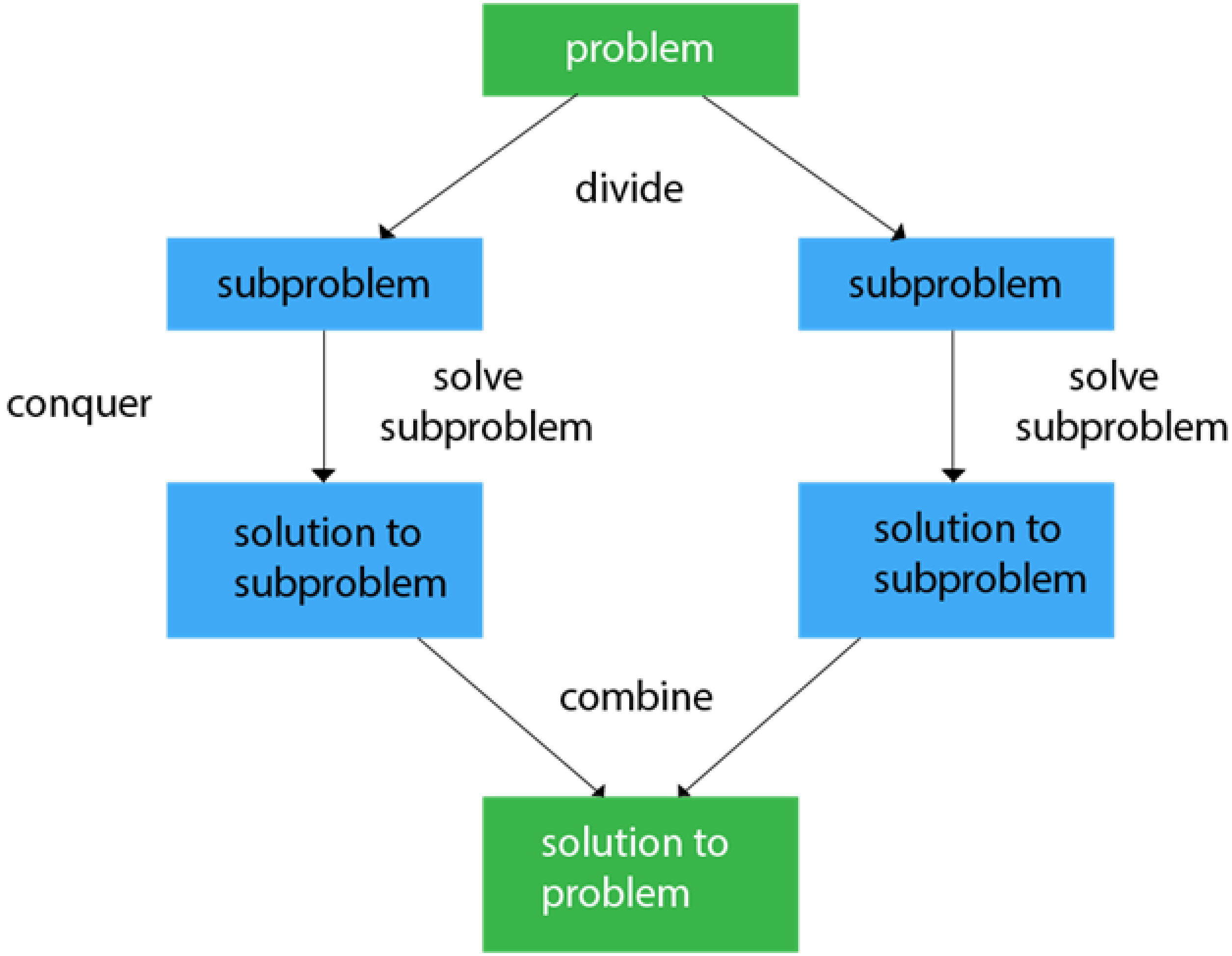
Recursion

National University of Computer and Emerging Sciences,
Islamabad

Divide-and-Conquer

- **Divide** the problem into a number of subproblems that are smaller instances of the
- same problem.
- **Conquer** the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.
- **Combine** the solutions to the subproblems into the solution for the original problem.

Divide and Conquer



Recursion (2)

- **To solve problem recursively**

```
1. Define the base case(s)
2. Define the recursive case(s)
   a) Divide the problem into smaller sub-problems
   b) Solve the sub-problems
   c) Combine results to get answer
```

Sub-problems solved as a recursive call to the same function

- **Sub-problem must be smaller than the original problem**
  - **Otherwise recursion never terminates**

Recursion

Recursion occurs when a function/procedure calls itself.

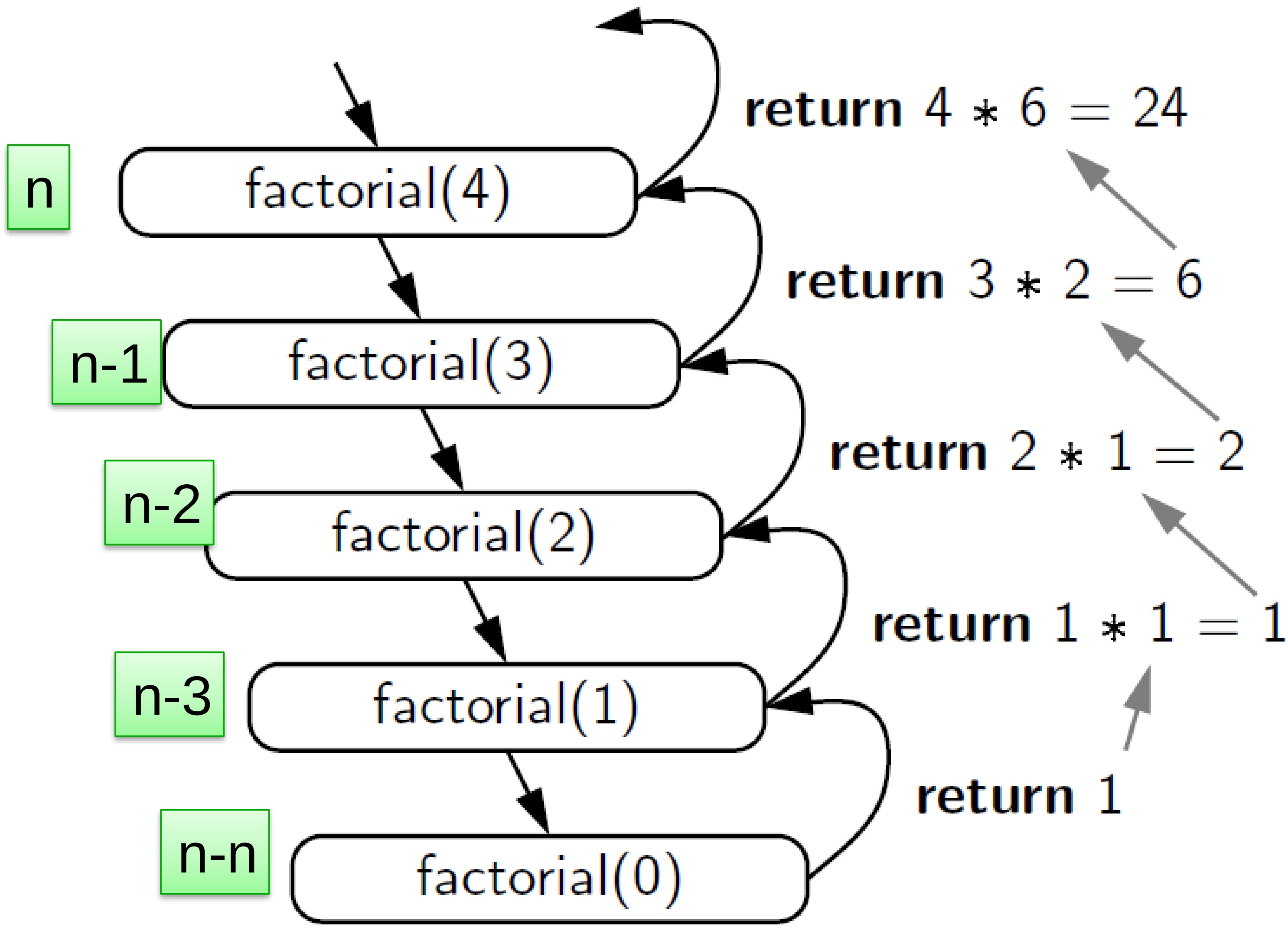
Many algorithms can be best described in terms of recursion.

Example: Factorial function

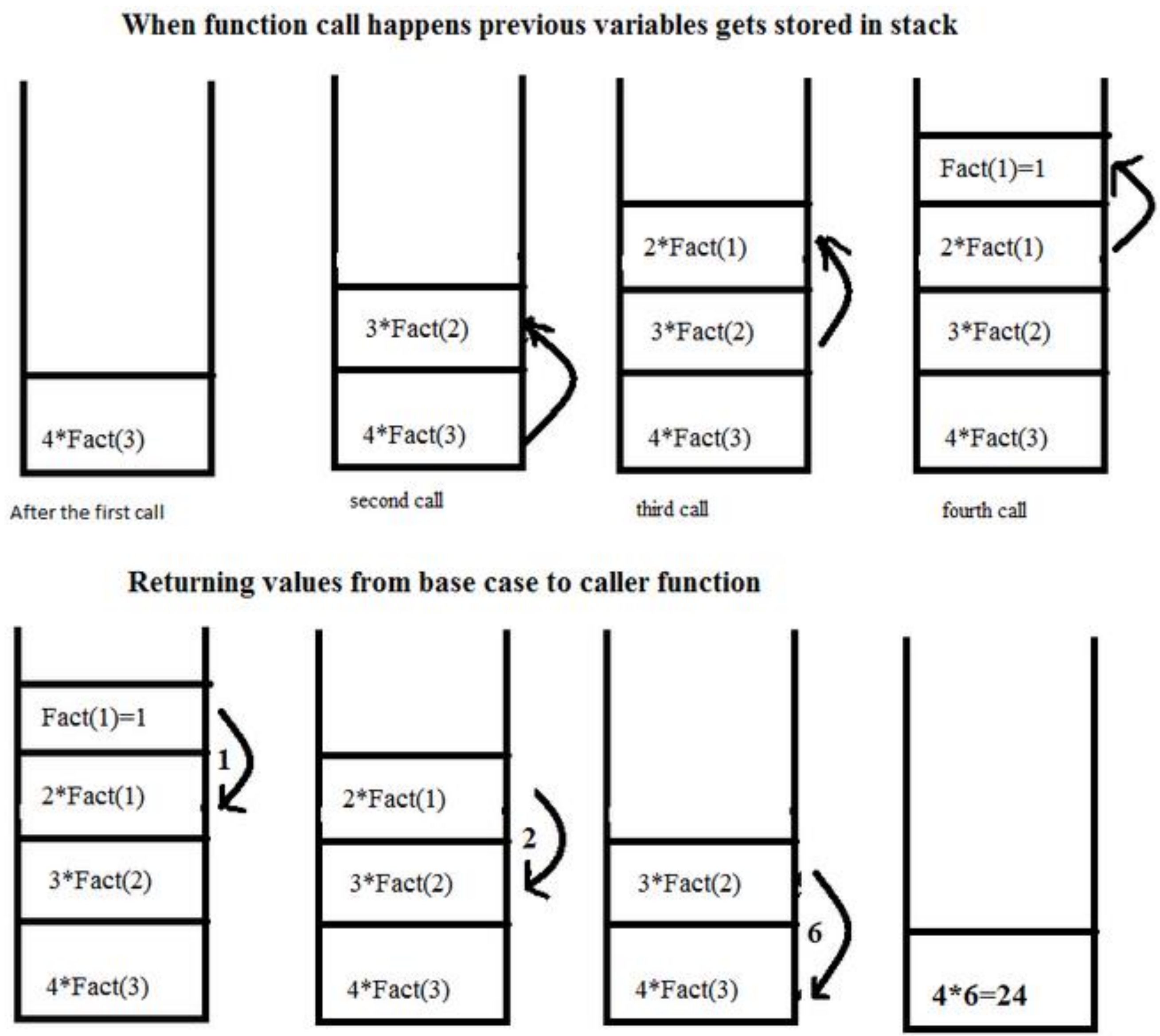
The product of the positive integers from *1* to *n* inclusive is called "**n factorial**", usually denoted by n!:

$$n! = 1 * 2 * 3 \ldots. (n-2) * (n-1) * n$$

Factorial function



Recursive Definition of the Factorial Function



Stack Overflow!

- **Recursive functions cannot use statically allocated local variables**
  - **Each instance of the function needs its own copies of local variables**

- **Most modern languages allocate local variables for functions on the run-time stack**

- **Calling a recursive function many times or with large arguments may result in stack overflow**

Recursive Definition of the Factorial Function

Recursive Definition of the Factorial Function

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

$5! = 5 * 4!$	$= 5 * 24 = 120$
$4! = 4 * 3!$	$= 4 * 3! = 4 * 6 = 24$
$3! = 3 * 2!$	$= 3 * 2! = 3 * 2 = 6$
$2! = 2 * 1!$	$= 2 * 1! = 2 * 1 = 2$
$1! = 1 * 0!$	$= 1 * 0! = 1$

Recursive Definition of the Fibonacci Numbers

The Fibonacci Sequence is the series of numbers:
1, 1, 2, 3, 5, 8, 13, 21, 34, ...

The Fibonacci numbers are a series of numbers as follows:

Recursive Definition of the Fibonacci Numbers

The Fibonacci Sequence is the series of numbers:
1, 1, 2, 3, 5, 8, 13, 21, 34, ...

The Fibonacci numbers are a series of numbers as follows:

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

$$fib(n) = \begin{cases} 1, & n <= 2 \\ fib(n-1) + fib(n-2), & n > 2 \end{cases}$$

fib(3) = 1 + 1 = 2
fib(4) = 2 + 1 = 3
fib(5) = 2 + 3 = 5

How do I write a recursive function?

- Determine the size factor
- Determine the base case(s)
  (the one for which you know the answer)
- Determine the general case(s)
  (the one where the problem is expressed as a smaller version of itself)
- Verify the algorithm
  (use the "Three-Question-Method")

Recursive Definition

```
int BadFactorial(n){
   int x = BadFactorial(n-1);
   if (n == 1)
      return 1;
   else
      return n*x;
}
```

What is the value of `BadFactorial(2)`?

We must make sure that recursion eventually stops, otherwise it runs forever:

Using Recursion Properly

For correct recursion we need two parts:

1. One (ore more) **base cases** that are not recursive, i.e. we can directly give a solution:
```
if (n==1)
    return 1;
```

2. One (or more) **recursive cases** that operate on smaller problems that get closer to the base case(s)
```
return n * factorial(n-1);
```

The base case(s) should always be checked before the recursive calls.

Counting Digits

- **19865 (5 Digit Number)**
- **386(3 Digit Number)**

Counting Digits

- **Recursive definition**

  digits(n) = 1 if (–9 <= n <= 9) Base Case

  1 + digits(n/10) otherwise Recursive case

- **Example**

  digits(321) =

  1 + digits(321/10) = 1 +digits(32) =

  1 + [1 + digits(32/10)] = 1 + [1 + digits(3)] =

  1 + [1 + (1)] =

  3

Counting Digits in C++

```
int numberofDigits(int n)
{
   if ((-10 < n) && (n < 10))
      return 1;
   else
      return 1 +
  numberofDigits(n/10);
}
```

Base Case

Recursive case

Evaluating Exponents Recursively

```
5 ^ 4 =?
power(5, 4)
```

Evaluating Exponents Recursively

```
int power(int k, int n) {
  // raise k to the power n
  if (n == 0)
    return 1;
  else
    return k * power(k, n - 1);
}
```

5 ^ 4 =?

power(5, 4)

625

5 * power(5, 3) = 5 ***5 ^ 3**

125

5 * power(5, 2) = 5 ***5 ^ 2**

25

5 * power(5, 1) = 5 ***5 ^ 1**

5

5 * power(5, 0) = 5*1

1

1

Evaluating Exponents Recursively

```
int power(int k, int n) {
  // raise k to the power n
  if (n == 0)
    return 1;
  else
    return k * power(k, n - 1);
}
```

Divide and Conquer

- Using this method each recursive subproblem is about one-half the size of the original problem
- If we could define `power` so that each subproblem was based on computing $k^{n/2}$ instead of k^{n-1} we could use the divide and conquer principle
- Recursive divide and conquer algorithms are often more efficient than iterative algorithms

Evaluating Exponents Using Divide and Conquer

```
int power(int k, int n) {
  // raise k to the power n
  if (n == 0)
    return 1;
  else{
    int t = power(k, n/2);
    if ((n % 2) == 0)
      return t * t;
    else
      return k * t * t;
}
```

Evaluating Exponents Using Divide and Conquer

5 ^ 4 =?

power(5, 4)

```
int power(int k, int n) {
  // raise k to the power n
  if (n == 0)
    return 1;
  else{
    int t = power(k, n/2);
    if ((n % 2) == 0)
      return t * t;
    else
      return k * t * t;
}
```

n=4,
k=5

power(5, **4**/2)
= power(5, 2)

n=4,
t=25

25*25=625

n=2,
k=5

power(5, **2**/2)
= power(5, 1)

5*5=25

n=2,
t=5

n=1,
k=5

power(5, **1**/2)
=power(5, **0)**

5*1*1

n=1,
t=1

n=0,
k=5

return 1

1

Disadvantages

- **May run slower.**
  - **Compilers**
  - **Inefficient Code**

- **May use more space.**

Advantages

- **More natural.**
- **Easier to prove correct.**
- **Easier to analyze.**
- **More flexible.**