



Introduction to Algorithms

Module Code: CELEN086

Lecture 3

(17/10/24)

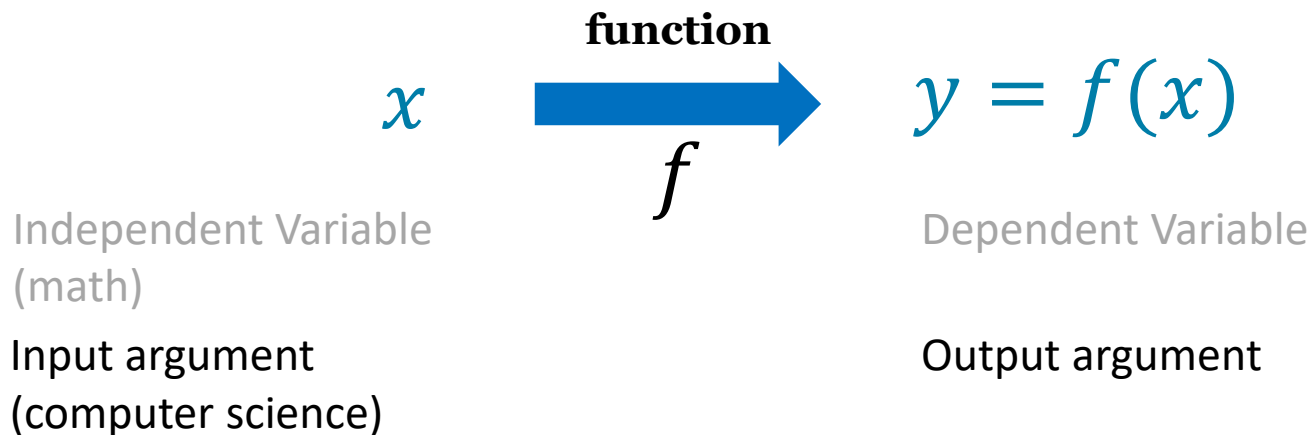
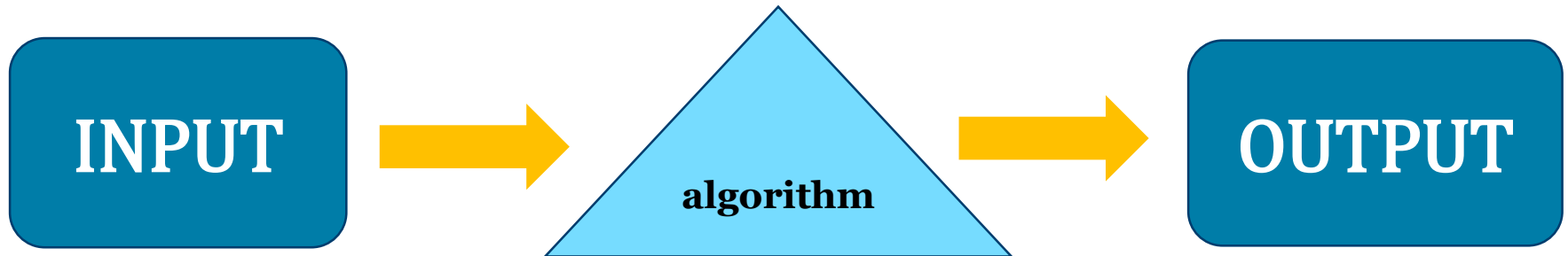
Lecturer & Convenor: Manish Dhyani

Email: Manish.Dhyani@nottingham.edu.cn

Topics to Cover Today

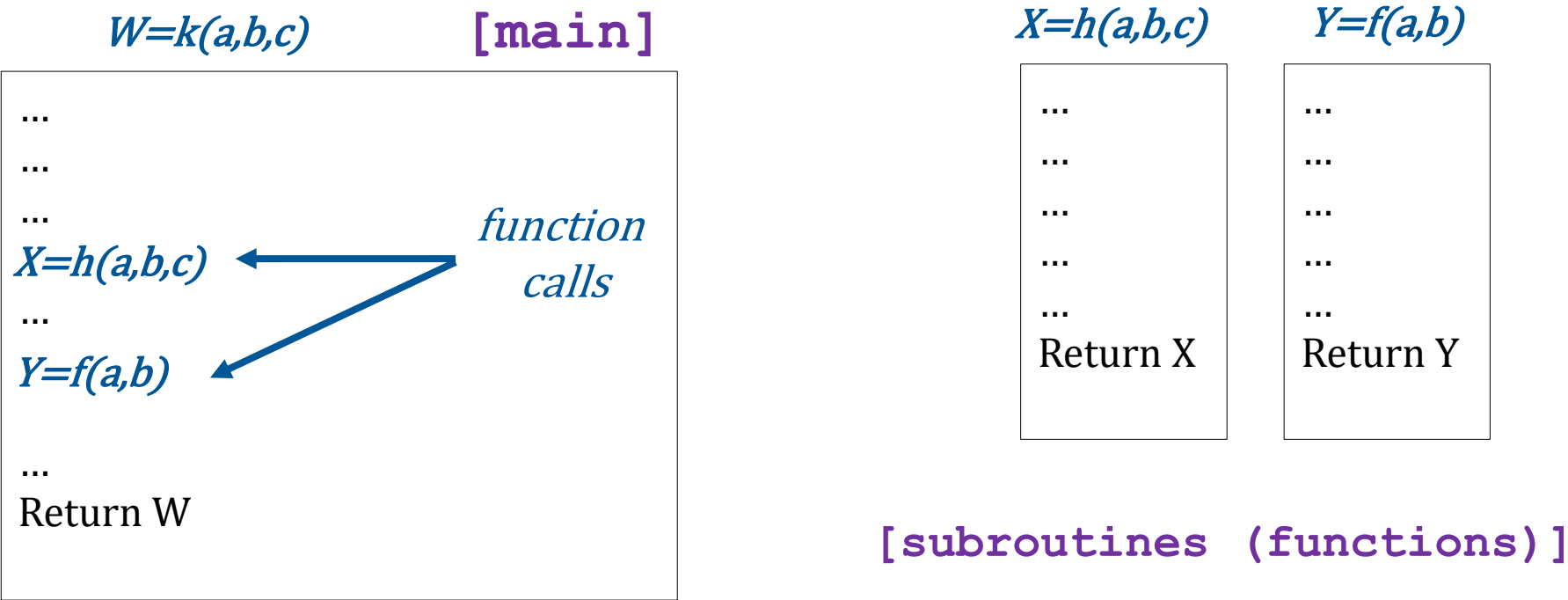
- ❖ Introduction to concept of recursion using factorial concept.
- ❖ Euclid Algorithm for finding GCD
- ❖ Defining and calling helper algorithm for recursive algorithm.
- ❖ Components of recursive algorithm

Algorithm vs. Function



Functional programming

In computer science, **functional programming** is a programming paradigm where programs are constructed by applying and composing functions.



What is Factorial?

The factorial of a non-negative integer n , written as $n!$, is the product of all positive integers from 1 to n .

- The factorial of n is

$$n \times (n - 1) \times (n - 2) \times \dots \times 2 \times 1$$

Example:

- $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$
- $3! = 3 \times 2 \times 1 = 6$

Special Case :

- $0!$ Is defined as 1.



Examples

$$\text{factorial}(0) = 1$$

$$\text{factorial}(1) = 1$$

$$\text{factorial}(2) = 2 \times 1 = 2$$

$$\text{factorial}(3) = 3 \times 2 \times 1 = 6$$

$$\text{factorial}(4) = 4 \times 3 \times 2 \times 1 = 24$$

$$\text{factorial}(5) = 5 \times 4 \times 3 \times 2 \times 1 = 120$$

➤ The factorial function counts

- the number of ways n people can form a queue.
- the number of anagrams of “Algorithm”.

Is it a good algorithm?

Algorithm MyFactorial(n)

Requires: A number n

Returns: The factorial of n

```
1: if  $n == 0$  then
2:   return 1
3: else if  $n == 1$  then
4:   return 1
5: else if  $n == 2$  then
6:   return 2 .....
7: endif
```

This is going nowhere!

We need a new ingredient!

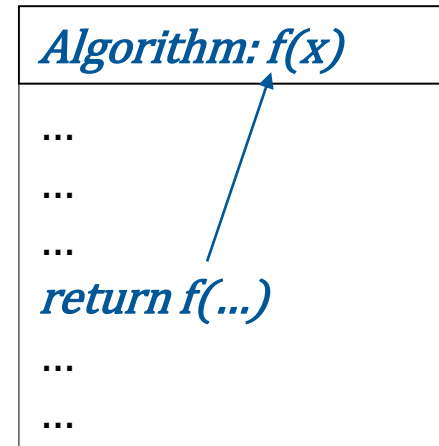
Recursion

An algorithm/function that **invokes/calls itself** is known as a **recursive algorithm/function** and this technique is known as **recursion**.

Sometimes a problem is difficult and complex to tackle head on.

If it can be divided into smaller versions instances (sub-problems) of the same problem, we may be able to conquer the subproblems and build them up to solve the entire problem.

(idea of Divide-and-Conquer technique)

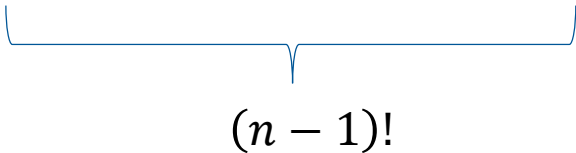


*$f(x)$ calls itself
inside its body structure*

Algorithm design

- Task: Let us design an algorithm (or a function $f(n)$) that computes the **factorial** $n!$ of a given non-negative integer number n .

Analysis:

$$n! = n \cdot (n-1) \cdot (n-2) \cdot \dots \cdot 2 \cdot 1,$$

$$(n-1)!$$

$$f(n) = n! = n \cdot (n-1)! = n \cdot f(n-1) \quad (\#)$$

We can reduce the problem computing $n!$ into a smaller one:

Computing $(n-1)!$

How to do it? Using **recursion**! Because we are repeating a similar routine, governed by the same formula (#) with $n-1$ replacing n :

$$f(n-1) = (n-1) \cdot f(n-2)$$

Algorithm design (Cont'd)

This is a process of **dividing** into smaller problem.

Yes! for the smallest problem, we already know the answer, which is the **base case**.

Does this process have an end?

$$\begin{array}{ccccccc}
 n! & \rightarrow & (n-1)! & \rightarrow & (n-2)! & \rightarrow & \dots & \rightarrow & 2! & \rightarrow & 1! & \rightarrow & 0! = 1 \\
 \xleftarrow{\times n} & & \xleftarrow{\times (n-1)} & & & & \xleftarrow{\times \dots} & & \xleftarrow{\times 2} & & \xleftarrow{\times 1} & &
 \end{array}$$

A process of **conquering** each smaller problem and building up the entire solution.

The formula $f(n) = n \cdot f(n-1)$ that governs the whole process is the **recursive formula**.

Algorithm: factorial(n)

Algorithm: **factorial**(n)

Requires: a non-negative integer n

Returns: n! the factorial of n

1. if n==0 then

2. return 1

3. else

4. return n***factorial**(n-1)

5. endif

base case

*recursive formula/
recursive step/ recursive call*

Line 4: this algorithm is calling itself, hence it is a recursive algorithm.

For each recursive call, the input argument is **decreased by 1**. It ensures that the recursive steps are approaching base case.

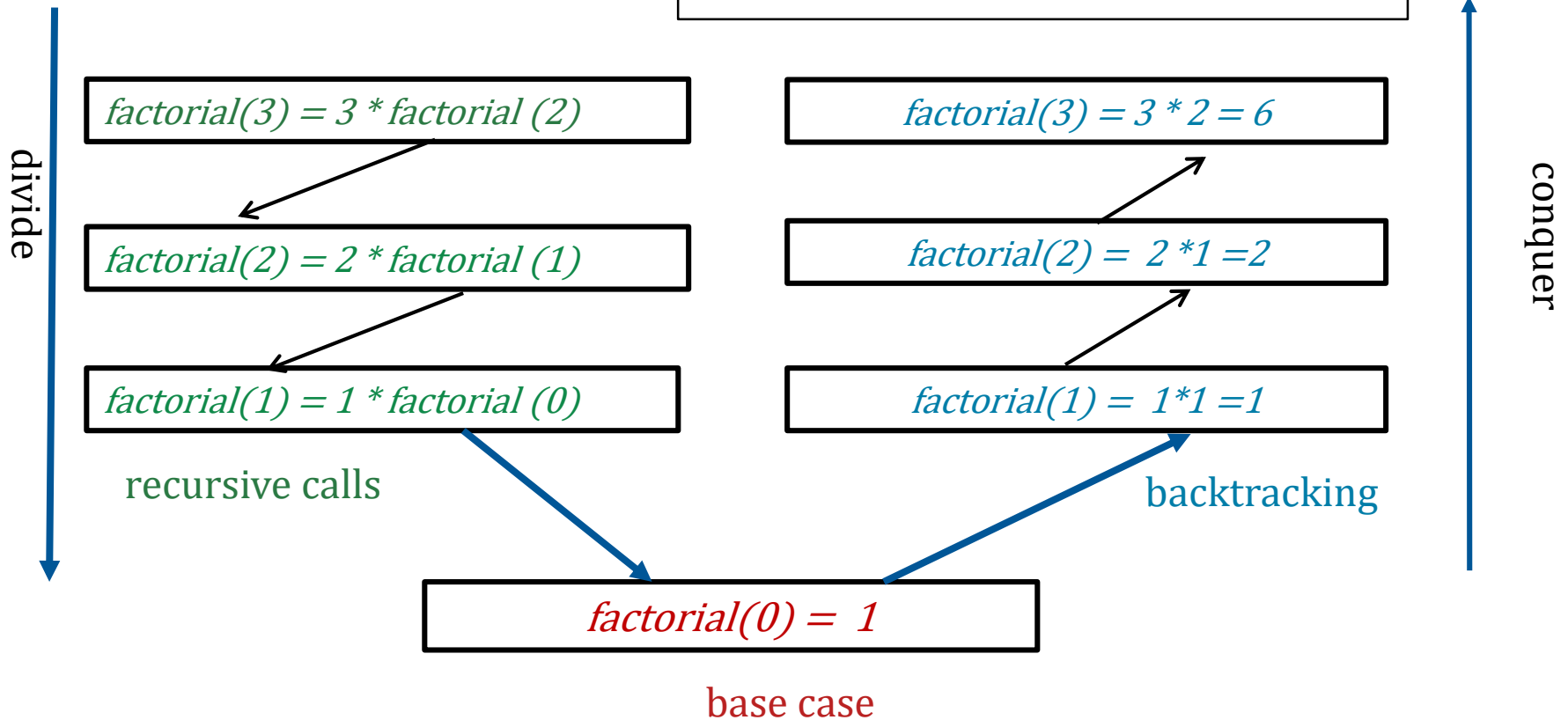
Note: a recursive algorithm can have more than one base cases, and more than one recursive calls.



Trace

$\text{factorial}(3) = 6$

```
1. if n==0
2.     return 1
3. else
4.     return n*factorial(n-1)
5. endif
```



Practise

Design an algorithm to compute the sum of digits of a positive integer n .

Expected Output:

For $n = 123$, the algorithm should return 6
(since $1+2+3=6$).

Algorithm: `sum_of_digits(n)`

Requires: a positive integer n

Returns: a positive integer i.e. sum of all the digits of n

Euclid's Algorithm

- Euclid's Algorithm is an efficient method for computing the **greatest common divisor (GCD)** of two integers, the largest number that divides them both without a remainder.
- It is one of the oldest algorithms in common use. Computations using this algorithm form part of the cryptographic protocols that are used to secure internet communications, and in methods for breaking these cryptosystems by factoring large composite numbers
- It is a **recursive** algorithm that finds the GCD simply and efficiently.

Greatest common divisor (GCD)

How to find the gcd of two numbers?

For example, to compute $\text{gcd}(12, 18)$

A trivial approach:

List all the divisors of 12: 1, 2, 3, 4, 6, 12

List all the divisors of 18: 1, 2, 3, 6, 9, 18

Common divisors of 12 and 18 are: 1, 2, 3, 6

The greatest common divisors of 12 and 18 is 6. i.e., $\text{gcd}(12, 18) = 6$

Greatest Common Divisor Properties

$$\text{gcd}(2022, 123450) = ?$$

We can observe that finding all divisors for large numbers are difficult.

In fact, Euclid's algorithm can solve the problem nicely, using the following properties of the GCD:

- **Property 1:** $\text{gcd}(x,y) = x$, if $x == y$;
- **Property 2:** $\text{gcd}(x,y) = \text{gcd}(x,y-x)$, if $y > x$;
- **Property 3:** $\text{gcd}(x,y) = \text{gcd}(x-y,y)$, if $x > y$.

For example:

$$\begin{aligned}\text{gcd}(12,18) &= \text{gcd}(12,18-12) \text{ using Property 2} \\ &= \text{gcd}(12,6) = \text{gcd}(12-6,6) \text{ using Property 3} \\ &= \text{gcd}(6,6) = 6. \text{ using Property 1}\end{aligned}$$



Properties of GCD

- **Property 1:** $\text{gcd}(x,y) = x$, if $x == y$;
- **Property 2:** $\text{gcd}(x,y) = \text{gcd}(x,y-x)$, if $y > x$;
- **Property 3:** $\text{gcd}(x,y) = \text{gcd}(x-y,y)$, if $x > y$.

$$\text{gcd}(2022, 123450) = ?$$

$$\text{gcd}(2022, 123450 - 2022) =$$

$$\text{gcd}(2022, 121428) =$$

$$\text{gcd}(2022, 119406) =$$

$$\text{gcd}(2022, 117384) =$$

$$\text{gcd}(2022, 115362) =$$

**More than 81 recursive call of gcd function
to get $\text{gcd}(2022, 123450) = 6$.**

$$\text{gcd}(6,6) = 6$$

Euclid's Algorithm

Algorithm: gcd(x,y)

Requires: two positive integer x and y

Returns: the greatest common divisor

```
1. if x == y  
2.     return x  
3. elseif x < y  
4.     return gcd(x, y-x)  
5. else  
6.     return gcd(x-y, y)  
7. end
```

base case

two recursive case

We should ask ourselves the following questions:

- do the recursive steps always approach the base case?
- to trace and check the algorithm thoroughly, how many different test cases do we need?

Improved Euclid's Algorithm

Algorithm: Euclid(x,y)

Requires: two positive integer x and y, $x \geq y$

Returns: the greatest common divisor

```
1. if y == 0
2.     return x // base case
3. else
4.     return Euclid(y, x mod y)
5. endif
```

*This algorithm requires that the first input argument is always **no less than** the second input argument.*

*You should make sure such requirement is always **satisfied** in all recursive steps.*

*Yes. Remainder $<$ Divisor.
Hence $y > (x \bmod y)$.*

Trace gcd(35,4) and Euclid(35,4), then you can find the latter is more efficient.

Euclid(35,4) x=35, y=4 Line 4: return Euclid(4, 35 mod 4)

Euclid(4,3) x=4, y=3 Line 4: return Euclid(3, 4 mod 3)

Euclid(3,1) x=3, y=1 Line 4: return Euclid(1, 0)

Euclid(1,0) x=1, y=0 Line 2: return 1 35 and 4 are Co-primes.

Improved Euclid's Algorithm

$$\text{gcd}(2022, 123450) = ?$$

How many Recursive call by using Euclid's Division Algorithm/Improved Euclid's Algorithm.

$$\text{gcd}(2022, 123450) =$$

6 recursive calls

$$\text{gcd}(123450, 2022) =$$

$$\text{gcd}(2022, 108) =$$

$$\text{gcd}(108, 78) =$$

$$\text{gcd}(78, 30) =$$

$$\text{gcd}(30, 18) =$$

$$\text{gcd}(18, 12) =$$

Where $12 \bmod 6 == 0$ (**base case**) so $\text{gcd}(12, 6) = 6$

Prime number

- A prime number is an integer greater than 1 that has only two divisors: 1 and the number itself.

Examples of smaller prime numbers: 2, 3, 5, 7, 11,...

It also has a wide range of applications in cryptography: e.g., data encryption

- We are interested in designing **recursive algorithm** that checks whether a given number is prime or not.
- In the example, the technique of **helper function** will also be introduced for functional programming.

Helper function

Helper function is a special subroutine (sub-function) written separately to be called in a main function.

Algorithm: func(x,y) [main]
Requires: two values x and y
Returns: ...

1. return funcHelper(x,y,m)

Algorithm: funcHelper(a,b,c) [main]
Requires: three values
Returns: ...

// when this sub-function is called,
// the values x, y, m are passed to
// variables a, b, c.

If the process for solving the problem is complex, we may need **extra variables**.

These variables are neither input nor output arguments. They **only work intermediately** for solving the problem.

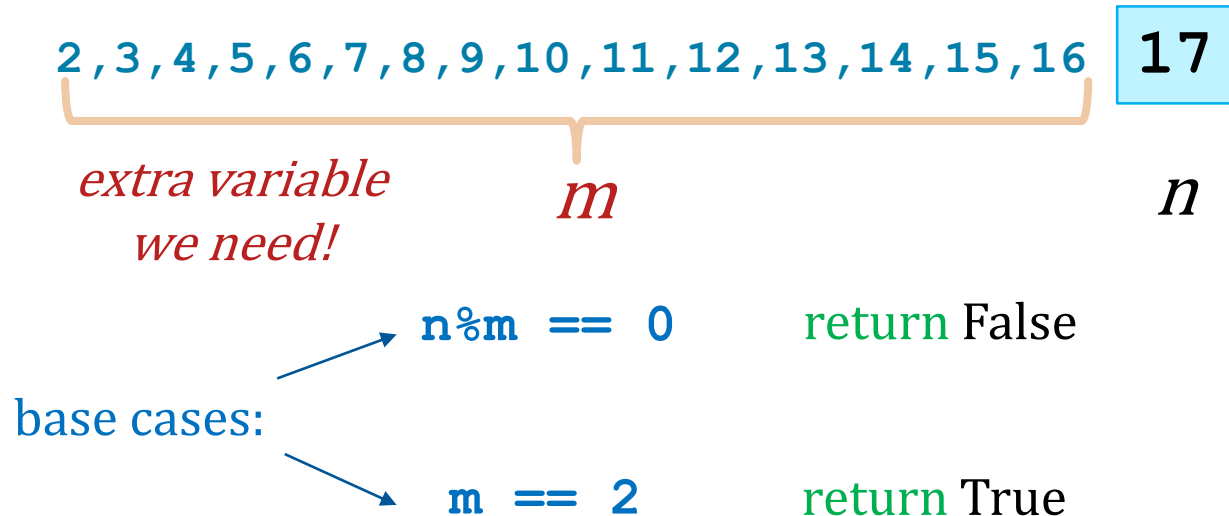
The users only care about the inputs/outputs. As programmer, we implement/hide these details in helper function.

Algorithm design

Design a recursive algorithm called *isPrime*(*n*) that takes a positive integer greater than 1 and returns True if *n* is prime; False otherwise.

User I/O Window

```
>> isPrime(17)
>> True
>> isPrime(170)
>> False
```



Algorithm: isPrime(n)

Algorithm: isPrime(n) [main]

Requires: a positive integer n , $n \geq 2$

Returns: True if n is prime; False otherwise

```
1. return isPrimeHelper(n, n-1)
```

When isPrimeHelper(n, m) is called, the local variable m will initially take value $n-1$.

Algorithm: isPrimeHelper(n, m)

Requires: two positive integers $n > m$

Returns: True or False

```
1. if n%m == 0
2.     return False
3. elseif m == 2
4.     return True
5. else
6.     return isPrimeHelper(n, m-1)
7. endif
```

two base cases

recursive call

Questions to think:

- Why are there two base cases? How do they correspond to our checking process? Trace the algorithm for isPrime(7) and isPrime(8) for details.
- Do the recursive steps approach base cases? Why?
- This algorithm has a defect. Can you fix it?

Practice Question

What is wrong in the following algorithm?

Algorithm: NoBase(n)

Requires : A number n

Returns : ??

1: return $n + \text{NoBase}(n - 1)$

Answer: There is no base case .

Note : The base case Must always be included in the recursive algorithm otherwise it will never end.

Practice Question

What is wrong in the following algorithm?

Algorithm: Loop(n)

Requires : A number n

Returns : ?

1: if $n == 0$ then

2: return 1

3: else

4: return $n \times \text{Loop}(n)$

Answer: Algorithm will go into infinite loop.

Note : Recursive case should approach the base case after finitely many steps.

Summary

- Recursion is when an algorithm is defined in terms of itself
- It allows us to define subproblems similar in form to the original
- Recursion always has two parts:
 - Base case
 - A problem that is closer to the solution
- Eventually, the base case is always called
- Without the base case, we would have infinite recursion
- Recursion makes some code shorter and easier to understand, such as input validation

Complete following activity to review today's Topic covered

<https://forms.office.com/r/2kmYrhZ5wS>

OR

Click here

Scan this

