

# COMP9020 Week 5

## Term 3, 2019

### Algorithmic analysis

# Summary of topics

- Motivation
- Standard approach
- Examples
- Simplifying with worst-case and big-O
- Recursive examples

# Summary of topics

- Motivation
- Standard approach
- Examples
- Simplifying with worst-case and big-O
- Recursive examples

# Algorithmic analysis: motivation

Want to compare algorithms – particularly ones that can solve *arbitrarily large* instances.

We would like to be able to talk about the **resources** (running time, memory, energy consumption) required by a program/algorithm as a function  $f(n)$  of some parameter  $n$  (e.g. the size) of its input.

## Example

How long does a given sorting algorithm take to run on a list of  $n$  elements?

time and space!!!!

# Issues

## Problems

- The exact resources required for an algorithm are difficult to pin down. Heavily dependent on:
  - Environment the program is run in (hardware, software, choice of language, external factors, etc)
  - Choice of inputs used

# Issues

## Problems

- The exact resources required for an algorithm are difficult to pin down. Heavily dependent on:
  - Environment the program is run in (hardware, software, choice of language, external factors, etc)
  - Choice of inputs used
- Cost functions can be complex, e.g.

$$2n \log(n) + (n - 100) \log(n)^2 + \frac{1}{2^n} \log(\log(n))$$

Need to identify the “important” aspects of the function.

# Order of growth

!!!!

## Example

Consider two time-cost functions:

- $f_1(n) = \frac{1}{10}n^2$  milliseconds, and
- $f_2(n) = 10n \log n$  milliseconds

Input size	$f_1(n)$	$f_2(n)$
100	0.01s	2s
1000	1s	30s
10000	1m40s	6m40s
100000	2h47m	1h23m
1000000	11d14h	16h40h
10000000	3y3m	8d2h

# Summary of topics

- Motivation
- **Standard approach**
- Examples
- Simplifying with worst-case and big-O
- Recursive examples



# Algorithmic analysis

Asymptotic analysis is about how costs **scale** as the input increases.

Standard (default) approach:

- Consider **asymptotic growth** of cost functions
- Consider **worst-case** (highest cost) inputs
- Consider **running time cost: number of elementary operations**

## NB

*Other common analyses include:*

- *Average-case analysis*
- *Space (memory) cost*

## Elementary operations

Informally: A single computational “step”; something that takes a constant number of computation cycles.

Examples:

- Arithmetic operations
- Comparison of two values
- Assignment of a value to a variable
- Accessing an element of an array
- Calling a function
- Returning a value
- Printing a single character

### NB

*Count operations up to a constant factor,  $O(1)$ , rather than an exact number.*

# Summary of topics

- Motivation
- Standard approach
- **Examples**
- Simplifying with worst-case and big-O
- Recursive examples

# Examples

## Example

Squaring a number (First version):

```
square( $n$ ) :  
    return  $n * n$ 
```

# Examples

## Example

Squaring a number (First version):

```
square( $n$ ) :  
    return  $n * n$      $O(1)$ 
```

# Examples

## Example

Squaring a number (First version):

```
square( $n$ ) :  
    return  $n * n$             $O(1)$ 
```

Running time:  $O(1)$

# Running time vs Execution time

Previous example shows one difference between running time and execution time.

In general, running time only *approximates* execution time:

- Simplifying assumptions about elementary operations
- Hidden constants in big-O
- Big-O only looks at limiting performance as  $n$  gets large.

## Examples

- Implementations of `square( $n$ )` will take longer as  $n$  gets bigger
- A program that “solves chess” will run in  $O(1)$  time.

# Examples

## Example

Squaring a number (Second version):

```
square( $n$ ) :  
   $r := 0$   
  for  $i = 1$  to  $n$  :  
     $r := r + n$   
  return  $r$ 
```



# Examples

## Example

Squaring a number (Second version):

```
square( $n$ ) :
```

```
   $r := 0$ 
```

```
  for  $i = 1$  to  $n$  :
```

```
     $r := r + n$ 
```

```
  return  $r$ 
```

$O(1)$

# Examples

## Example

Squaring a number (Second version):

```
square( $n$ ) :  
   $r := 0$   $O(1)$   
  for  $i = 1$  to  $n$  :  $O(1)$   
     $r := r + n$   
  return  $r$ 
```

# Examples

## Example

Squaring a number (Second version):

```
square( $n$ ) :  
   $r := 0$   $O(1)$   
  for  $i = 1$  to  $n$  :  $O(1)$   
     $r := r + n$   $O(1)$   
  return  $r$ 
```

# Examples

## Example

Squaring a number (Second version):

```
square( $n$ ) :  
   $r := 0$   $O(1)$   
  for  $i = 1$  to  $n$  :  $O(1)$   
     $r := r + n$   $O(1)$  |  $n$  times  
  return  $r$ 
```

# Examples

## Example

Squaring a number (Second version):

```
square( $n$ ) :  
   $r := 0$   $O(1)$   
  for  $i = 1$  to  $n$  :  $O(1)$   
     $r := r + n$   $O(1)$   
  return  $r$ 
```

$n$  times  $O(n)$

# Examples

## Example

Squaring a number (Second version):

```
square( $n$ ) :  
   $r := 0$   $O(1)$   
  for  $i = 1$  to  $n$  :  $O(1)$   
     $r := r + n$   $O(1)$   
  return  $r$   $O(1)$ 
```

$n$  times  $O(n)$

# Examples

## Example

Squaring a number (Second version):

```
square( $n$ ) :  
   $r := 0$   $O(1)$   
  for  $i = 1$  to  $n$  :  $O(1)$   
     $r := r + n$   $O(1)$  |  $n$  times  $O(n)$   
  return  $r$   $O(1)$ 
```

Running time:  $O(1) + O(n) + O(1) = O(n)$

# Examples

## Example

Cubing a number (using second squaring program):

```
cube( $n$ ) :  
   $r := 0$   
  for  $i = 1$  to  $n$  :  
     $r := r + \text{square}(n)$   
  return  $r$ 
```



# Examples

## Example

Cubing a number (using second squaring program):

```
cube( $n$ ) :  
   $r := 0$   $O(1)$   
  for  $i = 1$  to  $n$  :  $O(1)$   
     $r := r + \text{square}(n)$   
  return  $r$   $O(1)$ 
```

# Examples

## Example

Cubing a number (using second squaring program):

```
cube( $n$ ) :  
   $r := 0$   $O(1)$   
  for  $i = 1$  to  $n$  :  $O(1)$   
     $r := r + \text{square}(n)$   $O(1) + O(n)$   
  return  $r$   $O(1)$ 
```

# Examples

## Example

Cubing a number (using second squaring program):

```
cube( $n$ ) :  
   $r := 0$   $O(1)$   
  for  $i = 1$  to  $n$  :  
     $r := r + \text{square}(n)$   $O(1) + O(n)$  |  $n$  times  $O(n^2)$   
  return  $r$   $O(1)$ 
```

# Examples

## Example

Cubing a number (using second squaring program):

```
cube( $n$ ) :  
   $r := 0$   $O(1)$   
  for  $i = 1$  to  $n$  :  
     $r := r + \text{square}(n)$   $O(1) + O(n)$  |  $n$  times  $O(n^2)$   
  return  $r$   $O(1)$ 
```

Running time:  $O(1) + O(n^2) + O(1) = O(n^2)$

# Summary of topics

- Motivation
- Standard approach
- Examples
- Simplifying with worst-case and big-O
- Recursive examples

# Worst-case and big-O

Worst-case input assumption and big-O combine to *simplify* the analysis:

## Example

Sum of squares (Using second squaring program):

```
sumOfSquares( $n$ ) :  
   $r := 0$   
  for  $i = 1$  to  $n$  :  
     $r := r + \text{square}(i)$   
  return  $r$ 
```

## Worst-case and big-O

Worst-case input assumption and big-O combine to *simplify* the analysis:

### Example

Sum of squares (Using second squaring program):

```
sumOfSquares( $n$ ) :  
   $r := 0$   $O(1)$   
  for  $i = 1$  to  $n$  :  $O(1)$   
     $r := r + \text{square}(i)$   
  return  $r$   $O(1)$ 
```

## Worst-case and big-O

Worst-case input assumption and big-O combine to *simplify* the analysis:

### Example

Sum of squares (Using second squaring program):

```
sumOfSquares( $n$ ) :  
   $r := 0$   $O(1)$   
  for  $i = 1$  to  $n$  :  $O(1)$   
     $r := r + \text{square}(i)$   $O(i)$   $O(?)$   
  return  $r$   $O(1)$ 
```



## Worst-case and big-O

Worst-case input assumption and big-O combine to *simplify* the analysis:

### Example

Sum of squares (Using second squaring program):

```
sumOfSquares( $n$ ) :  
   $r := 0$   $O(1)$   
  for  $i = 1$  to  $n$  :  $O(1)$   
     $r := r + \text{square}(i)$   $O(n)$  |  $n$  times  $O(?)$   
  return  $r$   $O(1)$ 
```

## Worst-case and big-O

Worst-case input assumption and big-O combine to *simplify* the analysis:

### Example

Sum of squares (Using second squaring program):

```
sumOfSquares( $n$ ) :  
   $r := 0$   $O(1)$   
  for  $i = 1$  to  $n$  :  $O(1)$   
     $r := r + \text{square}(i)$   $O(n)$  |  $n$  times  $O(n^2)$   
  return  $r$   $O(1)$ 
```

Running time:  $O(1) + O(n^2) + O(1) = O(n^2)$

# Worst-case and big-O

Worst-case input assumption and big-O combine to *simplify* the analysis:

## Example

Finding an element ( $x$ ) in an array ( $L$ ) of length  $n$ :

```
find( $x, L$ ):  
    for  $i = 0$  to  $n - 1$ :  
        if  $L[i] == x$ :  
            return  $i$   
    return  $-1$ 
```

## Worst-case and big-O

Worst-case input assumption and big-O combine to *simplify* the analysis:

### Example

Finding an element ( $x$ ) in an array ( $L$ ) of length  $n$ :

```
find( $x, L$ ):  
  for  $i = 0$  to  $n - 1$ :       $O(1)$   
    if  $L[i] == x$ :           $O(1)$   
      return  $i$              $O(1)$   
  return  $-1$                  $O(1)$ 
```

## Worst-case and big-O

Worst-case input assumption and big-O combine to *simplify* the analysis:

### Example

Finding an element ( $x$ ) in an array ( $L$ ) of length  $n$ :

<code>find(<math>x, L</math>):</code>			
<code>for <math>i = 0</math> to <math>n - 1</math>:</code>	$O(1)$		
<code>if <math>L[i] == x</math>:</code>	$O(1)$	? times	$O(?)$
<code>return <math>i</math></code>	$O(1)$		
<code>return -1</code>			$O(1)$

## Worst-case and big-O

Worst-case input assumption and big-O combine to *simplify* the analysis:

### Example

Finding an element ( $x$ ) in an array ( $L$ ) of length  $n$ :

<code>find(<math>x, L</math>):</code>			
<code>for <math>i = 0</math> to <math>n - 1</math>:</code>	$O(1)$		
<code>if <math>L[i] == x</math>:</code>	$O(1)$	$O(n)$ times	$O(?)$
<code>return <math>i</math></code>	$O(1)$		
<code>return -1</code>			$O(1)$

## Worst-case and big-O

Worst-case input assumption and big-O combine to *simplify* the analysis:

### Example

Finding an element ( $x$ ) in an array ( $L$ ) of length  $n$ :

```
find( $x, L$ ) :  
  for  $i = 0$  to  $n - 1$  :       $O(1)$   
    if  $L[i] == x$  :           $O(1)$     $O(n)$  times    $O(n)$   
      return  $i$               $O(1)$   
  return  $-1$                   $O(1)$ 
```

Running time:  $O(n) + O(1) = O(n)$

## Worst-case and big-O

Worst-case input assumption and big-O combine to *simplify* the analysis:

### NB

*Simplifications might lead to sub-optimal bounds, may have to do a better analysis to get best bounds:*

- *Finer-grained upper bound analysis*
- *Analyse specific cases to find a matching lower bound (big- $\Omega$ )*

### NB

*Big- $\Omega$  is a **lower bound** analysis of the worst-case; NOT a “best-case” analysis.*



## Worst-case and big-O

Analyse specific cases to find a matching lower bound (big- $\Omega$ )

### Example

Let  $L_n$  be an  $n$ -element array of 0's.

Finding an element ( $x$ ) in an array ( $L$ ) of length  $n$ :

```
find( $x, L_n$ ):  
  for  $i = 0$  to  $n - 1$ :  
    if  $L[i] == x$ :  
      return  $i$   
  return  $-1$ 
```

## Worst-case and big-O

Analyse specific cases to find a matching lower bound (big- $\Omega$ )

### Example

Let  $L_n$  be an  $n$ -element array of 0's.

Finding an element ( $x$ ) in an array ( $L$ ) of length  $n$ :

```
find( $x, L_n$ ) :  
  for  $i = 0$  to  $n - 1$  :       $\Omega(1)$   
    if  $L[i] == x$  :           $\Omega(1)$   
      return  $i$                $\Omega(1)$   
  return  $-1$                    $\Omega(1)$ 
```

## Worst-case and big-O

Analyse specific cases to find a matching lower bound (big- $\Omega$ )

### Example

Let  $L_n$  be an  $n$ -element array of 0's.

Finding an element ( $x$ ) in an array ( $L$ ) of length  $n$ :

```
find( $x, L_n$ ) :  
  for  $i = 0$  to  $n - 1$  :       $\Omega(1)$   
    if  $L[i] == x$  :           $\Omega(1)$  |  $\Omega(n)$  times  
      return  $i$                $\Omega(1)$   
  return  $-1$                    $\Omega(1)$ 
```

## Worst-case and big-O

Analyse specific cases to find a matching lower bound (big- $\Omega$ )

### Example

Let  $L_n$  be an  $n$ -element array of 0's.

Finding an element ( $x$ ) in an array ( $L$ ) of length  $n$ :

```
find( $x, L_n$ ):  
  for  $i = 0$  to  $n - 1$ :  
    if  $L[i] == x$ :  
      return  $i$   
  return  $-1$ 
```

$\Omega(1)$		$\Omega(n)$ times	$\Omega(n)$
$\Omega(1)$			
$\Omega(1)$			
			$\Omega(1)$

## Worst-case and big-O

Analyse specific cases to find a matching lower bound (big- $\Omega$ )

### Example

Let  $L_n$  be an  $n$ -element array of 0's.

Finding an element ( $x$ ) in an array ( $L$ ) of length  $n$ :

<code>find(<math>x, L_n</math>):</code>			
for $i = 0$ to $n - 1$ :	$\Omega(1)$		
if $L[i] == x$ :	$\Omega(1)$	$\Omega(n)$ times	$\Omega(n)$
return $i$	$\Omega(1)$		
return $-1$			$\Omega(1)$

Running time of `find(1,  $L_n$ )`:  $\Omega(n)$

## Worst-case and big-O

Analyse specific cases to find a matching lower bound (big- $\Omega$ )

### Example

Let  $L_n$  be an  $n$ -element array of 0's.

Finding an element ( $x$ ) in an array ( $L$ ) of length  $n$ :

<code>find(<math>x, L_n</math>):</code>			
for $i = 0$ to $n - 1$ :	$\Omega(1)$		
if $L[i] == x$ :	$\Omega(1)$	$\Omega(n)$ times	$\Omega(n)$
return $i$	$\Omega(1)$		
return $-1$			$\Omega(1)$

Running time of `find(1,  $L_n$ )`:  $\Omega(n)$

Therefore, running time of `find( $x, L$ )`:  $\Theta(n)$

# Summary of topics

- Motivation
- Standard approach
- Examples
- Simplifying with worst-case and big-O
- Recursive examples

# Recursive examples

## Example

Factorial:

```
fact( $n$ ) :  
    if  $n == 0$  :  
        return 0  
    else :  
        return  $n * \text{fact}(n - 1)$ 
```



# Recursive examples

## Example

Factorial:

```
fact( $n$ ) :  
  if  $n == 0$  :  $O(1)$   
    return 0  $O(1)$   
  else :  
    return  $n * \text{fact}(n - 1)$   $O(1) + ?$ 
```

# Recursive examples

## Example

Factorial:

```
fact( $n$ ) :  
  if  $n == 0$  :  $O(1)$   
    return 0  $O(1)$   
  else :  
    return  $n * \text{fact}(n - 1)$   $O(1) + ?$ 
```

Running time for  $\text{fact}(n)$ :  $T(n)$

# Recursive examples

## Example

Factorial:

```
fact( $n$ ) :  
  if  $n == 0$  :  $O(1)$   
    return 0  $O(1)$   
  else :  
    return  $n * \text{fact}(n - 1)$   $O(1) + T(n - 1)$ 
```

Running time for  $\text{fact}(n)$ :  $T(n)$

# Recursive examples

## Example

Factorial:

```
fact(n) :  
  if n == 0 :            $O(1)$   
    return 0             $O(1)$   
  else :  
    return  $n * \text{fact}(n - 1)$     $O(1) + T(n - 1)$ 
```

Running time for  $\text{fact}(n)$ :  $T(n)$ , where:

$$T(0) \in O(1) + O(1) = O(1)$$

$$T(n) = T(n - 1) + O(1)$$

# Recursive examples

## Example

Factorial:

```
fact(n) :  
  if n == 0 :           O(1)  
    return 0           O(1)  
  else :  
    return n * fact(n - 1)  O(1) + T(n - 1)
```

Running time for  $\text{fact}(n)$ :  $T(n)$ , where:

$$\begin{aligned} T(0) &\in O(1) + O(1) = O(1) \\ T(n) &= T(n-1) + O(1) \\ &\in O(n) \end{aligned}$$

# Recursive examples

## Example

Factorial:

```
fact(n) :  
  if n == 0 :           O(1)  
    return 0           O(1)  
  else :  
    return n * fact(n - 1)  O(1) + T(n - 1)
```

Running time for  $\text{fact}(n)$ :  $T(n)$ , where:

$$\begin{aligned}T(0) &\in O(1) + O(1) = O(1) \\T(n) &= T(n - 1) + O(1) \\&\in O(n)\end{aligned}$$

Running time:  $T(n) \in O(n)$

# Recursive examples

## Example

Summing elements of a linked list (length  $n$ ):

```
sum(L) :  
  if L.isEmpty() :  
    return 0  
  else :  
    return L.data + sum(L.next)
```

# Recursive examples

## Example

Summing elements of a linked list (length  $n$ ):

```
sum(L) :  
  if L.isEmpty() : O(1)  
    return 0 O(1)  
  else :  
    return L.data + sum(L.next) O(1) +
```



# Recursive examples

## Example

Summing elements of a linked list (length  $n$ ):

```
sum(L) :  
  if L.isEmpty() : O(1)  
    return 0 O(1)  
  else :  
    return L.data + sum(L.next) O(1) +
```

Running time for `sum(L)`:  $T(n)$

# Recursive examples

## Example

Summing elements of a linked list (length  $n$ ):

```
sum(L) :  
  if L.isEmpty() :  $O(1)$   
    return 0  $O(1)$   
  else :  
    return L.data + sum(L.next)  $O(1) + T(n - 1)$ 
```

Running time for `sum(L)`:  $T(n)$

# Recursive examples

## Example

Summing elements of a linked list (length  $n$ ):

```
sum(L) :  
  if L.isEmpty() : O(1)  
    return 0 O(1)  
  else :  
    return L.data + sum(L.next)  $O(1) + T(n - 1)$ 
```

Running time for  $\text{sum}(L)$ :  $T(n)$ , where:

$$T(0) \in O(1) + O(1) = O(1)$$

$$T(n) = T(n - 1) + O(1)$$

# Recursive examples

## Example

Summing elements of a linked list (length  $n$ ):

```
sum(L) :  
  if L.isEmpty() : O(1)  
    return 0 O(1)  
  else :  
    return L.data + sum(L.next)  $O(1) + T(n - 1)$ 
```

Running time for  $\text{sum}(L)$ :  $T(n)$ , where:

$$\begin{aligned} T(0) &\in O(1) + O(1) = O(1) \\ T(n) &= T(n - 1) + O(1) \\ &\in O(n) \end{aligned}$$

## Recursive examples

### Example

Insertion sort (L has  $n$  elements):

```
sort(L) :  
  if L.isEmpty() :  
    return L  
  else :  
    L2 := sort(L.next)  
    insert L.data into L2  
    return L2
```

## Recursive examples

### Example

Insertion sort (L has  $n$  elements):

```
sort(L) :  
  if L.isEmpty() :  $O(1)$   
    return L  $O(1)$   
  else :  
    L2 := sort(L.next)  
    insert L.data into L2  
    return L2  $O(1)$ 
```

## Recursive examples

### Example

Insertion sort (L has  $n$  elements):

```
sort(L) :  
  if L.isEmpty() :  $O(1)$   
    return L  $O(1)$   
  else :  
     $L2 := \text{sort}(L.\text{next})$   
    insert L.data into L2  $O(n)$   
    return L2  $O(1)$ 
```

## Recursive examples

### Example

Insertion sort (L has  $n$  elements):

```
sort(L) :  
  if L.isEmpty() :  $O(1)$   
    return L  $O(1)$   
  else :  
    L2 := sort(L.next)  $T(n - 1)$   
    insert L.data into L2  $O(n)$   
    return L2  $O(1)$ 
```

Running time for `sort(L)`:  $T(n)$



## Recursive examples

### Example

Insertion sort (L has  $n$  elements):

```
sort(L) :  
  if L.isEmpty() :  $O(1)$   
    return L  $O(1)$   
  else :  
    L2 := sort(L.next)  $T(n - 1)$   
    insert L.data into L2  $O(n)$   
    return L2  $O(1)$ 
```

Running time for `sort(L)`:  $T(n)$ , where:

$$T(0) \in O(1) + O(1) = O(1)$$

$$T(n) = T(n - 1) + O(n) + O(1)$$

## Recursive examples

### Example

Insertion sort (L has  $n$  elements):

```
sort(L) :  
  if L.isEmpty() :  $O(1)$   
    return L  $O(1)$   
  else :  
    L2 := sort(L.next)  $T(n - 1)$   
    insert L.data into L2  $O(n)$   
    return L2  $O(1)$ 
```

Running time for `sort(L)`:  $T(n)$ , where:

$$\begin{aligned}T(0) &\in O(1) + O(1) = O(1) \\T(n) &= T(n - 1) + O(n) + O(1) \\&\in O(n^2)\end{aligned}$$

# Recursive examples

## Example

Euclidean algorithm for  $\text{gcd}(m, n)$  ( $N = m + n$ ):

```
gcd( $m, n$ ) :  
  if  $m > n$  :  
    return gcd( $m - n, n$ )  
  else if  $n > m$  :  
    return gcd( $m, n - m$ )  
  else :    return  $m$ 
```

# Recursive examples

## Example

Euclidean algorithm for  $\text{gcd}(m, n)$  ( $N = m + n$ ):

```
gcd( $m, n$ ) :  
  if  $m > n$  :  $O(1)$   
    return gcd( $m - n, n$ )  
  else if  $n > m$  :  $O(1)$   
    return gcd( $m, n - m$ )  
  else :  $O(1)$  return  $m$ 
```

# Recursive examples

## Example

Euclidean algorithm for  $\text{gcd}(m, n)$  ( $N = m + n$ ):

```
gcd( $m, n$ ) :  
  if  $m > n$  :  $O(1)$   
    return gcd( $m - n, n$ )  
  else if  $n > m$  :  $O(1)$   
    return gcd( $m, n - m$ )  
  else :  $O(1)$     return  $m$ 
```

Running time for  $\text{gcd}(m, n)$ :  $T(N)$

# Recursive examples

## Example

Euclidean algorithm for  $\text{gcd}(m, n)$  ( $N = m + n$ ):

```
gcd( $m, n$ ) :  
  if  $m > n$  :  
    return gcd( $m - n, n$ )  
  else if  $n > m$  :  
    return gcd( $m, n - m$ )  
  else :  
    return  $m$ 
```

$O(1)$   
 $\leq T(N - 1)$   
 $O(1)$   
 $\leq T(N - 1)$   
 $O(1)$

Running time for  $\text{gcd}(m, n)$ :  $T(N)$

## Recursive examples

### Example

Euclidean algorithm for  $\text{gcd}(m, n)$  ( $N = m + n$ ):

```
gcd( $m, n$ ) :  
  if  $m > n$  :  
    return gcd( $m - n, n$ )  
  else if  $n > m$  :  
    return gcd( $m, n - m$ )  
  else :  
    return  $m$ 
```

$O(1)$   
 $\leq T(N - 1)$   
 $O(1)$   
 $\leq T(N - 1)$   
 $O(1)$

Running time for  $\text{gcd}(m, n)$ :  $T(N)$ , where:

$$\begin{aligned} T(1) &\in O(1) \\ T(N) &\leq T(N - 1) + O(1) \end{aligned}$$

## Recursive examples

### Example

Euclidean algorithm for  $\text{gcd}(m, n)$  ( $N = m + n$ ):

```
gcd(m, n) :  
  if m > n :  
    return gcd(m - n, n) O(1)  
  else if n > m : O(1)  
    return gcd(m, n - m) ≤ T(N - 1)  
  else : O(1)  
    return m ≤ T(N - 1)
```

Running time for  $\text{gcd}(m, n)$ :  $T(N)$ , where:

$$\begin{aligned} T(1) &\in O(1) \\ T(N) &\leq T(N-1) + O(1) \\ &\in O(N) \end{aligned}$$



# Recursive examples

## Example

Euclidean algorithm for  $\text{gcd}(m, n)$  ( $N = m + n$ ):

Running time:  $O(N)$

## NB

$N$  is not the input **size**. Input size is  $\log(m) + \log(n)$

? ?

# Recursive examples

## Example

Faster Euclidean algorithm for  $\text{gcd}(m, n)$  ( $N = m + n$ ):

```
gcd( $m, n$ ) :  
  if  $m > n > 0$  :  
    return gcd( $m \% n, n$ )  
  else if  $n > m > 0$  :  
    return gcd( $m, n \bmod opm$ )  
  else :  
    return max( $m, n$ )
```

# Recursive examples

## Example

Faster Euclidean algorithm for  $\text{gcd}(m, n)$  ( $N = m + n$ ):

```
gcd( $m, n$ ) :  
  if  $m > n > 0$  :  $O(1)$   
    return gcd( $m \% n, n$ )  
  else if  $n > m > 0$  :  $O(1)$   
    return gcd( $m, n \text{ mod } opm$ )  
  else :  $O(1)$   
    return max( $m, n$ )
```

# Recursive examples

## Example

Faster Euclidean algorithm for  $\text{gcd}(m, n)$  ( $N = m + n$ ):

```
gcd( $m, n$ ) :  
  if  $m > n > 0$  :  $O(1)$   
    return gcd( $m \% n, n$ )  
  else if  $n > m > 0$  :  $O(1)$   
    return gcd( $m, n \text{ mod } opm$ )  
  else :  $O(1)$   
    return max( $m, n$ )
```

Running time for  $\text{gcd}(m, n)$ :  $T(N)$

# Recursive examples

## Example

Faster Euclidean algorithm for  $\text{gcd}(m, n)$  ( $N = m + n$ ):

```
gcd(m, n) :  
  if m > n > 0 :  
    return gcd(m % n, n) O(1)  
  else if n > m > 0 :  
    return gcd(m, n mod opm) ≤ T(N/2)  
  else :  
    return max(m, n) O(1)
```

$O(1)$        $????$

Running time for  $\text{gcd}(m, n)$ :  $T(N)$

## Recursive examples

### Example

Faster Euclidean algorithm for  $\text{gcd}(m, n)$  ( $N = m + n$ ):

```
gcd( $m, n$ ) :  
  if  $m > n > 0$  :  $O(1)$   
    return gcd( $m \% n, n$ )  $\leq T(N/2)$   
  else if  $n > m > 0$  :  $O(1)$   
    return gcd( $m, n \bmod opm$ )  $\leq T(N/2)$   
  else :  $O(1)$   
    return max( $m, n$ )
```

Running time for  $\text{gcd}(m, n)$ :  $T(N)$ , where:

## Recursive examples

### Example

Faster Euclidean algorithm for  $\text{gcd}(m, n)$  ( $N = m + n$ ):

```
gcd(m, n) :  
  if m > n > 0 :  
    return gcd(m % n, n) O(1)  
  else if n > m > 0 : O(1)  
    return gcd(m, n mod opm) ≤ T(N/2)  
  else : O(1)  
    return max(m, n) ≤ T(N/2)
```

Running time for  $\text{gcd}(m, n)$ :  $T(N)$ , where:

$$\begin{aligned} T(1) &\in O(1) \\ T(N) &\leq T(N/2) + O(1) \end{aligned}$$

# Recursive examples

## Example

Faster Euclidean algorithm for  $\text{gcd}(m, n)$  ( $N = m + n$ ):

```
gcd(m, n) :  
  if m > n > 0 :  
    return gcd(m % n, n) O(1)  
  else if n > m > 0 : O(1)  
    return gcd(m, n mod opm) ≤ T(N/2)  
  else : return max(m, n) ≤ T(N/2)  
                                O(1)
```

Running time for  $\text{gcd}(m, n)$ :  $T(N)$ , where:

$$\begin{aligned} T(1) &\in O(1) \\ T(N) &\leq T(N/2) + O(1) \\ &\in O(\log N) \end{aligned}$$