University of
Central Lancashire
UCLan

CO2412
Computational Thinking

Pseudocode and program analysis
Recursive functions

Where opportunity creates success

Module overview

Upon successful completion of this module, a student will be able to:

1) Use methods including logic and probability to reason about algorithms and data structures;
2) Compare, select, and justify algorithms and data structures for a given problem;
3) Analyse the computational complexity of problems and the efficiency of algorithms;
4) Use a range of notations to represent and analyse problems;
5) Implement and test algorithms and data structures.

program analysis

algorithm design

graphs and trees

logic

formal languages

complexity

randomness and probability

University of
Central Lancashire
UCLan

Pseudocode and program analysis

CO2412

12th October 2021

Levels of abstraction in program analysis

Program implementation (code)

accessible to automated analysis; formal verification may be possible

different programming languages entail variation in data structures, etc.

Algorithm description (pseudocode)

accessible to analysis by humans; e.g., **efficiency** of the algorithm

informal representation, independent of implementation and architecture

Levels of abstraction in program analysis

Binary executable (equivalently, script + executable interpreter)	
performance, i.e., **resource requirements**, on given hardware	influenced by compiler/interpreter choice and configuration, etc.
Program implementation (code)	
accessible to automated analysis; formal verification may be possible	different programming languages entail variation in data structures, etc.
Algorithm description (pseudocode)	
accessible to analysis by humans; e.g., **efficiency** of the algorithm	informal representation, independent of implementation and architecture
Problem statement	
open to theoretical investigation; **complexity**: best possible efficiency	proofs of upper or lower bounds apply to any potential algorithm

Code development cycle

- Specify
  - Function specification – what it should do
  - Non-functional specification – **how well** it should do it
- Design
  - Select appropriate algorithms and data structures
    - Consider effectiveness/correctness – *does it do what it is supposed to?*
    - Consider efficiency
      - Size
      - Speed
- Implement
  - Create solution at low level

Design by contract

- Specify
  - Function specification – what it should do
  - Non-functional specification – **how well** it should do it
- Design
  - Select appropriate algorithms and data structures
    - Consider effectiveness/correctness – *does it do what it is supposed to?*
    - Consider efficiency
      - Size
      - Speed
- Implement
  - Create solution at low level

"contracts" between specifier, designer, and programmer

We need a way of specifying algorithms and their outcomes

Design by contract

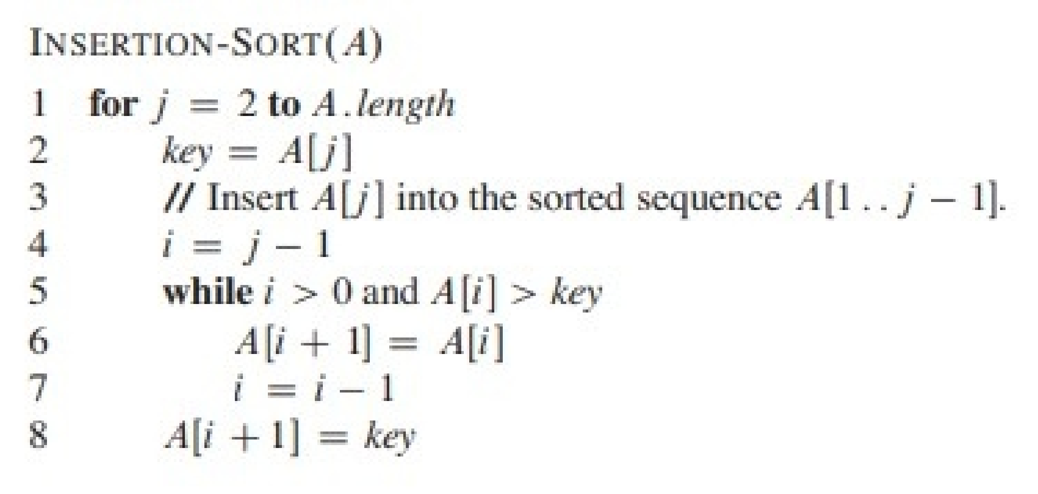
- Specify
  - Function specification – what it should do
  - Non-functional specification – **how well** it should do it
- Design
  - Select appropriate algorithms and data structures
    - Consider effectiveness/correctness – *does it do what it is supposed to?*
    - Consider efficiency
      - Size
      - Speed
- Implement
  - Create solution at low level
- Evaluate
  - Debug, assess for syntactic & semantic correctness
  - Check performance (i.e., resource requirements)

"contracts" between specifier, designer, and programmer

We need a way of specifying algorithms and their outcomes

Pseudocode as an informal specification

Algorithms are best discussed using **pseudocode**:



```
INSERTION-SORT($A$)
for $j = 2$ to $A.length$
    $key = A[j]$
    // Insert $A[j]$ into the sorted sequence $A[1 \,..\, j-1]$.
    $i = j - 1$
    while $i > 0$ and $A[i] > key$
        $A[i+1] = A[i]$
        $i = i - 1$
    $A[i+1] = key$
```

Since the analysis of an algorithm should occur at a level of abstraction higher than that of its practical implementation (as code), simple generally comprehensible notations are often more suitable than syntactically correct code in any given programming language.

Pseudocode as an informal specification

code

```
int prod = 1;
for(int i=0; i < n; i++) prod *= fact[i];
```

```
prod = 1
for i in range(n):
    prod *= fact[i]
```

Program code is intended for computational processing by a compiler or interpreter.

Pseudocode is a representation of the semantics (meaning) of the code for a human reader.

The C/C++ code on top and the Python code at the bottom have equivalent outcomes if the initial state at the beginning of the block is equivalent.

As algorithms, they can be given a joint representation.

Pseudocode as an informal specification

code

```
int prod = 1;
for(int i=0; i < n; i++) prod *= fact[i];
```

```
prod = 1
for i in range(n):
    prod *= fact[i]
```

pseudocode

```
input: int n, int array fact
                 with ≥n elements
output: int prod

  prod ← 1
  for int i in 0 to n–1
    prod ← fact[i] * prod
  end for
```

Program flow graphs[1]

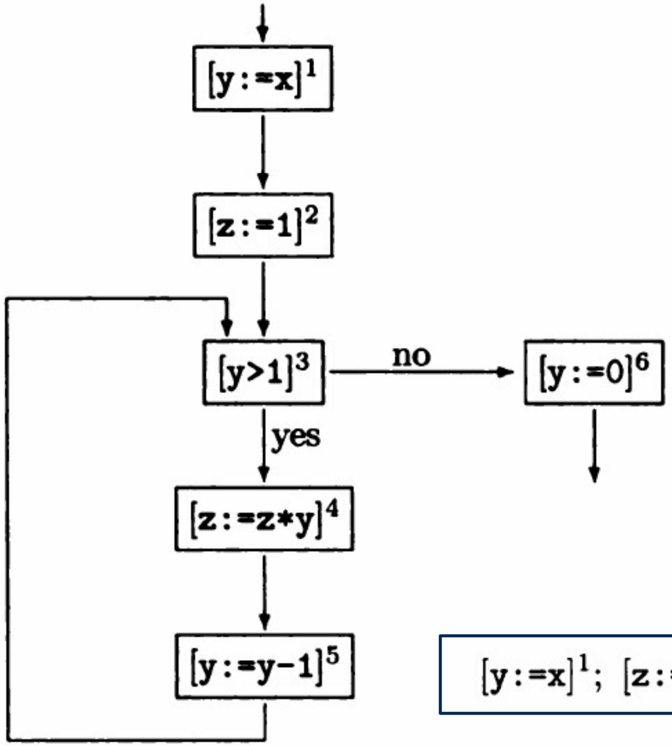


Figure 1.2: Flow graph for the factorial program.

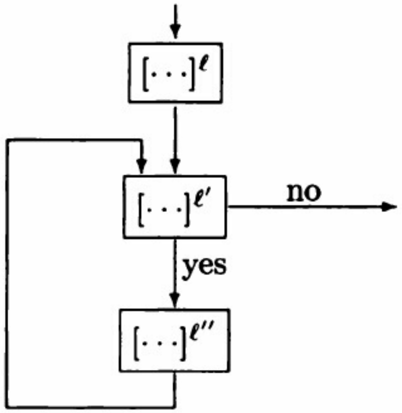


Figure 2.2: A schematic flow graph.

$[y:=x]^1;\ [z:=1]^2;\ \texttt{while}\ [y>1]^3\ \texttt{do}\ ([z:=z*y]^4;\ [y:=y-1]^5);\ [y:=0]^6$

[1] F. Nielson, H. Riis Nielson, C. Hankin, *Principles of Program Analysis*, Heidelberg: Springer, **2005**.

Decomposition

In many procedural programming languages, including C and Python, code blocks that can be called from other code blocks are called **functions**. Employing functions allows jumping to code that solve a certain task in a robust way, without needing to use the undesirable "goto" statement.[1]

Breaking complex problems down into smaller pieces can make them more manageable. In procedural (and object oriented) programming languages, functions (and methods) are typically used for that purpose.

$$\text{overall algorithm} \equiv \text{combine\_functions}(\ \text{function}_1, \text{function}_2, \ldots\ )$$

If the functions (or methods) for the smaller tasks have been **designed by contract** and are known to fulfill their respective purpose, e.g., supported by unit testing, it becomes easier to establish the correctness of the overall algorithm.

[1] E. W. Dijkstra, "Go to statement considered harmful," *Communications of the ACM* 11(3), 147, **1968**.

Procedural programming

In many procedural programming languages, including C and Python, code blocks that can be called from other code blocks are called **functions**.

Role of functions in procedural programming:

- Functions are named
- Each function has a distinct task
- It may have its own variables
- It may call another function
- It may return a value
- It may accept arguments

  x = multiply(n, $fact$)
- Function **parameters** are the variables listed in the function's definition. Function **arguments** are the values passed to the function, which are assigned to the function's parameters at runtime.

Procedural programming

In many procedural programming languages, including C and Python, code blocks that can be called from other code blocks are called **functions**. However, do not confuse **procedural programming** (as a programming paradigm) with **functional programming**, a name given to a very different approach (LISP, etc.).

- Functions are named
- Each function has a distinct task
- It may have its own variables
- It may call another function
- It may return a value
- It may accept arguments

  x = multiply(n, $fact$)

- Function **parameters** are the variables listed in the function's definition. Function **arguments** are the values passed to the function, which are assigned to the function's parameters at runtime.

Procedural programming

code

```
int prod = 1;
for(int i=0; i < n; i++) prod *= fact[i];
```

```
prod = 1
for i in range(n):
    prod *= fact[i]
```

pseudocode

```
input: int n, int array fact
                with ≥n elements
output: int prod

  prod ← 1
  for int i in 0 to n–1
    prod ← fact[i] * prod
  end for
```

Procedural programming

code

```
int multiply(int n, int* fact)
{
  int prod = 1;
  for(int i=0; i < n; i++) prod *= fact[i];
  return prod;
}
```

```
def multiply(n, fact):
   prod = 1
   for i in range(n):
      prod *= fact[i]
   return prod
```

pseudocode

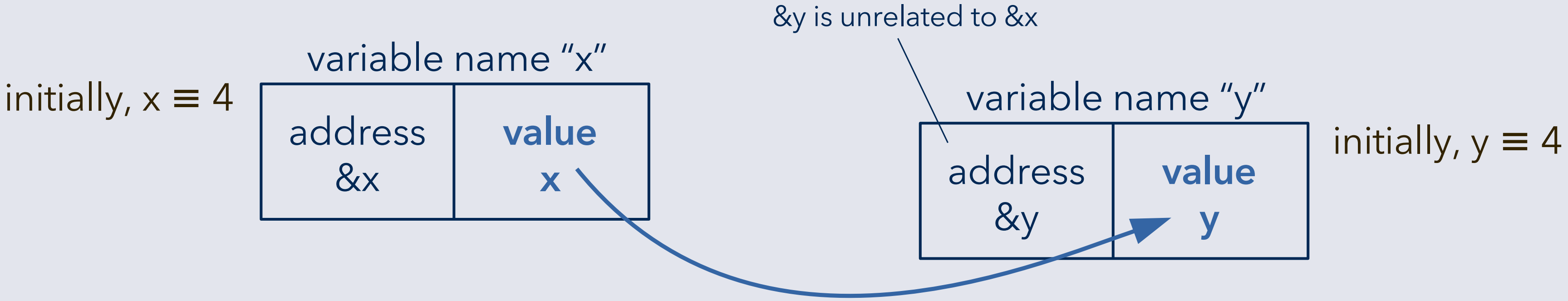
```
function multiply
input: int n, int array fact
                      with ≥n elements
output: int prod

  prod ← 1
  for int i in 0 to n–1
    prod ← fact[i] * prod
  end for
  return prod
end function
```

Pass by value and pass by reference

Two major ways in which arguments can be passed to functions:

Argument passing by value



&y is unrelated to &x

initially, x ≡ 4

variable name “x”

address &x

value x

variable name “y”

address &y

value y

initially, y ≡ 4

Argument passing by reference

initially, x ≡ 4

variable name “x”

address &x

value x

variable name “y”

address &y

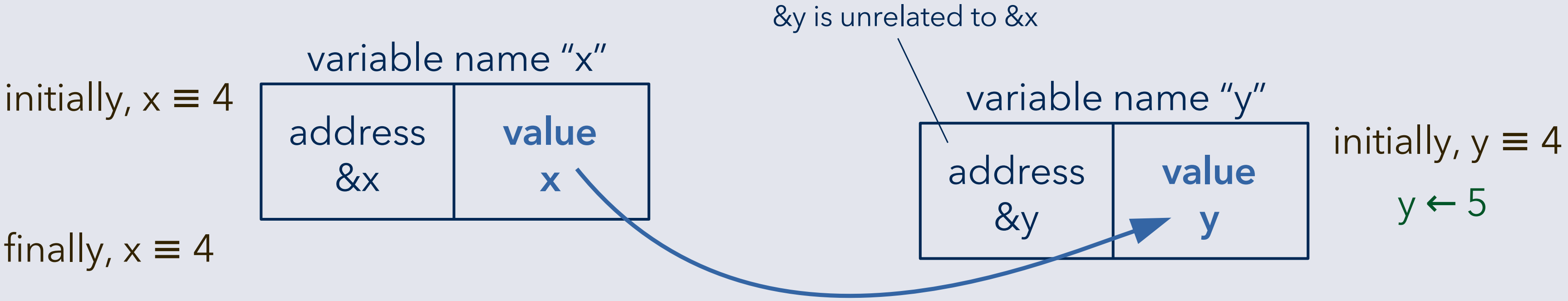
value y

initially, y ≡ &x,
hence *y ≡ 4

Pass by value and pass by reference

Two major ways in which arguments can be passed to functions:

Argument passing by value



&y is unrelated to &x

variable name "x"

initially, x ≡ 4

address &x	**value** **x**

finally, x ≡ 4

variable name "y"

address &y	**value** **y**

initially, y ≡ 4

y ← 5

Argument passing by reference

variable name "x"

initially, x ≡ 4

address **&x**	value x

finally, x ≡ 5

variable name "y"

address &y	**value** **y**

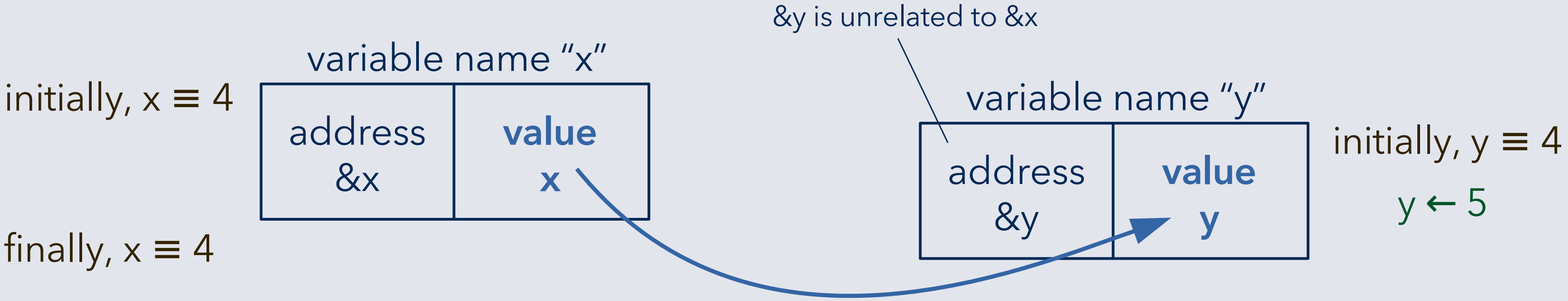
initially, y ≡ &x,
hence *y ≡ 4

*y ← 5

Pass by value and pass by reference

Two major ways in which arguments can be passed to functions:

Argument passing by value



&y is unrelated to &x

variable name “x”

initially, x ≡ 4

address &x	**value** **x**

finally, x ≡ 4

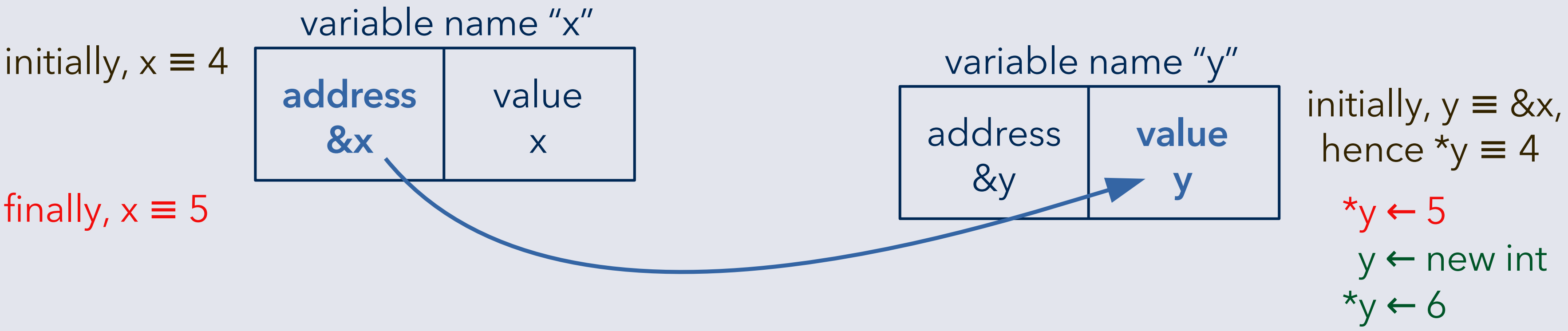
variable name “y”

address &y	**value** **y**

initially, y ≡ 4

y ← 5

Argument passing by reference



variable name “x”

initially, x ≡ 4

address **&x**	value x

finally, x ≡ 5

variable name “y”

address &y	**value** **y**

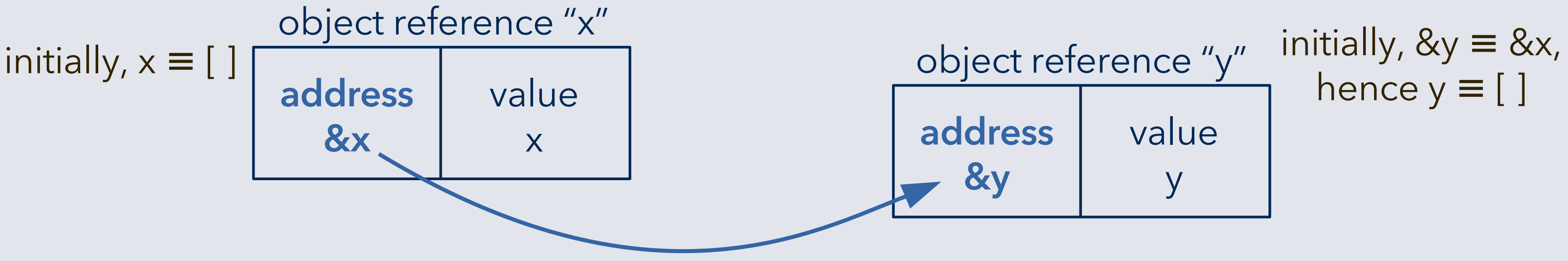
initially, y ≡ &x,
hence *y ≡ 4

*y ← 5
y ← new int
*y ← 6

Pass by object reference

In Python, object references are passed by value (i.e., “pass by object reference”):

Argument passing by object reference in Python (similarly, in Java)



initially, x ≡ [ ]

object reference “x”

address **&x**	value x

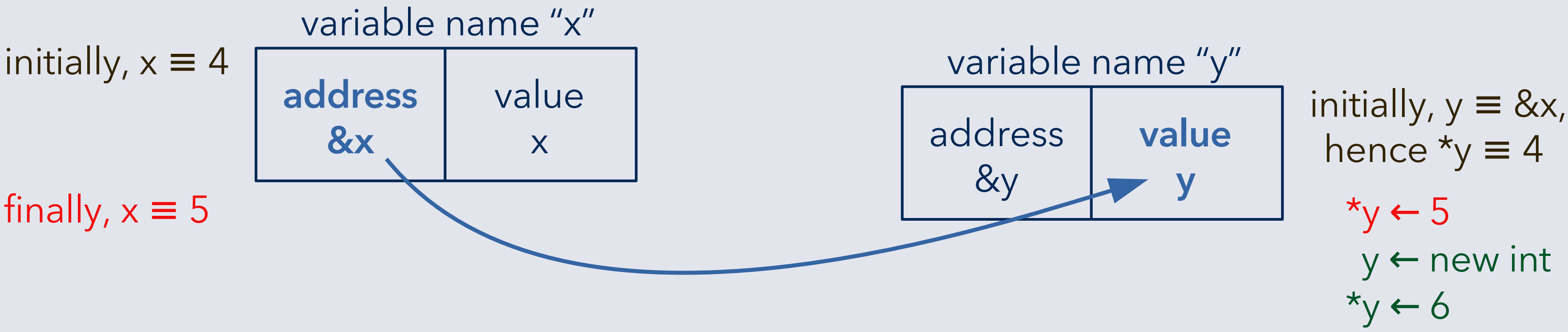
object reference “y”

address **&y**	value y

initially, &y ≡ &x,
hence y ≡ [ ]

function call f(x)

Argument passing by reference



initially, x ≡ 4

variable name “x”

address **&x**	value x

finally, x ≡ 5

variable name “y”

address &y	**value** **y**

initially, y ≡ &x,
hence *y ≡ 4

*y ← 5
y ← new int
*y ← 6

Pass by object reference

In Python, object references are passed by value (i.e., “pass by object reference”):

Argument passing by object reference in Python (similarly, in Java)

initially, x ≡ [ ]

object reference “x”

address &x | value x

finally, x ≡ [1]

object reference “y”

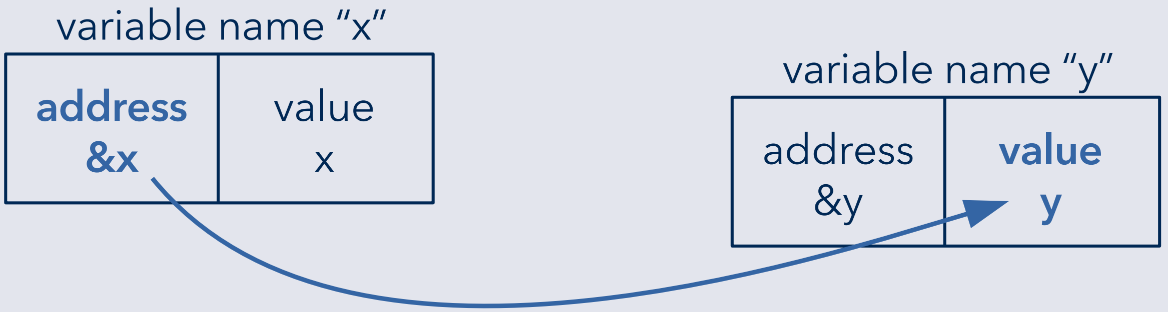
address &y | value y

initially, &y ≡ &x,
hence y ≡ [ ]

y.append(1)

function call f(x)

Argument passing by reference



initially, x ≡ 4

variable name “x”

address &x | value x

finally, x ≡ 5

variable name “y”

address &y | **value y**

initially, y ≡ &x,
hence *y ≡ 4

*y ← 5
y ← new int
*y ← 6

Pass by object reference

In Python, object references are passed by value (i.e., “pass by object reference”):

Argument passing by object reference in Python (similarly, in Java)

initially, x ≡ [ ]

finally, x ≡ [1]

initially, &y ≡ &x,
hence y ≡ [ ]

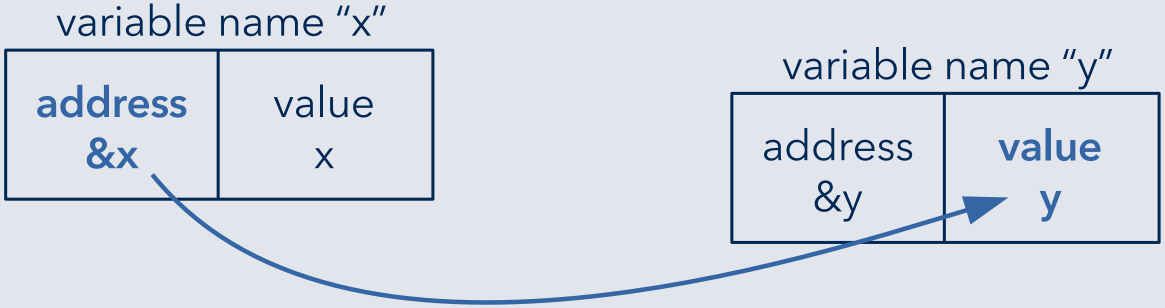
y.append(1)

y ← [1, 2]

function call f(x)

assigning a new
object to y changes
the address!

Argument passing by reference



initially, x ≡ 4

finally, x ≡ 5

initially, y ≡ &x,
hence *y ≡ 4

*y ← 5

y ← new int

*y ← 6

Recursive functions

CO2412

12th October 2021

Recursive function definitions

Recursion can be used to define a function, e.g., for the geometric series

$$f_q(0) = 1 \quad \text{and} \quad f_q(k) = q^k + f_q(k - 1).$$

The approach is applicable to any domain that is defined, or can be constructed, by induction.

Example: The set of integers $\mathbb{N}$ can be constructed by stating that $1 \in \mathbb{N}$ (base case), and for any $k \in \mathbb{N}$, there is a successor element $k+1 \in \mathbb{N}$.

It is applied by reducing a problem instance for a given argument value k to that for another, more elementary argument value $k' < k$. Here, the operator $<$ signifies some order indicating closeness to the base case (smallest element).

Such constructions are often used to define mathematical sequences.

Recursive function definitions

Recursion can be used to define a function, e.g., for the geometric series

$$f_q(0) = 1 \quad \text{and} \quad f_q(k) = q^k + f_q(k - 1).$$

The approach is applicable to any domain that is defined, or can be constructed, by induction.

Example: The se
at $1 \in \mathbb{N}$ (base
case), and for an

It is applied by re
nt value k to
that for another,
ne operator <
signifies some order indicating closeness to the base case (smallest element).

```
def geometric_series(q, k):
   if k>0:
      return q**k + geometric_series(q, k-1)
   else:
      return 1
```

Such constructions are often used to define mathematical sequences.

Recursive function definitions

Recursion can be used to define a function, e.g., for the geometric series

$f_q(0) = 1$ and $f_q(k) = q^k + f_q(k - 1)$; **$f_q(k) = (1 - q^{k+1}) / (1 - q)$.**

The approach is applicable to any domain that is defined, or can be constructed, by induction.

Example: The set of integers $\mathbb{N}$ can be constructed by stating that $1 \in \mathbb{N}$ (base case), and for any $k \in \mathbb{N}$, there is a successor element $k+1 \in \mathbb{N}$.

It is applied by reducing a problem instance for a given argument value k to that for another, more elementary argument value $k' < k$. Here, the operator $<$ signifies some order indicating closeness to the base case (smallest element).

Such constructions are often used to define mathematical sequences.
Recursively defined functions are not always best computed by recursion.

Recursion as an algorithm design strategy

Recursion is the process of defining the solution to a problem (or the solution to a problem) in terms of a simpler or smaller instance of the same problem.

Image from: https://www.therussianstore.com/blog/the-history-of-nesting-dolls

Recursion is a form of decomposition:

solution(k) ≡ recursive_step(solution($< k$))

Recursion as an algorithm design strategy

Recursion is the process of defining the solution to a problem (or the solution to a problem) in terms of a simpler or smaller instance of the same problem.

The base case (or a base case) is reached when the problem has been simplified to the utmost

Image from: https://www.therussianstore.com/blog/the-history-of-nesting-dolls

Recursion is a form of decomposition:

solution(k) ≡ recursive_step(solution($< k$))

solution($\perp$) ≡ base_case_solution

Recursion as an algorithm design strategy

Recursion is the process of defining the solution to a problem (or the solution to a problem) in terms of a simpler or smaller instance of the same problem.

The base case (or a base case) is reached when the problem has been simplified to the utmost

Multiple recursion decomposes a problem into more than one simplified instance

Image from: https://www.therussianstore.com/blog/the-history-of-nesting-dolls

Recursion is a form of decomposition:

$$\text{solution}(\ k\ )\ \equiv\ \text{recursive\_step}(\ \text{solution}_1(< k),\ \text{solution}_2(< k),\ \dots\ )$$

$$\text{solution}(\ \perp\ )\ \equiv\ \text{base\_case\_solution}$$

Multiple recursion example

The **Fibonacci numbers** constitute a mathematical sequence that is defined by multiple recursion:

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_k &= F_{k-1} + F_{k-2}, \quad \text{for } k > 1 \end{aligned}$$

0, 1, 1, 2, 3, 5, 8, 13, …

While the definition is most conveniently given in the form of a **recursion**, the numerical implementation would usually proceed by **iteration**. Compare the code employing a loop with that obtained by a direct calque of the definition.

In [31]:

```python
import time
k = 40

start = time.time()
print("Fibonacci nr.", k, "=", fibonacci_iter(k))
end = time.time()
print("Time required for iterative execution:", end - start, "s")

start = time.time()
print("Fibonacci nr.", k, "=", fibonacci_recur(k))
end = time.time()
print("Time required for naive recursive execution:", end - start, "s")
```

```
Fibonacci nr. 40 = 102334155
Time required for iterative execution: 0.00019598007202148438 s
Fibonacci nr. 40 = 102334155
Time required for naive recursive execution: 30.47756600379944 s
```

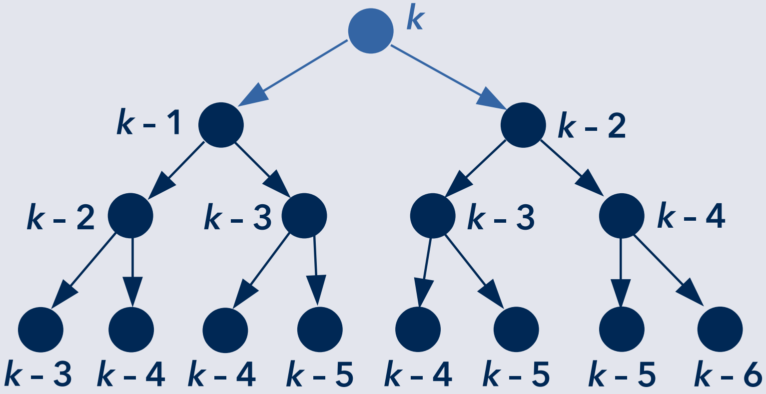
Multiple recursion example

The **Fibonacci numbers** constitute a mathematical sequence that is defined by multiple recursion:

$$\boxed{\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_k &= F_{k-1} + F_{k-2}, \text{ for } k > 1 \end{aligned}}$$

0, 1, 1, 2, 3, 5, 8, 13, …

While the definition is most conveniently given in the form of a **recursion**, the numerical implementation would usually proceed by **iteration**. Compare the code employing a loop with that obtained by a direct calque of the definition.



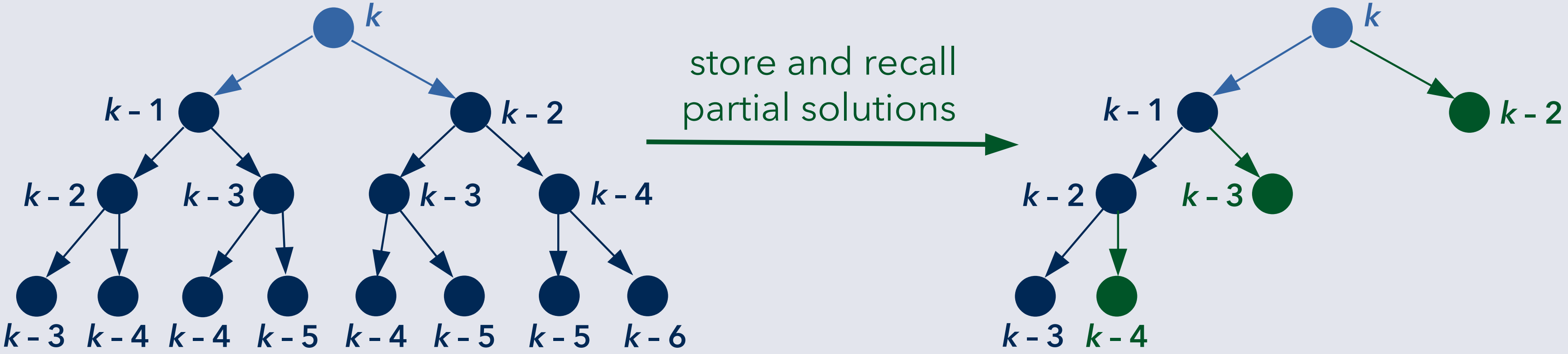
Multiple recursion example

The **Fibonacci numbers** constitute a mathematical sequence that is defined by multiple recursion:

$$\boxed{\begin{aligned} F_0 &= 0\\ F_1 &= 1\\ F_k &= F_{k-1} + F_{k-2}, \quad \text{for } k > 1 \end{aligned}}$$

0, 1, 1, 2, 3, 5, 8, 13, …

While the definition is most conveniently given in the form of a **recursion**, the numerical implementation would usually proceed by **iteration**. Compare the code employing a loop with that obtained by a direct calque of the definition.



Converting an iterative to a recursive solution

Loop structures can be equivalently transformed into recursions as follows:

iterative implementation

```
function do_iteratively
input: argv
output: retv

declaration of local variables work


while condition
   [single-step body]
end while
return retv
```

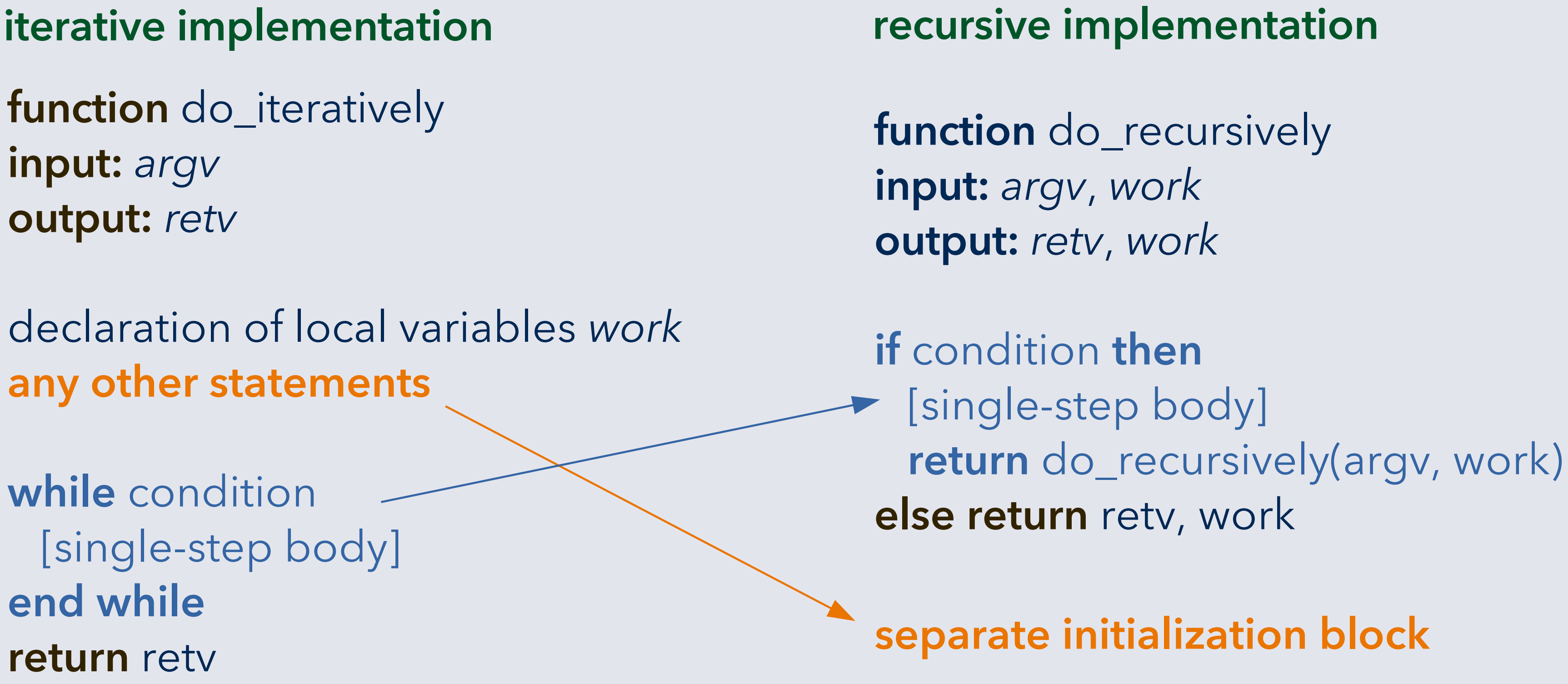
recursive implementation

```
function do_recursively
input: argv, work
output: retv, work

if condition then
   [single-step body]
   return do_recursively(argv, work)
else return retv, work
```

Converting an iterative to a recursive solution

Loop structures can be equivalently transformed into recursions as follows:



iterative implementation

```
function do_iteratively
input: argv
output: retv

declaration of local variables work
any other statements

while condition
  [single-step body]
end while
return retv
```

recursive implementation

```
function do_recursively
input: argv, work
output: retv, work

if condition then
  [single-step body]
  return do_recursively(argv, work)
else return retv, work

separate initialization block
```

Example: Python code for prime factor decomposition.

Converting an iterative to a recursive solution

Loop structures can be equivalently transformed into recursions as follows:

iterative implementation

```
def primfact_iter(n):
    factors = []
    i = 2
    while i <= n**(1/2):
        while n%i == 0:
            factors.append(i)
            n /= i
        i += 1
    if n != 1:
        factors.append(n)
    return factors
```

recursive implementation

```
def primfact_recur_body(n, factors, i):
    if i <= n**(1/2):
        while n%i == 0:
            factors.append(i)
            n /= i
        i += 1
        return primfact_recur_body(n, factors, i)
    else:
        return n, factors, i
```

Example: Python code for prime factor decomposition.

Converting an iterative to a recursive solution

Loop structures can be equivalently transformed into recursions as follows:

iterative implementation

```
def primfact_iter(n):
    factors = []
    i = 2
    while i <= n**(1/2):
        while n%i == 0:
            factors.append(i)
            n /= i
        i += 1
    if n != 1:
        factors.append(n)
    return factors
```

recursive implementation

```
def primfact_recur_body(n, factors, i):
    if i <= n**(1/2):
        while n%i == 0:
            factors.append(i)
            n /= i
        i += 1
```

```
def primfact_recur(n):
    factors = []
    i = 2
    n, factors, i = primfact_recur_body(n, factors, i)
    if n != 1:
        factors.append(n)
    return factors
```

Example: Python code for prime

Converting a recursive to an iterative solution

For **simple recursion over** ℕ, transformation into loop form is straightforward:

```
def geometric_series_recur(q, k):
   if k>0:
      return q**k + geometric_series_recur(q, k-1)
   else:
      return 1
```

- begin with the base case
- apply loop construct to work upward

$$\boxed{f_q(k) = \sum_{0 \leq j \leq k} q^j}$$

Converting a recursive to an iterative solution

For **simple recursion over** ℕ, transformation into loop form is straightforward:

```
def geometric_series_recur(q, k):
    if k>0:
        return q**k + geometric_series_recur(q, k-1)
    else:  # i.e., if k == 0
        return 1
```

- begin with the base case
- apply loop construct to work upward

$$\boxed{f_q(k) = \sum_{0 \le j \le k} q^j}$$

```
def geometric_series_iter(q, k):
    j = 0
    retv = 1
    while k > j:
        j += 1
        retv += q**j
    return retv
```

For **multiple recursions** or over domains with a more complex structure, loop-based equivalents can also be constructed, but in a less straightforward way.

University of
Central Lancashire
UCLan

CO2412
Computational Thinking

Pseudocode and program analysis
Recursive functions

Where opportunity creates success