University of Central Lancashire
UCLan

CO2412 Computational Thinking

Formal verification
Performance and efficiency

Where opportunity creates success

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 evaluating algorithms and their outcomes

Formal verification

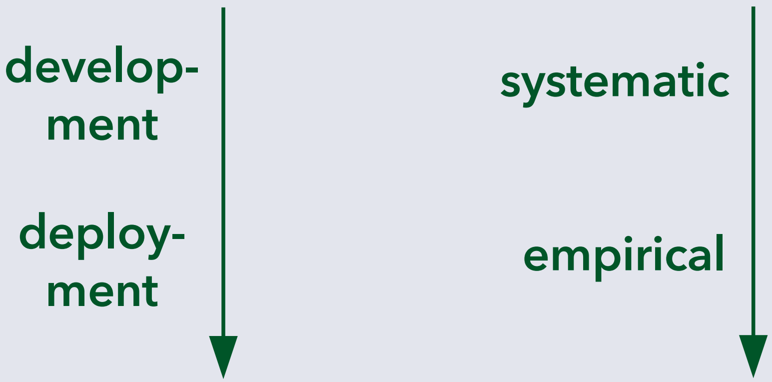
CO2412

19th October 2021

Verification, validation, and testing

Verification: Proof that the developed product complies with its specification.

- Where possible, provide a **rigorous logical/mathematical proof**; alternatively, provide documents following agreed standards/procedures.



Testing: Use-case driven evaluation of the final (or alpha, or beta) product.

- The considered **use cases** should be **representative**.
- They should be as unrelated as possible to any concrete scenarios considered during development, including the validation process.

Verification, validation, and testing

Verification: Proof that the developed product complies with its specification.

- Where possible, provide a **rigorous logical/mathematical proof**; alternatively, provide documents following agreed standards/procedures.

development → **deployment**

systematic → **empirical**

developer-driven → **user-driven**

Testing: Use-case driven evaluation of the final (or alpha, or beta) product.

- The considered **use cases** should be **representative**.
- They should be as unrelated as possible to any concrete scenarios considered during development, including the validation process.
- Ideally, **conducted by prospective users**; if unavailable, "play the user."

University of Central Lancashire UCLan

Verification, validation, and testing

Verification: Proof that the developed product complies with its specification.

- Where possible, provide a **rigorous logical/mathematical proof**; alternatively, provide documents following agreed standards/procedures.

Validation: Empirical evaluation to what extent user the requirements are met.

- All requirements need to be covered and demonstrated at least once.
- Ideally, **requirements** are not identical with the specification. They should be user-oriented; *e.g.*, epics and user stories in a requirements analysis from agile software engineering. **Feedback from users** is needed.

Testing: Use-case driven evaluation of the final (or alpha, or beta) product.

- The considered use cases should be **representative**.
- They should be **as unrelated as possible to** any concrete scenarios considered during development, including **the validation process**.
- Ideally, **conducted by prospective users**; if unavailable, “play the user.”

Verification, validation, and testing

Verification: Proof that the developed product complies with its specification.

- Where possible, provide a **rigorous logical/mathematical proof**; alternatively, provide documents following agreed standards/procedures.

Vali **met.**

ce.
hould
ysis

> **Remark**
>
> **Verification** always has the meaning that something is demonstrated to be true, particularly by logical reasoning. **Validation** and **testing** have many meanings to different communities; the distinction here is common in AI (*e.g.*, validation set *vs.* test set).

Testing: Use-case driven evaluation of the final (or alpha, or beta) product.

- The considered use cases should be **representative**.
- They should be **as unrelated as possible to** any concrete scenarios considered during development, including **the validation process**.
- Ideally, **conducted by prospective users**; if unavailable, "play the user."

Verification, validation, and testing

Verification: Proof that the developed product complies with its specification.

- Where possible, provide a **rigorous logical/mathematical proof**; alternatively, provide documents following agreed standards/procedures.

Vali … **met.**

… ce.
… hould
… ysis

> **Note**
>
> - The above is what we mean by **formal verification**.
> - There can be no verification without a **specification**.
> - **It can be done by humans, using code or pseudocode.**

Testing: Use-case driven evaluation of the final (or alpha, or beta) product.

- The considered use cases should be **representative**.
- They should be **as unrelated as possible to** any concrete scenarios considered during development, including **the validation process**.
- Ideally, **conducted by prospective users**; if unavailable, "play the user."

Verification, validation, and testing

Verification: Proof that the developed product complies with its specification.

– Where possible, provide a **rigorous logical/mathematical proof**; alternatively, provide documents following agreed standards/procedures.

Valic … **met.**

… ce.
… hould
… ysis

Testi … .

… er."

Note

- The above is what we mean by **formal verification**.
- There can be no verification without a **specification**.
- It can be done by humans, using code or pseudocode.
- It can also be done computationally (**automated verification**); in that case, either the programming language must be restricted severely, or it is only a **model of the program** that can be verified.
- The latter is known as **model checking**. It is limited by the accuracy and extent of the information provided in the model.

Program flow graphs[1]

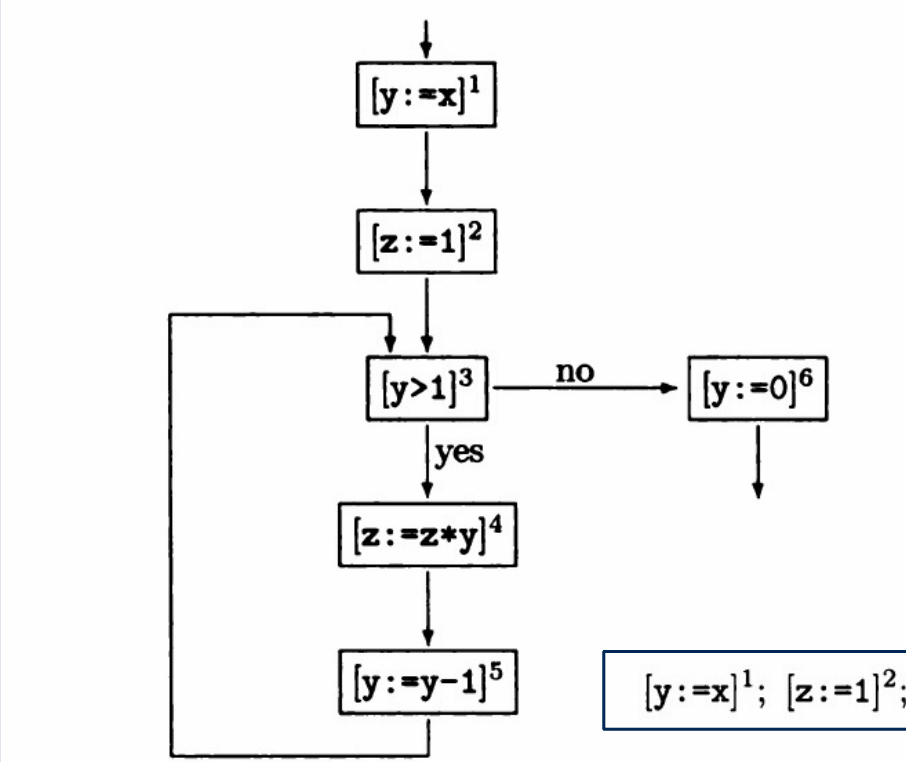


Figure 1.2: Flow graph for the factorial program.

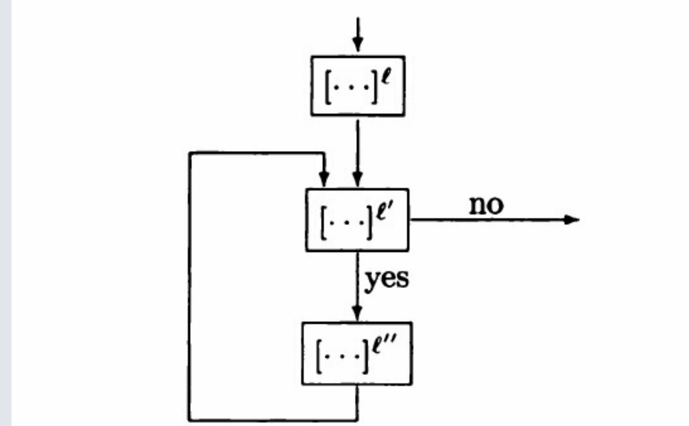


Figure 2.2: A schematic flow graph.

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

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

Preconditions and postconditions

For purposes of formal analysis, the program flow is analysed step by step, e.g., at the instruction (statement) level, at the level of blocks of code that form a coherent unit, or at the level of functions or methods.

Precondition: State of the program at a point directly before the considered unit. This may include assumptions taken from the design contract or specification.

Postcondition: State of the program at a point directly after the considered unit, assuming that the precondition was fulfilled at the point directly before it.

Example

As part of a development project, we need a function grtfrac(x, y) that takes two floating-point arguments and returns the one with the greater fractional part; *e.g.*, grtfrac(2.7, 3.6) is to return 2.7, because ".7" is greater than ".6". In design by contract, the caller, not the called method needs to guarantee the precondition.

Preconditions and postconditions

For purposes of formal analysis, the program flow is analysed step by step, e.g., at the instruction (statement) level, at the level of blocks of code that form a coherent unit, or at the level of functions or methods.

Precondition: State of the program at a point directly before the considered unit. This may include assumptions taken from the design contract or specification.

Postcondition: State of the program at a point directly after the considered unit, assuming that the precondition was fulfilled at the point directly before it.

Example

As part of a development project, we need a function grtfrac(x, y) that takes **two floating-point arguments** and returns the one with the greater fractional part; *e.g.*, grtfrac(2.7, 3.6) is to return 2.7, because ".7" is greater than ".6". **In design by contract, the caller, not the called method needs to guarantee the precondition.**

Preconditions and postconditions

For purposes of formal analysis, the program flow is analysed step by step, e.g., at the instruction (statement) level, at the level of blocks of code that form a coherent unit, or at the level of functions or methods.

Precondition: State of the program at a point directly before the considered unit. This may include assumptions taken from the design contract or specification.

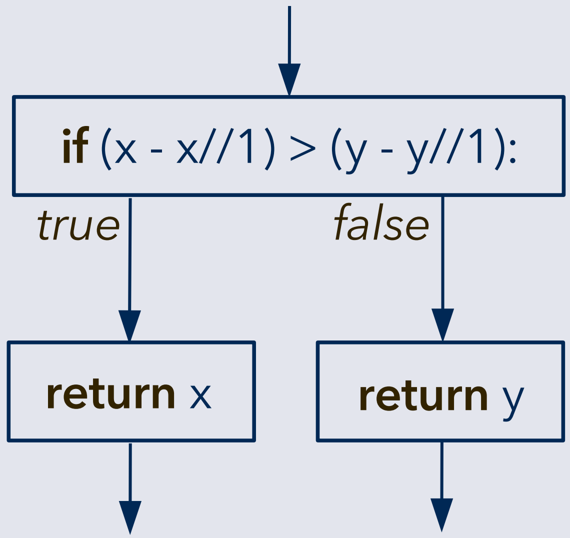
Postcondition: State of the program at a point directly after the considered unit, assuming that the precondition was fulfilled at the point directly before it.

Example

As part of a development project, we need a function grtfrac(x, y) that takes two floating-point arguments and returns the one with the greater fractional part; *e.g.*, grtfrac(–2.7, –1.8) is to return –2.7, because ".3" (or –0.7) is greater than ".2" (or –0.8).

Preconditions and postconditions

```
def grtfrac(x, y):
    if (x - x//1) > (y - y//1):
        return x
    else:
        return y
```

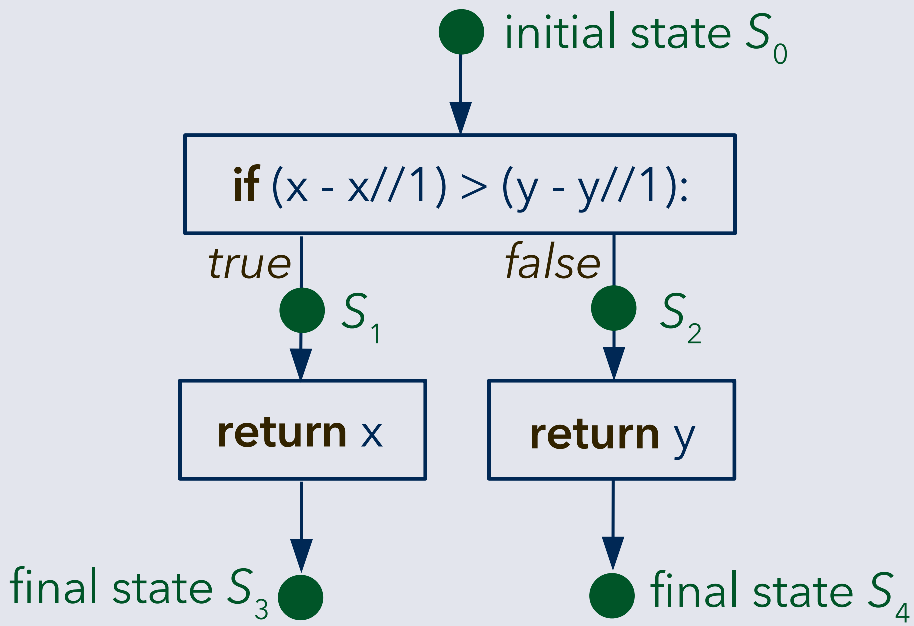


Example

As part of a development project, we need a function grtfrac(x, y) that takes two floating-point arguments and returns the one with the greater fractional part; *e.g.*, grtfrac(2.7, 3.6) is to return 2.7, because ".7" is greater than ".6".

Preconditions and postconditions

```
def grtfrac(x, y):
    if (x - x//1) > (y - y//1):
        return x
    else:
        return y
```

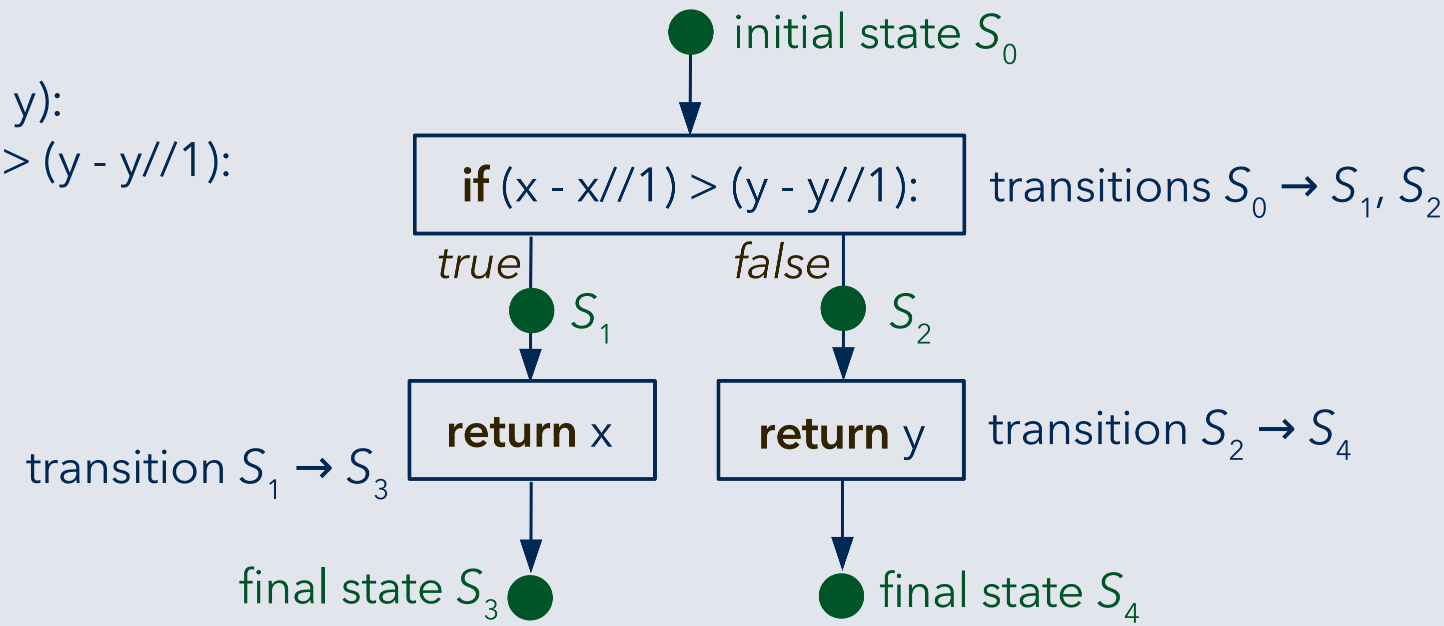


Example

As part of a development project, we need a function grtfrac(x, y) that takes two floating-point arguments and returns the one with the greater fractional part; *e.g.*, grtfrac(2.7, 3.6) is to return 2.7, because ".7" is greater than ".6".

Preconditions and postconditions

```
def grtfrac(x, y):
    if (x - x//1) > (y - y//1):
        return x
    else:
        return y
```

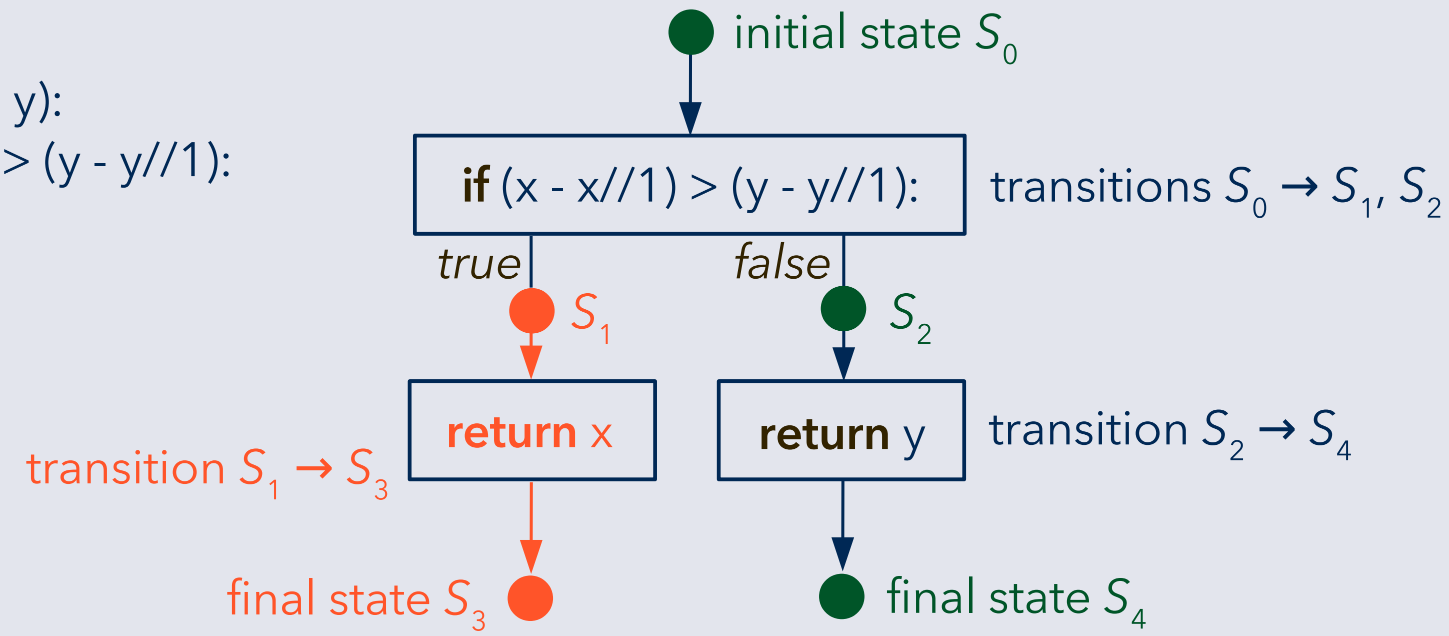


Example

As part of a development project, we need a function grtfrac(x, y) that takes two floating-point arguments and returns the one with the greater fractional part; *e.g.*, grtfrac(2.7, 3.6) is to return 2.7, because ".7" is greater than ".6".

Preconditions and postconditions

```
def grtfrac(x, y):
    if (x - x//1) > (y - y//1):
        return x
    else:
        return y
```



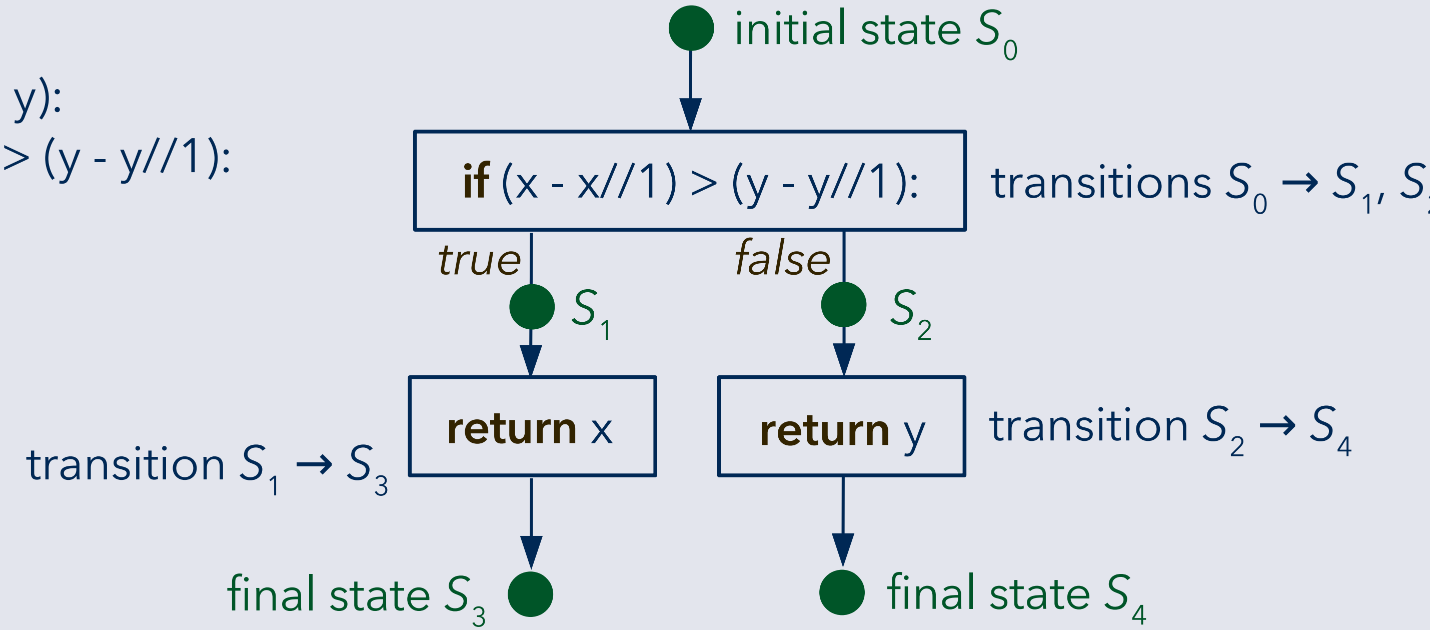
Note

Consider the statement "**return** x" from transition $S_1 \rightarrow S_3$:

- The execution state S_1 is the **precondition** of $S_1 \rightarrow S_3$.
- The execution state S_3 is the **postcondition** of $S_1 \rightarrow S_3$.

Preconditions and postconditions

```
def grtfrac(x, y):
    if (x - x//1) > (y - y//1):
        return x
    else:
        return y
```



S_0: x and y are floating-point numbers (by contract!).

S_1: x, y as above; the fractional part of x is greater than that of y.

S_2: x, y as above; the fractional part of y is greater than that of x, or equal.

S_3: The fractional part of x is the greater one, and x was returned.

S_4: The fractional part of y is the greater one (or they are equal); y was returned.

Problem: Matching natural numbers

```
def natmatch_iter(x, y):
    for i in range(len(x)):
        for j in range(i+1, len(x)):
            if (x[i]+x[j] == y) and (x[i] != x[j]):
                return [x[i], x[j]]
    return []
```

Examples:

If x is [17, 10, 4, 1] and y is 21, return any of [17, 4] or [4, 17].

If x is the same as above and y is 12, return the empty list [].

Specification

The function takes a list of natural numbers *x* as its first argument and a natural number *y* as its second argument. If in the list x, there are elements a and b which are not equal and add up to exactly y, the list [a, b] is returned; otherwise, [] is returned.

Problem: Matching natural numbers

```
def natmatch_iter(x, y):
   for i in range(len(x)):
      for j in range(i+1, len(x)):
         if (x[i]+x[j] == y) and (x[i] != x[j]):
            return [x[i], x[j]]
   return []
```

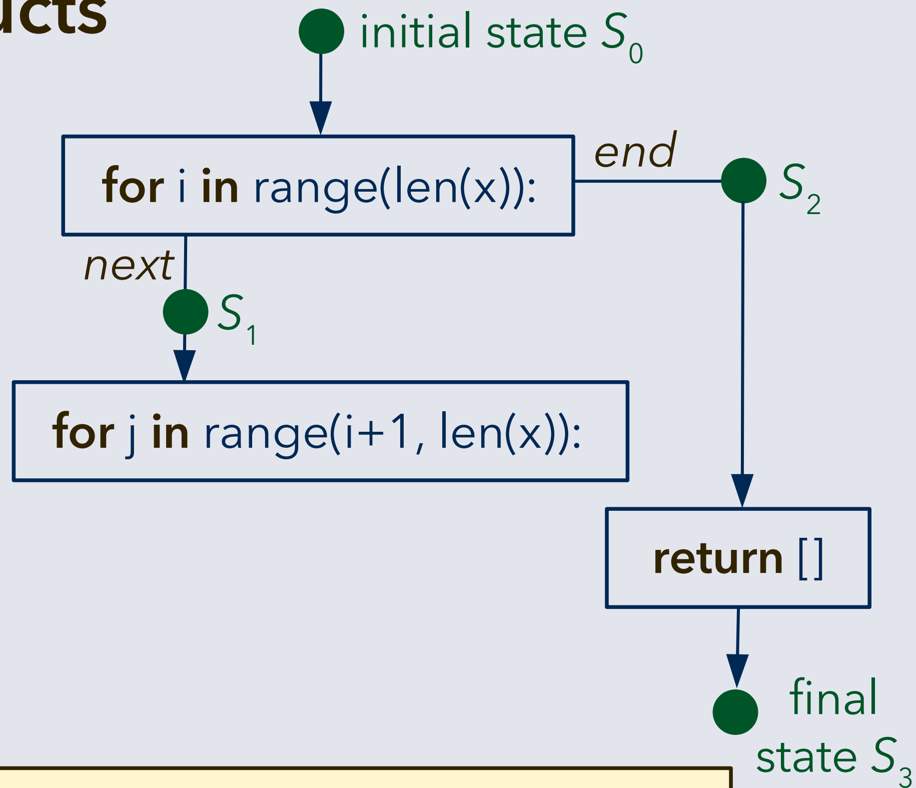
```
def natmatch_recur_core(x, y, l):
   if 1 >= l:
      return []
   else:
      for i in range(l-1):
         if (x[i]+x[l-1] == y) and (x[i] != x[l-1]):
            return [x[i], x[l-1]]
      return natmatch_recur_core(x, y, l-1)

def natmatch_recur(x, y):
   return natmatch_recur_core(x, y, len(x))
```

Specification

The function takes a list of natural numbers *x* as its first argument and a natural number *y* as its second argument. If in the list x, there are elements a and b which are not equal and add up to exactly y, the list [a, b] is returned; otherwise, [] is returned.

Verification of loop constructs

```
def natmatch_iter(x, y):
    for i in range(len(x)):
        for j in range(i+1, len(x)):
            if (x[i]+x[j] == y) and (x[i] != x[j]):
                return [x[i], x[j]]
    return []
```

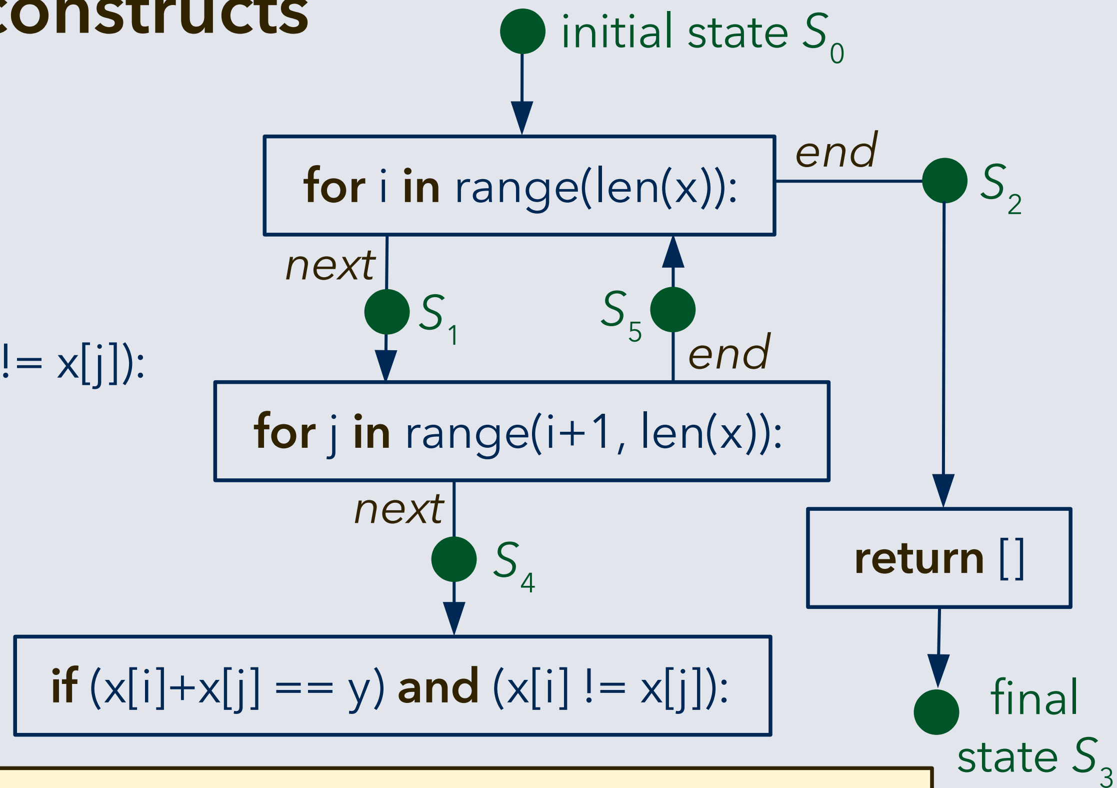


Specification

The function takes a list of natural numbers *x* as its first argument and a natural number *y* as its second argument. If in the list x, there are elements a and b which are not equal and add up to exactly y, the list [a, b] is returned; otherwise, [] is returned.

Verification of loop constructs

```
def natmatch_iter(x, y):
    for i in range(len(x)):
        for j in range(i+1, len(x)):
            if (x[i]+x[j] == y) and (x[i] != x[j]):
                return [x[i], x[j]]
    return []
```

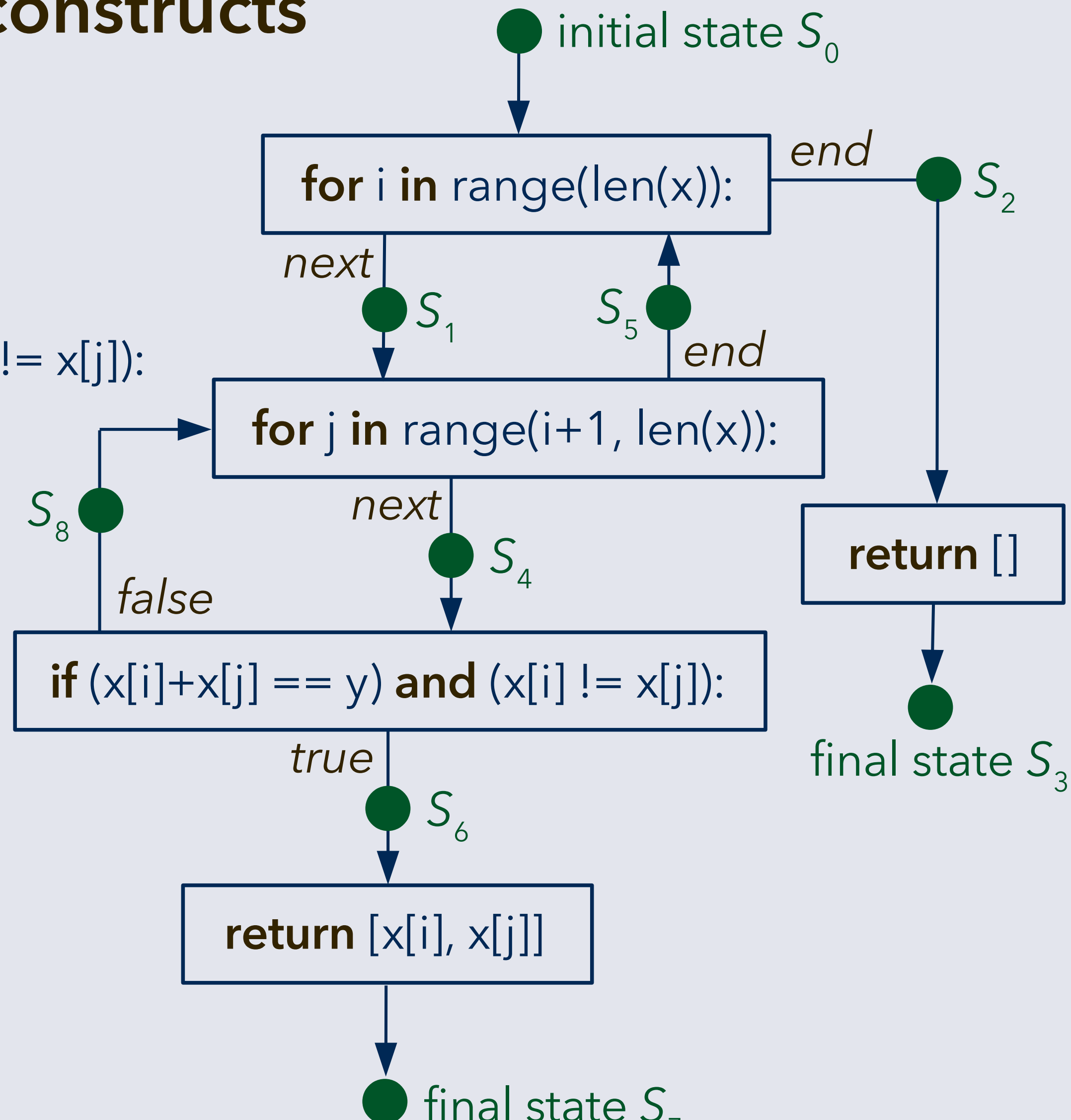


Specification

The function takes a list of natural numbers *x* as its first argument and a natural number *y* as its second argument. If in the list x, there are elements a and b which are not equal and add up to exactly y, the list [a, b] is returned; otherwise, [] is returned.

Verification of loop constructs

```
def natmatch_iter(x, y):
    for i in range(len(x)):
        for j in range(i+1, len(x)):
            if (x[i]+x[j] == y) and (x[i] != x[j]):
                return [x[i], x[j]]
    return []
```



Specification

The function takes a list *x* and a natural number *y* as arguments. If in the list x, there are elements a and b which are not equal and add up to y, the list [a, b] is returned; otherwise, [] is returned.

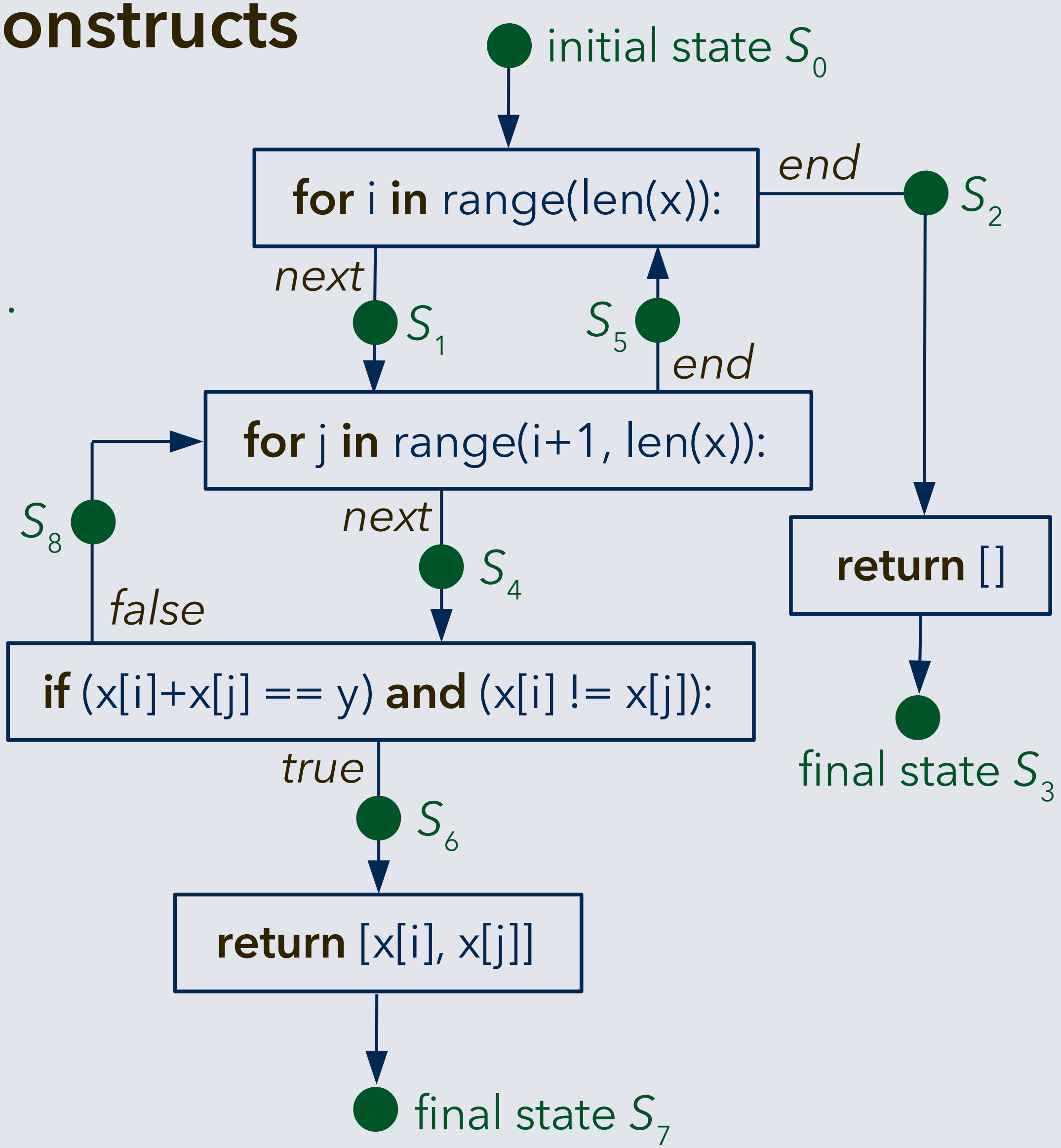
Verification of loop constructs

S_0: x and y given as specified.

S_1, first iteration: i ≡ 0, len(x) ≥ 1.

S_4, first iteration: i ≡ 0, j ≡ 1, len(x) ≥ 2.

S_8, first iteration: As above, and x[0] + x[1] do not match.



Verification of loop constructs

S_0: x and y given as specified.

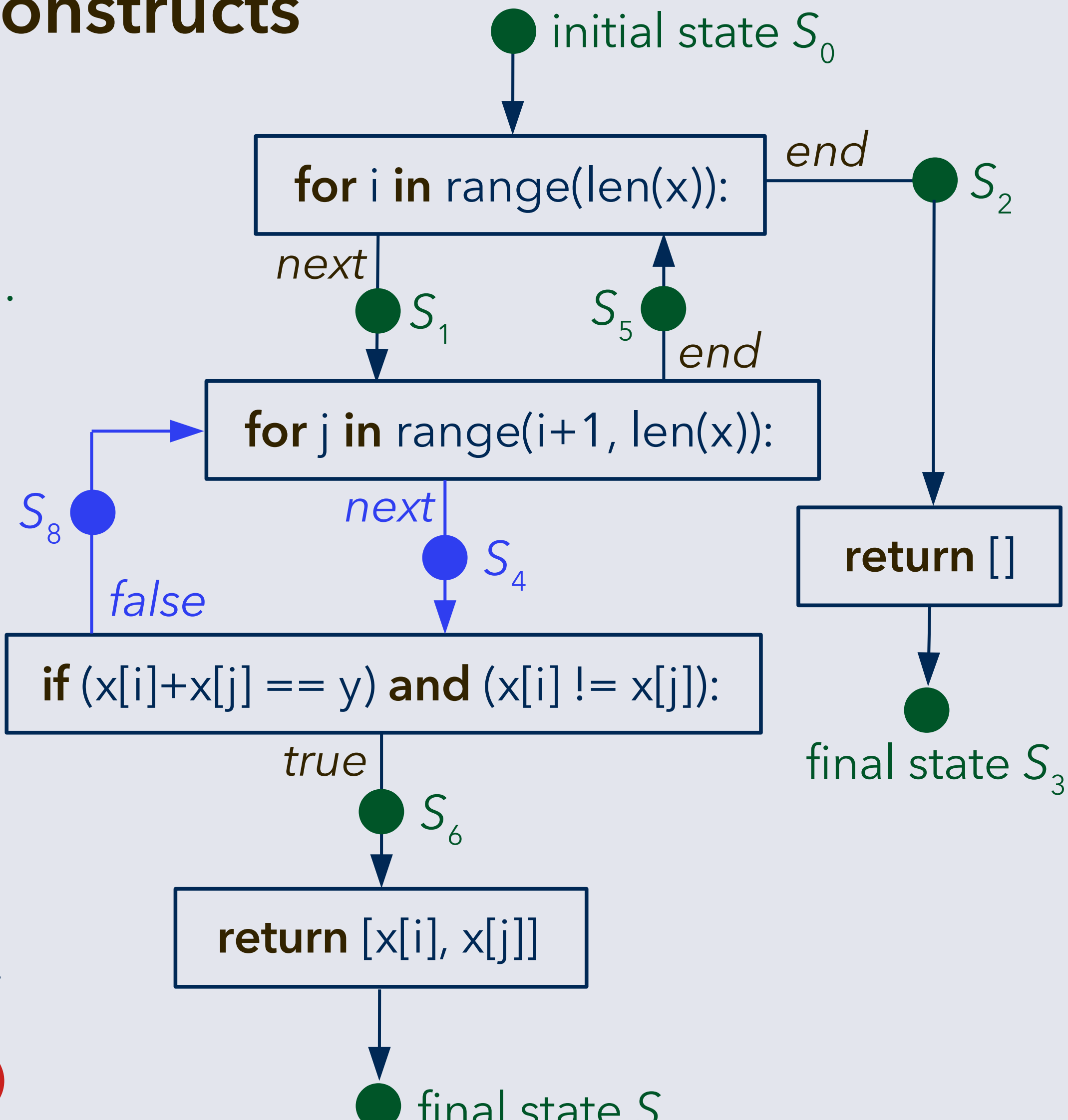
S_1, first iteration: i ≡ 0, len(x) ≥ 1.

S_4, first iteration: i ≡ 0, j ≡ 1, len(x) ≥ 2.

S_8, first iteration: As above, and x[0] + x[1] do not match.

Task now: Find a **loop invariant**, one that holds every time that the execution states S_4 and S_8 inside the inner loop are visited.

(Still assuming that i remains 0.)



Verification of loop constructs

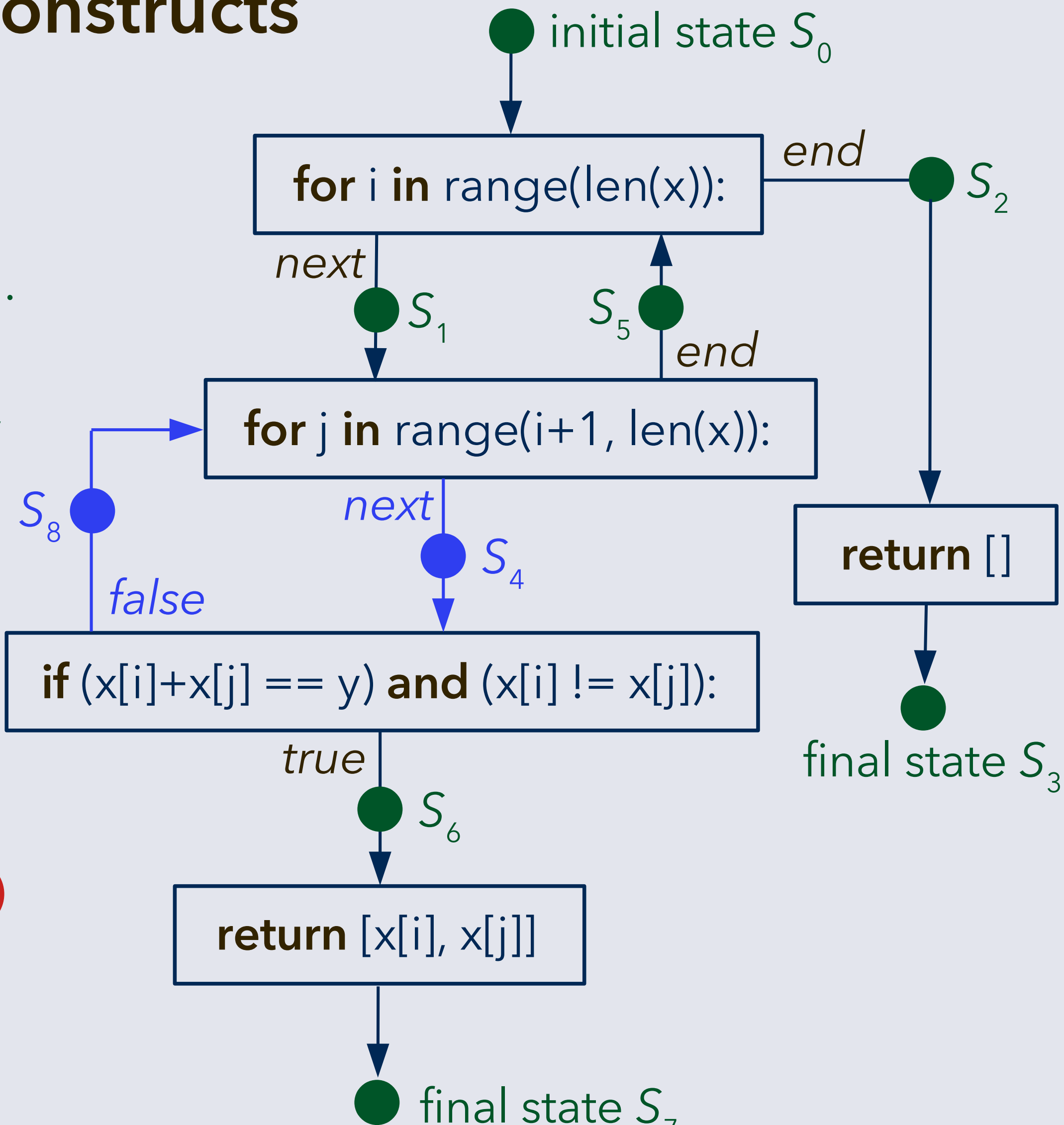
S_0: x and y given as specified.

S_1, first iteration: i ≡ 0, len(x) ≥ 1.

***S*₄, invariant**: i ≡ 0, 0 < j < len(x), and x[0] does not match with any x[k] for indices k < j.

***S*₈, invariant**: As above, but x[0] does not match with any x[k] for indices k ≤ j.

(Still assuming that i remains 0.)



Verification of loop constructs

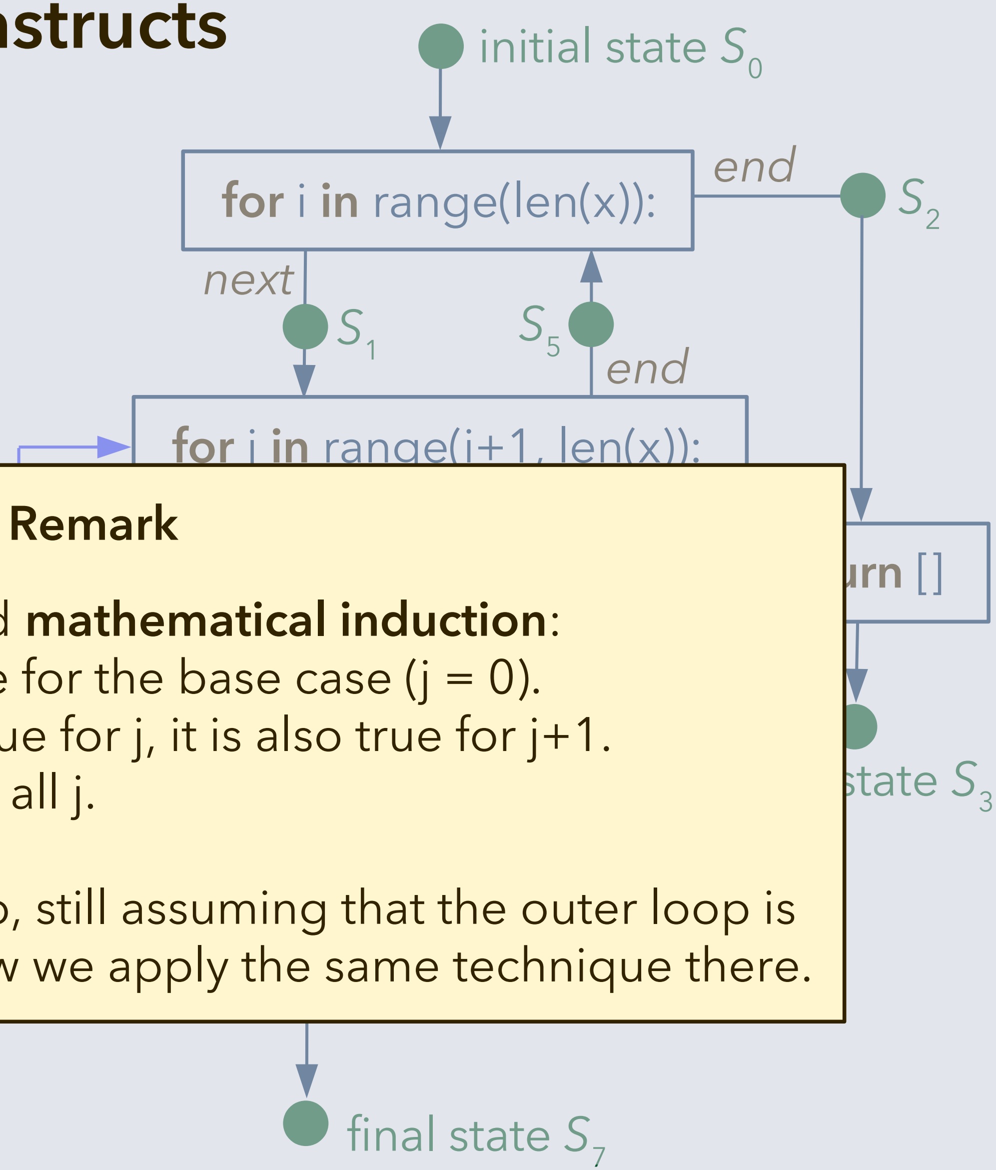
S_0: x and y given as specified.

S_1, first iteration: i ≡ 0, len(x) ≥ 1.

S_4, invariant: i ≡ 0, 0 < j < len(x),
and
any x

S_8, in
x[0]
x[k] f

(Still



Remark

This proof technique is called **mathematical induction**:

- We know that it is true for the base case (j = 0).
- We know that if it is true for j, it is also true for j+1.
- Therefore it is true for all j.

We did this for the inner loop, still assuming that the outer loop is in its first iteration (i = 0). Now we apply the same technique there.

Verification of loop constructs

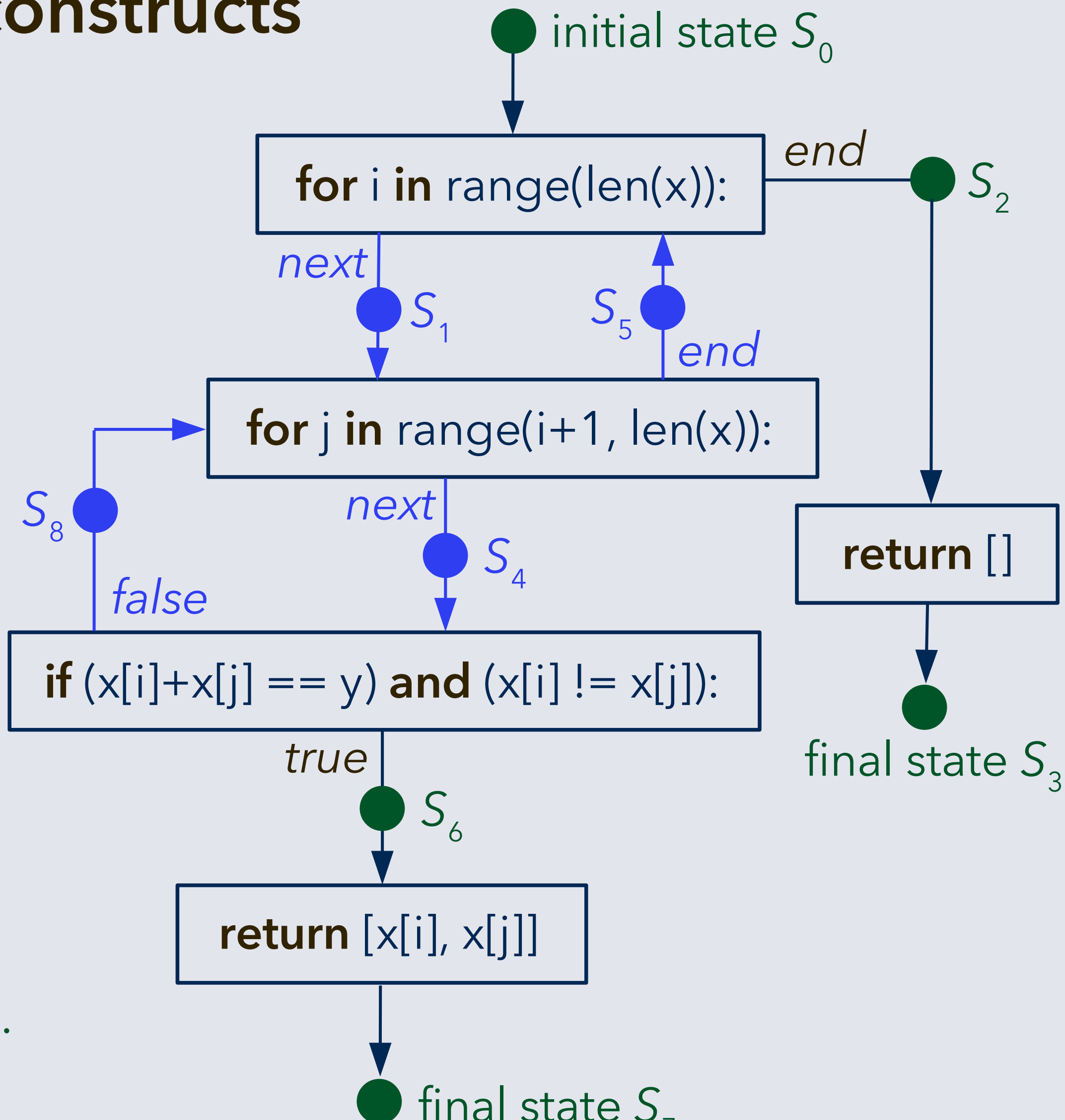
S_0: x and y given as specified.

S_1, for i ≡ 0: i ≡ 0, len(x) ≥ 1.

S_4, for i ≡ 0: i ≡ 0, 0 < j < len(x), and x[0] does not match with any x[k] for indices k < j.

S_8, for i ≡ 0: As above, but x[0] does not match with any x[k] for indices k ≤ j.

S_5, for i ≡ 0: i ≡ 0, and after trying all indices 0 < j < len(x), x[0] was not found to match any.



Verification of loop constructs

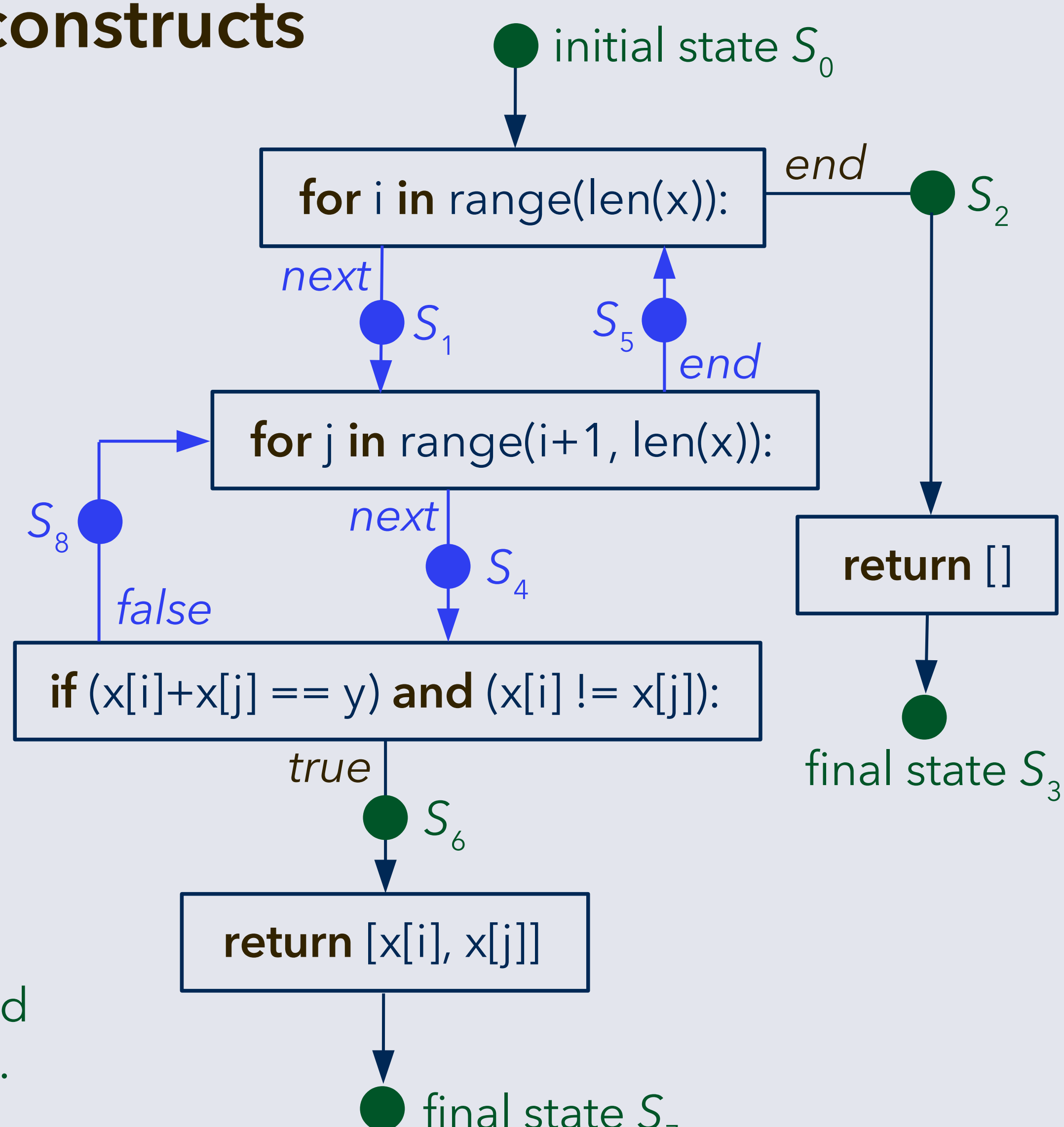
S_0: x and y given as specified.

S_1, invariant: 0 ≤ i < len(x).

S_4, invariant: 0 ≤ i < len(x), i < j < len(x), all indices smaller than i did not yield a match, and x[i] does not match with any x[k] for indices i < k < j.

S_8, invariant: As above, and x[i] does not match with any x[k] for indices i < k ≤ j.

S_5, invariant: As above, and we now know that x[i] does not yield a match with any other element. (And neither did any smaller i.)

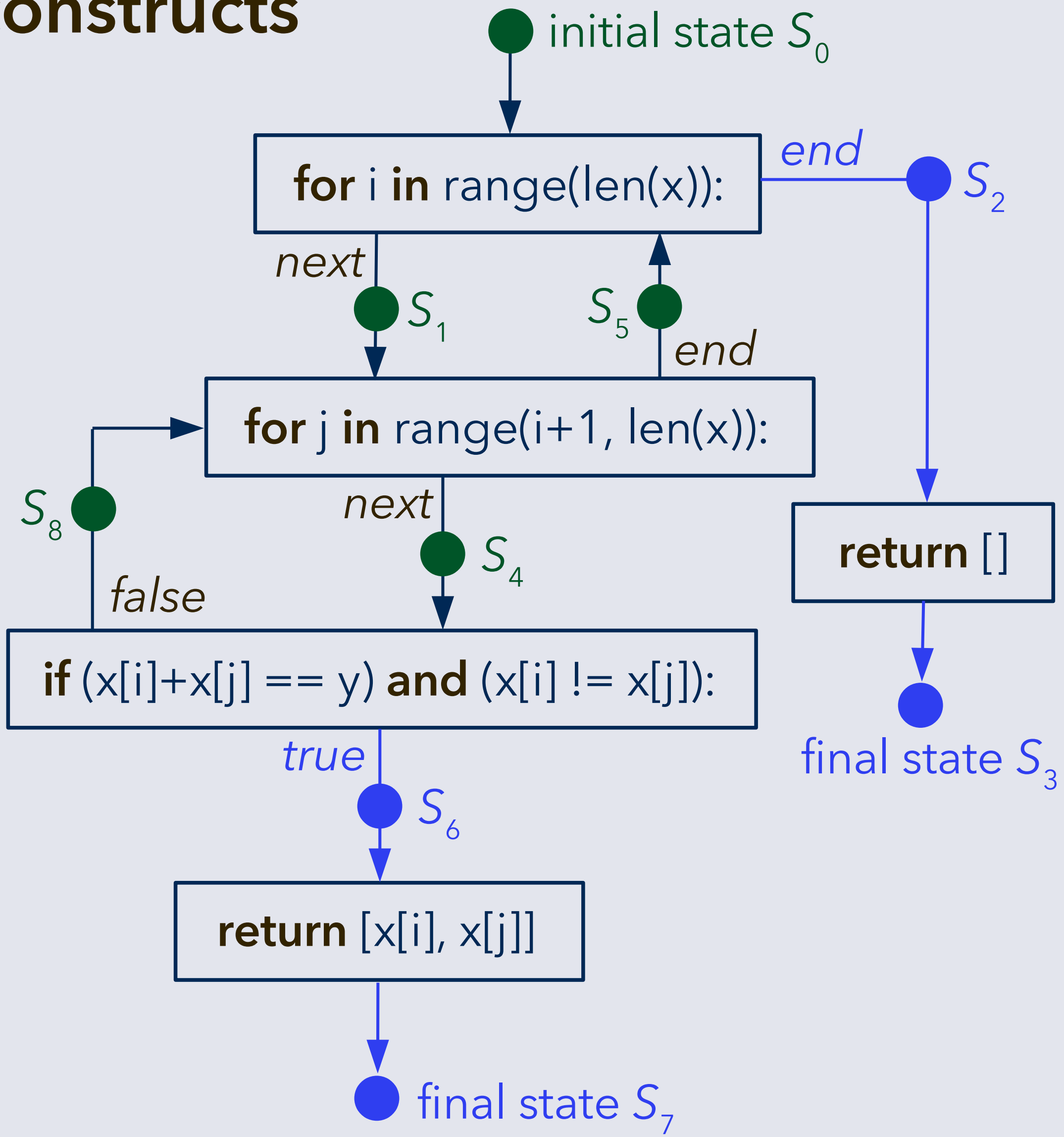


Verification of loop constructs

S_0: x and y given as specified.

S_1: 0 ≤ i < len(x).

S_5: No combination of any x[i] that was tried so far, with any x[j] from the list where i < j, produces a valid match.



Verification of loop constructs

S_0: x and y given as specified.

S_1: $0 \le i < len(x)$.

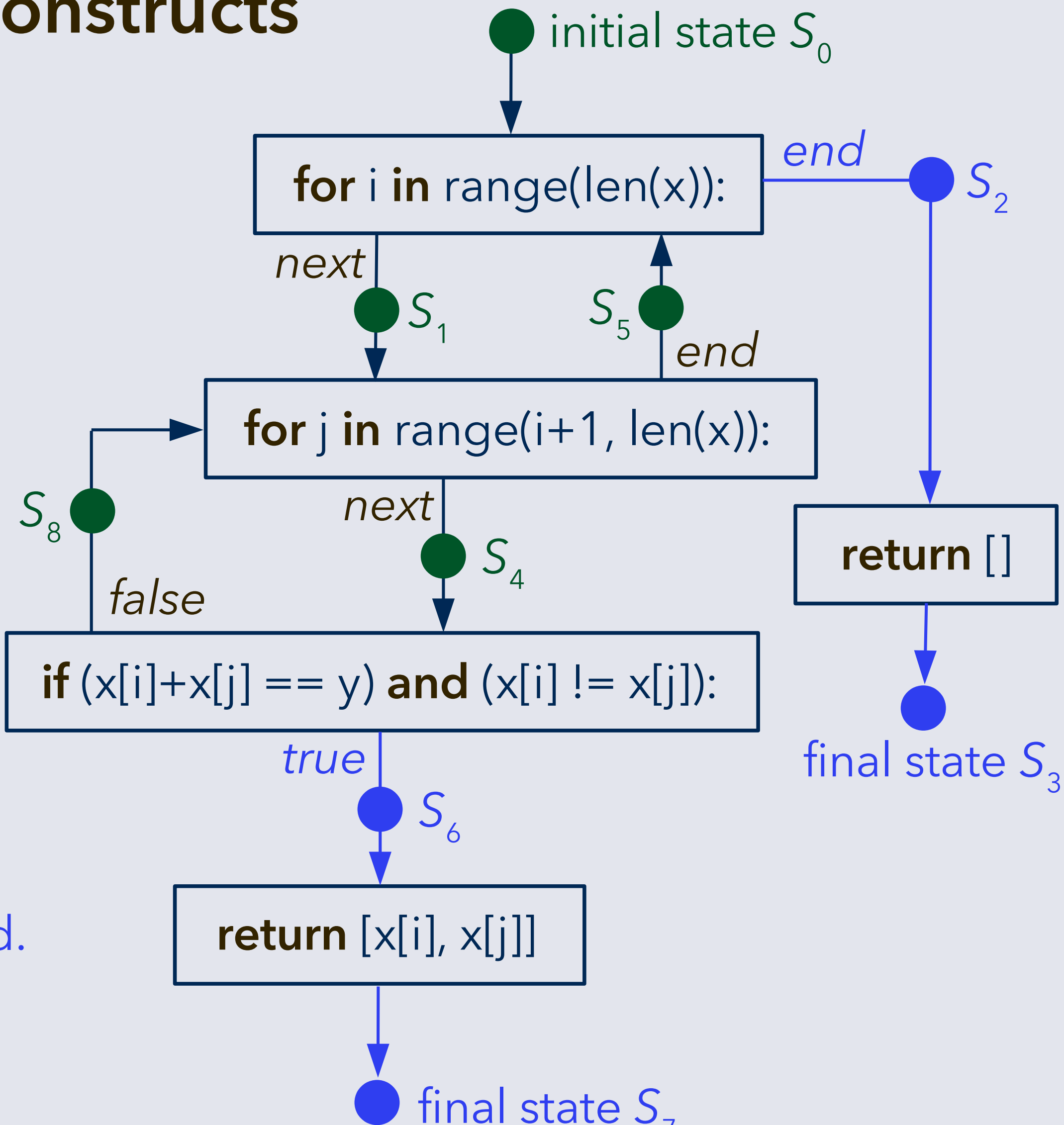
S_5: No combination of any x[i] that was tried so far, with any x[j] from the list where i < j, produces a valid match.

S_6: x[i] and x[j] are a match.

S_7: Match found and returned.

S_2: End of list, all pairs of elements were tried, none matched.

S_3: No match; [] was returned.



Performance and efficiency

CO2412

19th October 2021

University of Central Lancashire UCLan

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

Performance as a function of the problem size

Usually we are not interested in the resource requirements of a single execution, but in understanding how the requirements behave as a function of a characteristic quantity, the **problem size** ***n***, that describes the magnitude of the task.

We distinguish between:

- **Time requirements**, describing the computing time. Where possible, this should be expressed in terms of actual CPU time (+ I/O time); the operating system will usually distribute CPU time between multiple processes.
- **Memory (or space) requirements**, describing the memory allocated to the program; depending on definition, this may include I/O size.

Performance as a function of the problem size

Usually we are not interested in the resource requirements of a single execution, but in understanding how the requirements behave as a function of a characteristic quantity, the **problem size *n***, that describes the magnitude of the task.

We distinguish between:

- **Time requirements**, describing the computing time. Where possible, this should be expressed in terms of actual CPU time (+ I/O time); the operating system will usually distribute CPU time between multiple processes.
- **Memory (or space) requirements**, describing the memory allocated to the program; depending on definition, this may include I/O size.

- **Worst-case performance**, which for any given problem size n corresponds to the input/special case of size n with the greatest requirements.
- **Average-case performance**, over many representative cases of size *n*.

There is also "best-case performance," but usually not as an evaluation criterion.

Algorithm efficiency as a function of problem size

Usually we are not interested in the efficiency of an algorithm for a single input value, but in understanding how the efficiency behaves as a function of a characteristic quantity, the **problem size *n***, that describes the magnitude of the task.

We distinguish between:

- **Time efficiency measure(s)**, describing CPU time in an abstract way; one possible measure for it is the number of code/pseudocode instructions.
- **Space or memory efficiency measure(s)**, describing the memory in an abstract way, *e.g.*, by the number of elementary values stored in variables, data structures, or files; this usually excludes the initial input.

- **Worst-case efficiency**, which for any given problem size n corresponds to the special case of size n with the greatest computing time/memory.
- **Average-case efficiency**, over all (or many representative) cases of size *n*.

There is also "best-case efficiency," but usually not as an evaluation criterion.

Algorithm efficiency as a function of problem size

Usually we are not interested in the efficiency of an algorithm for a single input value, but in understanding how the efficiency behaves as a function of a characteristic quantity, the **problem size** n, that describes the magnitude of the task.

We distinguish between:

Observation

Performance analysis is carried out by measurements; it is usually very hard to determine the worst case, therefore it is common to describe the **average-case performance**, *e.g.*, from random input.

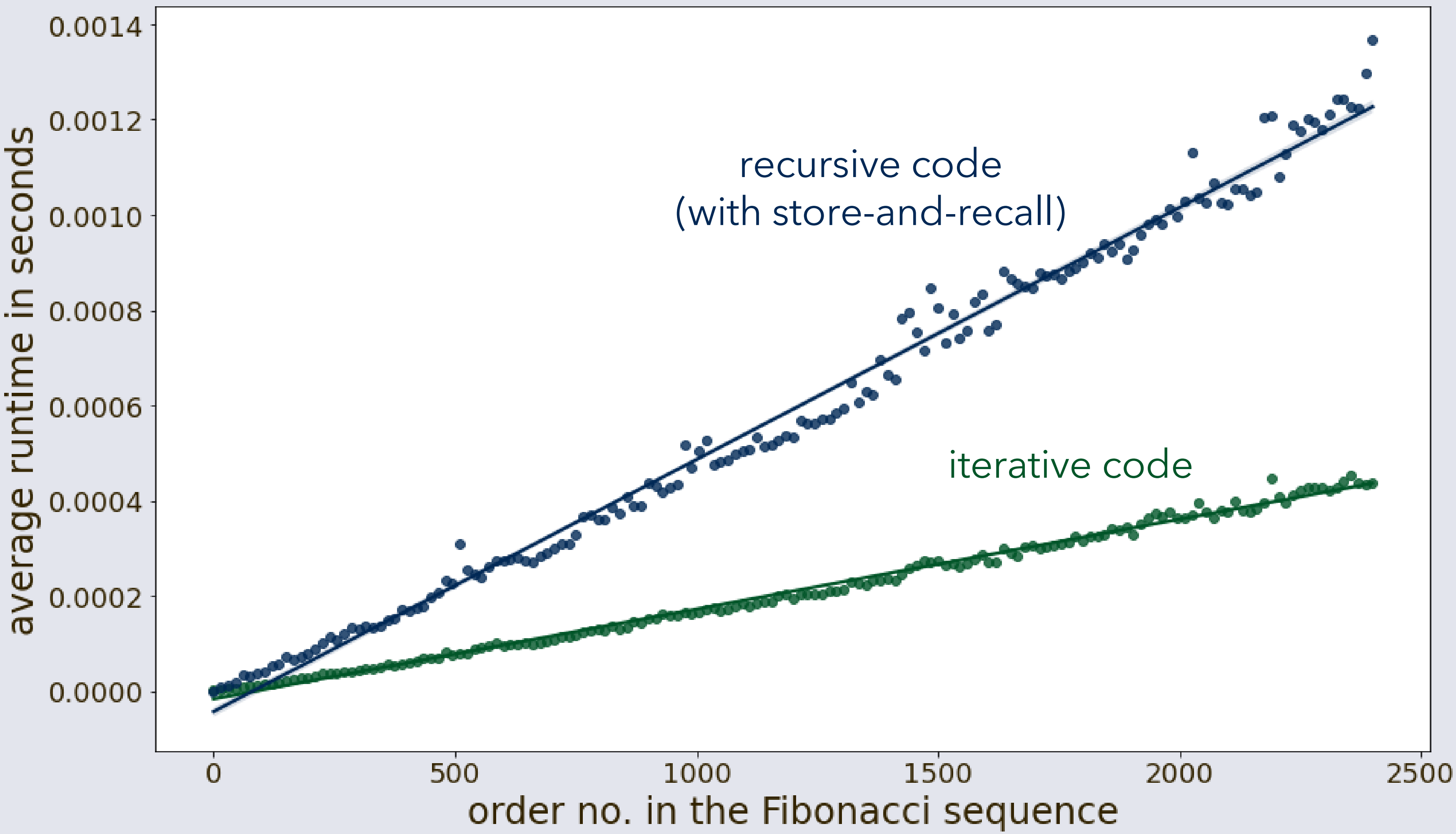
Algorithm efficiency can consider both the average and the worst case, but the average case usually requires a statistical analysis. Statements on the **worst case** can be very straightforward.

y; one
tions.
n an
aria-

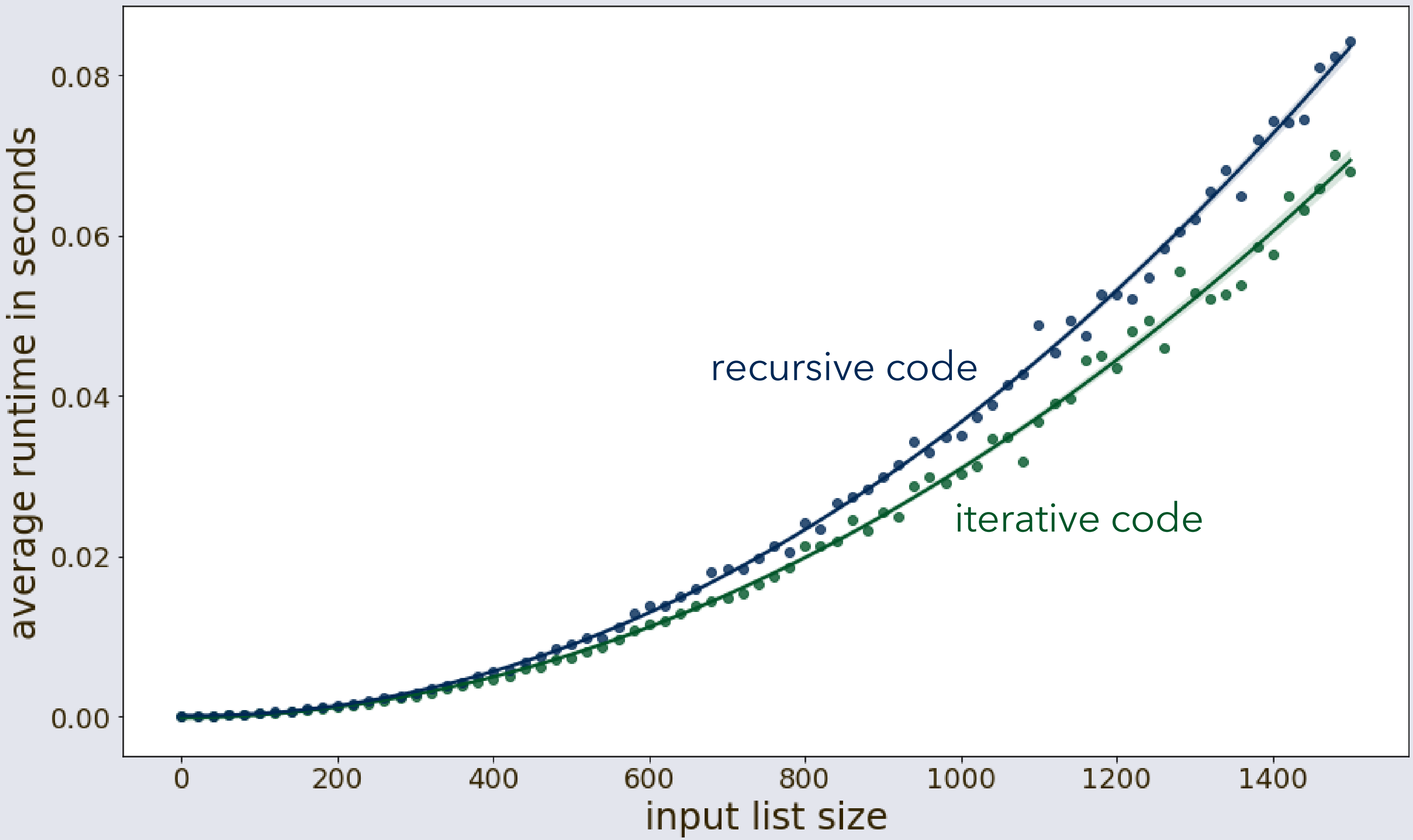
nds
ory.
size n.

There is also "best-case efficiency," but usually not as an evaluation criterion.

Example: Fibonacci sequence



Example: Number matching



Asymptotic efficiency (or performance)

Often we are most interested in the **qualitative scaling behaviour** of algorithms.

For this purpose, **Landau notation** is used,[1] also known as "big O notation." For any given efficiency or requirements measure, this is obtained as follows:

- Eliminate all except the leading contribution, *i.e.*, the one that dominates the measure for large values of n. It is the one that grows fastest:
  - From $3n^3 + 12n + 17$, we retain only $3n^3$.
  - From $16 \cdot 2^n + 5n^3$, we retain only $16 \cdot 2^n$.
  - If you are unsure, insert $n = 1000$ and see which term is greatest.

- Eliminate constant coefficients; $3n^3$ becomes $O(n^3)$, $16 \cdot 2^n$ becomes $O(2^n)$.

[1] Named for Edmund Landau (1877 – 1938) who developed this notation for infinitesimal calculus.

Asymptotic efficiency (or performance)

Often we are most interested in the **qualitative scaling behaviour** of algorithms.

For this purpose, **Landau notation** is used,[1] also known as "big O notation." For any given efficiency or requirements measure, this is obtained as follows:

- Eliminate all except the leading contribution, *i.e.*, the one that dominates the measure for large values of n. It is the one that grows fastest:
  - From $3n^3 + 12n + 17$, we retain only $3n^3$.
  - From $16 \cdot 2^n + 5n^3$, we retain only $16 \cdot 2^n$.
  - If you are unsure, insert $n = 1000$ and see which term is greatest.
- Eliminate constant coefficients; $3n^3$ becomes $O(n^3)$, $16 \cdot 2^n$ becomes $O(2^n)$.

If an algorithm includes $3n^3 + 12n + 17$ instructions in the worst case, we can say, it is in time efficiency class $O(n^3)$, or simply, it has time efficiency $O(n^3)$.

[1]Named for Edmund Landau (1877 – 1938) who developed this notation for infinitesimal calculus.

University of Central Lancashire
UCLan

CO2412 Computational Thinking

Formal verification
Performance and efficiency

Where opportunity creates success