University of Central Lancashire
UCLan

CO2412
Computational Thinking

Formal verification #2
Algorithmic efficiency #2
Terminology and building a glossary

Where opportunity creates success

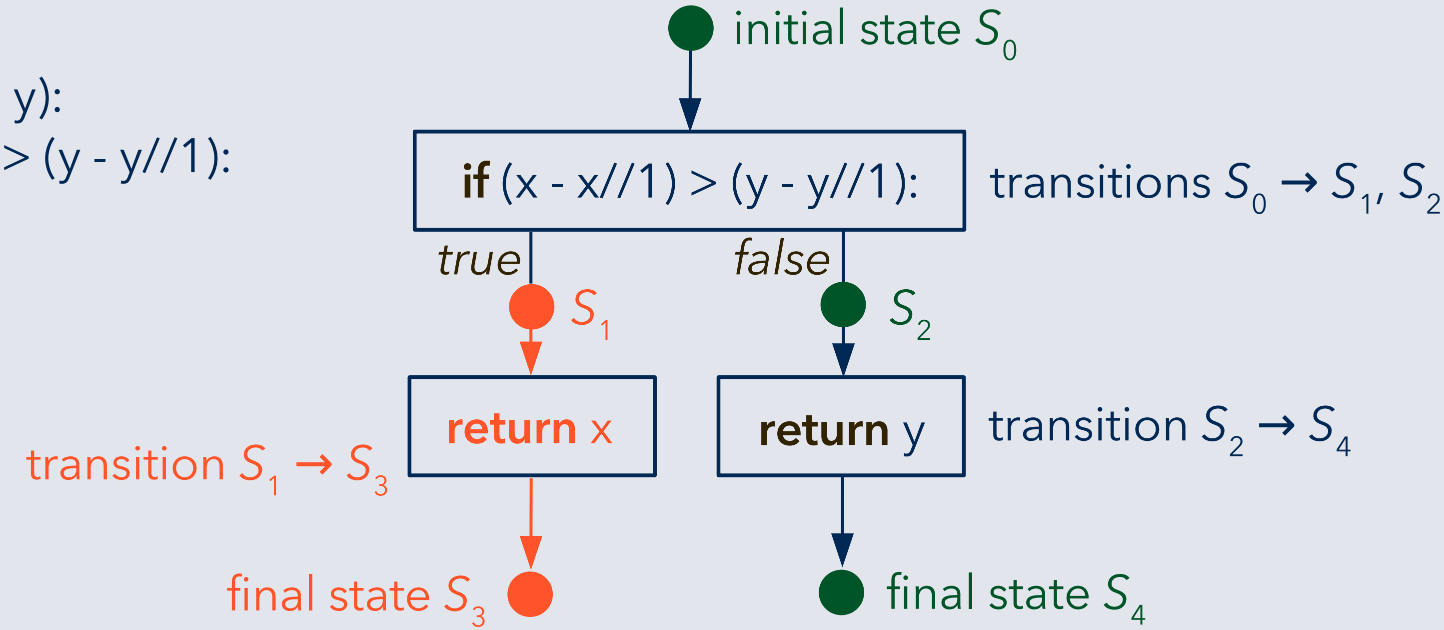
Formal verification #2

CO2412

26th October 2021

Preconditions and postconditions

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



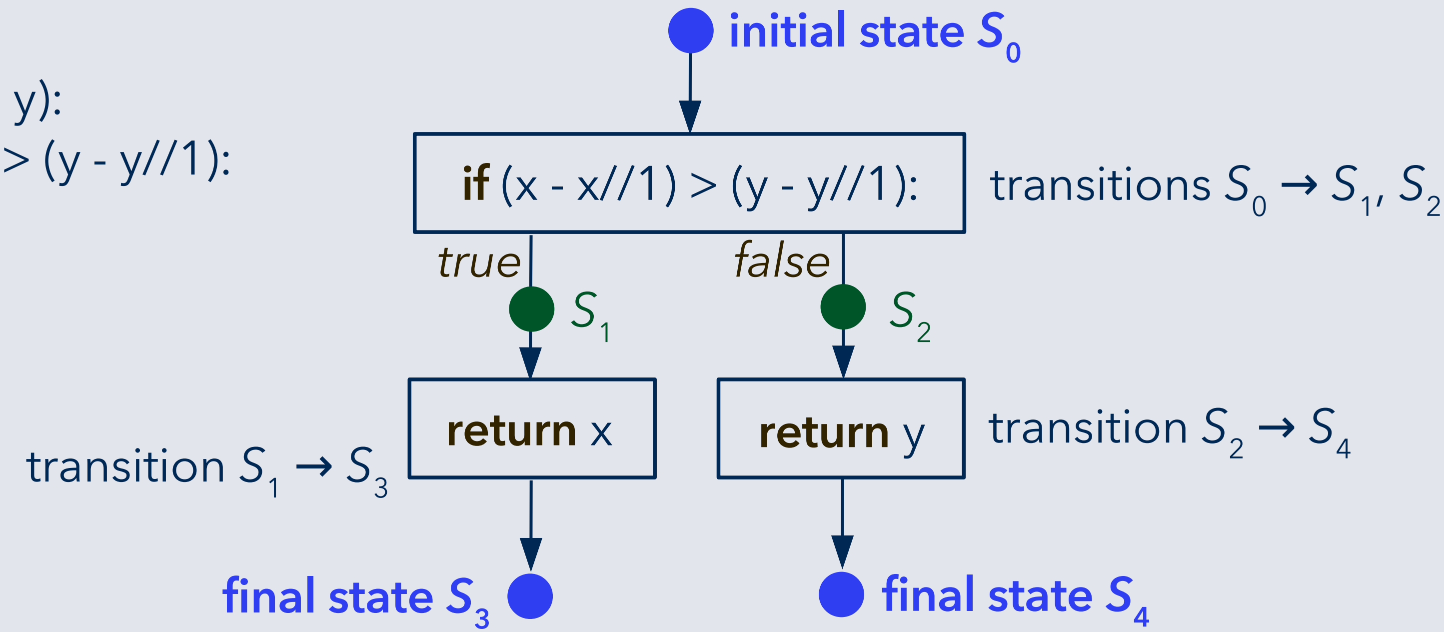
Note

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 $S_1 \rightarrow S_3$.

Initial and final conditions matching the specification

```
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 specification).

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 greater (or they are equal); y was returned.

Loop invariants

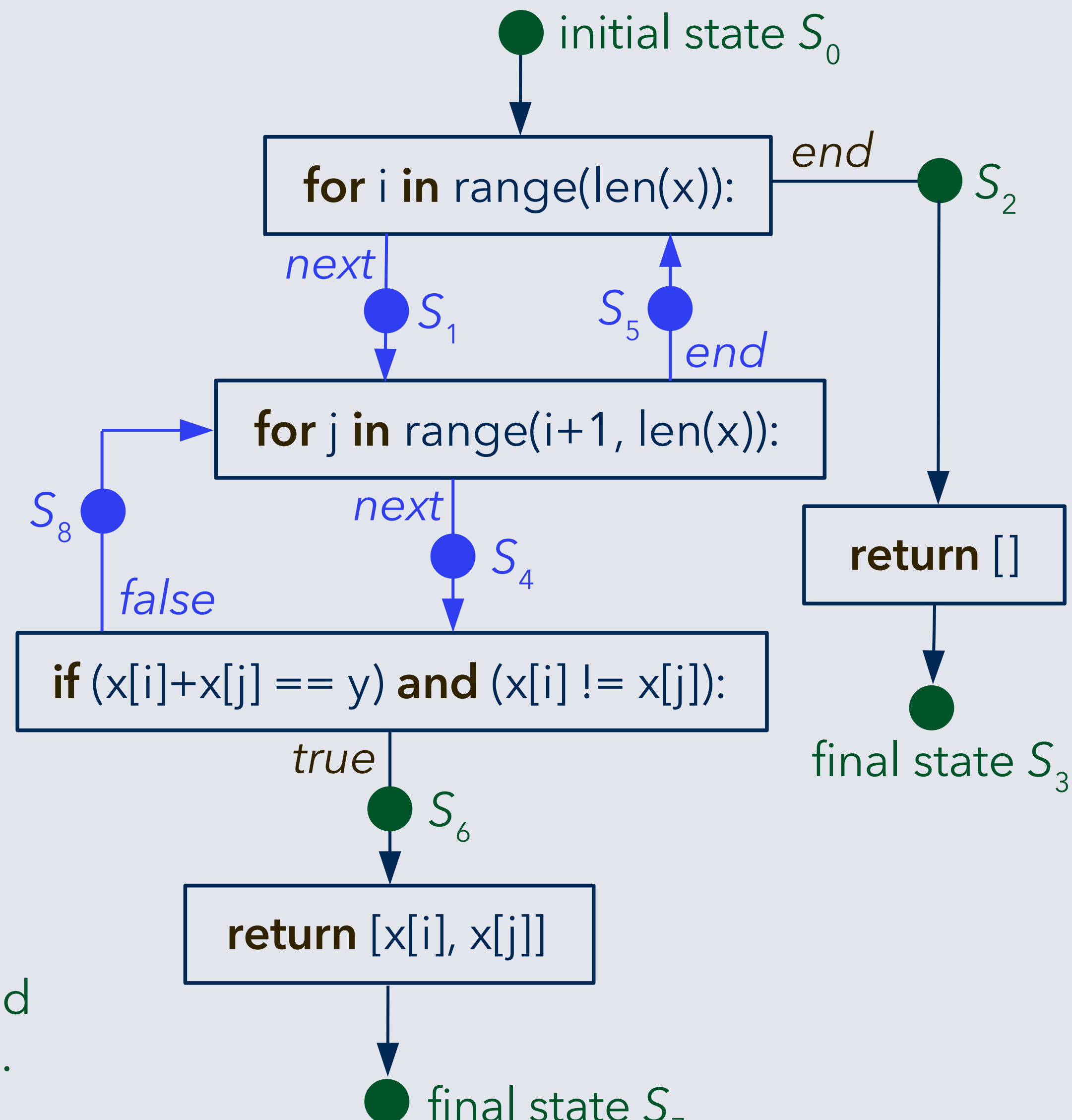
S_0: x and y given as specified.

***S_1*, invariant**: $0 \leq i < \text{len}(x)$.

***S_4*, invariant**: $0 \leq i < \text{len}(x)$, $i < j < \text{len}(x)$, 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 $i < k \leq j$.

***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.)



Initial and final conditions

S_0: x and y given as specified.

S_1: 0 ≤ i < len(x).

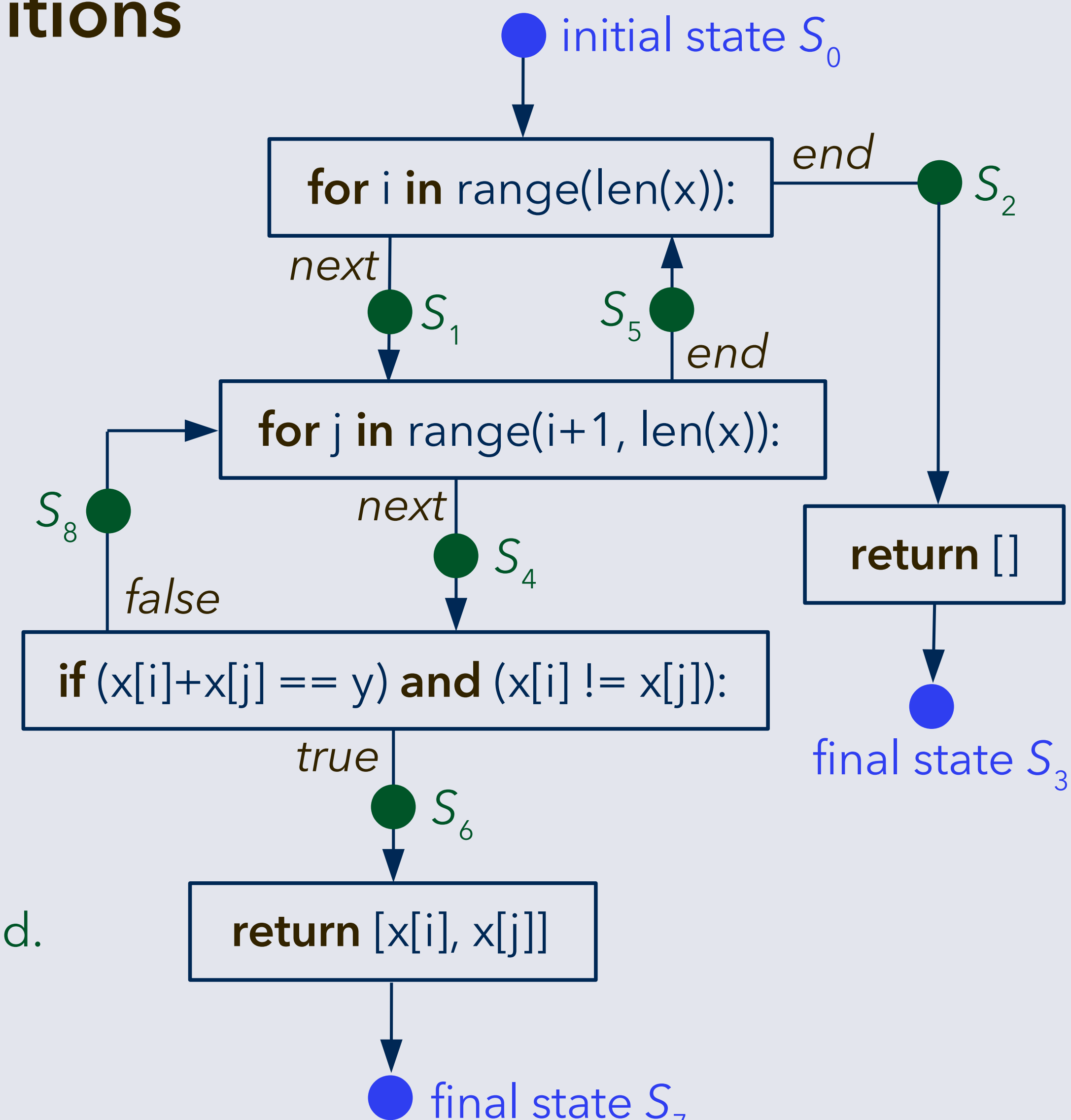
S_5: No combination of any x[i] that was tried so far, with any x[j] from the list where i < j, produces a valid match.

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.

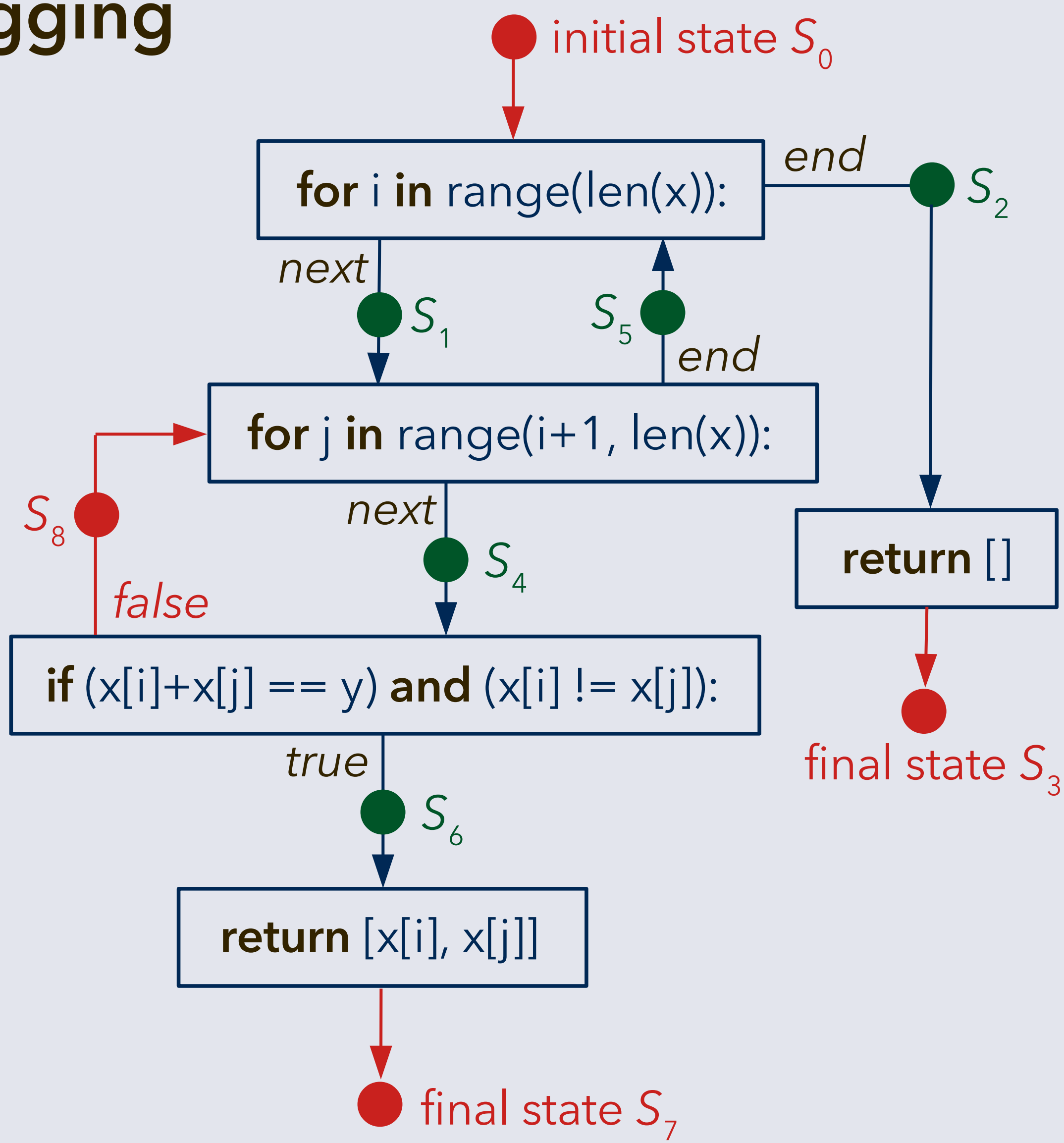
S_3: No match; [] was returned.



Application to debugging

Breakpoints at:

- S_0, output x and y
- S_3, output status message
- S_7, output i, j, x[i], x[j], and their sum
- S_8, output i, j, x[i], x[j], and their sum



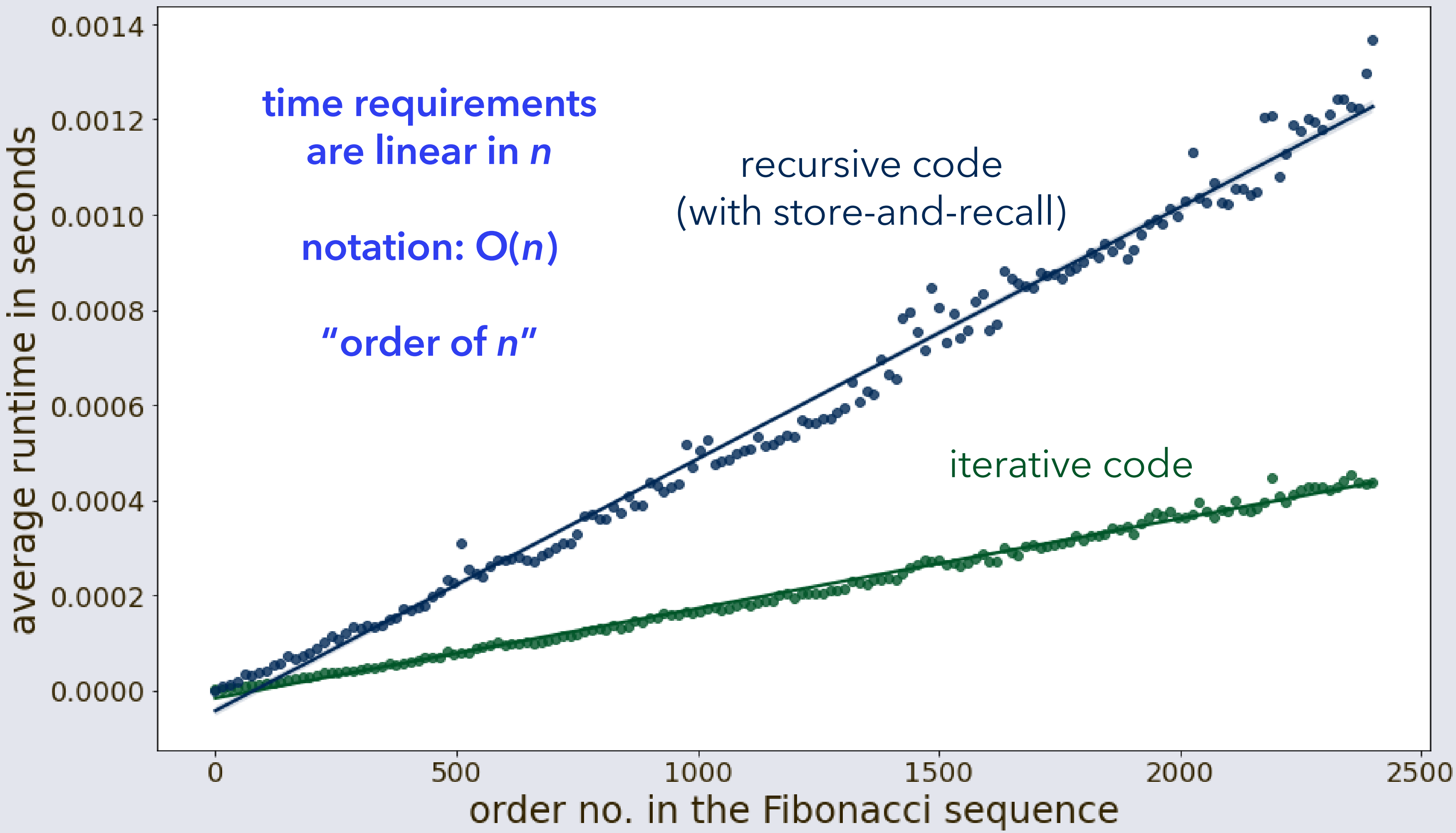
Algorithmic efficiency #2

CO2412

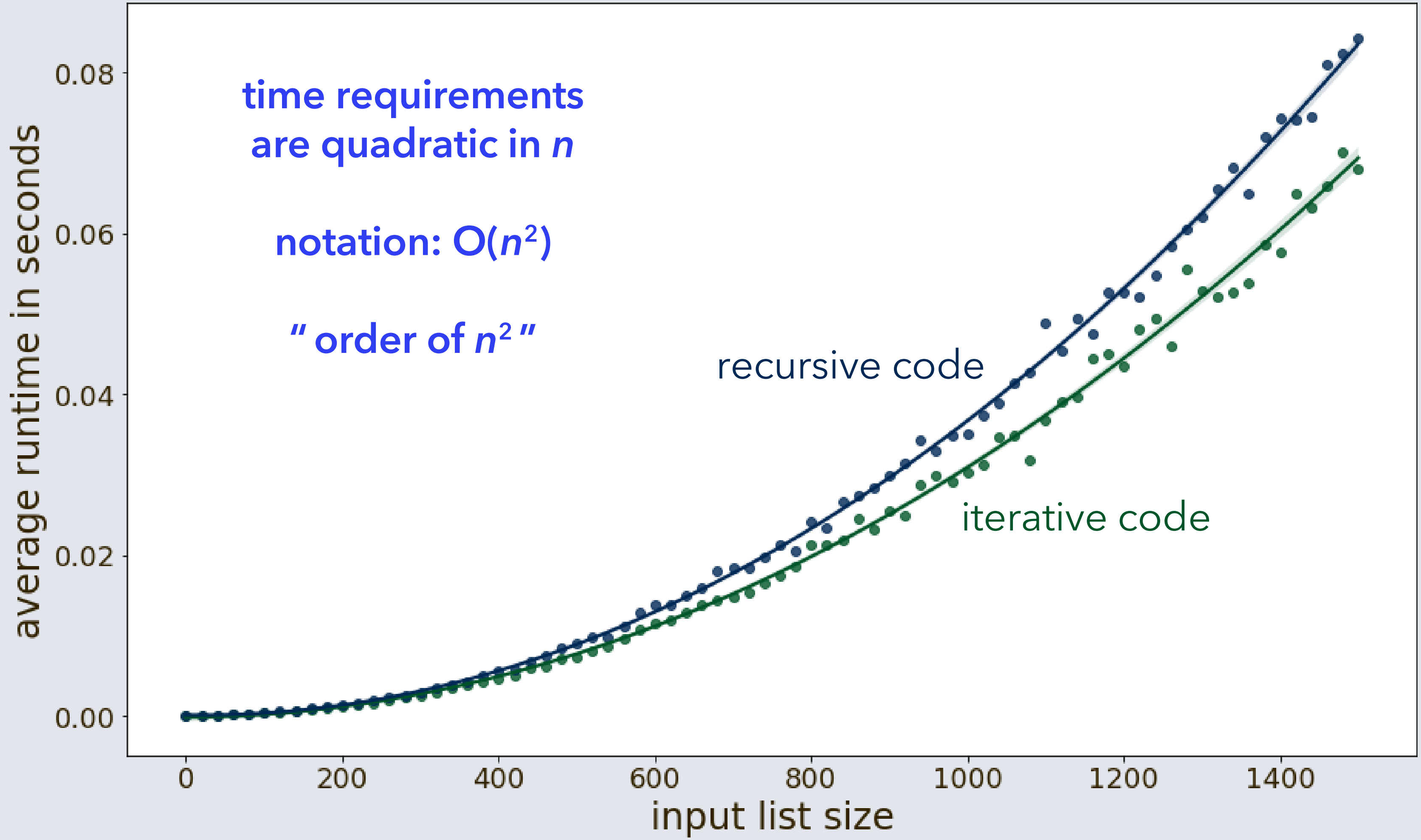
26th October 2021

University of Central Lancashire UCLan

Average performance of Fibonacci codes



Average performance of number-matching codes



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:

Remark

There is no universal rule for how the **problem size *n*** should be defined. It is up to the person analysing an algorithm to define it appropriately. It should describe how complicated the task is.

Common choices are the length of the input (*e.g.*, if given as an array or string), the value passed as of one of the arguments of a function, or the number of elements stored in a data structure.

; one
ons.
an
ia-

nds
ory.
ize n.

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

Asymptotic efficiency

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 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

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 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.

Asymptotic efficiency

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 measure, this is obtained as follows:

inates

eatest.

$O(2^n)$.

Observation

Landau notation describes the "shape of the curve" for great n.

The asymptotic efficiency class of an algorithm is the same as the asymptotic performance class of any reasonable implementation.

For most algorithms, the distinction between the average and the worst case disappears if considered in terms of Landau notation.

If a … can say, it is in time efficiency class $O(n^{\cdot})$, or simply, it has time efficiency $O(n^{\cdot})$.

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

Asymptotic efficiency

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

For t … For
any g …

– … nates

Examples

The Fibonacci algorithms have $O(n)$ time and space efficiency.

The number matching algorithms have $O(n^2)$ time efficiency; the iterative one has $O(1)$ space efficiency, the recursive one $O(n)$.

- 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 any leading factors; $3n^3$ 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.

Asymptotic efficiency

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

For t … For any g …

– … nates

– 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 any leading factors; $3n^3$ becomes $O(n^3)$, $16{\cdot}2^n$ becomes $O(2^n)$.

If an … n say, i …

[1]Name … culus.

Examples

The Fibonacci algorithms have $O(n)$ time and space efficiency.

The number matching algorithms have $O(n^2)$ time efficiency; the iterative one has $O(1)$ space efficiency,[1] the recursive one $O(n)$.

Note

[1]Unless you count the input size, which would contribute in $O(n)$. This is why input size is usually excluded from space efficiency.

Asymptotic efficiency

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

Examples

The Fibonacci algorithms have $O(n)$ time and space efficiency.

The number matching algorithms have $O(n^2)$ time efficiency; the iterative one has $O(1)$ space efficiency,[1] the recursive one $O(n)$.

Is this the best possible asymptotic efficiency, or can it be done in a better way? This is a topic both for **algorithm design** (find better solutions) and **complexity theory** (prove general lower bounds).

For t … For
any g …

… nates

… atest.

– Eliminate any leading factors; $3n^3$ becomes $O(n^3)$, $16\cdot 2^n$ becomes $O(2^n)$.

Note

[1]Unless you count the input size, which would contribute in $O(n)$. This is why input size is usually excluded from space efficiency.

If an … n
say, i …

[1]Name … culus.

Why does the Fibonacci algorithm take linear time?

def fibonacci_iter(n):	
fibo = [0, 1]	2 instructions
for k **in** range(2, n+1):	loop executed n – 1 times: – 1 instruction for the loop index
fibo.append(fibo[k-1] + fibo[k-2])	– 4 instructions
return fibo[n]	1 instruction

$5(n - 1) + 3 \;=\; 5n - 2$ instructions

O(n) time efficiency

Why does the Fibonacci algorithm take linear time?

def fibonacci_iter(n):	
fibo = [0, 1]	2 instructions
for k **in** range(2, n+1):	loop executed $n - 1$ times: – 1 instruction for the loop index
fibo.append(fibo[k-1] + fibo[k-2])	– 4 instructions
return fibo[n]	1 instruction
	$5(n - 1) + 3 = 5n - 2$ instructions
	O(n) time efficiency

The number of "instructions" assumed above is rather arbitrary. Asymptotic efficiency analysis simplifies this. In particular, any constants become "O(1)".

Why does the Fibonacci algorithm take linear time?

def fibonacci_iter(n):	
fibo = [0, 1]	**O(1) instructions**
for k **in** range(2, n+1):	loop executed **O(*n*) times**:
fibo.append(fibo[k-1] + fibo[k-2])	– **O(1) instructions**
return fibo[n]	**O(1) instructions**
	$O(n \cdot 1) + O(1) = O(n)$ instructions
	O(*n*) time efficiency

The number of "instructions" assumed above is rather arbitrary. Asymptotic efficiency analysis simplifies this. In particular, any constants become "O(1)".

Why does our matching code take quadratic time?

```
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 []
```

Note: Input size *n* given by len(x)

loop executed $O(n)$ times:

- loop executed $O(n)$ times:
  - $O(1)$ instructions
  - $O(1)$ optional instructions

$O(1)$ optional instructions

$O(n) \cdot O(n \cdot 1) + O(1) = O(n^2)$ instructions

$O(n^2)$ time efficiency

Memory efficiency evaluation

```
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 []
```

Note: Input size *n* given by len(x)

1 variable (i); used over all iterations

- 1 variable (j); over all iterations
  - no new variables
  - no new variables

no new variables

---

2 variables overall, therefore O(1)

O(1) memory efficiency

Memory efficiency evaluation

```
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]]
```

Note: Input size *n* given by len(x)

1 variable (i); used over all iterations

- 1 variable (j); over all iterations
  - no new variables
  - no new variables

no new variables

---

2 variables overall, therefore O(1)

O(1) memory efficiency

If we include memory requirements for storing the input, this gives $n + 3$, therefore $O(n)$. It is common **not to include the input**, since it existed before; it does not need any **additional** memory.

Landau notation: Examples

1. Eliminate all except the leading contribution, *i.e.*, the one that dominates the measure for large values of n; the one that grows fastest:

2. Eliminate constant coefficients; replace them by a factor 1.

Efficiency measure as a function of *n*	**Landau notation for the measure**
$24\boldsymbol{n}^2 + 4n + 600$	$O(n^2)$
$7\boldsymbol{n}^{1/2} + 3$	$\boxed{n^{1/2} = \sqrt{n}}$
$(n + 1)(n + 2)$	
$3(n^{1/2} + 5 \log n) \cdot n$	

Landau notation: Examples

> **1. Eliminate all except the leading contribution**, *i.e.*, the one that dominates the measure for large values of n; the one that grows fastest:
>
> **2. Eliminate constant coefficients**; replace them by a factor 1.

Efficiency measure as a function of n	Landau notation for the measure
$24\boldsymbol{n}^2 + \cancel{4n} + \cancel{600}$	$O(n^2)$
$7\boldsymbol{n}^{1/2} + \cancel{3}$	$O(n^{1/2})$
$(n + 1)(n + 2)$	
$3(n^{1/2} + 5 \log n)\cdot n$	

Landau notation: Examples

1. Eliminate all except the leading contribution, *i.e.*, the one that dominates the measure for large values of n; the one that grows fastest:

2. Eliminate constant coefficients; replace them by a factor 1.

Efficiency measure as a function of n	Landau notation for the measure
$24\boldsymbol{n^2} + \cancel{4}n + \cancel{600}$	$O(n^2)$
$7\boldsymbol{n^{1/2}} + \cancel{3}$	$O(n^{1/2})$
$(n+1)(n+2) = \boldsymbol{n^2} + \cancel{3}n + \cancel{2}$	$O(n) \cdot O(n) = O(n^2)$
$3(\boldsymbol{n^{1/2}} + 5 \,\boldsymbol{\log n})\cdot \boldsymbol{n}$	

Remark on logarithms

In general, the logarithm is the inverse operation to exponentiation; both require a base. However, for "log x," a base is often assumed from context.

$$\boxed{y = b^x \quad \Leftrightarrow \quad x = \log_b y}$$

Convention in engineering and natural sciences

If no base is given, $\log n$ means $\log_{10} n$, i.e., the decimal or decadic logarithm.

$\log_{10} 1 = 0$, $\log_{10} 10 = 1$, $\log_{10} 100 = 2$, $\log_{10} 1000 = 3$, …

Remark on logarithms

In general, the logarithm is the inverse operation to exponentiation; both require a base. However, for "log x," a base is often assumed from context.

$$\boxed{y = b^x \quad \Leftrightarrow \quad x = \log_b y}$$

Convention in engineering and natural sciences

If no base is given, $\log n$ means $\log_{10} n$, i.e., the decimal or decadic logarithm.

$\log_{10} 1 = 0,\ \log_{10} 10 = 1,\ \log_{10} 100 = 2,\ \log_{10} 1000 = 3, \ldots$

Convention in mathematics

If no base is given, $\log n$ means $\ln n$, the natural logarithm (base $e = 2.71828\ldots$).

$\ln 1 = 0,\ \ln e = 1,\ \ln e^2 = 2,\ \ln e^3 = 3, \ldots$ $\boxed{y = e^x \quad \Leftrightarrow \quad y = \exp(x) \quad \Leftrightarrow \quad x = \ln y}$

Remark on logarithms

In general, the logarithm is the inverse operation to exponentiation; both require a base.

$$\boxed{\frac{\log_p n}{\log_q n} = \log_p q = \text{const.}}$$

Convention in engineering and natural sciences

If no base is given, $\log n$ means $\log_{10} n$, i.e., the decimal or decadic logarithm.

$\log_{10} 1 = 0,\ \log_{10} 10 = 1,\ \log_{10} 100 = 2,\ \log_{10} 1000 = 3, \ldots$

Convention in theoretical computer science

If no base is given, $\log n$ means $\log_2 n$, i.e., the binary logarithm.

$\log_2 1 = 0,\ \log_2 2 = 1,\ \log_2 4 = 2,\ \log_2 256 = 8,\ \log_2 1024 = 10, \log_2 65536 = 16, \ldots$

Landau notation: Examples

1. Eliminate all except the leading contribution, *i.e.*, the one that dominates the measure for large values of n; the one that grows fastest:

2. Eliminate constant coefficients; replace them by a factor 1.

Efficiency measure as a function of n | **Landau notation for the measure**

n	=	1	4	16	64	256	1024	4096	…
$\log n$	=	0	2	4	6	8	10	12	
$n^{1/2}$	=	1	2	4	8	16	32	64	

$3(\boldsymbol{n^{1/2}} + 5 \log n)\cdot\boldsymbol{n}$ $\qquad$ $O(n^{1/2}) \cdot O(n) = O(n^{1/2} \cdot n^1) = O(n^{3/2})$

… or simply $O\left(n\sqrt{n}\right)$

Time efficiency classification: Example

Specification: The function has two arguments, a **list x** containing n = len(x) integer numbers, where multiple elements are allowed to have the same value, and a single-digit integer **0 ≤ y ≤ 9**. The function determines three numbers:

- **q_1, the number of elements** of x with **y as their final digit**.
  If the same number occurs twice in the list, it also counts twice.
  In other words, q_1 is the number of indices i such that “x[i] % 10 == y”.

  For x = [**7**, 9, 4, **17**, **7**, 3] and y = 7, the value of q_1 would be 3.

  This corresponds to the three indices 0, 3, and 4.

Time efficiency classification: Example

Specification: The function has two arguments, a **list x** containing n = len(x) integer numbers, where multiple elements are allowed to have the same value, and a single-digit integer **$0 \leq y \leq 9$**. The function determines three numbers:

- **q_1, the number of elements** of x with **y as their final digit**.
  If the same number occurs twice in the list, it also counts twice.
  In other words, q_1 is the number of indices i such that "x[i] % 10 == y".
- **q_2, the number of combinations of two indices** i and j, with i ≠ j, such that the product **x[i] · x[j] has the remainder y** upon division by 10. In other words, q_2 is the number of ordered pairs (i, j) with "x[i]*x[j] % 10 == y".
  As a consequence, each pair counts twice, once as (i, j), once as (j, i).

  For x = [7, 9, 4, 17, 7, 3] and y = 7, the value of q_2 would be 2.

  This corresponds to the two ordered pairs of indices (1, 5) and (5, 1).

Time efficiency classification: Example

Specification: The function has two arguments, a **list x** containing n = len(x) integer numbers, where multiple elements are allowed to have the same value, and a single-digit integer **0 ≤ y ≤ 9**. The function determines three numbers:

- ***q_1*, the number of elements** of x with **y as their final digit**.
  If the same number occurs twice in the list, it also counts twice.
  In other words, q_1 is the number of indices i such that "x[i] % 10 == y".
- ***q_2*, the number of combinations of two indices** i and j, with i ≠ j, such that the product **x[i] · x[j] has the remainder y** upon division by 10. In other words, q_2 is the number of ordered pairs (i, j) with "x[i]*x[j] % 10 == y".
  As a consequence, each pair counts twice, once as (i, j), once as (j, i).
- ***q_3*, the number of combinations of three indices** i, j, k such that the product **x[i] · x[j] · x[k] has y as its final digit**; x[i], x[j], x[k] may be the same, but i, j, k must be three different indices. Every such triple occurs in six permutations: (i, j, k), (i, k, j), (j, i, k), (j, k, i), (k, i, j), (k, j, I) – they count as six.

The function returns a list containing the three values [q_1, q_2, q_3].

Time efficiency classification: Example

Specification: The function has two arguments, a **list x** containing n = len(x) integer numbers, where multiple elements are allowed to have the same value, and a single-digit integer **0 ≤ y ≤ 9**. The function returns the list $[q_1, q_2, q_3]$.

Problem size defined as n = len(x).

```
def mod10_count_naive(x, y):
    q1, q2, q3 = 0, 0, 0
    for i in range(len(x)):
        if x[i] % 10 == y:
            q1 += 1
        for j in range(len(x)):
            if i == j:
                continue
            elif (x[i]*x[j]) % 10 == y:
                q2 += 1
            for k in range(len(x)):
                if i == k or j == k:
                    continue
                elif (x[i]*x[j]*x[k]) % 10 == y:
                    q3 += 1
    return [q1, q2, q3]
```

Time efficiency classification: Example

Specification: The function has two arguments, a **list x** containing n = len(x) integer numbers, where multiple elements are allowed to have the same value, and a single-digit integer **0 ≤ y ≤ 9**. The function returns the list $[q_1, q_2, q_3]$.

Problem size defined as n = len(x).

```
def mod10_count_naive(x, y):
    q1, q2, q3 = 0, 0, 0
    for i in range(len(x)):
        if x[i] % 10 == y:
            q1 += 1
        for j in range(len(x)):
            if i == j:
                continue
            elif (x[i]*x[j]) % 10 == y:
                q2 += 1
            for k in range(len(x)):
                if i == k or j == k:
                    continue
                elif (x[i]*x[j]*x[k]) % 10 == y:
                    q3 += 1
    return [q1, q2, q3]
```

done once

done n times

done n^2 times

done n^3 times

done once

Time efficiency classification: Example

Specification: The function has two ar…
co…
int…
wh…
ele…
all…
sa…
single-digit integer **0 ≤ y ≤ 9**. The function returns the list [q_1, q_2, q_3].

Problem size defined as *n* = len(x).

```
def mod10_count_naive(x, y):
    q1, q2, q3 = 0, 0, 0
    …)):
        …
        …n(x)):
            …
            …10 == y:
                q2 += 1
            for k in range(len(x)):
                if i == k or j == k:
                    continue
                elif (x[i]*x[j]*x[k]) % 10 == y:
                    q3 += 1
    return [q1, q2, q3]
```

done once

done *n* times

done n^2 times

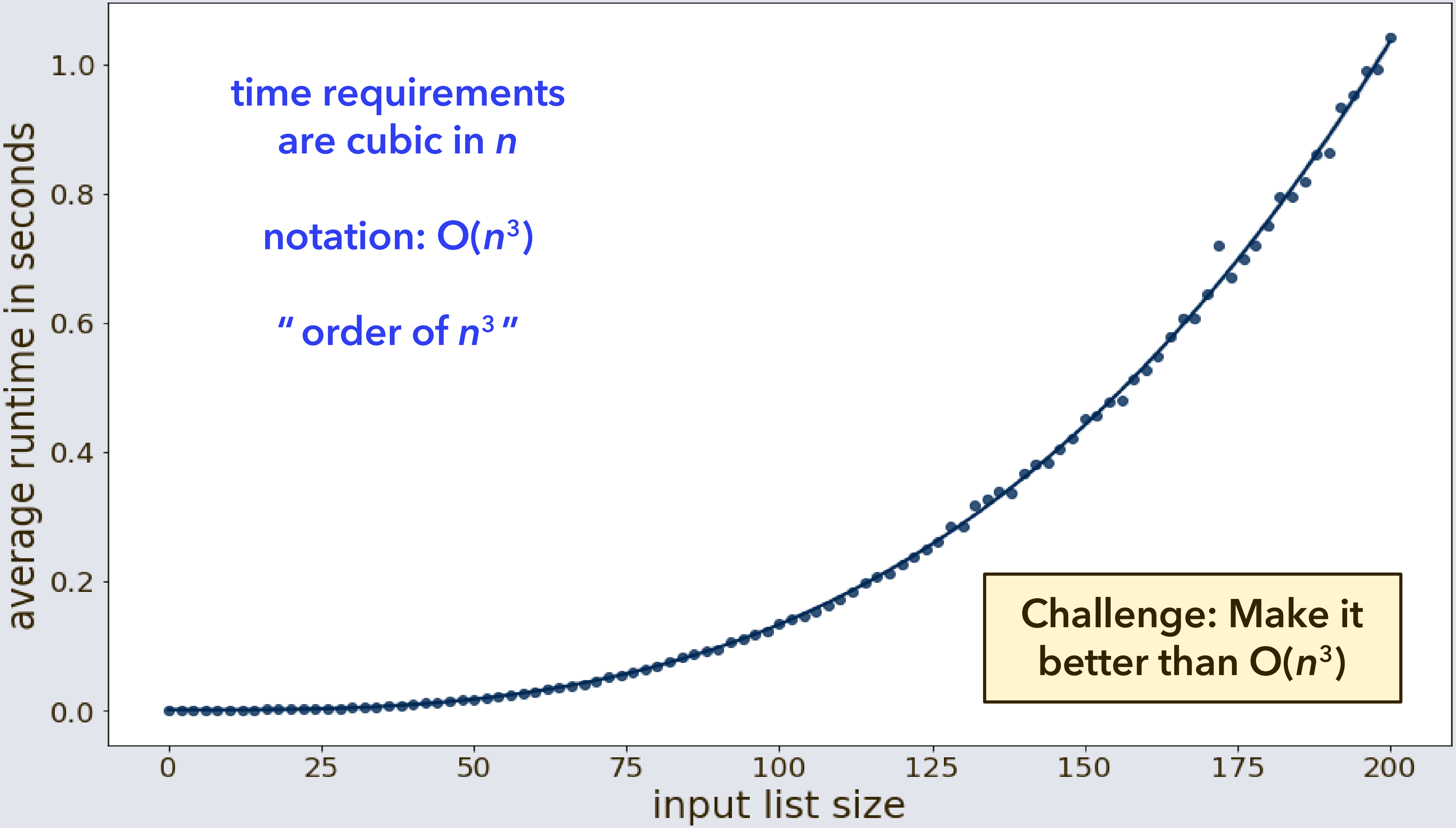
done n^3 times

done once

Eliminate all except the leading contribution, *i.e.*, the one that dominates the measure for large values of *n*; the one that grows fastest.

$$O(n^3) + O(n^2) + O(n) + O(1) = O(n^3)$$

Average performance of q_1, q_2, q_3 computations



time requirements are cubic in n

notation: $O(n^3)$

" order of n^3 "

Challenge: Make it better than $O(n^3)$

Terminology and building a glossary

CO2412

26th October 2021

University of Central Lancashire
UCLan

CO2412
Computational Thinking

Formal verification #2
Algorithmic efficiency #2
Terminology and building a glossary

Where opportunity creates success