University of Central Lancashire
UCLan

CO2412 Computational Thinking

Review of module parts 1 to 3
End-of-year reflection
Tutorial 2.3 discussion

Where opportunity creates success

Review of module parts 1 to 3

University of Central Lancashire UCLan

CO2412

14th December 2021

Module structure

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**

Part 1: Program analysis

On the topic of **program analysis**, we have:

- Considered the space (memory) and time efficiency of algorithms;
- Described asymptotic scaling behaviour using Landau $O(n)$ notation;
- Analysed algorithms formally via pre-/postconditions of statements.

Part 1: Program analysis

On the topic of **program analysis**, we have:

- Considered the space (memory) and time efficiency of algorithms;
- Described asymptotic scaling behaviour using Landau $O(n)$ notation;
- Analysed algorithms formally via pre-/postconditions of statements.

common efficiency classes:

iteration vs. recursion

constant, O(1)

program flow graphs

linear, O(n)

time & space efficiency

O(n log n)

Landau ("big O") notation

quadratic, O(n^2)

Part 2: Algorithm design

On the topic of **algorithm design**, we have:

- Compared and applied algorithm design strategies such as recursion, divide-and-conquer, greedy algorithms, dynamic programming;
- Looked at common data structures and their specification and implementation;
- Applied algorithm design to sorting as a highly relevant use case.

Part 2: Algorithm design

On the topic of **algorithm design**, we have:

- Compared and applied algorithm design strategies such as recursion, divide-and-conquer, greedy algorithms, dynamic programming;
- Looked at common data structures and their specification and implementation;
- Applied algorithm design to sorting as a highly relevant use case.

algorithm design strategies:

brute-force algorithms

greedy algorithms

divide and conquer

dynamic programming

sequential data structures:

(static) array

dynamic array

singly linked list

doubly linked list

Sorting algorithms: Selection sort

Selection sort: Greedy algorithm

Sorting algorithm that keeps **selecting the smallest remaining** element:

Test list: [35, 16, 58, 3, 11, 106, 15, 55, 7, 81, 1]

Step 1: [1] → Step 2: [1, 3] → Step 3: [1, 3, 7] → Step 4: [1, 3, 7, 11]

→ Step 5: [1, 3, 7, 11, 15] → Step 6: [1, 3, 7, 11, 15, 16] → …

→ Step 11: [1, 3, 7, 11, 15, 16, 35, 55, 58, 81, 106]

Sorting algorithms: Insertion sort

Insertion sort: Greedy algorithm

Sorting algorithm that keeps **inserting the next element** into a sorted list:

Test list: [35, 16, 58, 3, 11, 106, 15, 55, 7, 81, 1]

Step 1: [35] → Step 2: [16, 35] → Step 3: [16, 35, 58] → Step 4: [3, 16, 35, 58]

→ Step 5: [3, 11, 16, 35, 58] → Step 6: [3, 11, 16, 35, 58, 106] → …

→ Step 11: [1, 3, 7, 11, 15, 16, 35, 55, 58, 81, 106]

Sorting algorithms: Mergesort

University of Central Lancashire UCLan

Mergesort: Divide-and-conquer algorithm

sublist_size = 1

20 22 4 89 110 52 60 79 58 9 87
20 22 4 89 110 52 60 79 58 9 87

20 22 4 89 110 52 60 79 58 9 87
20 22 4 89 110 52 60 79 58 9 87

20 22 4 89 110 52 60 79 58 9 87
20 22 4 89 52 110 60 79 58 9 87

20 22 4 89 52 110 60 79 58 9 87
20 22 4 89 52 110 60 79 58 9 87

20 22 4 89 52 110 60 79 58 9 87
20 22 4 89 52 110 60 79 9 58 87

20 22 4 89 52 110 60 79 9 58 87

sublist_size = 2

20 22 4 89 52 110 60 79 9 58 87
4 20 22 89 52 110 60 79 9 58 87

4 20 22 89 52 110 60 79 9 58 87
4 20 22 89 52 60 79 110 9 58 87

4 20 22 89 52 60 79 110 9 58 87
4 20 22 89 52 60 79 110 9 58 87

sublist_size = 4

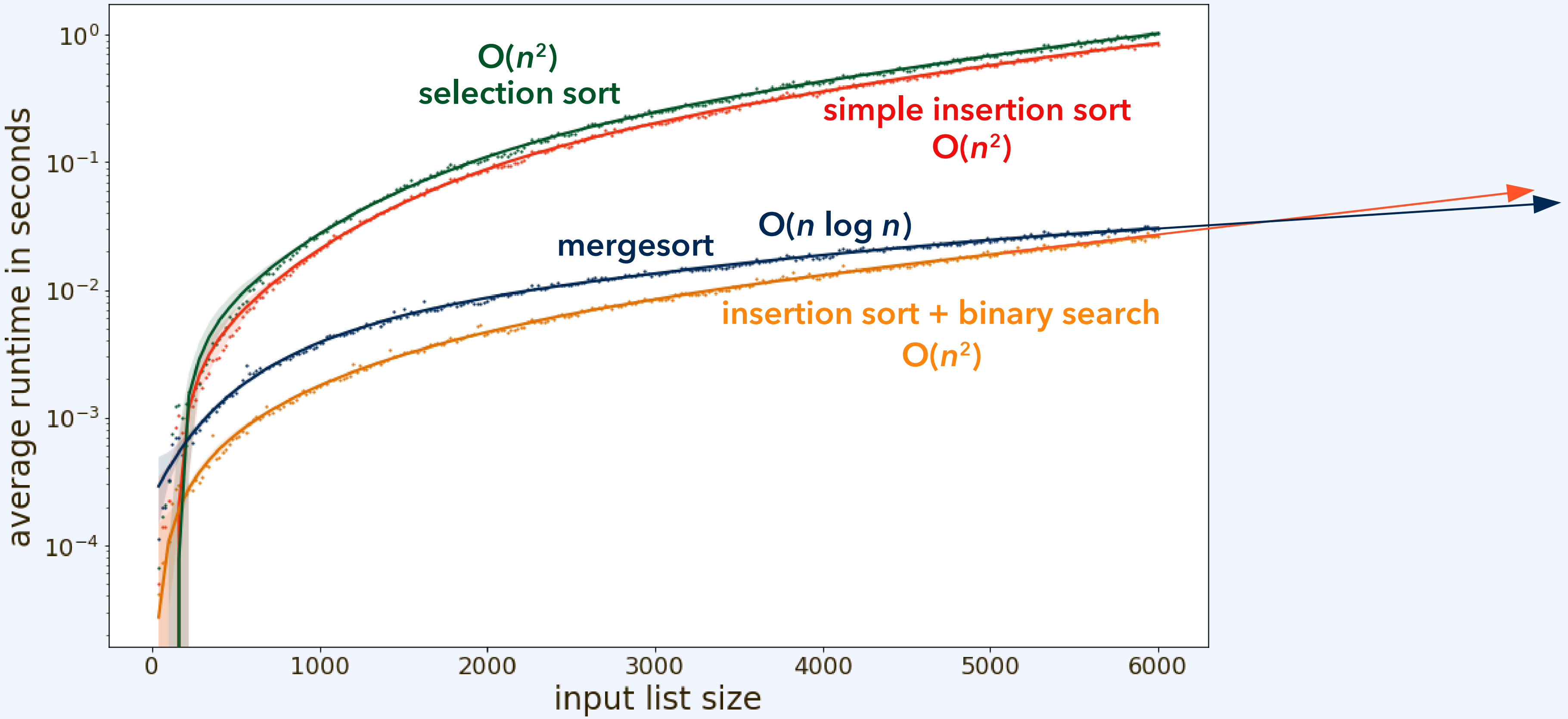
4 20 22 89 52 60 79 110 9 58 87
4 20 22 52 60 79 89 100 9 58 87

4 20 22 52 60 79 89 100 9 58 87

sublist_size = 8

4 20 22 52 60 79 89 100 9 58 87
4 9 20 22 52 58 60 79 87 89 100

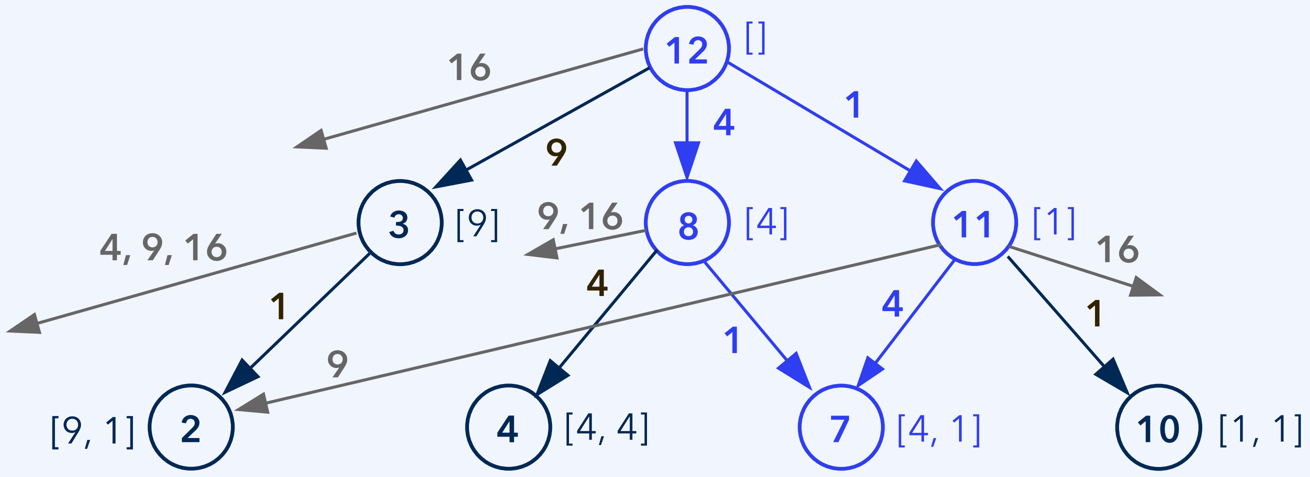
Sorting algorithms: Performance



Part 2: Algorithm design

Revising the concepts

What is the difference between dynamic programming and divide-and conquer?



Part 3: Graphs and trees

On the topic of **graphs and trees**, we have:

- Introduced graph theory and its basic definitions and concepts, including trees as a special case;
- Addressed basic tasks/problems when dealing with graphs, *e.g.*, computing shortest paths, strategies for graph traversal, and the application of trees to sorting and searching;
- Discussed numerical representations of graphs as data structures.

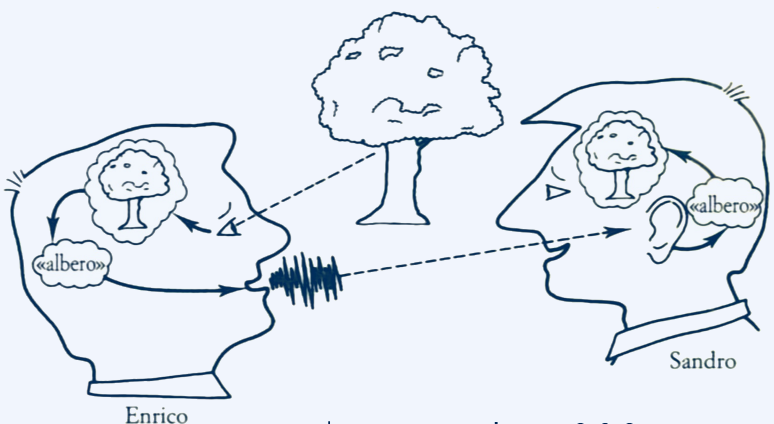


Fig. from R. Jackendoff, *Patterns in the Mind* (Italian translation).

Part 3: Graphs and trees

On the topic of **graphs and trees**, we have:

- Introduced graph theory and its basic definitions and concepts, including trees as a special case;
- Addressed basic tasks/problems when dealing with graphs, *e.g.*, computing shortest paths, strategies for graph traversal, and the application of trees to sorting and searching;
- Discussed numerical representations of graphs as data structures.

binary search **graph** **adjacency list**

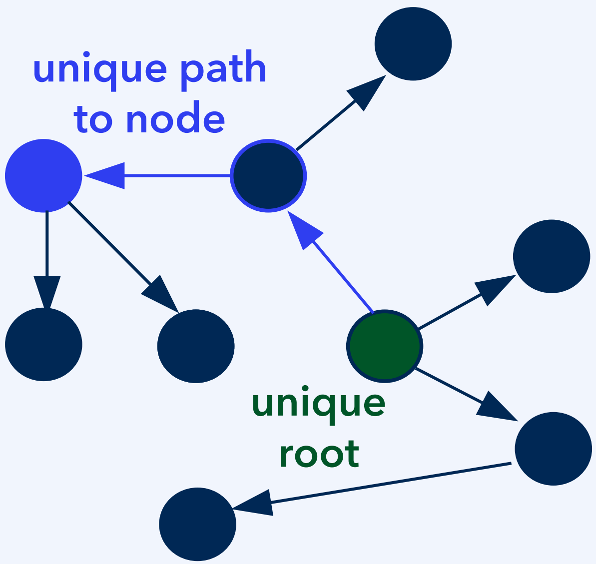
binary search tree **traversal** **incidence list**

balanced tree **spanning tree** **adjacency matrix**

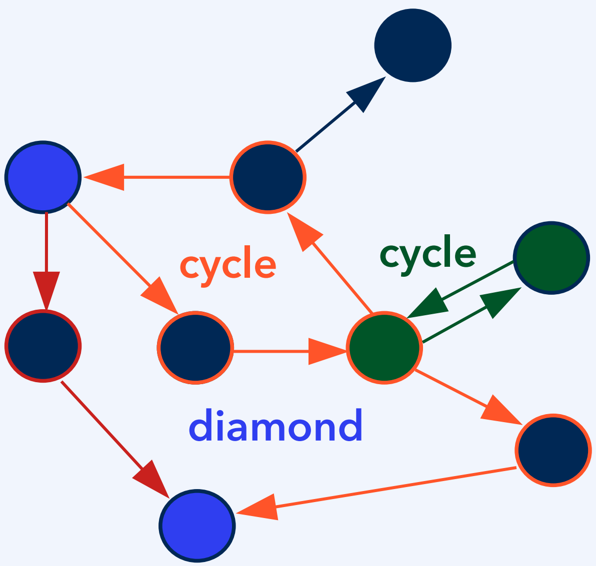
Part 3: Graphs and trees

Trees as a special kind of graph, and graphs as a generalization of trees

tree (a kind of graph)



graph



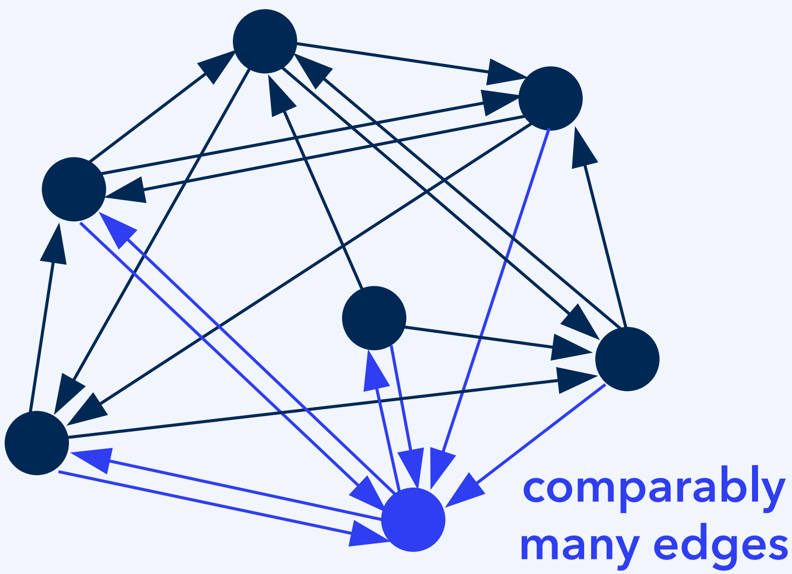
Definition ("tree"; in the literature, also: "out-tree" or "rooted tree")

A tree is a graph with a root and a unique path from the root to each node.

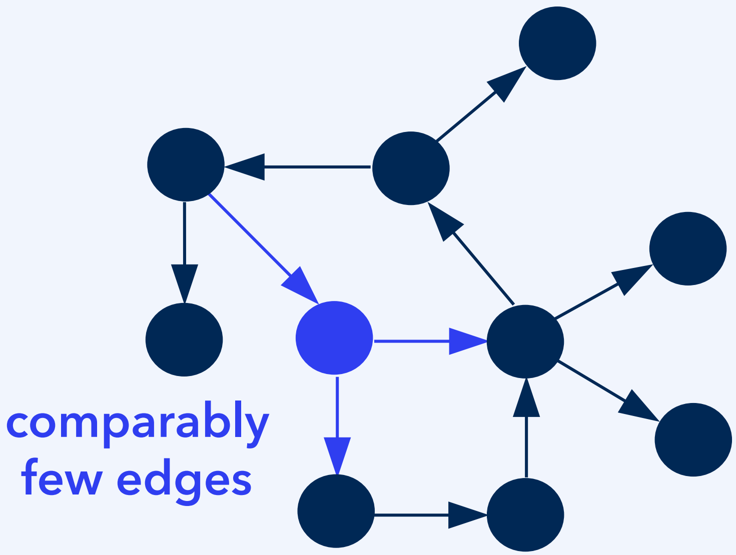
Graphs as data structures: Implementation

Neighbour lists, implemented as **adjacency or incidence lists**, are most suitable for **sparse graphs**. Matrix-like data structures are best for **dense graphs**.

dense graphs

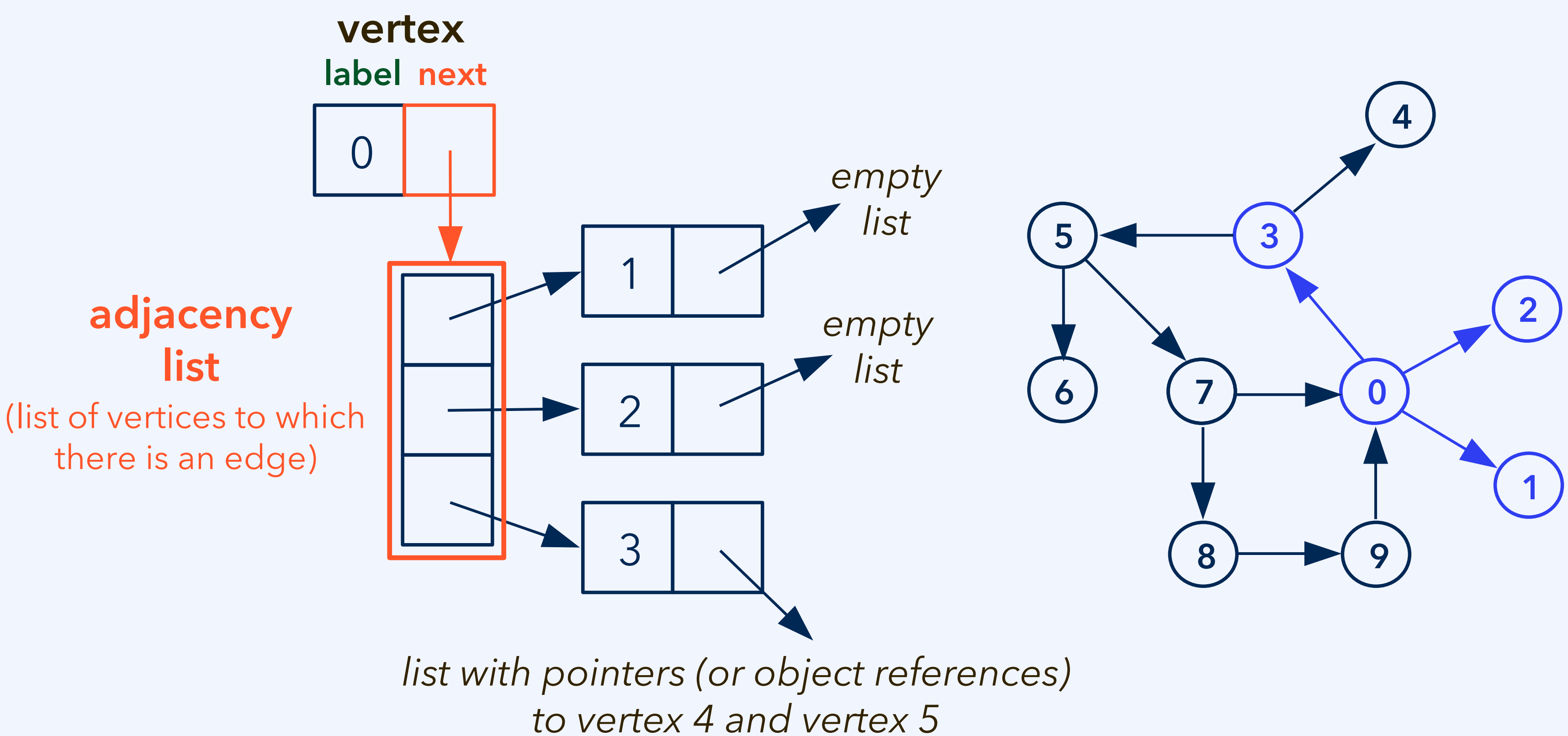


sparse graphs



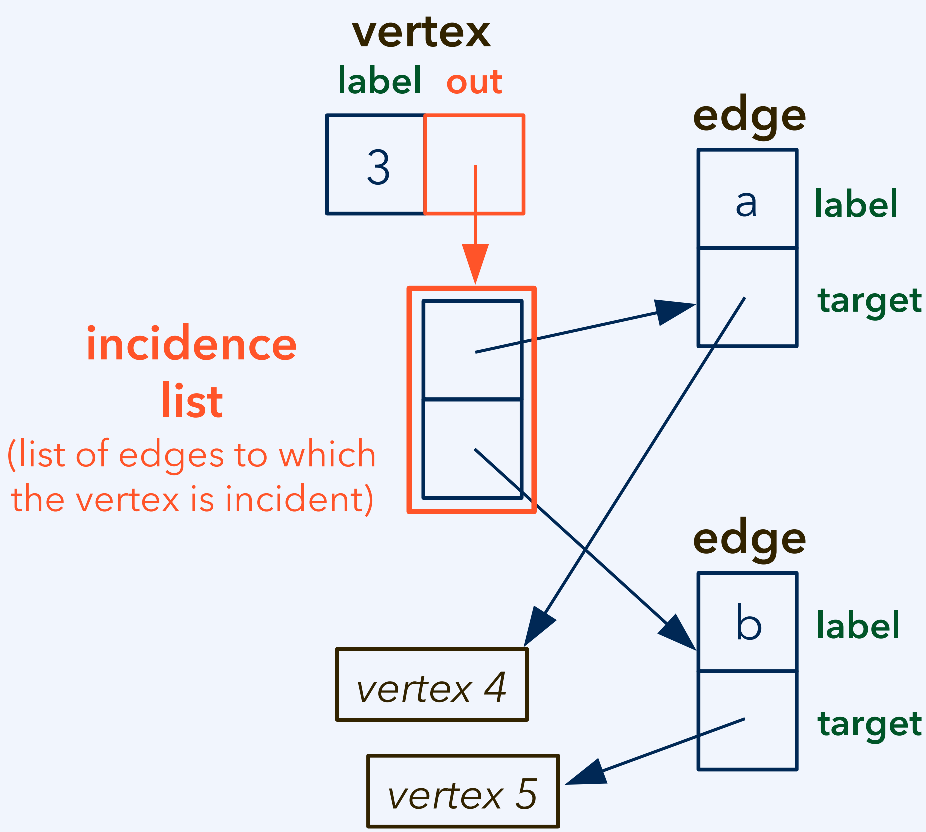
Graphs as data structures: Implementation

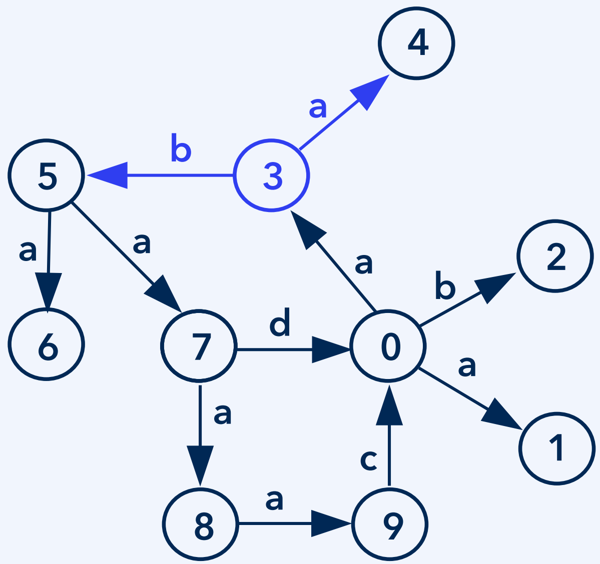
Remark: This construction is particularly suitable for tree data structures, since trees are sparse graphs (in-degree ≤ 1), and they normally contain data items.



Graphs as data structures: Implementation

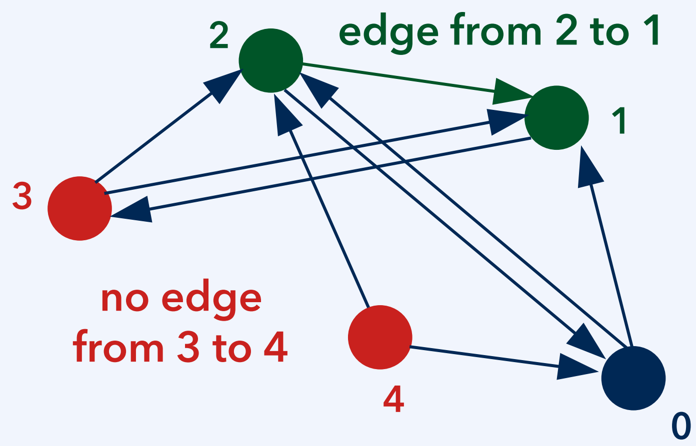
For adjacency lists or incidence lists, a variety of data structures can be used, *e.g.*, dynamic arrays. They need not be sequential data structures.





Graphs as data structures: Implementation

Matrix-like data structures in Python include lists of lists (*i.e.*, 2D dynamic arrays), if the numpy library is used, two-dimensional static arrays. For graphs, the most relevant data structure of this type is the **adjacency matrix**.



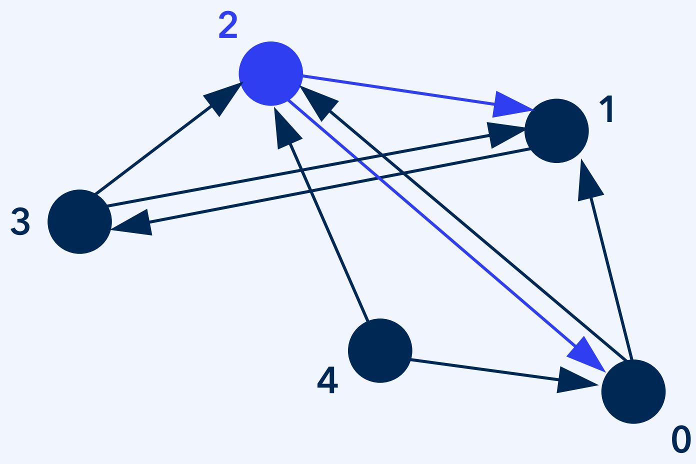

```
adj = [   [0,   1,   1,   0,   0],
          [0,   0,   0,   1,   0],
          [1,   1,   0,   0,   0],
          [0,   1,   1,   0,   0],
          [1,   0,   1,   0,   0]   ]
```

adj[2][1] = **1**, or **True**

adj[3][4] = **0**, or **False**

Graphs as data structures: Implementation

Matrix-like data structures in Python include lists of lists (*i.e.*, 2D dynamic arrays), if the numpy library is used, two-dimensional static arrays. For graphs, the most relevant data structure of this type is the **adjacency matrix**.




```
adj = [  [0,  1,  1,  0,  0],   out of node 0
         [0,  0,  0,  1,  0],   out of node 1
         [1,  1,  0,  0,  0],   out of node 2
         [0,  1,  1,  0,  0],   out of node 3
         [1,  0,  1,  0,  0]  ]
```

For a sparse graph, the vast majority of entries in the 2D array/matrix is zero. Adjacency matrices are commonly only used when expecting a **dense graph**.

Part 3: Graphs and trees

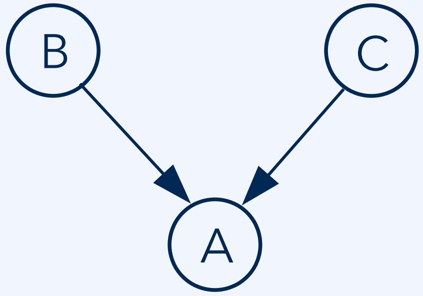
Revising the concepts

We define a tree to be a graph with:

- one unique root node;
- one unique path from the root node to each node.

Assume that in a graph there is a node with two incoming edges.

Why is it impossible that such a graph is a tree?



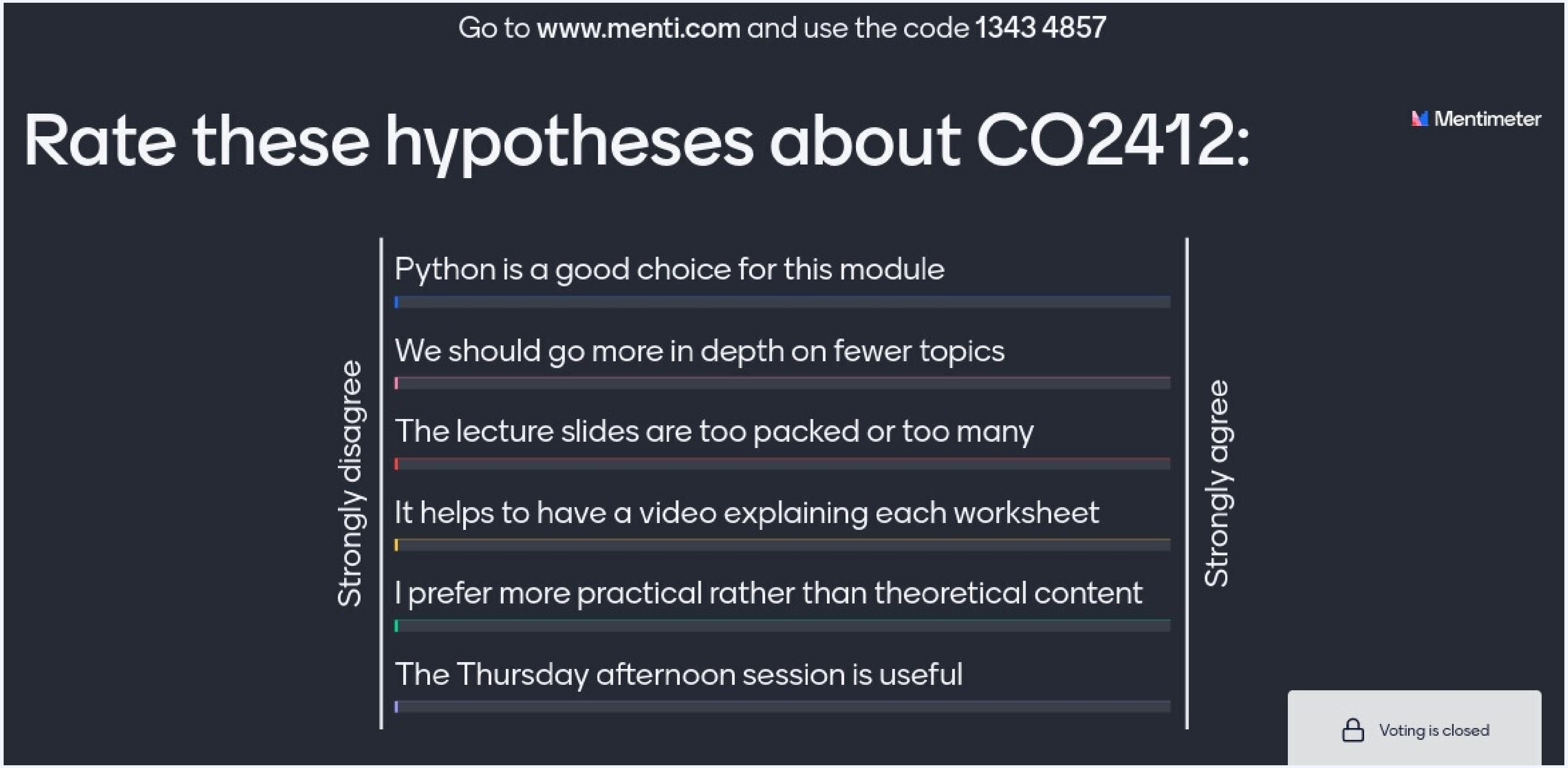
End-of-year reflection

CO2412

14th December 2021

University of Central Lancashire UCLan

Help improve the module for the coming year



https://www.menti.com/ with code 1343 4857

Tutorial 2.3 discussion

CO2412

14th December 2021

Problem 2.3.1: Performance of doubly linked lists

A list with *n* elements is given.

Iterate over the whole list, and for each element:

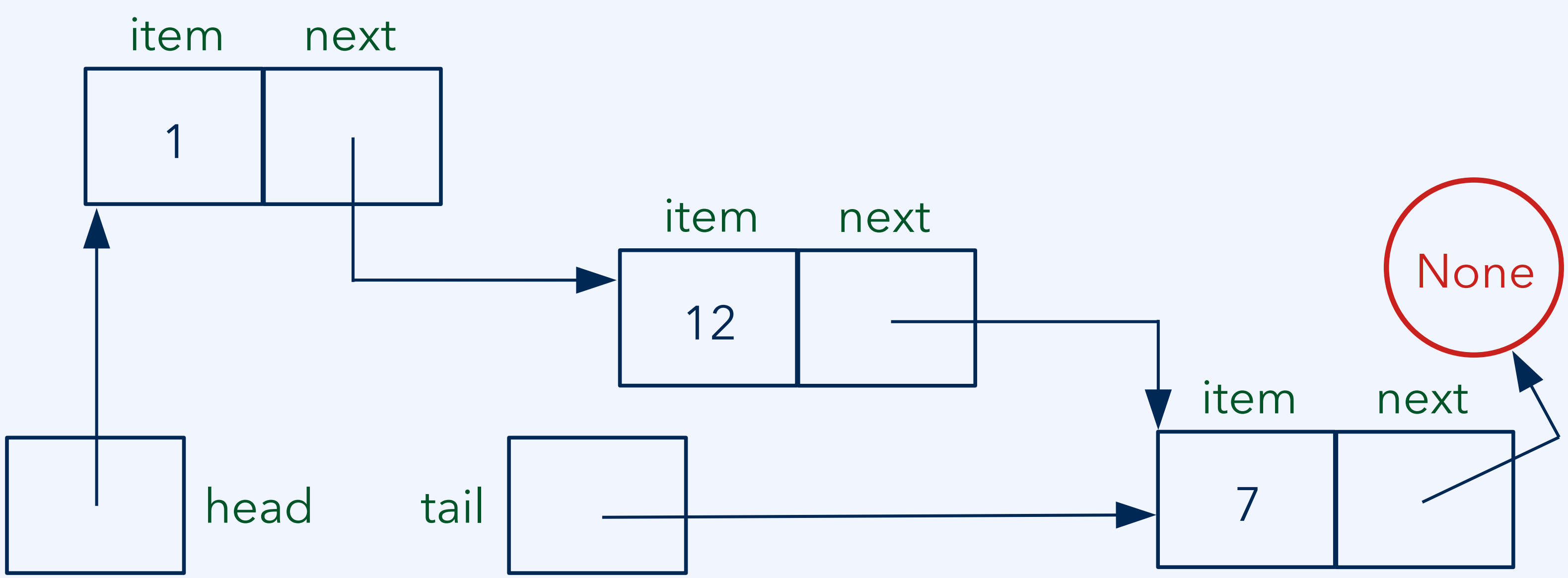
- If it is a multiple of 3, delete it from the list;
- If it has a remainder of 1 upon division by three, do nothing;
- If it has a remainder of 2, insert a copy of the element right next to it.

In this way, *e.g.*, [19, 12, 20, 12, 4] is modified to become [19, 20, 20, 4].

Problem 2.3.1: Performance of doubly linked lists

In a **singly linked list**, each node contains a data item and a reference (or pointer) to the **next node**. This facilitates traversal in **one direction**, namely forward, and **inserting** a new data item **after** any given node, in constant time.

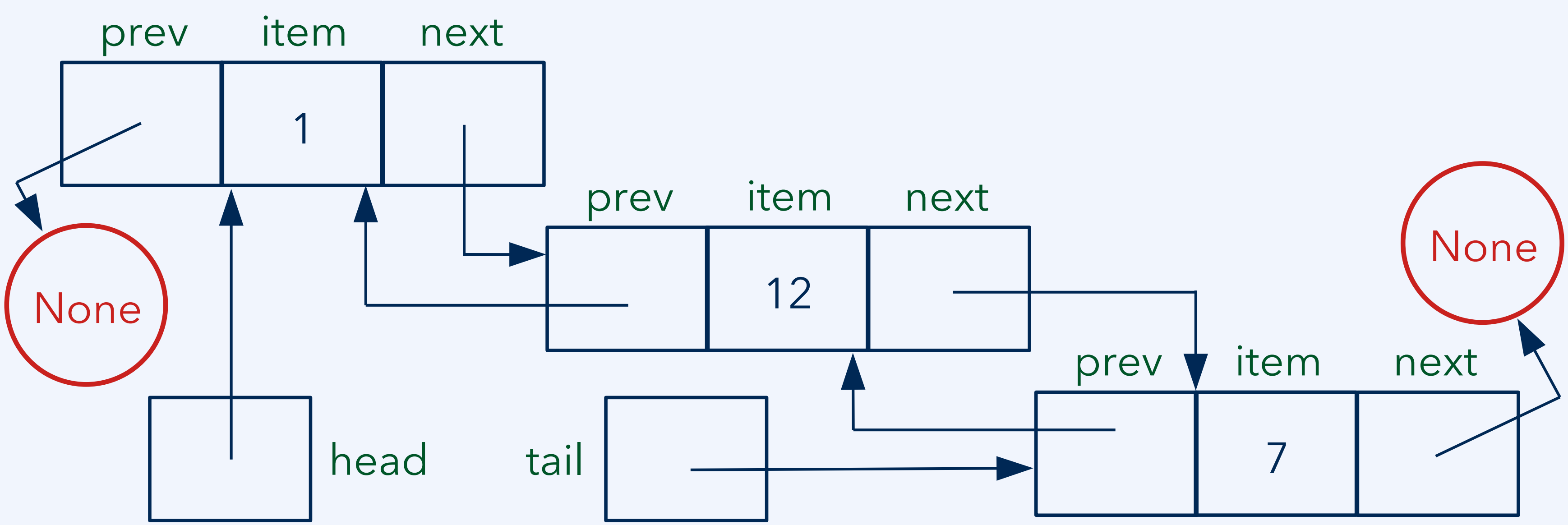
Singly linked lists require two variables per data item (item and next).



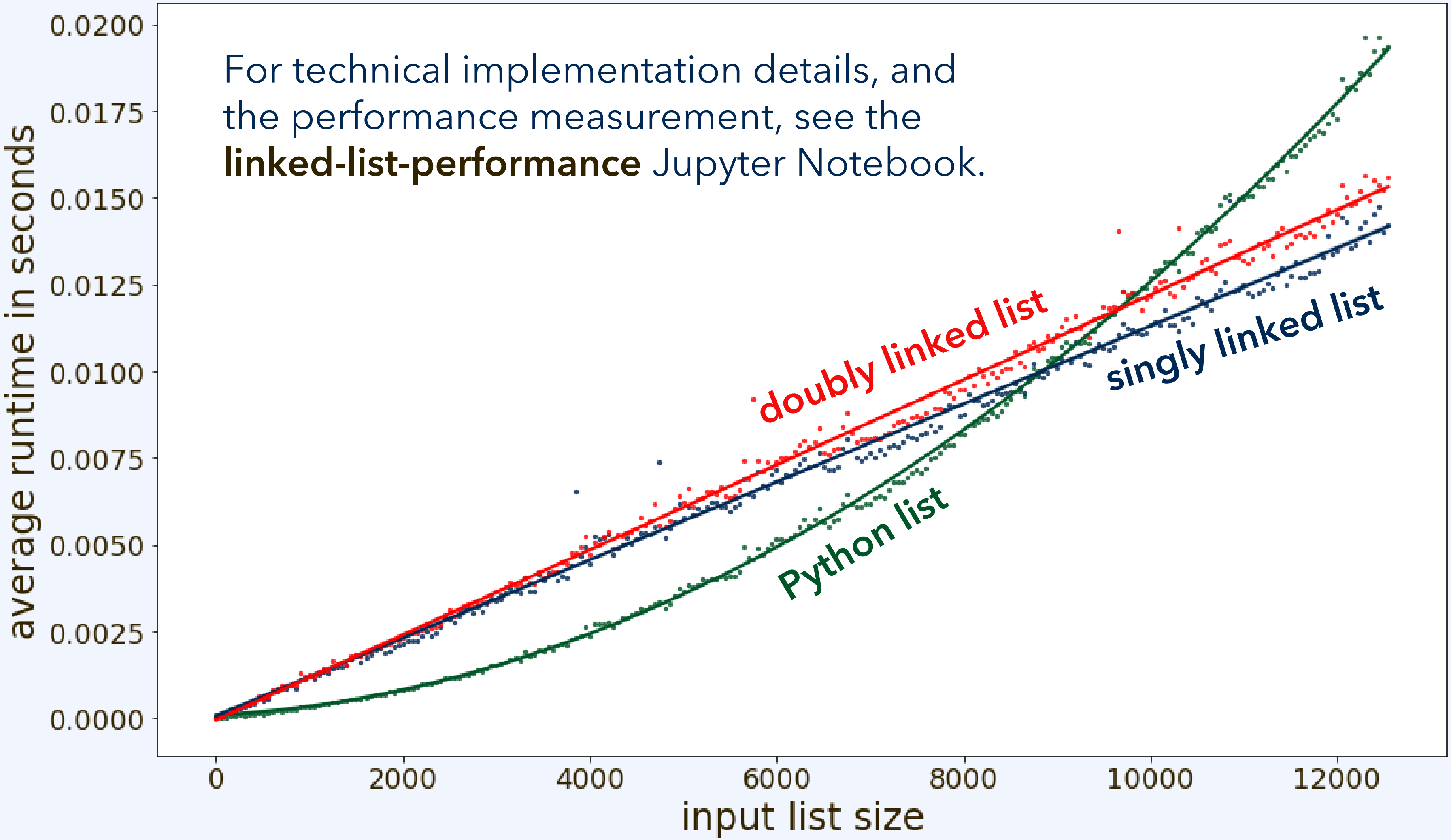
Problem 2.3.1: Performance of doubly linked lists

In a **doubly linked list**, each node additionally contains a reference to the **previous node**. This facilitates traversal in **both directions and inserting** a new data item **before** any given node (rather than only after it), all in constant time.

Singly linked lists require two variables per data item (item and next).
Doubly linked lists require three variables per data item (prev, item, and next).



Problem 2.3.1: Performance of doubly linked lists



Problem 2.3.2: Dantzig's algorithm

Greedy algorithm for the knapsack problem:

- There is a limited capacity *c*.
- Loadable items each have a weight *w*[*i*] and a value *v*[*i*].
- Dantzig's algorithm selects them in descending order of *v*[*i*] / *w*[*i*].
- The algorithm terminates when no more items fit into the capacity.

The question was: Does this algorithm always determine the best solution?

capacity 8

value 20
weight 5

value 12
weight 4

value 12
weight 4

value 3
weight 3

Problem 2.3.2: Dantzig's algorithm

Greedy algorithm for the knapsack problem:

- There is a limited capacity *c*.
- Loadable items each have a weight *w*[*i*] and a value *v*[*i*].
- Dantzig's algorithm selects them in descending order of *v*[*i*] / *w*[*i*].
- The algorithm terminates when no more items fit into the capacity.

The question was: Does this algorithm always determine the best solution?

value 20 *value 3*

total cargo value: 23

capacity 8

20/5 = 4 | **12/4 = 3** | **12/4 = 3** | **3/3 = 1**

value 20 | *value 12* | *value 12* | *value 3*

weight 5 | **weight 4** | **weight 4** | **weight 3**

Problem 2.3.2: Dantzig's algorithm

Greedy algorithm for the knapsack problem:

- There is a limited capacity *c*.
- Loadable items each have a weight *w*[*i*] and a value *v*[*i*].
- Dantzig's algorithm selects them in descending order of *v*[*i*] / *w*[*i*].
- The algorithm terminates when no more items fit into the capacity.

The question was: Does this algorithm always determine the best solution?

value 20	*value 3*	**total cargo value: 23**

capacity 8

Optimal solution, not found by Dantzig's algorithm:

value 12	*value 12*	**total cargo value: 24**

capacity 8

University of Central Lancashire
UCLan

CO2412
Computational Thinking

Review of module parts 1 to 3
End-of-year reflection
Tutorial 2.3 discussion

Where opportunity creates success