FIT1045/FIT1053 Algorithmic Problem Solving – Tutorial 5.

Objectives

After this tutorial, you should be able to:

- Give the definition of important graph terms.
- Reason about copying nested objects in Python.
- Identify simple loop invariants.
- Use decomposition to simplify problems.

Prepared Question

When wiring a building, electricians often have to solve the problem: how can I connect all electrical components in this building while minimising the amount of wire used?

An electrician is trying to solve this problem for the building shown in Figure 1. Circles represent an electrical component, and lines indicate that it is possible to install a connecting wire between two components (for example, it is possible to install a wire between the switch and the light; but not the light and the toaster). You can assume that every connection takes exactly 1 metre of wire. Every component must be connected, either directly or indirectly, to the main.

1. What abstract data type allows us to translate this into a computational problem?
2. Give a potential wiring of this building that minimises the amount of wire the electrician must use. You may describe this wiring in any way that you like.

You must attempt both parts to receive the mark.

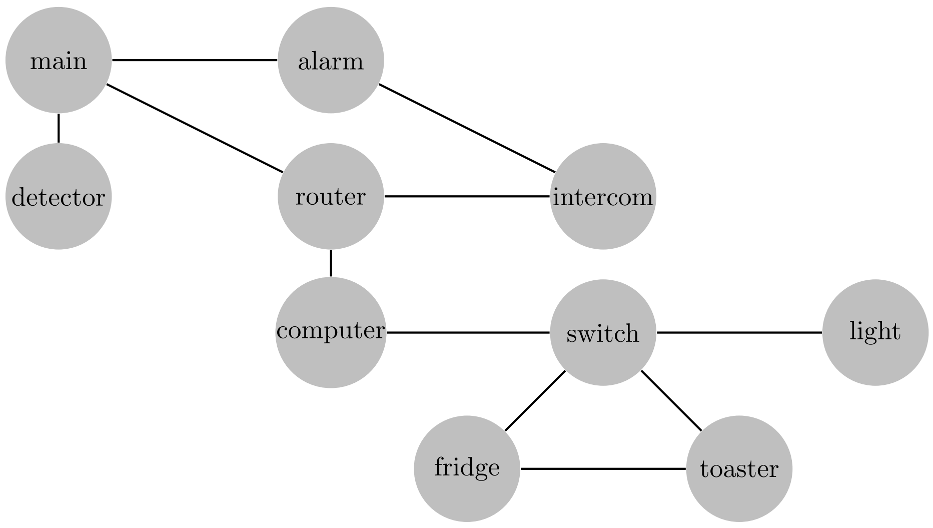


Figure 1: Electrical Components in a Building

Workbook

§5 Code traces and §7 Graphs.

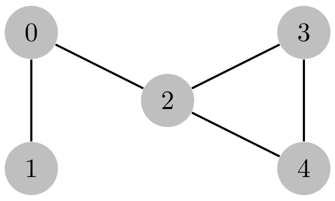
While your tutors mark the prepared question, attempt the above Workbook sections. Your tutors will go through some of these questions with the class.

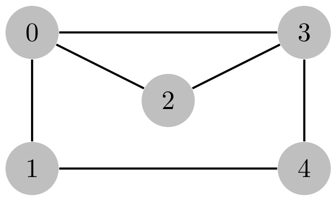
Warm-up

Do not solve these questions by running in Python.

Graphs: For each of the following two graphs:

- What is the adjacency matrix of the graph?
- How many cycles does the graph have?
- How many paths are there between vertices `1` and `4`
- How many spanning trees does the graph have?





Loop invariants: For both of the following Python programs, what are the loop exit conditions, what are loop invariants that (together with the loop exit condition) show that the function produces the correct output:

-
  ```
  def find_max(lst):
      """
      Input : non-empty list lst of
              comparable elements
      Output: maximum element of lst
      """
      res = lst[0]
      i = 1
      while i < len(lst):
          if lst[i] > res:
              res = lst[i]
          i = i+1

      return res
  ```

-
  ```
  def sum_of_ints(n):
      """
      Input : positive integer n
      Output: sum 1+...+n
      """
      res = 0
      i = 1
      while i <= n:
          res = res + i
          i = i+1

      return res
  ```

Copying lists: What does `x` yield at the end of the code block?

-
  ```
  from copy import copy
  x = ['Fellowship', 'Towers', 'King']
  lst = copy(x)
  lst.append('Hobbit')
  ```

-
  ```
  from copy import copy
  x = [['Menace', 'Clones', 'Sith'],
      ['Hope', 'Empire', 'Return']]
  lst = copy(x)
  lst[-1][-1] = 'Jedi'
  ```

Magic Squares

A magic square is a table with n rows and n columns such that each square is filled with distinct positive integers i where $1 \leq i \leq n^2$, and the sum of the integers in each row, in each column, and the two main diagonals ie equal (let's denote the sum of a column (or row or diagonal) of an $n \times n$ square as *magic(n)*).

For example, table 1 is a 5×5 magic square where each row, column and main diagonal add to 65

Table 1: 5×5 magic square, $magic(5) = 65$

11	24	7	20	3	**65**
4	12	25	8	16	**65**
17	5	13	21	9	**65**
10	18	1	14	22	**65**
23	6	19	2	15	**65**
65	**65**	**65**	**65**	**65**	**65**

Describe an algorithm which takes as input a magic square in the form of a list of lists (where each row is a single list, and the table is a list of rows) and returns `True` if the table is a magic square and `False` otherwise. Remember to make use of decomposition.

Deep Copy

Assignment statements in Python do not create new objects. Instead, assignment statements bind names to objects. This is not generally a problem when the object is immutable, but for *mutable* objects like lists this can create problems, particularly when we want to create copies. To deal with this, Python has a function called `deepcopy` that creates a fully independent copy of the original list.[1]

Copying lists with depth exactly 2

In your group design an algorithm that accepts a nested list with depth of exactly 2, and returns a deep copy of the list. Focus on articulating the necessary steps as dot-points. Once you have designed an algorithm, try implementing it in Python.

An example of a nested list with depth of exactly 2: `[['a', 'b', 'c'], ['d'], ['e', 'f']]`

Copying lists with depth at-most 2

In your group design an algorithm that accepts a nested list with depth of at-most 2, and returns a deep copy of the list. Focus on articulating the necessary steps as dot-points. Once you have designed an algorithm, try implementing it in Python. You may find the Python function `isinstance(object, type)` useful.

An example of a nested list with depth of exactly 2: `['a', [ 'b', 'c'], 'd', ['e'], 'f']`

Decomposition

Now consider the following task; we have a list of strings (representing numbers) and we would like to determine how similar each number is to the average. If you are familiar with statistics you may be familiar with the concept of *variance* and this is what we want to compute. This involves computing the average of a list and then for each element in that list get the difference between it and the average, square it and add this to a sum.

As a group, first list the functions which will be necessary for this (try to ensure one function represents one task (a good hint is that if you could plausibly use some part of a function somewhere else that part should be a function too)). For each function list:

- name
- purpose
- input
- output

[1] Read about `deepcopy` here: `https://docs.python.org/3/library/copy.html`.

Each group member will be responsible for creating functions for (at least) one of those you came up with. If there are not enough functions, you may like to go back and reconsider if tasks can be broken down further.

Once your team has finished creating these functions, run through your work for a simple example and see if it works. If something goes wrong, troubleshoot as a group and help the developer figure out how to fix the issue.

Invariants (compulsory for FIT1053)

Consider an urn that is filled with black and white balls. At each step take two balls out until there is only one ball. If both balls have the same colour, throw them away and put another black ball into the urn. If they have different colours, throw the black ball away and put the white ball back into the urn. What can be said about the colour of the final ball in relation to the original number of black and white balls?