Exercises on algorithms

Author(s)
Silvio Peroni – silvio.peroni@unibo.it
Department of Classical Philology and Italian Studies, University of Bologna, Bologna, Italy

Abstract

These lecture notes introduce additional exercises about algorithms that can be solved by using Python.

Introduction

In this lecture, we introduce several exercises. Each exercise asks one to develop a particular algorithm, that can be written in Python. As a suggestion, first, try to solve the algorithm without any additional help. In case of issues or difficulties, try to google for possible approaches to solve it, and recreate a particular solution for answering the exercise.

Of course, feel free to test any algorithm in Python to be sure it is working as expected – i.e. use the *test-driven development* approach for checking this aspect, as demanded during the course. Finally, share your solution on the GitHub repository of the course, creating a new issue, or answer directly to an existing issue if some of your colleagues have already created it for the exercise in consideration.

Checking numbers

Write the function `def is_odd(int_number)` in Python that takes an integer number as input and returns *True* if the number is odd, *False* otherwise.

Floor division

Write the function `def floor_division(dividend, divisor)` that takes the dividend and the divisor as input and returns the division between the two and returns only the integer part of the division, without considering the fractional part. For instance, $\frac{5}{2}$ will return *2*, as well as $\frac{6}{3}$.

Note that it is not possible to use the Python built-in operator `//` in the algorithm implementation.

Finding the maximum

Write the function `def find_max(number_collection)` that takes a collection (a list, a set, etc.) of numbers as input and returns the greatest number it contains.

Finding the minimum

Write the function `def find_min(number_collection)` that takes a collection (a list, a set, etc.) of numbers as input and returns the lowest number it contains.

Prime factorization

Write the function `def prime_factorization(int_number)` that takes an integer number as input and returns its prime factorization, i.e. a dictionary specifying as keys the prime factor numbers of the input integer and as keys how many times that prime number is actually used in the factorization. For instance, the prime factorization of the number *60* is *5 • 3 • 2 • 2*, thus resulting in the following dictionary: `{5: 1, 3: 1, 2: 2}`.

Palindromos

Write the function `def is_palindromic(word)` that takes a word as input and returns *True* if it is a palindrome, *False* otherwise. For instance, "anna" and "madam" are both palindromic words.

Common substring

Write the function `def longest_common_substring(s1, s2)` that takes two strings as input and returns a new string which is the longest common substring contained in both the

input strings. For instance, specifying the strings `"this is new, guys!"` and `"it is new, fellows!"` as input, the algorithm should return the string `" is new, "`. If no common substring exists, the empty string `""` is returned.

String distance

Write the function `def levenshtein_distance(s1, s2)` that takes two strings as input and returns the minimum number of edit operations on characters that are needed to transform the first string into the second one. For instance, considering the strings `"house"` and `"home"`, the function should return 2: substitute the character *u* with the character *m* (first operation: `"house" -> "homse"`), and remove the character *s* (second operation: `"homse" -> "home"`).

Another distance metric for strings

The *Hamming distance* between two strings of equal length is the number of positions at which the corresponding characters are different. Thus, it measures the minimum number of substitutions required to change one string into the other.

Write the function `def hamming_distance(s1, s2)` which takes two strings as input and that calculates the Hamming distance if the strings have the same length, otherwise it returns the smallest string.

List items as keys in a dictionary

Write the function `def algorithm(dictionary, key_list)` that takes a dictionary and a list of strings as input and checks if each string in the list is a key of a pair in the dictionary. All the values of the pairs in the dictionary that have been matched by any key contained in the input list are added to a set, that is returned at the end of the algorithm.

Binary search

Write the function `def binary_search(item, ordered_list, start, end)`, that takes an item to search (i.e. item), an ordered list and a starting and ending positions in the list as input, and returns the position of item in the list if it is included in it, and `None` otherwise. The approach implemented by the binary search is described as follows. First, it checks if the middle element of the list between `start` and `end` (included) is equal to `item`, and returns its position in this case. Otherwise:

1. if the middle element is lesser than `item`, the search is executed in the part of the list that follows the middle element; otherwise,

2. if the middle element is greater than `item` the search is executed in the part of the list that precedes the middle element.

Fibonacci search

Write the function `def fibonacci_search(item, ordered_list, start, end)` that takes an item, an ordered list, a starting position and an ending position as input and returns the position the item in the list if it is included between the start and end position specified. It is a particular adaptation of the binary search, where the list is actually divided into two parts that have sizes that are consecutive Fibonacci numbers – thus, the difficult part is to calculate the middle position, while the search process will follow the same rules of the binary search. Note that the length n of the list on which to find the item should be a Fibonacci number, i.e. $n = fib(m)$. If *n* is not a Fibonacci number, then let *fib(m)* be the smallest number in the Fibonacci sequence that is greater than *n*. That *m* is used for determining how to recursively call the algorithm of the two parts of the list.

For instance, if we have to find a book in a list of 19 book titles, *m* will be set to *8* (since *fib(8) = 21 > 19*), and an ancillary index `mid` (identifying the position of the middle element) is obtained by using the Fibonacci number at *m - 1*.
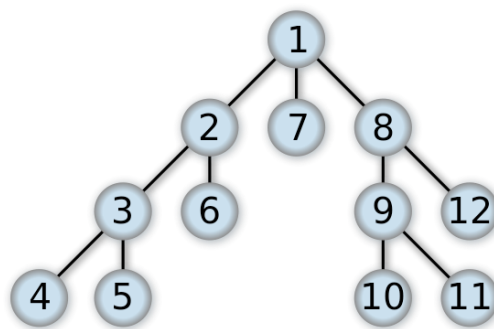
Breadth-first search

Write the function `def breadth_first_search(item, input_tree)` that takes an item to search (e.g. a name) and a node in a tree where each node contains a value, returns the node where the item is stored. The search on the tree starts from the root. At a given node, the algorithm checks if any of its children refers to the item we are looking for. In this case, the algorithm returns the child containing the item, otherwise, it repeats the same operation for all its child nodes, from the left-most to the right-most. The order in which the nodes are visited is shown in Figure 1.
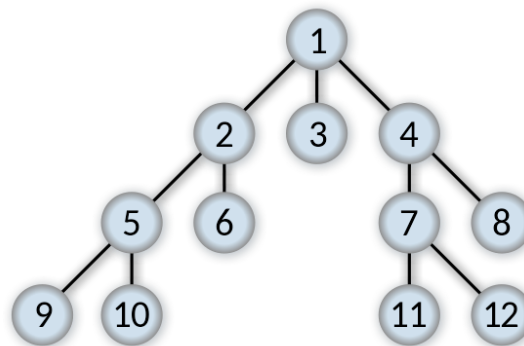
Figure 1. How the nodes are visited according to a breadth-first search. Picture by Alexander Drichel, source: https://commons.wikimedia.org/wiki/File:Breadth-first-tree.svg.

Depth-first search

Write the function `def depth_first_search(item, node)` that takes an item to search (e.g. a name) and a node in a tree where each node contains a value, and returns the node where the item is stored. The search on the tree starts from the root. At a given node, the algorithm checks if any of its children that has not been visited yet (from the left-most to the right-most) refers to the item we are looking for. In this case, the algorithm returns the child containing the item, otherwise, it repeats the same operation for the next non-visited child. The order in which the nodes are visited is shown in Figure 2.

Figure 2. How the nodes are visited according to a depth-first search. Picture by Alexander Drichel, source: https://commons.wikimedia.org/wiki/File:Depth-first-tree.svg.

Bubble sort

Write the function `def bubble_sort(input_list)` that takes an unordered list as input and returns it ordered. The idea behind the bubble sort is the following one: at every step of the algorithm, it compares a pair of adjacent items. If they are in the right order, analyses the next pair directly, otherwise it swaps them before analysing the following pair. The algorithm stops after having compared all the possible pairs in the list twice, so as to be sure that all of them have been put in the right position.

Efficiency with bubble sort

Write the function `def bubble_sort_efficient(input_list)`, which is similar to the previous one, but it actually stops if no swaps are possible at a certain iteration.

Selection sort

Write the function `def selection_sort(input_list)` that takes an unordered list as input and returns it ordered. The selection sort works in the following way: it splits the list into two sublists, i.e. that one having the items already sorted (on the left, initially empty) and the one with the items in the wrong position (on the right). Every iteration, it looks for the smallest item in the part of the unsorted sublist and swaps it with the left-most unsorted element.

Readability of English texts

The *automated readability index* (*ARI*) is a readability test for English texts, designed to gauge the understandability of a text by representing the US grade level needed to comprehend such text. The formula for calculating the ARI is the following one:

$$4.71 * \frac{chars}{words} + 0.5 * \frac{words}{sentences} - 21.43$$

where *chars* is the number of letters and numbers, *words* is the number of token, and *sentences* is the number of sentences. Non-integer scores are always rounded up to the nearest whole number, so a score of 10.1 or 10.6 would be converted to 11.

Write the function `def ari(text)` which takes a string representing a text in input and returns the ARI for that text. As a simplification, the input text can be composed only by English characters, numbers, commas, semicolons, colons, and full stops, and no abbreviation (such as "e.g.") can be used.

Importance of words in a document

In information retrieval, the *term frequency–inverse document frequency* (or *tf-idf*) is a numerical statistic that is intended to reflect how important a word is to a document in a corpus. It is based on two functions:

- the term frequency, `def tf(t, d)`, which counts the number of times a term `t` occurs in document `d`;
- the inverse document frequency, `idf(t, d_list)`, which measures whether a term `t` is common or rare across all the documents in the list `d_list`, calculated as the logarithm of the division between the total number of documents in the list and the number of documents that contains the term `t`.

Thus, the *tf-idf* of a term `t` in a document `d` included in a collection of document `d_list` is simply the multiplication between its term frequency and its inverse document frequency.

Write the function `def tfidf(t, d, d_list)` which takes a string `t` representing a term, a string `d` representing a document, and a list of strings `d_list` representing a collection of documents which includes also `d`, and that returns the *tf-idf* of the input term according to the document in that document list. As a simplification, all the input strings are composed only by lowercase English alphabetic characters with no punctuation. The logarithm function `log` is available in Python within the module `math` (`from math import log`) and takes the number on which to calculate the logarithm as input.

Selecting activities

Write the function `def activity_selector(input_dict)` that takes a dictionary of activities – i.e. `{ "activity1": { "start": "2017-12-25T09:34:55", "end": "2017-12-26T16:14:01" }, ... }` – as input and returns the list of the maximum number of activities that can be performed by someone assuming that it cannot work on more than one activity at a particular time.

# Selecting activities: reprise

Write the function `def weighted_activity_selector(input_dict)` that takes a dictionary of activities – i.e. `{ "activity1": { "start": "2017-12-25T09:34:55", "end": "2017-12-26T16:14:01", "weight": 7 }, ... }` – as input and returns the list of the number of activities that can be performed by someone assuming that it cannot work on more than one activity at a particular time. In addition, the sum of all the weights of the selected activities must be maximum (i.e. is the highest possible).

Tower of Hanoi

Write the function `def solve_hanoi(stack1, stack2, stack3)` that takes three stacks as input depicting the Tower of Hanoi puzzle, shown in Figure 3, and that returns the solution to the puzzle.

The goal of this solitaire is to recreate the same tower initially put the first rod in the third one, by following three simple rules:

● at every iteration, only one disk can be moved;
● a move allows one to take a disk from a stack and place it in another one;
● no disk may be placed on top of a smaller disk.
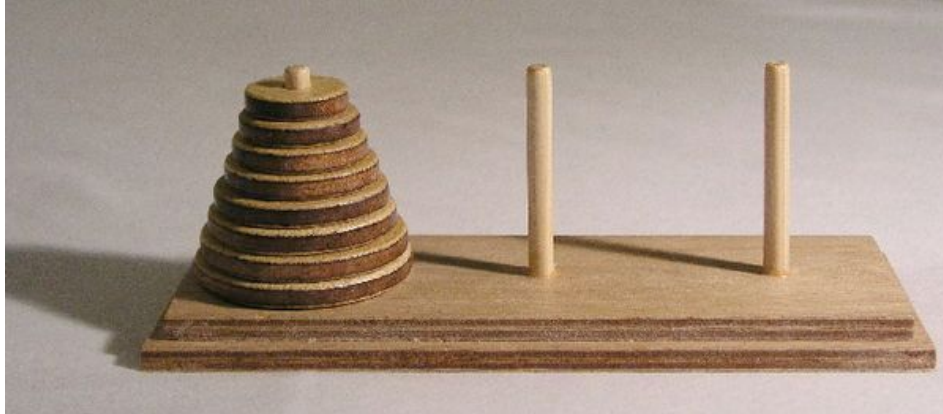
Figure 3. A picture of the initial status of the Tower of Hanoi puzzle. Picture by Ævar Arnfjörð Bjarmason, source: https://commons.wikimedia.org/wiki/File:Tower_of_Hanoi.jpeg.

Sudoku

Write the function `def solve_sudoku(dict_of_cells, last_move=None)` that takes as input a dictionary of cells – e.g. `{(0, 0): None, (1, 0): 5, ...}`, each defining a particular cell in a sudoku 9x9 board – and the last move done, and returns the solution to the puzzle. It is worth mentioning that each key in the dictionary identifies a particular cell in terms of x-y positions defined as tuples (from `(0, 0)` to `(8, 8)`), and the value associated with each key is either a number (from 1 to 9) or `None` if no number is specified. A possible initial state of the sudoku board is introduced in Figure 4.

Figure 4. A possible initial state of a sudoku puzzle. Picture by Auguel, source: https://commons.wikimedia.org/wiki/File:Sudoku_Puzzle_by_L2G-20050714_standardized_layout.svg.

The rules of the game are pretty simple: each number can only occur once in each row, column, and 3x3 box indicated in Figure 4 by means of bolder borders.

Knight's tour

Write the function `def solve_knights_tour(dict_of_cells, last_move=None)` that takes as input a dictionary of cells – e.g. `{(0, 0): False, (1, 0): False, ...}`, each defining a particular cell in a chess 5x5 board – and the last move done, and returns the solution to the puzzle. It is worth mentioning that each key in the dictionary identifies a particular cell in terms of x-y positions defined as tuples (from `(0, 0)` to `(4, 4)`), and the value associated with each key is either *True* if the cell has been previously occupied by the knight or *False* otherwise. In particular, starting from the centre cell of the 5x5 board (i.e. `(2, 2)`, initially set to *True*), the algorithm should find the moves that allow the knight to visit every cell only once. It is worth mentioning that the knight moves to a cell that is two cell away horizontally and one cell vertically, or two cells vertically and one cell horizontally.

Shortest path

Write the function `def shortest_path(input_graph, start_node, end_node)`, that takes a undirected and weighted graph (i.e. each edge has a cost for being traversed), a start node (e.g. a city), and an end node (e.g. another city) in input, and returns the optimal list of edges (e.g. the roads that one should take) that must traversed for reaching the end node from the start one with the minimal sum of the weights.