Brute-force algorithms

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

Keywords
Betty Holberton; Go; Linear search; Insertion sort

Abstract

These lecture notes introduce the notion of *brute-force algorithms* with the implementation of two algorithms of this kind: *linear search* and *insertion sort*. The historic hero introduced in these notes is Betty Holberton, who was one of the first programmers of the ENIAC, and one of the key people for the development of several programming languages and algorithms for sorting objects.

Historic hero: Betty Holberton

[Frances Elizabeth – knows as Betty – Holberton](#), shown in [Figure 1](#), was one of the first programmers of the [Electronic Numerical Integrator and Computer (ENIAC)](#), the earliest electronic and general-purpose computer developed between 1943 and 1946 with the funds of the United States Army. In addition, she was also involved in the development of several programming languages, such as [COBOL](#) and [FORTRAN](#), and the creator of the first statistical analysis tool, that was used for analysing the data of the United States Census in 1950.

She dedicated a huge part of her work in the development of algorithms for [sorting the elements in a list](#). The activity of sorting things is a typical human activity, which can be of course automatised by means of computers, and it is a desirable property to have for addressing several tasks.
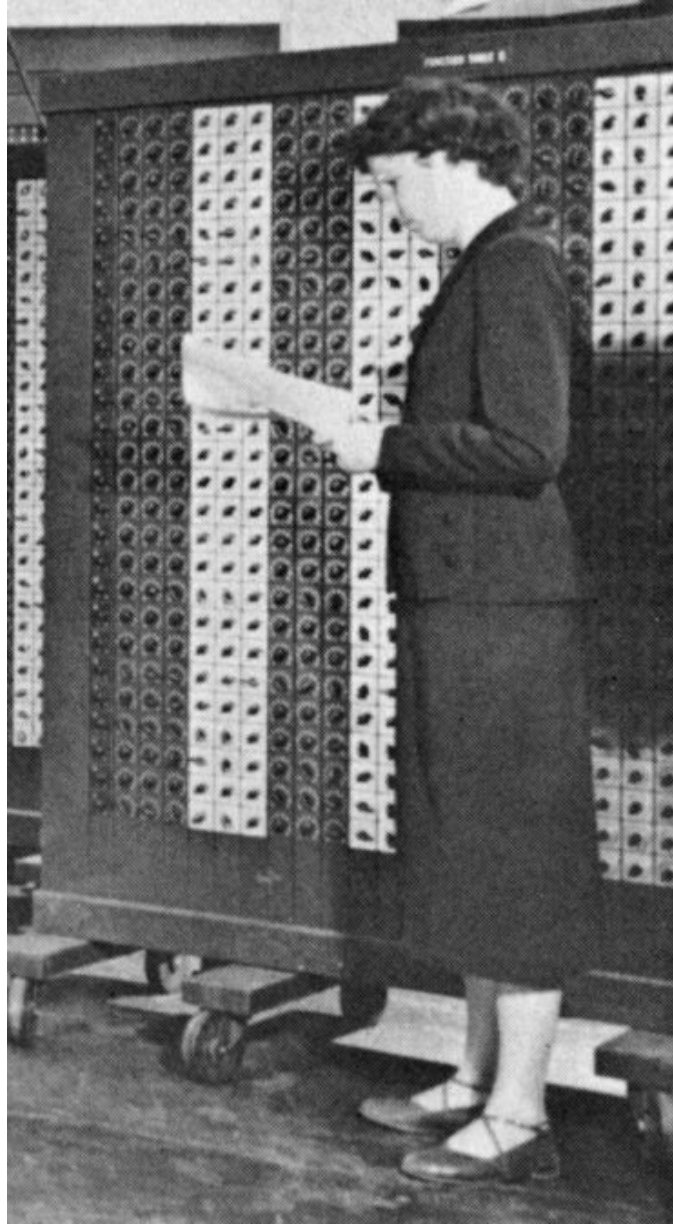
Figure 1. Picture of Betty Holberton in front of the ENIAC. Source: https://commons.wikimedia.org/wiki/File:Betty_Holberton.jpg.

Of course, sorting things is an expensive task, in particular if you have to order billions of items. However, having such items sorted in some way is crucial for several tasks that we can perform on the list that contains them. For instance, in libraries, books are ordered according to specific guidelines – e.g. Dewey classification for clustering books according to particular fields, where each cluster contains books ordered according to the authors' name and the book title. With this kind of order, the librarian can find a requested title avoiding to look all the billion books available one by one, thus saving a huge amount of time, after all. Thus, to sort things in advance is a winning practice if one has to search these things several times in the future.

May the (brute) force be with you

In this lecture, for the very first time, we start to talk about problem-solving methods. In Computer Science – and, generally speaking, in Computational Thinking – *problem-solving* is the activity of creating a computer-interpretable process (i.e. an algorithm) for solving some given problem, e.g. ordering alphabetically all the books in a library. There exist several different methods for solving problems that have been proposed in Computer Science, and all of those are grouped into general categories. Probably, the simpler category of problem-solving methods is the *brute-force* approach.
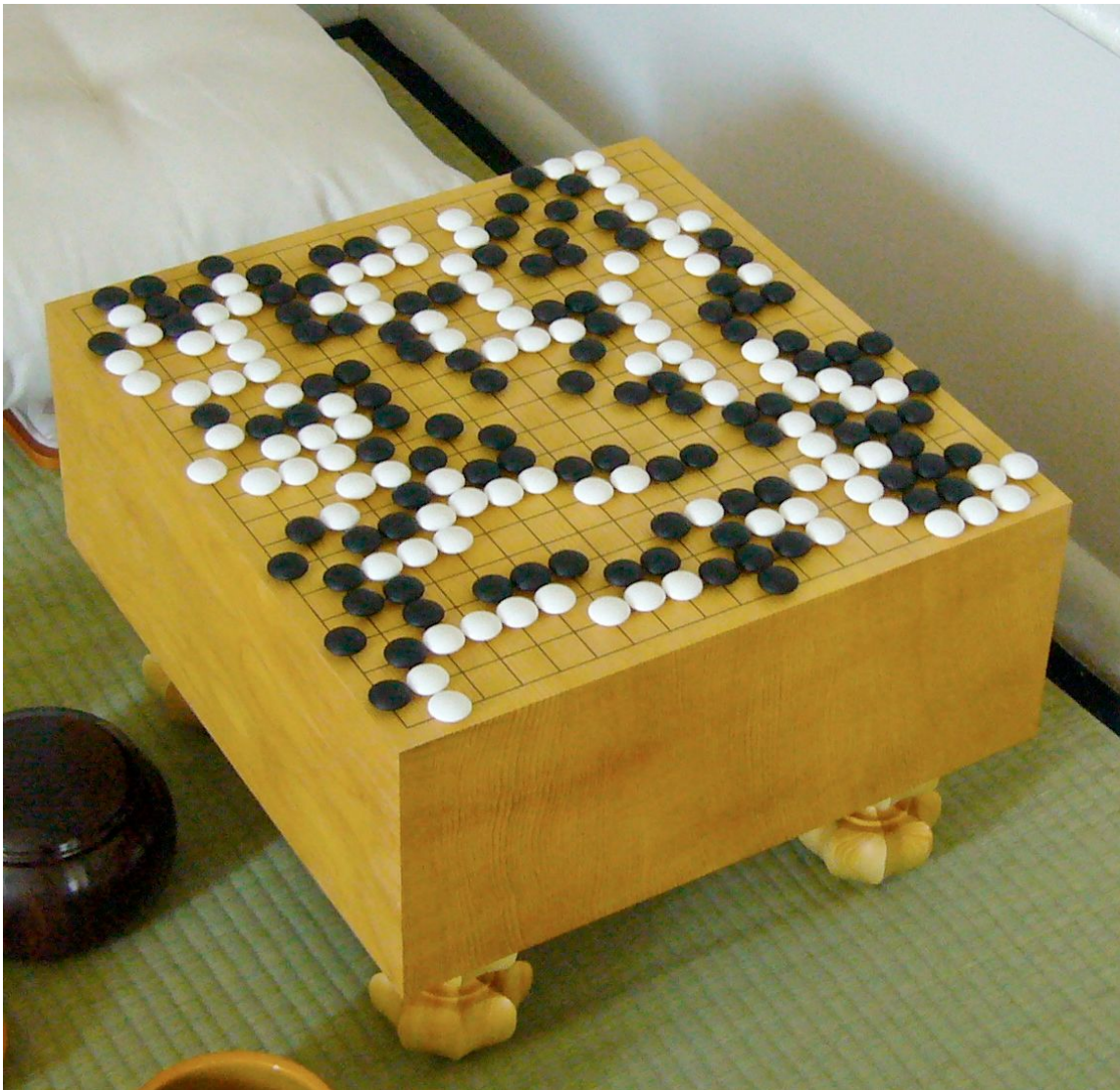

Figure 2. The game of Go, which cannot be solved efficiently by means of a pure brute-force approach. Picture by Goban1, source: https://commons.wikimedia.org/wiki/File:FloorGoban.JPG.

Brute-force algorithms are these processes that reach the perfect solution of a problem by analysing all the possible candidates that may provide a solution to a certain computational problem, and then check if each candidate solves the problem question. There are advantages and disadvantages in adopting such kind of approach. Usually, a brute-force approach is simple to implement and it will always find a solution to the computational problem, by considering *iteratively* all the possible solutions one by one. However, its computational cost depends strictly on the number of possible candidate solutions that are available. Thus, often, for practical problems with a huge solution space, it is a rather slow, even if simple, approach to adopt. A good suggestion is to use such approach when the problem size is small.

Abstract strategy board games, such as Chess or Go, belong to that set of computational problems that have a quite huge solution space. For instance, for writing a brute-force algorithm which is able to play appropriately the game (and, hopefully, win), one needs to consider all the possible legal moves that are available on the board (shown in Figure 2). According to John Tromp, the number of all the possible legal moves in Go was determined to be *208168199381979984699478633344862770286522453884530548425639456820927419612738015378525648451698519643907259916015628128546089888314427129715319317557736620397247064840935* [Tromp, 2016]. That makes a brute-force approach totally intractable, even for an electronic computer.

Python has two alternatives for creating iterative blocks: *foreach loops* and *while loops*. The first kind of iteration mechanism is provided in Python by means of a `for` statement, which is briefly illustrated in Listing 1. In practice, all the instructions within the `for` block are repeated for each item included in a collection – any of them: a list, a queue, etc.

```
for item in <collection>:
    # do something using the current item
```
Listing 1. The general structure of a foreach loop in Python.

```
from collections import deque

def stack_from_list(input_list):
    output_stack = deque()  # the stack to create

    # Iterate each element in the input list and add it to the stack
    for item in input_list:
        output_stack.append(item)

    return output_stack
```
Listing 2. A simple function that takes a list as input and creates a stack with all the values the list contains – by using a foreach loop. The source code of this listing is available as part of the material of the course.

The foreach loop is very useful when we want to iterate on all the elements of a list for applying some sort of operation on each of them, or to find a particular value – something that we discuss in more detail in Section "Linear Search". For instance, we can use a foreach loop for creating a stack with all the elements included in a list, as shown by the simple algorithm in Listing 2.

```
while <condition>:
    # do something until the condition is true
```
Listing 3. The general structure of a while loop in Python.

The while loop, instead, works in a slightly different way. Python allows us to create it by using a `while` statement (as shown in Listing 3), which basically will repeat all the instructions that are contained in such block until the condition specified is true. For instance, it is possible to use a `while` statement for implementing the `run_forever` function that maps the Turing machine introduced in the lecture notes "Computability". Listing 4 shows one of its possible implementation in Python.

```
def run_forever():
    value = 0
    while value >= 0:
        value = value + 1
```
Listing 4. Another simple algorithm that sums *1* to a starting value indefinitely. The source code of this listing is available as part of the material of the course.

In the following sections, we reuse some of these iterations in order to implement two brute-force algorithms for searching the position of an item in a list, and for ordering a list. These are known as *linear search* and *insertion sort*.

Linear search

Searching the position in which a particular value is in a list is a common operation, which has applications in several real-life task. For instance, consider again the example of the library introduced in Section "Historic hero: Betty Holberton". The typical operation performed by a librarian, once she has received a particular request for a book, is to consult the catalogue of all the books managed by the library so as to find the appropriate location where the requested book is placed in the library. This is a sort of real application of the aforementioned abstract problem of searching a value in a list, which is formally defined as follows:

Computational problem: find the position of the first occurrence of a value within a list.

While there are several approaches that can be used to find an element in a list, we focus on a particular algorithm, called _linear search_. This approach is pretty simple. The idea is to iterate over all the items contained in an input list one by one, and checking if each of them is equal to the value we are looking for, specified as input. Once the value has been found, its position in the list is then returned. If the value is not contained in the list, no position is returned at all.

There are some aspects of the description of the linear search algorithm that must be clarified before providing its implementation in Python. The first one is the fact that an item in a list has a specific position. Which is something quite natural, if you think about it. However, in the previous lecture, we have not mentioned how to get such position. In addition to that, there is this aspect typical of Computer Science, that wants to number every position starting from _0_, instead of _1_. Thus, for instance, looking at the books in Figure 3, Terry Pratchett's _The Carpet People_ has position _0_, James Ponti's _Dead City_ has position _1_, and so on.
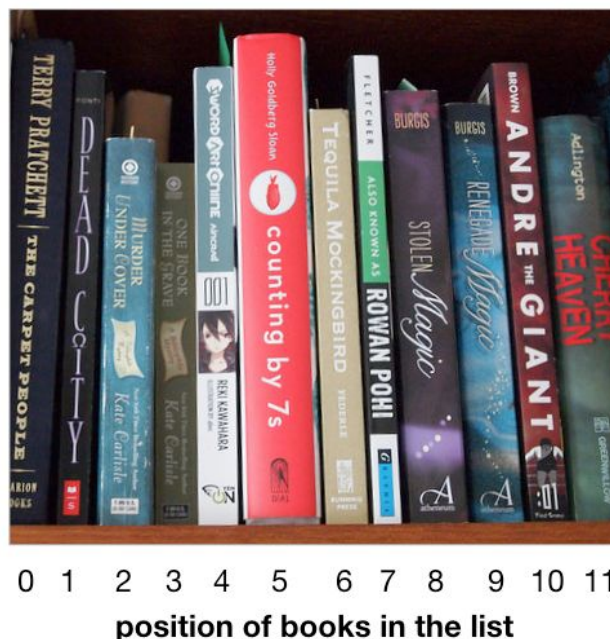

Figure 3. The position numbers assigned to a book of a list according to the typical Computer Science habit – which starts numbering from _0_.

```
for <var_item_1>, <var_item_2>, ... in <collection of tuples>:
    # here you can use directly the variables defining
    # the various items in the tuple
```
Listing 5. How Python allows one to decouple tuples in foreach loops involving a collection of tuples, by assigning a variable to each item in the tuple.

Python allows one to use the function `enumerate(<list>)` for retrieving the current position of an item in a list that is accessed by means of a foreach loop. This function takes as input a list of values, and returns a _kind_ of list (it is an _enumerate object_: it behaves like a list, but it is

not a list) of *tuples*, with two elements each: the first element is the position of the item in consideration within the list, while the second element is the item itself. In Python, a tuple is created by specifying comma-separated values between round brackets – for instance, `my_tuple = (0, 1, 2, 3, 4, 5)` assigns a tuple with six numbers to `my_tuple`. While tuples could be perceived similar to lists, they actually are not. The main difference with lists is that tuples do not provide any way for updating them with a new value, e.g. by using the *append* operation like it happens with lists. Thus, once a tuple is created, it stays forever.

```python
# Test case for the algorithm
def test_linear_search(input_list, value_to_search, expected):
    result = linear_search(input_list, value_to_search)
    if expected == result:
        return True
    else:
        return False


# Code of the algorithm
def linear_search(input_list, value_to_search):
    # iterate all the items in the input list,
    # getting also their position on the list
    for position, item in enumerate(input_list):
        # check if the current item is equal to the value to search
        if item == value_to_search:
            # if so, the position of the current item is returned
            # and the algorithm stops
            return position


# Three different test runs
print(test_linear_search([1, 2, 3, 4, 5], 3, 2))
print(test_linear_search(["Alice", "Catherine", "Bob", "Charles"],
                         "Denver", None))
print(test_linear_search(["Ron", "Harry", "Hermione"], "Ron", 0))
```

Listing 6. The linear search algorithm described in Python, including its test case. The source code of this listing is available <u>as part of the material of the course</u>.

Considering these aspects, running the function `enumerate(list(["a", "b", "c"]))` will return the following enumeration of tuples: `enumerate([(0, "a"), (1, "b"), (2, "c")])`, where the first item of each tuple is the position that the second item of the tuple has in the original list. In addition, Python allows us to decouple the items in a tuple by specifying names for each item with variables created in the `for` statement on-the-fly, as shown in <u>Listing 5</u>. Thus, for instance, a foreach loop like `for position, item in enumerate([(0, "a"), (1, "b"), (2, "c")])` will assign *0* to `position` and *"a"* to `item` in the first

iteration, *1* to `position` and *"b"* to `item` in the second iteration, and *2* to `position` and *"c"* to `item` in the third iteration.

```
list_of_books = list(["Coraline", "American Gods",
                      "The Graveyard Book", "Good Omens",
                      "Neverwhere"])

linear_search(list_of_books, "The Graveyard Book")

# FOREACH LOOP EXECUTION
# enumerate(input_list) will result in:
# enumerate([(0, "Coraline"), (1, "American Gods"),
#            (2, "The Graveyard Book"), (3, "Good Omens"),
#            (4, "Neverwhere")])
#
# Iteration 1
# position = 0
# item = "Coraline"
# item == value_to_search is False
# Continue to the next iteration
#
# Iteration 2
# position = 1
# item = "American Gods"
# item == value_to_search is False
# Continue to the next iteration
#
# Iteration 3
# position = 2
# item = "The Graveyard Book"
# item == value_to_search is True
#  Return  the  position  (value  2)  and  end  the  execution  of  the
algorithm
```
Listing 7. An example of the execution steps of the linear search algorithm implemented.

The second aspect that is implicit in the description of the linear search algorithm introduced before is that the algorithm should not return any position if the value to search is not present in the list. In Python, this can be done implicitly by not returning anything (i.e. by not ever executing the statement `return`) or explicitly by returning a particular object that means *nothing*, i.e. `None`.

At this point, we have all the ingredients for developing the linear search algorithm in Python. This is illustrated in <u>Listing 6</u>. In order to understand how the algorithm works, we have prepared

a description of the various execution steps (shown in [Listing 7](#)) of the foreach loop introduced in the algorithm. This would allow the reader to have a first glance at how iterative loops work from a purely computational point of view.

Insertion sort

As already mentioned in [Section "Historic hero: Betty Holberton"](#), the task of ordering a sequence of items is a quite important operation we usually have to deal with in everyday life. Recalling the example of the library, having the books sorted will make the operation of searching them more efficient, and would allow us to avoid to use naive approaches for the search, e.g. the one introduced in [Section "Linear search"](#), which is quite expensive if we have billion of books to check.
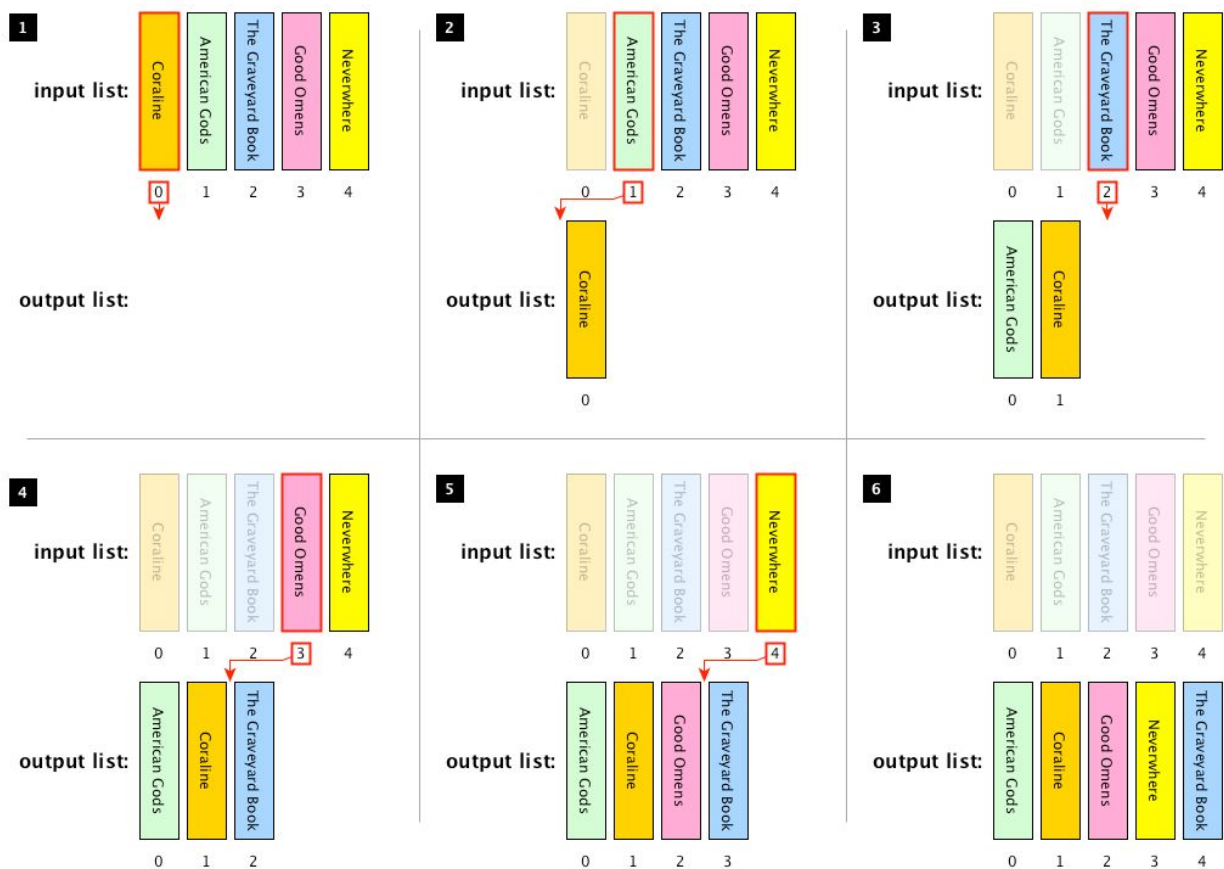

Figure 4. The execution of the insertion sort algorithm using the following list of book titles as input: *Coraline*, *American Gods*, *The Graveyard Book*, *Good Omens*, *Neverwhere*. The book highlighted by a bold red border is the one currently selected in the particular iteration of the algorithm, and the red arrow shows where it should be placed in the output list. When a book title in the input list is shown in transparency it means that it has been already considered in some previous iterations, and it has been already placed in the output list in the correct order.

In this section we propose one particular brute-force algorithm for addressing the following computational problem:

Computational problem: sort all the items in a given list.

The algorithm that we want to use for addressing the aforementioned computational problem is called _insertion sort_. It is one of the simpler sorting algorithms to implement, and it is quite efficient for small lists. The idea behind this algorithm is the following. It considers the items in the list one by one, in the order they have been placed. Thus, at each iteration, it removes one item from the input list, it finds the right location for it looking at the elements on its left only – that have been already ordered in previous iterations – and it inserts it in the right location. It finishes once all the items in the list have been considered. An example of the execution of this algorithm is shown in Figure 4.

In order to provide a Python implementation of this algorithm, we need to introduce two functions and one additional operation that can be used on lists. The first function we need to use is `range(<stop_number>)`. It takes a non negative number as input and returns a _kind_ of list (i.e. a _range object_ behaving like a list) of all numbers from _0_ to the one preceding `stop_number`. Thus, for instance, `range(4)` returns the `range([0, 1, 2, 3])`, while `range(0)` returns the empty range object `range([])`.

The other function is `reversed(<input_list>)`, that takes a list as input and returns a _kind_ of list (i.e. an _iterator_ for the list, from the last item to the first one) with all the elements sorted in the opposite order. Thus, for instance, `reversed(list([0, 1, 2, 3]))` returns `iterator([3, 2, 1, 0])`. These two functions can be used in combination so as to obtain, at a certain iteration of the insertion sort algorithm, the positions of the items that have been already ordered in the past iterations by specifying the position of the current item considered – e.g. `reversed(range(2))` returns `iterator(range([1, 0]))` starting from the position (i.e. _2_) of the third item in the input list.

In addition, we need a way for selecting an item in a list and for inserting an item in a specific position in a list. For addressing these tasks, Python makes available the additional list methods `<list>[<position>]` and `<list>.insert(<position>, <item>)`. In particular, the former returns the item in the list at the specified position – e.g. if we have the list `my_list = list(["a", "b", "c"])`, `my_list[1]` returns _"b"_. The latter method allows one to put `<item>` in the position specified, and it shifts all the elements with position greater than or equal to `<position>` of one position – e.g., by applying `my_list.insert(1, "d")`, the list in `my_list` is modified as follows: `list(["a", "d", "b", "c"])`.

At this point, we have all the ingredients for developing the insertion sort algorithm in Python. This is illustrated in Listing 8.

```python
def insertion_sort(input_list):
    result = list()  # A new empty list where to store the result

    # iterate all the items on the input list
    for item in input_list:

        # initialise the position where to insert the item
        # at the end of the result list
        insert_position = len(result)

        # iterate, in reverse order, all the positions of all the
        # items already included in the result list
        for prev_position in reversed(range(insert_position)):

            # check if the current item is less than the one in
            # prev_position in the result list
            if item < result[prev_position]:
                # if it is so, then the position where to insert the
                # current item becomes prev_position
                insert_position = prev_position

        # the item is inserted into the position found
        result.insert(insert_position, item)

    return result  # the ordered list is returned
```
Listing 8. The insertion sort algorithm described in Python. The source code of this listing is available as part of the material of the course, and includes also the test case of the algorithm.

Exercises

1. Write down the execution steps of `linear_search(list(["Coraline", "American Gods", "The Graveyard Book", "Good Omens", "Neverwhere"]), "The Sandman")`, as explained in Listing 7.
2. Create the test case for the algorithm introduced in Listing 2.
3. Write in Python the function `def my_enumerate(input_list)` which behave like the built-in function `enumerate()` introduced in Section "Linear search" and returns a proper list, and accompany the function with the related test case. It is not possible to use the built-in function `enumerate()` in the implementation.
4. Write in Python the function `def my_range(stop_number)` which behave like the built-in function `range()` introduced in Section "Insertion sort" and returns a proper list,

and accompany the function with the related test case. It is not possible to use the built-in function `range()` in the implementation.

5. Write in Python the function `def my_reversed(input_list)` which behave like the built-in function `reversed()` introduced in [Section "Insertion sort"](#) and returns a proper list, and accompany the function with the related test case. It is not possible to use the built-in function `reversed()` in the implementation.

References

Tromp, J. (2016). Counting Legal Positions in Go. http://tromp.github.io/go/legal.html (last visited 10 November 2017)