Theory: Algorithms in Python

28 minutes 7 / 7 problems solved

Start practicing

As you know, algorithms are language-independent, meaning that the same algorithm can be implemented in different programming languages. Since your goal is to learn Python, we will consider how to implement basic algorithms using this language and its features.

§1. Implementing algorithms in Python

Python has many libraries that provide ready-to-use functions for various problems. Algorithms are not an exception. You can find most of the widely-used algorithms in the Python standard library or in the external packages. Usually, an algorithm is a standalone function that may either be implemented directly in Python or represent a wrapper that calls a function written in another programming language.

Python standard library contains such functions as `sorted`, `min`, `max`, which represent algorithms for sorting and finding the minimum/maximum values in a collection of elements. There are also modules that unite the algorithms of one subject area, for example:

- `math` is a module with mathematical algorithms. For example, `math.sqrt` is an algorithm that calculates the square root of a number.
- `collections` is a module that extends and improves the functionality of standard collections such as `list` or `tuple`. An example is `collections.deque`, an efficient implementation of a double-ended queue data structure.

Although most standard algorithms are already present in libraries and packages, we will learn to implement some of them ourselves. This will help you practice your Python skills and understand how these algorithms work under the hood.

§2. A particular example

Let's implement a simple algorithm in Python and consider some typical problems that arise during the process. Below is an algorithm that finds the index of the maximum element in a list of numbers:

```
def indexof_max(numbers):
    index = 0

    for i in range(1, len(numbers)):
        if numbers[i] > numbers[index]:
            index = i

    return index
```

The `indexof_max` function takes a list of numbers as an input and returns the index of the maximum element in this list. The algorithm works as follows:

- First, we initialize the variable `index` with zero indicating that the element with this index is a candidate to be the maximum.
- Then, we start iterating through the list, compare the current element with the maximum element, and update the index if necessary.
- Finally, we return the index of the maximum element.

§3. Invoking the function

The function above can be invoked as follows:

```
numbers = [2, 3, 5, 1, 2, 0, 4]
index = indexof_max(numbers)
print(index)  # 2
```

Current topic:

- ✓ Algorithms in Python

Topic depends on:

- ✓ Computer algorithms
- ✓ Indexes 7★ Stage 3
- ✓ Functional decomposition 7★ Stage 1
- ✓ List comprehension 5★

Topic is required for:

- Linear search in Python
- Searching a substring in Python
- Selection sort in Python
- Merge sort in Python
- ✓ Stack in Python
- ✓ Queue in Python
- Recursion in Python
- Quick sort in Python

Table of contents:

Here, we create a list of integers and apply the described function. The maximum element is equal to 5 and its index is 2.

Let's try to use lists of other sizes:

```
numbers = [7, 9, 5]
index = indexof_max(numbers)
print(index)  # 1
```

Here, the list `numbers` consist of three elements. The index of the maximum element is 1. Below is how the function works for a list consisting of a single element:

```
numbers = [7]
index = indexof_max(numbers)
print(index)  # 0
```

§4. Considering corner cases

Looks like the function works well for the above lists. Now, let's try to answer the following questions:

- What will happen if a list contains several elements equal to the maximum value?
- What will happen if a list is empty?
- What will happen if a list contains not only numbers but also strings?

Let's go through these questions one by one.

First, the function finds the index of the first maximum element in a list:

```
numbers = [3, 1, 9, 9, 2]
index = indexof_max(numbers)
print(index)  # 2
```

This is because we use **greater than (>)** operator and not **greater than or equal** (>=). In some cases, you may need to find the last index of the maximum element. You can easily modify the provided function to do that.

Secondly, if a list is empty, the function returns 0:

```
empty_list = []
index = indexof_max(empty_list)
print(index)  # 0
```

This happens because at the very start we define the index as 0. Such behavior is ambiguous: it doesn't allow us to distinguish between the case when the element with the index 0 is the maximum and the case when the input list is empty.

To fix the issue, let's return a special value of -1 when the input list is empty. The function should be modified in the following way:

```
def indexof_max(numbers):
    if not numbers:
        return -1

    # the rest of the algorithm
    ...
```

Now it works as follows:

```
empty_list = []
index = indexof_max(empty_list)
print(index)  # -1
```

The last question is a bit trickier to answer. What will be the result of the code below?

```
numbers = [1, 2, 3, "four", 5, 6]
print(indexof_max(numbers))
```

After invoking the code, we will get the `TypeError` exception with the following error message:

```
'>' not supported between instances of 'str' and 'int'
```

The problem is that the interpreter doesn't know how to compare a string with an integer. This issue may also be fixed, but we will discuss it in another topic.

§5. Summary

Python has a wide range of resources that we can use to work with algorithms: from standard library features to special libraries and modules. In the following topics, we will learn about some basic algorithms and implement them ourselves to improve our Python skills and understand the internal logic of these algorithms. While implementing algorithms, it's important to take into account all kinds of situations, so always think about corner cases!