FRUIZIONE E UTILIZZO DEI MATERIALI DIDATTICI

- **E’ vietata** la **copia**, la **rielaborazione**, la **riproduzione** dei contenuti e immagini presenti nelle lezioni in qualsiasi forma

- **E’ inoltre vietata** la **diffusione**, la **redistribuzione** e la **pubblicazione** dei contenuti e immagini, incluse le registrazioni delle videolezioni con qualsiasi modalità e mezzo non autorizzati espressamente
dall'autore o da Unica

Exercise

Write a function that adds the values of an array, skipping all the values equal to 13 and the value immediately after 13.

Example:

```
[1, 13, 10, 1, 13, 13, 13, 10, 1, 13] -> 3
```

Exercise

Write a function that adds the values of an array, skipping all the values equal to 13 and the value immediately after 13.

Example:

```
[1, 13, 10, 1, 13, 13, 13, 10, 1, 13] -> 3
```

Solution: `state/sum_skip_01`

The proposed solutions have a common defect: they are 'ad hoc' solutions. They do not allow us to define a general criterion to solve a class of similar problems.

Let's consider such kind of problems:

- Sum the values in a list, excluding the values between two specific values in the sequence.
  I.e., given the list $[1, 3, 7, \mathbf{10}, \mathbf{10}, 9, 3, 7, \mathbf{10}, 9, 4]$ sum all values skipping those between 7 and 9 in the sequence.

Let's consider such kind of problems:

- Write a function that accepts a list of characters and prints them according to the following rules:
  - the function starts printing lowercase characters
  - when it encounters the **u** character, prints uppercase until it encounters the **l** character
  - characters between two **h**'s are not printed,

i.e.,input/output

```
['a','o', 'u', 'm', 'n', 'l', 'o','h', 'x', 'y', 'h', 'a', 'b', 'c']
['a','o',  ? , 'M', 'N', 'l', 'o', ?,            ? , 'a', 'b', 'c']
```

Let's consider such kind of problems:

- A game character gains a superpower if he finds the three amulets **a, b, c** in that order. He can use the superpower three times, then he loses the three amulets and has to start over.
  Variants: he gains superpowers only if he finds the objects **a, b** consecutively, i.e.,

```
x1, a, b, x2, c,...
x1, a, b, c,...
```
are correct sequence.

```
x1, a, x2, b, c,...
```
This is not correct.

It is easy to see a **common pattern** in these problems. We have an **object with a state**.
The behavior of the object depends on the input and the state. The input causes a state transition.

"sum all values skipping those between 7 and 9 in the sequence."

$$[1, 3, 7, \mathbf{10}, \mathbf{10}, 9, 3, 7, \mathbf{10}, 9, 4]$$

It is useful to analyze a possible strategy to solve this class of problems.

It is also useful to better describe these problems, because the presented description is ambiguous.

For example, in problem no. 2, will the characters **u** and **l** be printed, or are they only used to convey commands?
If a **u** character is encountered between the two **h**'s, the object switches to the 'uppercase' state even if it does not print the **u** character?

```
['a', 'o', 'u', 'm', 'n', 'l', 'o','h', 'x', 'y', 'h', 'a', 'b', 'c']
['a', 'o',  ? , 'M', 'N', 'l', 'o', ?,              ? , 'a', 'b', 'c']
```

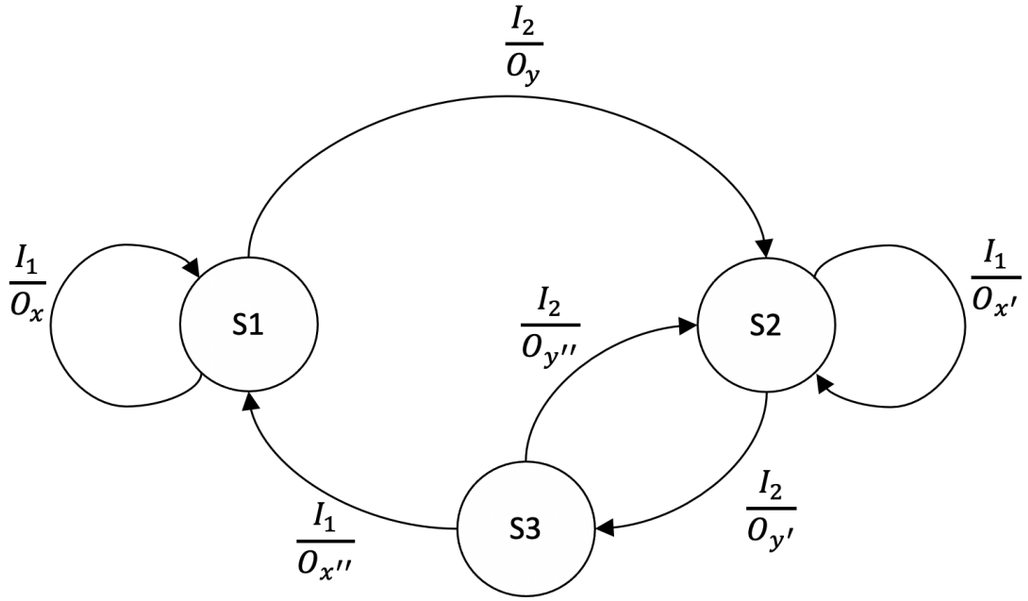
A finite-state machine[(1)] (FSM) is an abstract machine that can be in one of a finite number of states. The FSM can change from one state to another (transition) in response to some inputs.

State-transition table

(S: state, I: input, O: output)

Current state \ Input	I_1	I_2	…	I_n
S_1	S_i/O_x	S_j/O_y	…	S_k/O_z
S_2	$S_{i'}/O_{x'}$	$S_{j'}/O_{y'}$	…	$S_{k'}/O_{z'}$
…	…	…	…	…
S_m	$S_{i''}/O_{x''}$	$S_{j''}/O_{z''}$	…	$S_{k''}/O_{z''}$

State diagram

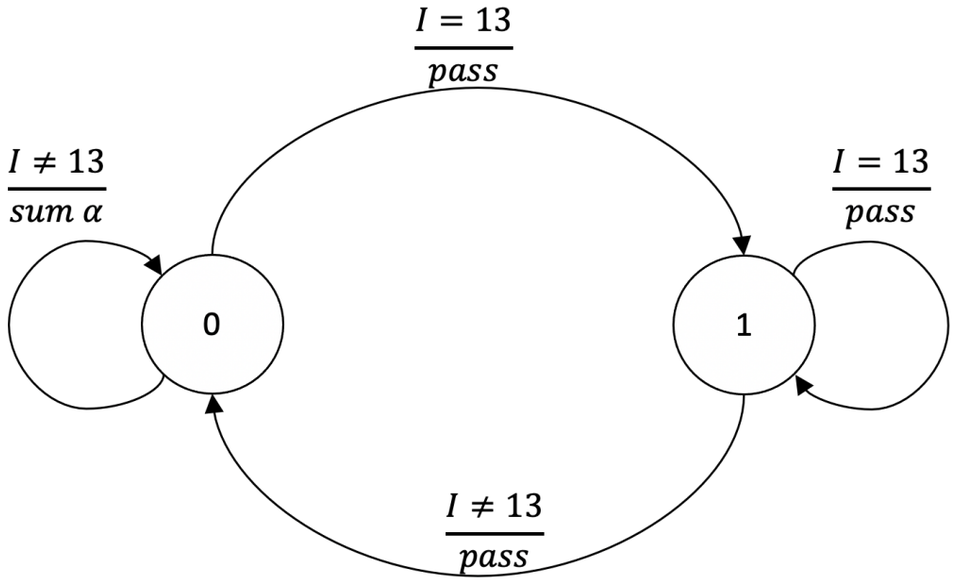


---

(1) https://en.wikipedia.org/wiki/Finite-state_machine. A Mealy machine the output values are determined both by its current state and the current inputs. In a Moore machine the output values are determined solely by its current state. It is also possible to associate actions with a state: an **entry action**, performed when entering the state, and an **exit action**, performed when exiting the state.

We can describe the `sum_skip` exercise (that skip the value 13 and the next element) in this way (solution: `state/sum_skip_02`. Try to solve it!)

- The object has two states, 0 and 1.
- In state 0, if the input is $\neq 13$ the object adds the values in input, and the object remains in state 0. If the input is 13, the value is skipped, and the object changes to state 1.
- In state 1, the object does not sum the values in input. If the input is 13, the object remains in state 1. If the input is other than 13, the object changes to state 0 (and do not print the value).



The solution in `state/sum_skip_02` uses an integer variable to store the state, and a **conditional structure** to manage the action and the state transition.

You can implement this solution using a function or a class.

Discussion.

In simple cases, explicit conditional logic is often fine.

Problems:

Discussion.

In simple cases, explicit conditional logic is often fine.

Problems:

1. The conditional logic is **error-prone**. It is easy to make errors, especially when there are complex, nested if-elif-else branches.

Discussion.

In simple cases, explicit conditional logic is often fine.

Problems:

1. The conditional logic is **error-prone**. It is easy to make errors, especially when there are complex, nested if-elif-else branches.
2. The solution **doesn't scale**. In large state machines, the code could extend for pages and pages of conditional statements. Applying changes to this code can be very difficult and can lead to maintenance problems. Simply adding a new state implies changing several functions.

Discussion.

In simple cases, explicit conditional logic is often fine.

Problems:

1. The conditional logic is **error-prone**. It is easy to make errors, especially when there are complex, nested if-elif-else branches.
2. The solution **doesn't scale**. In large state machines, the code could extend for pages and pages of conditional statements. Applying changes to this code can be very difficult and can lead to maintenance problems. Simply adding a new state implies changing several functions.
3. **Duplication**. The conditional logic tends to be repeated, with small variations, in all functions that access the state variable. Duplication leads to error-prone maintenance.

Discussion.

In simple cases, explicit conditional logic is often fine.

Problems:

1. The conditional logic is **error-prone**. It is easy to make errors, especially when there are complex, nested if-elif-else branches.
2. The solution **doesn't scale**. In large state machines, the code could extend for pages and pages of conditional statements. Applying changes to this code can be very difficult and can lead to maintenance problems. Simply adding a new state implies changing several functions.
3. **Duplication**. The conditional logic tends to be repeated, with small variations, in all functions that access the state variable. Duplication leads to error-prone maintenance.
4. **No separation of concerns**. There is no clear separation between the code of the state machine itself and the actions associated with the various events.

Another approach makes use of a **transition table** (`state/sum_skip_03`).

python implementation: the transition table is a dictionary

- **key**: tuple (state, input)
- **value**: a dictionary containing the action and the next state.

```
transition_table = {
    (state_0, input_0): {action: action, next_state: next_state},
    (state_0, input_1): {action: action, next_state: next_state},
    # ...
    (state_n, input_k): {action: action, next_state: next_state},
}
```

How to handle cases in which the tuple is not present? For example, it may happen that the input is not explicit.
It is precisely the case of the proposed exercise, in which the input can be $v == 13$ or $v \neq 13$, but all values other than 13 are not explicitly listed.

Another approach makes use of a **transition table** (`state/sum_skip_03`).

python implementation: the transition table is a dictionary

- **key**: state
- **value**: a dictionary with:
  - **key**: input
  - **value**: a dictionary containing the action and the next state.

```
transition_table = {
    state_0: {
        input_0: {action: action, next_state: next_state},
        input_1: {action: action, next_state: next_state},
        default_input: {action: action, next_state: next_state}
        },
    state_1: {
        input_0: {action: action, next_state: next_state},
        input_1: {action: action, next_state: next_state},
        default_input: {action: action, next_state: next_state}
    }}
```

This solution allows us to easily manage the 'default' input using `table.get(key, default)`.

The design based on a transition table solves the previous problems.

1. It scales well. Independent of the size of the state machine, the code for a state transition is just a table-lookup.
2. No code duplication.
3. Easy to modify. When adding a new state, the change is limited to the transition table.
4. Easy to understand. A well structured transition table provides a good overview of the complete lifecycle.
5. Separation of concerns.

The problem is greatly simplified if each input does not result in actions but **only state transitions**.

The transition table solution is **excellent** if we have **well-defined inputs** and **no actions** to perform after the state transition.

If there are **actions** to be performed after the transition, and these actions may have **different parameters depending on the state and input**, the transition table becomes more complicated, losing some initial elegance and clarity.