Exercise 1: Invariant checking I
In the book, you have seen an algorithm for invariant checking by forward depth-first search (Section 3.3.1, Algorithm 4). Below, we give a recursive version of this algorithm.

Algorithm 1: DFS-based invariant checking

```
Algorithm 1: DFS-based invariant checking
input : a finite transition system TS and a propositional formula \( \Phi \)
output: “yes” if \( TS \models \text{always } \Phi \), otherwise “no” and a counterexample

\[
R := \emptyset; \quad \text{// set of states ("Reachable")}
U := \varepsilon; \quad \text{// stack of states ("Unfinished")}
\]

forall \( s \in I \) do
  if DFS\((s, \Phi)\) then
    return("no", reverse(U)); \quad \text{// path from } s \text{ to error state}
  end

return("yes"); \quad \text{// } TS \models \text{"always } \Phi \text{"}
```

```
function DFS\((s, \Phi)\)
  push\((s, U)\);
  if \( s \notin R \) then
    \[
    R := R \cup \{s\}; \quad \text{// mark } s \text{ as reachable}
    \]
    if \( s \not\models \Phi \) then
      return("true"); \quad \text{// } s \text{ is an error state}
    else
      forall \( s' \in \text{Post}(s) \) do
        if DFS\((s', \Phi)\) then
          return("true"); \quad \text{// } s' \text{ lies on a path to an error state}
        end
      end
    end
  end
  pop\((U)\);
  return("false");
end
```
Apply this algorithm to the following transition system whose set of atomic propositions is $AP = \{a, b\}$. The invariant $\Phi$ to be checked is the propositional logical formula $a$.

Whenever you iterate over a set of states, always take state $s_i$ before state $s_j$ if $i$ is smaller than $j$.

Present the execution of the algorithm by writing down the contents of the set $R$ and the stack $U$ directly before every call to the function $\text{DFS}$.

Is the sequence of the contents of $R$ and $U$ different for the algorithm in the book?

![Transition System Diagram]

**Exercise 2: Invariant checking II**

The “DFS-based invariant checking” algorithm presented above (or in the book) always computes a minimal bad prefix. However, the algorithm does not necessarily compute a bad prefix of minimal total length (there might be two minimal bad prefixes of different length). What is an example that shows that the prefix that is returned does not always have minimal total length?

For this purpose, provide the following.

- A transition system that has three states $s_0, s_1, s_2$.
- An invariant.
- The (non-minimal) bad prefix that is computed by the algorithm that uses the following convention for iterating over a set of states. Always take state $s_i$ before state $s_j$ if $i$ is smaller than $j$.
- A minimal bad prefix.

**Exercise 3: Invariant checking III**

Give an algorithm (in pseudocode, analogously to the algorithm presented above or in the book) for invariant checking such that, in case the invariant is refuted, a bad prefix of minimal total length is provided as an error indication.

The algorithm should terminate for all finite transition systems.

*Hint:* You may modify the algorithm presented above (or in the book) appropriately. You may also want to use two data structures: A queue and a map.

A queue is a list with two operations:

- void $\text{add}(\text{Element})$ adds a new element at the end.
- $\text{Element} \ \text{remove}()$ removes the element at the front (FIFO principle).
A map behaves like a partial function. That is, it stores a value for a given key. It has the following operations:

- `void add(Key, Value)` adds a new mapping from a key to a value.
- `Value get(Key)` returns the value for the given key.
- `boolean has(Key)` returns `true` iff the map stores a value for the given key.

You can use the map to store a predecessor state for a given state. This can be helpful for constructing the bad prefix in the end.

**Exercise 4: Properties of the closure**
Prove that for the closure operator `closure` defined in Definition 3.26 the following inclusion (resp. equality) holds for every linear time property `P`.

(a) $P \subseteq closure(P)$

(b) $closure(P) = closure(closure(P))$ (the closure operator `closure` is idempotent)