Getting started I

May 19, 2019

1 Getting started I

We will explain basics of ACT-R and pyactr on several very simple models/minds that play Memory.

2 Model 1 - introduction to the goal buffer and production rules

The first model will be a mind that makes just one action - it will plan to turn one card in Memory. It will check that the game to be played is Memory. If so, it will take an action. Let's assume that there are 10 cards that can be uncovered by pressing numbers 1,2,...9,0. Thus, the action should be to press a key with one of the numbers.

We create this mind as follows:

1. we import the pyactr package and initialize a model (mind)
2. we create knowledge that the mind has; this knowledge has two parts:
    a) the goal that will be in the mind at the point we start the simulation
    b) the production knowledge, consisting of one production rule

When this is done, we will run the simulation

First, let us import the pyactr package and initialize the mind (point 1).

```
In [1]: import pyactr as actr

        playing_memory = actr.ACTRModel()
```

Now we will create the knowledge that the mind has (point 2).

First, its current goal. The goal is seen as a container (in ACT-R terminology, a buffer) that carries a chunk. A chunk, in turn, is a list of attribute-value pairs. (Attributes are called slots in ACT-R.)

So, let's specify the chunk that the goal will carry. We do that first by defining a chunk type, that is, by defining what slots a chunk in the goal buffer can carry. We will define a chunk type "playgame" and assume that there are two slots in that chunk type: "game" and "activity".

```
In [2]: actr.chunktype("playgame", "game, activity")
```

After defining the chunk type, we can specify the token that the goal buffer will have at the start of modeling/simulation. The starting point only specifies what game we are playing. Let's call this starting chunk "initial_chunk".

```
In [3]: initial_chunk = actr.makechunk(typename="playgame", game="memory")
```

Note that the chunk has a value specified for "game", but no value for "activity". The activity will be decided by the mind through production knowledge. Before we get there, we have to set the goal buffer in the model and add the initial chunk into it. This will represent the current goal of the mind.

```
In [4]: goal = playing_memory.set_goal("goal")
```

```
In [5]: goal.add(initial_chunk)
```

Let us check that the initial chunk is in the goal buffer:

```
In [6]: print(goal)
```

```
{playgame(activity= , game= memory)}
```

Note that in printing, we automatically get to see all slots, but some slots might have no values.

Let us now move onto production knowledge, which consist of production rules. Production rules are conditionalized actions. We will have only one rule: IF the goal is to play Tic-Tac-Toe and there is no activity yet, THEN plan an activity, namely, plan to press 1.

The rule is specified below. It has to be inputed as a string, in which the part before "==>" describes the condition, and the part after "==>" is the action. Here, we say that if the chunk in the goal buffer has memory as the value of the slot "game" and no activity assigned, it will introduce a new activity as its goal, "press" (a key).

Note how it is done: 1. The first line in the string, "=goal>", specifies what buffer we condition on. 2. The lines "isa playgame" up to "activity None" describe the chunk that must be in the buffer, otherwise the rule will not fire. Each line is a slot-value pair and the first line, "isa playgame" specifies what chunk type this is. Slot-value pairs can appear in any order. 3. The line following "==>", "=goal>", specifies what buffer will be modified if the rule fires (in our case, the goal buffer will be modified). 4. The last two lines describe how the chunk in the buffer is modified. Again, each line corresponds to one slot-value pair, and the line "isa playgame" specifies what chunk type the modified chunk is.

Note that the method returns the actual rule, which gets printed in a condense, readable format.

```
In [7]: playing_memory.productionstring(name="startplaying", string="""
        =goal>
        isa  playgame
        game memory
        activity None
        ==>
        =goal>
        isa playgame
        activity presskey""") #this rule will be modified later
```

```
Out[7]: {'=goal': playgame(activity= None, game= memory)}
        ==>
        {'=goal': playgame(activity= presskey, game= )}
```

This is all we need to do to specify the model. We can now run the simulation.

```
In [8]: simulation_game = playing_memory.simulation()
```

```
In [9]: simulation_game.run()
```

```
(0, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0, 'PROCEDURAL', 'RULE SELECTED: startplaying')
(0.05, 'PROCEDURAL', 'RULE FIRED: startplaying')
(0.05, 'goal', 'MODIFIED')
(0.05, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.05, 'PROCEDURAL', 'NO RULE FOUND')
```

The simulation outputs a series of actions (called a trace). Each line in the trace specifies three things: the time at which the action took place (in seconds), what part of ACT-R model is affected, and what action is carried out.

Running a (successful) simulation is usually the end point of ACT-R modeling. However, after we are done we can explore final stages of the mind. For instance, we can check what the goal buffer looks like at the end of the simulation. This confirms that our model worked correctly: it modified the value of "activity" in the goal buffer to pressing a key.

```
In [10]: print(goal)
```

```
{playgame(activity= presskey, game= memory)}
```

A short side note on Python. Any buffer (including the goal buffer) is like a set (it inherits from Set) and several standard set operations are possible to apply to it. For instance, we can pop its only element from it.

```
In [11]: final_chunk = goal.pop()
         print(final_chunk)
```

```
playgame(activity= presskey, game= memory)
```

Chunks inherit from named tuples. We can explore, for example, their length, we can iterate through them, we can check values of particular slots etc.

```
In [12]: len(final_chunk)
```

```
Out[12]: 2
```

```
In [13]: for slot_value in final_chunk:
             print(slot_value)
```

```
('activity', 'presskey')
('game', memory)
```

```
In [14]: print(final_chunk.game)
```

```
memory
```

3 Model 2 - introduction to the motor module

Our previous mind/model could fire one production rule. The rule planned a new activity, pressing a key. However, the model could not in any way carry out that action in the environment. That is, the model was just internal representation with no peripherals.

We will now let the mind modify the environment through the motor module. The motor module can simulate a key press on a keyboard.

The motor module will be affected in a new production rule. The production rule checks that our activity is "presskey". If this is so, the action will be carried out by a new buffer, manual (this is the buffer of the motor module). The buffer consists of a special chunk, "_manual", which is pre-specified in pyactr. The chunk has two slots, the slot "cmd", which describes the action (with a value "press_key"), and the slot "key", which specifies what key will be pressed.

Here is the rule. As we will see in a second, this production rule is not completely correct, but it is a good starting point.

```
In [15]: playing_memory.productionstring(name="presskey", string="""
         =goal>
         isa  playgame
         game memory
         activity presskey
         ==>
         +manual>
         isa _manual
         cmd press_key
         key 1""") #this rule is NOT correct
```

```
Out[15]: {'=goal': playgame(activity= presskey, game= memory)}
         ==>
         {'+manual': _manual(cmd= press_key, key= 1)}
```

Let us now run the model. We have to insert the starting chunk in the goal buffer and run the simulation. (By default, every simulation starts at time 0s.)

```
In [16]: goal.add(initial_chunk)
```

```
In [17]: simulation_game = playing_memory.simulation()
         simulation_game.run()
```

```
(0, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0, 'PROCEDURAL', 'RULE SELECTED: startplaying')
(0.05, 'PROCEDURAL', 'RULE FIRED: startplaying')
(0.05, 'goal', 'MODIFIED')
(0.05, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.05, 'PROCEDURAL', 'RULE SELECTED: presskey')
(0.1, 'PROCEDURAL', 'RULE FIRED: presskey')
(0.1, 'manual', 'COMMAND: press_key')
(0.1, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.1, 'PROCEDURAL', 'RULE SELECTED: presskey')
(0.15, 'PROCEDURAL', 'RULE FIRED: presskey')
```

```
(0.15, 'manual', 'COMMAND: press_key')
(0.15, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.15, 'PROCEDURAL', 'RULE SELECTED: presskey')
(0.2, 'PROCEDURAL', 'RULE FIRED: presskey')
(0.2, 'manual', 'COMMAND: press_key')
(0.2, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.2, 'PROCEDURAL', 'RULE SELECTED: presskey')
(0.25, 'PROCEDURAL', 'RULE FIRED: presskey')
(0.25, 'manual', 'COMMAND: press_key')
(0.25, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.25, 'PROCEDURAL', 'RULE SELECTED: presskey')
(0.3, 'PROCEDURAL', 'RULE FIRED: presskey')
(0.3, 'manual', 'COMMAND: press_key')
(0.3, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.3, 'PROCEDURAL', 'RULE SELECTED: presskey')
(0.35, 'PROCEDURAL', 'RULE FIRED: presskey')
(0.35, 'manual', 'COMMAND: press_key')
(0.35, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.35, 'PROCEDURAL', 'RULE SELECTED: presskey')
(0.4, 'PROCEDURAL', 'RULE FIRED: presskey')
(0.4, 'manual', 'COMMAND: press_key')
(0.4, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.4, 'PROCEDURAL', 'RULE SELECTED: presskey')
(0.45, 'PROCEDURAL', 'RULE FIRED: presskey')
(0.45, 'manual', 'COMMAND: press_key')
(0.45, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.45, 'PROCEDURAL', 'RULE SELECTED: presskey')
(0.5, 'PROCEDURAL', 'RULE FIRED: presskey')
(0.5, 'manual', 'COMMAND: press_key')
(0.5, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.5, 'PROCEDURAL', 'RULE SELECTED: presskey')
(0.55, 'PROCEDURAL', 'RULE FIRED: presskey')
(0.55, 'manual', 'COMMAND: press_key')
(0.55, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.55, 'PROCEDURAL', 'RULE SELECTED: presskey')
(0.6, 'PROCEDURAL', 'RULE FIRED: presskey')
(0.6, 'manual', 'COMMAND: press_key')
(0.6, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.6, 'PROCEDURAL', 'RULE SELECTED: presskey')
(0.65, 'PROCEDURAL', 'RULE FIRED: presskey')
(0.65, 'manual', 'COMMAND: press_key')
(0.65, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.65, 'PROCEDURAL', 'RULE SELECTED: presskey')
(0.7, 'PROCEDURAL', 'RULE FIRED: presskey')
(0.7, 'manual', 'COMMAND: press_key')
(0.7, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.7, 'PROCEDURAL', 'RULE SELECTED: presskey')
(0.75, 'PROCEDURAL', 'RULE FIRED: presskey')
```

```
(0.75, 'manual', 'COMMAND: press_key')
(0.75, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.75, 'PROCEDURAL', 'RULE SELECTED: presskey')
(0.8, 'PROCEDURAL', 'RULE FIRED: presskey')
(0.8, 'manual', 'COMMAND: press_key')
(0.8, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.8, 'PROCEDURAL', 'RULE SELECTED: presskey')
(0.85, 'PROCEDURAL', 'RULE FIRED: presskey')
(0.85, 'manual', 'COMMAND: press_key')
(0.85, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.85, 'PROCEDURAL', 'RULE SELECTED: presskey')
(0.9, 'PROCEDURAL', 'RULE FIRED: presskey')
(0.9, 'manual', 'COMMAND: press_key')
(0.9, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.9, 'PROCEDURAL', 'RULE SELECTED: presskey')
(0.95, 'PROCEDURAL', 'RULE FIRED: presskey')
(0.95, 'manual', 'COMMAND: press_key')
(0.95, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.95, 'PROCEDURAL', 'RULE SELECTED: presskey')
```

```
/home/jakub/Documents/moje/computations and corpora/python/pyactr/pyactr/simulation.py:167: Us
  warnings.warn("Process in %s interupted" % name)
```

Ok, this looks promising, but not entirely right. The first two rules apply correctly, but afterwards the model gets into a loop: it fires the rule "presskey", but before the key 1 can be pressed, it fires the rule again, and again and again. That the model never gets a chance to actually press the button can be seen from the warning that the manual process is (repeatedly) interrupted.

The problem is that the action of the rule "presskey" only affects the environment. Since the internal stage of the model did not change, the rule can fire again and again, creating a loop.

We can avoid this behavior by specifying two actions in the rule: 1. The manual buffer has to press a key 2. The goal buffer has to be modified; in particular, we can delete its activity since it has been taken over by the motor module

```
In [18]: playing_memory.productionstring(name="presskey", string="""
         =goal>
         isa  playgame
         game memory
         activity presskey
         ==>
         +manual>
         isa _manual
         cmd press_key
         key 1
         =goal>
         isa playgame
         activity None""") #this rule works correctly
```

```
Out[18]: {'=goal': playgame(activity= presskey, game= memory)}
         ==>
         {'+manual': _manual(cmd= press_key, key= 1), '=goal': playgame(activity= None, game= 
```

Let's see how this works.

```
In [19]: goal.add(initial_chunk)
```

```
In [20]: simulation_game = playing_memory.simulation()
         simulation_game.run()
```

```
(0, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0, 'PROCEDURAL', 'RULE SELECTED: startplaying')
(0.05, 'PROCEDURAL', 'RULE FIRED: startplaying')
(0.05, 'goal', 'MODIFIED')
(0.05, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.05, 'PROCEDURAL', 'RULE SELECTED: presskey')
(0.1, 'PROCEDURAL', 'RULE FIRED: presskey')
(0.1, 'goal', 'MODIFIED')
(0.1, 'manual', 'COMMAND: press_key')
(0.1, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.1, 'PROCEDURAL', 'RULE SELECTED: startplaying')
(0.15, 'PROCEDURAL', 'RULE FIRED: startplaying')
(0.15, 'goal', 'MODIFIED')
(0.15, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.15, 'PROCEDURAL', 'RULE SELECTED: presskey')
(0.2, 'PROCEDURAL', 'RULE FIRED: presskey')
(0.2, 'goal', 'MODIFIED')
(0.2, 'manual', 'COMMAND: press_key')
(0.2, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.2, 'PROCEDURAL', 'RULE SELECTED: startplaying')
(0.25, 'PROCEDURAL', 'RULE FIRED: startplaying')
(0.25, 'goal', 'MODIFIED')
(0.25, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.25, 'PROCEDURAL', 'RULE SELECTED: presskey')
(0.3, 'PROCEDURAL', 'RULE FIRED: presskey')
(0.3, 'goal', 'MODIFIED')
(0.3, 'manual', 'COMMAND: press_key')
(0.3, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.3, 'PROCEDURAL', 'RULE SELECTED: startplaying')
(0.35, 'PROCEDURAL', 'RULE FIRED: startplaying')
(0.35, 'goal', 'MODIFIED')
(0.35, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.35, 'PROCEDURAL', 'RULE SELECTED: presskey')
(0.4, 'PROCEDURAL', 'RULE FIRED: presskey')
(0.4, 'goal', 'MODIFIED')
(0.4, 'manual', 'COMMAND: press_key')
(0.4, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.4, 'PROCEDURAL', 'RULE SELECTED: startplaying')
```

```
(0.45, 'PROCEDURAL', 'RULE FIRED: startplaying')
(0.45, 'goal', 'MODIFIED')
(0.45, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.45, 'PROCEDURAL', 'RULE SELECTED: presskey')
(0.5, 'PROCEDURAL', 'RULE FIRED: presskey')
(0.5, 'goal', 'MODIFIED')
(0.5, 'manual', 'COMMAND: press_key')
(0.5, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.5, 'PROCEDURAL', 'RULE SELECTED: startplaying')
(0.55, 'PROCEDURAL', 'RULE FIRED: startplaying')
(0.55, 'goal', 'MODIFIED')
(0.55, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.55, 'PROCEDURAL', 'RULE SELECTED: presskey')
(0.6, 'PROCEDURAL', 'RULE FIRED: presskey')
(0.6, 'goal', 'MODIFIED')
(0.6, 'manual', 'COMMAND: press_key')
(0.6, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.6, 'PROCEDURAL', 'RULE SELECTED: startplaying')
(0.65, 'PROCEDURAL', 'RULE FIRED: startplaying')
(0.65, 'goal', 'MODIFIED')
(0.65, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.65, 'PROCEDURAL', 'RULE SELECTED: presskey')
(0.7, 'PROCEDURAL', 'RULE FIRED: presskey')
(0.7, 'goal', 'MODIFIED')
(0.7, 'manual', 'COMMAND: press_key')
(0.7, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.7, 'PROCEDURAL', 'RULE SELECTED: startplaying')
(0.75, 'PROCEDURAL', 'RULE FIRED: startplaying')
(0.75, 'goal', 'MODIFIED')
(0.75, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.75, 'PROCEDURAL', 'RULE SELECTED: presskey')
(0.8, 'PROCEDURAL', 'RULE FIRED: presskey')
(0.8, 'goal', 'MODIFIED')
(0.8, 'manual', 'COMMAND: press_key')
(0.8, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.8, 'PROCEDURAL', 'RULE SELECTED: startplaying')
(0.85, 'PROCEDURAL', 'RULE FIRED: startplaying')
(0.85, 'goal', 'MODIFIED')
(0.85, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.85, 'PROCEDURAL', 'RULE SELECTED: presskey')
(0.9, 'PROCEDURAL', 'RULE FIRED: presskey')
(0.9, 'goal', 'MODIFIED')
(0.9, 'manual', 'COMMAND: press_key')
(0.9, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.9, 'PROCEDURAL', 'RULE SELECTED: startplaying')
(0.95, 'PROCEDURAL', 'RULE FIRED: startplaying')
(0.95, 'goal', 'MODIFIED')
(0.95, 'PROCEDURAL', 'CONFLICT RESOLUTION')
```

```
(0.95, 'PROCEDURAL', 'RULE SELECTED: presskey')
```

```
/home/jakub/Documents/moje/computations and corpora/python/pyactr/pyactr/simulation.py:167: Use
  warnings.warn("Process in %s interupted" % name)
```

Ok, this is better but it is still not perfect. Instead of looping on the "presskey" rule, we created a two-rule loop: "startplaying" + "presskey". This might not be bad in its own right but it is a problem here. The two rules keep looping so quickly that the model never gets a chance to actually press the button (this is signaled in the warning again).

To avoid this issue, we have to modify our original rule "startplaying". We have to modify it, so that "startplaying" only fires if the manual buffer is not trying to press a key. The rule doing just that is given below. The crucial new bit is in the lines 6 and 7. These lines query on the status of the manual buffer. Querying is signaled as "?manual>", and the line "state free" checks in what state the buffer is - "state free" means that the buffer must not carry out any action, otherwise the rule cannot fire (apart from this, two other common queries are "state busy", which checks whether the buffer is busy, and "state error", which checks whether the action of the buffer ended up in an error - this last option is irrelevant in case of the manual buffer, though).

```
In [21]: playing_memory.productionstring(name="startplaying", string="""
         =goal>
         isa  playgame
         game memory
         activity None
         ?manual>
         state free
         ==>
         =goal>
         isa playgame
         activity presskey""")

Out[21]: {'?manual': {'state': 'free'}, '=goal': playgame(activity= None, game= memory)}
         ==>
         {'=goal': playgame(activity= presskey, game= )}

In [22]: goal.add(initial_chunk)

In [23]: simulation_game = playing_memory.simulation()
         simulation_game.run()
```

```
(0, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0, 'PROCEDURAL', 'RULE SELECTED: startplaying')
(0.05, 'PROCEDURAL', 'RULE FIRED: startplaying')
(0.05, 'goal', 'MODIFIED')
(0.05, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.05, 'PROCEDURAL', 'RULE SELECTED: presskey')
(0.1, 'PROCEDURAL', 'RULE FIRED: presskey')
(0.1, 'goal', 'MODIFIED')
```

```
(0.1, 'manual', 'COMMAND: press_key')
(0.1, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.1, 'PROCEDURAL', 'NO RULE FOUND')
(0.35, 'manual', 'PREPARATION COMPLETE')
(0.35, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.35, 'PROCEDURAL', 'NO RULE FOUND')
(0.4, 'manual', 'INITIATION COMPLETE')
(0.4, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.4, 'PROCEDURAL', 'NO RULE FOUND')
(0.5, 'manual', 'KEY PRESSED: 1')
(0.5, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.5, 'PROCEDURAL', 'NO RULE FOUND')
(0.65, 'manual', 'MOVEMENT FINISHED')
(0.65, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.65, 'PROCEDURAL', 'RULE SELECTED: startplaying')
(0.7, 'PROCEDURAL', 'RULE FIRED: startplaying')
(0.7, 'goal', 'MODIFIED')
(0.7, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.7, 'PROCEDURAL', 'RULE SELECTED: presskey')
(0.75, 'PROCEDURAL', 'RULE FIRED: presskey')
(0.75, 'goal', 'MODIFIED')
(0.75, 'manual', 'COMMAND: press_key')
(0.75, 'manual', 'PREPARATION COMPLETE')
(0.75, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.75, 'PROCEDURAL', 'NO RULE FOUND')
(0.8, 'manual', 'INITIATION COMPLETE')
(0.8, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.8, 'PROCEDURAL', 'NO RULE FOUND')
(0.9, 'manual', 'KEY PRESSED: 1')
(0.9, 'PROCEDURAL', 'CONFLICT RESOLUTION')
(0.9, 'PROCEDURAL', 'NO RULE FOUND')
```

Now we finally got somewhere. The trace shows that a key was pressed (at 0.5s). You can also see that pressing a key has various stages. First, the action is prepared (at 0.35s), then it is initiated (at 0.4s), and at 0.5s the key is pressed. After that, the movement is finished (it is assumed that the fingers of the model are at the home row of the keyboard, so that finishing the movement entails that the finger has moved back to its original position, a key in the home row). Details are not important here (the motor model simulating typing is taken in ACT-R from EPIC; check EPIC or ACT-R documentation for justification of these stages).

Notice that we still have a loop here. Unlike in the previous cases, though, the loop does not block key presses from taking place.

Ok, this does not look too bad, but it is not much realistic either. Our mind does not seem to be engaging in playing Memory, our mind is a maniac who keeps pressing the key 1 until the time runs out. This is not a game (unless you consider a repeated pressing of a "1" a game). How to make it look more like Memory? Read on.