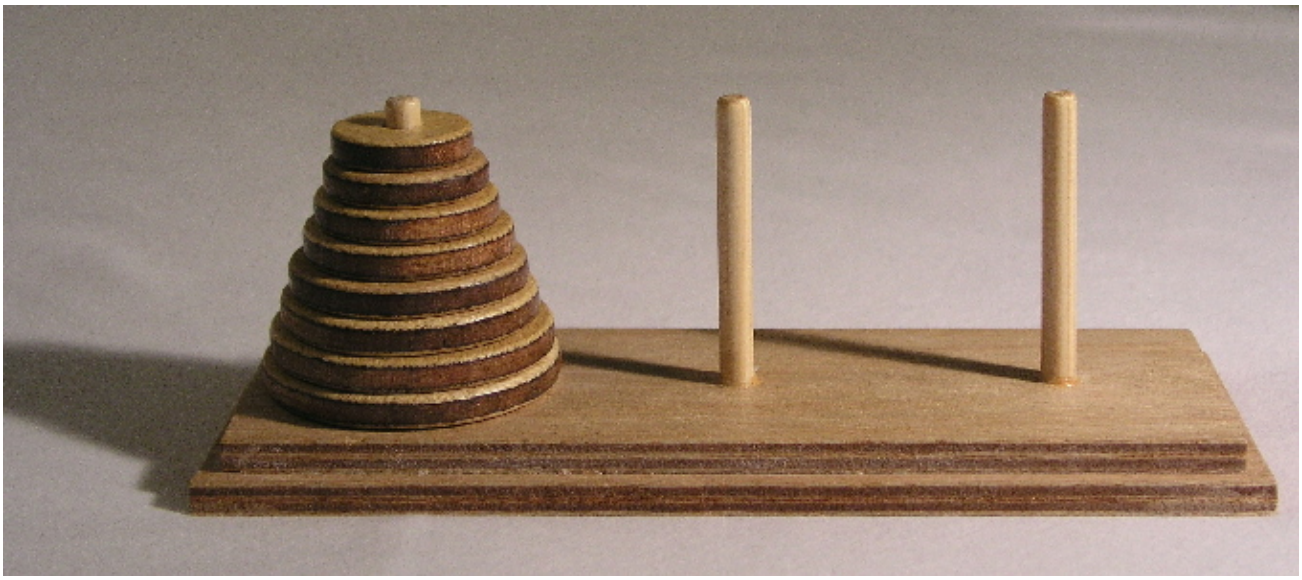
The tower of Hanoi

The tower of hanoi is a famous puzzle.



The game consists of three rods with disks stacked on top of them. The puzzle will start with all disks in a stack on one of the rods (like in the picture). The goal is to move all the discs to a single stack on the last rod.

To move the disks, you have to follow the following rules:

- You can move only one disk at a time.
- For each move, you have to take the upper disk from one of the stacks, and place it on top of another stack or empty rod.
- You cannot place a larger disk on top of a smaller disk.

This notebook will define a Julia implementation of the puzzle. It's up to you to write an algorithm that solves it.

Setting up the game pieces

Edit or run this notebook

What does a Julia implementation look like? We're not really interested in writing code that will manipulate physical disks. Our final goal is a function that will give us a *recipe* to solve the tower of hanoi, which is just a list of moves to make. Because of that, we can use a lot of abstraction in our implementation, and keep the data structures as simple as possible.

To start, we have to define some representation of the disks and the stacks. The disks have one important property, which is that they are ordered. We can use integers to represent them.

```
num_disks = 8
```

```
num_disks = 8
```

(Side note: the number of disks is arbitrary. When testing your function, you may want to set it to 1 or 2 to start.)

```
all_disks = 1:8
```

```
all_disks = 1:num_disks
```

A single stack can be represented as an array with all the disks in it. We will list them from top to bottom.

```
first_stack = [1, 2, 3, 4, 5, 6, 7, 8]
```

```
first_stack = collect(all_disks)
```

Now we have to make three of those.

```
starting_stacks = [[1, 2, 3, 4, 5, 6, 7, 8], [], []]
```

```
starting_stacks = [first_stack, [], []]
```

Defining the rules

Now that we have our "game board", we can implement the rules.

To start, we make two functions for states. A state of the game is just an array of stacks.

We will define a function that checks if a state is okay according to the rules. To be legal, all the stacks should be in the correct order, so no larger disks on top of smaller disks.

Another good thing to check: no disks should have appeared or disappeared since we started!

```
islegal (generic function with 1 method)
```

```
function islegal(stacks)
    order_correct = all(issorted, stacks)

    #check if we use the same disk set that we started with

    disks_in_state = sort([disk for stack in stacks for disk in stack])
    disks_complete = disks_in_state == all_disks

    order_correct && disks_complete
end
```


Another function for states: check if we are done! We can assume that we already checked if the state was legal. So we know that all the disks are there and they are ordered correctly. To check if we are finished, we just need to check if the last stack contains all the disks.

```
iscomplete (generic function with 1 method)
```

```
function iscomplete(stacks)
    last(stacks) == all_disks
end
```

Now the only rules left to implement are the rules for moving disks.

We could implement this as another check on states, but it's easier to just write a legal `move` function. Your solution will specify moves for the `move` function, so this will be the only way that the stacks are actually manipulated. That way, we are sure that nothing fishy is happening.

We will make our `move` function so that its input consists of a state of the game, and instructions for what to do. Its output will be the new state of the game.

So what should those instructions look like? It may seem intuitive to give a *disk* that should be moved, but that's more than we need. After all, we are only allowed to take the top disk from one stack, and move it to the top of another. So we only have to say which *stacks* we are moving between.

(Note that the `move` function is okay with moving a larger disk on top of a smaller disk. We already implemented that restriction in `islegal`.)

```
move (generic function with 1 method)
```

```
function move(stacks, source::Int, target::Int)
    #check if the from stack if not empty
    if isempty(stacks[source])
        error("Error: attempted to move disk from empty stack")
    end

    new_stacks = deepcopy(stacks)

    disk = popfirst!(new_stacks[source]) #take disk
    pushfirst!(new_stacks[target], disk) #put on new stack

    return new_stacks
end
```


Solving the problem

We have implemented the game pieces and the rules, so you can start working on your solution.

To do this, you can fill in the `solve(stacks)` function. This function should give a solution for the given `stacks`, by moving all the disks from stack 1 to stack 3.

As output, `solve` should give a recipe, that tells us what to do. This recipe should be an array of moves. Each moves is a `(source, target)` tuple, specifying from which stack to which stack you should move.

For example, it might look like this:

```
wrong_solution (generic function with 1 method)
```

```
function wrong_solution(stacks)::Array{Tuple{Int, Int}}
    return [(1,2), (2,3), (2,1), (1,3)]
end
```

Now you can work on building an actual solution. Some tips:

- `solve(stacks)` can keep track of the board if you want, but it doesn't have to.
- The section below will actually run your moves, which is very useful for checking them.
- If you want, you can change `num_disks` to 1 or 2. That can be a good starting point.

```
solve (generic function with 2 methods)
```

```
function solve(start = starting_stacks)::Array{Tuple{Int, Int}}

    #what to do?

    return []
end
```

Checking solutions

This is where we can check a solution. We start with a function that takes our recipe and runs it.

```
run_solution (generic function with 2 methods)
```

```
function run_solution(solver::Function, start = starting_stacks)
    moves = solver(deepcopy(start)) #apply the solver

    all_states = Vector{Any}(undef, length(moves) + 1)
    all_states[1] = start

    for (i, m) in enumerate(moves)
        try
            all_states[i + 1] = move(all_states[i], m[1], m[2])
        catch
            all_states[i + 1] = missing
        end
    end

    return all_states
end
```

You can use this function to see what your solution does.

If `run_solution` tries to make an impossible move, it will give `missing` from that point onwards. Look at what happens in the `wrong_solution` version and compare it to the moves in `wrong_solution`.

```
▸ [[[1, 2, 3, 4, 5, ⋯ more ,8], [], []], [[2, 3, 4, 5, 6, 7, 8], [1], []], [[2, 3, 4, 5,
```

```
run_solution(wrong_solution)
```

```
▸ [[[1, 2, 3, 4, 5, ⋯ more ,8], [], []]]
```

```
run_solution(solve)
```

Now that we have way to run a recipe, we can check if its output is correct. We will check if all the intermediate states are legal and the final state is the finished puzzle.

cł

```
fa
```

Th
s
fu
de
w
ye
K
w
or
it