homework 7, version 3

Homework 7: *Epidemic modeling*

`18.S191`, Fall 2023

This notebook contains *built-in, live answer checks*! In some exercises you will see a coloured box, which runs a test case on your code, and provides feedback based on the result. Simply edit the code, run it, and the check runs again.

Feel free to ask questions!

Initializing packages

When running this notebook for the first time, this could take up to 15 minutes. Hang in there!

```
using Plots, PlutoUI
```

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

```
function bernoulli(p::Number)
    rand() < p
end
```

Exercise 1: *Agent-based model for an epidemic outbreak – types*

In this and the following exercises we will develop a simple stochastic model for combined infection and recovery in a population, which may exhibit an **epidemic outbreak** (i.e. a large spike in the number of infectious people). The population is **well mixed**, i.e. everyone is in contact with everyone else. [An example of this would be a small school or university in which people are constantly moving around and interacting with each other.]

The model is an **individual-based** or **agent-based** model: we explicitly keep track of each individual, or **agent**, in the population and their infection status. For the moment we will not keep track of their position in space; we will just assume that there is some mechanism, not included in the model, by which they interact with other individuals.

Exercise 1.1

Each agent will have its own **internal state**, modelling its infection status, namely "susceptible", "infectious" or "recovered". We would like to code these as values `S`, `I` and `R`, respectively. One way to do this is using an **enumerated type** or **enum**. Variables of this type can take only a pre-defined set of values; the Julia syntax is as follows:

```
@enum InfectionStatus S I R
```

We have just defined a new type `InfectionStatus`, as well as names `S`, `I` and `R` that are the (only) possible values that a variable of this type can take.

👉 Define a variable `test_status` whose value is `S`.

```
test_status = S::InfectionStatus = 0
```

```
test_status = S
```

👉 Use the `typeof` function to find the type of `test_status`.

```
Enum InfectionStatus:
S = 0
I = 1
R = 2
```

```
typeof(test_status)
```

👉 Convert `x` to an integer using the `Integer` function. What value does it have? What values do `I` and `R` have?

```
(0, 1, 2)
```

```
Integer(S),Integer(I),Integer(R)
```

Exercise 1.2

For each agent we want to keep track of its infection status and the number of *other* agents that it infects during the simulation. A good solution for this is to define a *new type* `Agent` to hold all of the information for one agent, as follows:

```
Agent
```

```
begin
    mutable struct Agent
        status::InfectionStatus
        num_infected::Int64
    end
    function Agent()
        return Agent(S,0)
    end
end
```

When you define a new type like this, Julia automatically defines one or more **constructors**, which are methods of a generic function with the *same name* as the type. These are used to create objects of that type.

👉 Use the `methods` function to check how many constructors are pre-defined for the `Agent` type.

3 methods for type constructor:

- Main.var"workspace#21".Agent() in Main.var"workspace#21" at C:\Users\Dell\.julia\pluto_notebooks\hw7_098eb60f.jl#==#ae4ac4b4-041f-11eb-14f5-1bcde35d18f2:6
- Main.var"workspace#21".Agent(status::**Main.var"workspace#3".InfectionStatus**, num_infected::**Int64**) in Main.var"workspace#21" at C:\Users\Dell\.julia\pluto_notebooks\hw7_098eb60f.jl#==#ae4ac4b4-041f-11eb-14f5-1bcde35d18f2:3
- Main.var"workspace#21".Agent(status, num_infected) in Main.var"workspace#21" at C:\Users\Dell\.julia\pluto_notebooks\hw7_098eb60f.jl#==#ae4ac4b4-041f-11eb-14f5-1bcde35d18f2:3

```
1 methods(Agent)
```

👉 Create an agent `test_agent` with status `S` and `num_infected` equal to 0.

```
test_agent =   Agent(S::InfectionStatus = 0, 0)
```

```
1 test_agent = Agent(S,0)
```

👉 For convenience, define a new constructor (i.e. a new method for the function) that takes no arguments and creates an `Agent` with status `S` and number infected 0, by calling one of the default constructors that Julia creates. This new method lives *outside* (not inside) the definition of the `struct`. (It is called an **outer constructor**.)

(In Pluto, a struct definition and an outer constructor need to be combined in a single cell using a `begin end` block.)

Let's check that the new method works correctly. How many methods does the constructor have now?

```
Agent(S::InfectionStatus = 0, 0)
```

```
1 Agent()
```

Exercise 1.3

👉 Write functions `set_status!(a)` and `set_num_infected!(a)` which modify the respective fields of an `Agent`. Check that they work. [Note the bang (" ! ") at the end of the function names to signify that these functions *modify* their argument.]

```
set_status! (generic function with 1 method)
```

```
1 function set_status!(agent::Agent, new_status::InfectionStatus)
2     # your code here
3     agent.status=new_status
4 end
```

Got it!

Splendid!

👉 We will also need functions `is_susceptible` and `is_infected` that check if a given agent is in those respective states.

`is_susceptible (generic function with 1 method)`

```
function is_susceptible(agent::Agent)

    return agent.status==S ? true : false
end
```

`is_infected (generic function with 1 method)`

```
function is_infected(agent::Agent)

    return agent.status==I ? true : false
end
```

Got it!

Good job!

Exercise 1.4

👉 Write a function `generate_agents(N)` that returns a vector of `N` freshly created `Agent`s. They should all be initially susceptible, except one, chosen at random (i.e. uniformly), who is infectious.

`generate_agents (generic function with 1 method)`

```
function generate_agents(N::Integer)

    agent=Agent()
    vec_agent=fill(agent,N)
    for i in 1:N
        new_agents=Agent()
        vec_agent[i]=new_agents
    end
    infected_agent=rand(vec_agent)
    set_status!(infected_agent,I)
    return vec_agent
end
```

```
[Agent(S::InfectionStatus = 0, 0), Agent(S::InfectionStatus = 0, 0), Agent(I::InfectionStatu
```

```
generate_agents(3)
```

Got it!

Great! 🎉

We will also need types representing different infections.

Let's define an (immutable) `struct` called `InfectionRecovery` with parameters `p_infection` and `p_recovery`. We will make it a subtype of an abstract `AbstractInfection` type, because we will define more infection types later.

```
abstract type AbstractInfection end
```

```
struct InfectionRecovery <: AbstractInfection
    p_infection
    p_recovery
end
```

Exercise 1.5

👉 Write a function `interact!` that takes an affected `agent` of type `Agent`, an `source` of type `Agent` and an `infection` of type `InfectionRecovery`. It implements a single (one-sided) interaction between two agents:

- If the `agent` is susceptible and the `source` is infectious, then the `source` infects our `agent` with the given infection probability. If the `source` successfully infects the other agent, then its `num_infected` record must be updated.
- If the `agent` is infected then it recovers with the relevant probability.
- Otherwise, nothing happens.

```
interact! (generic function with 1 method)
```

```
function interact!(agent::Agent, source::Agent, infection::InfectionRecovery)

    # your code here
    if agent.status==S && source.status==I && rand()<infection.p_infection
        set_status!(agent,I)
        source.num_infected += 1
    elseif agent.status==I && rand()<infection.p_recovery
        set_status!(agent,R)
    end
end
```

Play around with the test case below to test your function! Try changing the definitions of `agent`, `source` and `infection`. Since we are working with randomness, you might want to run the cell multiple times.

```
(agent = Agent(I::InfectionStatus = 1, 0), source = Agent(I::InfectionStatus = 1, 1))
```

```
let
    agent = Agent(S, 0)
    source = Agent(I, 0)
    infection = InfectionRecovery(0.9, 0.5)

    interact!(agent, source, infection)

    (agent=agent, source=source)
end
```

Got it!

Your function treats the **susceptible** agent case correctly!

Got it!

Your function treats the **infectious** agent case correctly!

Got it!

Your function treats the **recovered** agent case correctly!

Exercise 2: *Agent-based model for an epidemic outbreak – Monte Carlo simulation*

In this exercise we will build on Exercise 2 to write a Monte Carlo simulation of how an infection propagates in a population.

Make sure to re-use the functions that we have already written, and introduce new ones if they are helpful! Short functions make it easier to understand what the function does and build up new functionality piece by piece.

You should not use any global variables inside the functions: Each function must accept as arguments all the information it requires to carry out its task. You need to think carefully about what the information each function requires.

Exercise 2.1

👉 Write a function `step!` that takes a vector of `Agent`s and an `infection` of type `InfectionRecovery`. It implements a single step of the infection dynamics as follows:

- Choose two random agents: an `agent` and a `source`.
- Apply `interact!(agent, source, infection)`.
- Return `agents`.

```
step! (generic function with 1 method)
```

```
function step!(agents::Vector{Agent}, infection::InfectionRecovery)
    vec_agents= rand(agents,2)
    interact!(vec_agents[1],vec_agents[2],infection)
    return agents

end
```

👉 Write a function `sweep!`. It runs `step!` N times, where N is the number of agents. Thus each agent acts, on average, once per sweep; a sweep is thus the unit of time in our Monte Carlo simulation.

```
sweep! (generic function with 1 method)
```

```
function sweep!(agents::Vector{Agent}, infection::AbstractInfection)
    for i in 1:length(agents)
        step!(agents,infection)
    end
end
```

👉 Write a function `simulation` that does the following:

1. Generate the N agents.
2. Run `sweep!` a number T of times. Calculate and store the total number of agents with each status at each step in variables `S_counts`, `I_counts` and `R_counts`.
3. Return the vectors `S_counts`, `I_counts` and `R_counts` in a **named tuple**, with keys `S`, `I` and `R`.

You've seen an example of named tuples before: the `student` variable at the top of the notebook!

Feel free to store the counts in a different way, as long as the return type is the same.

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

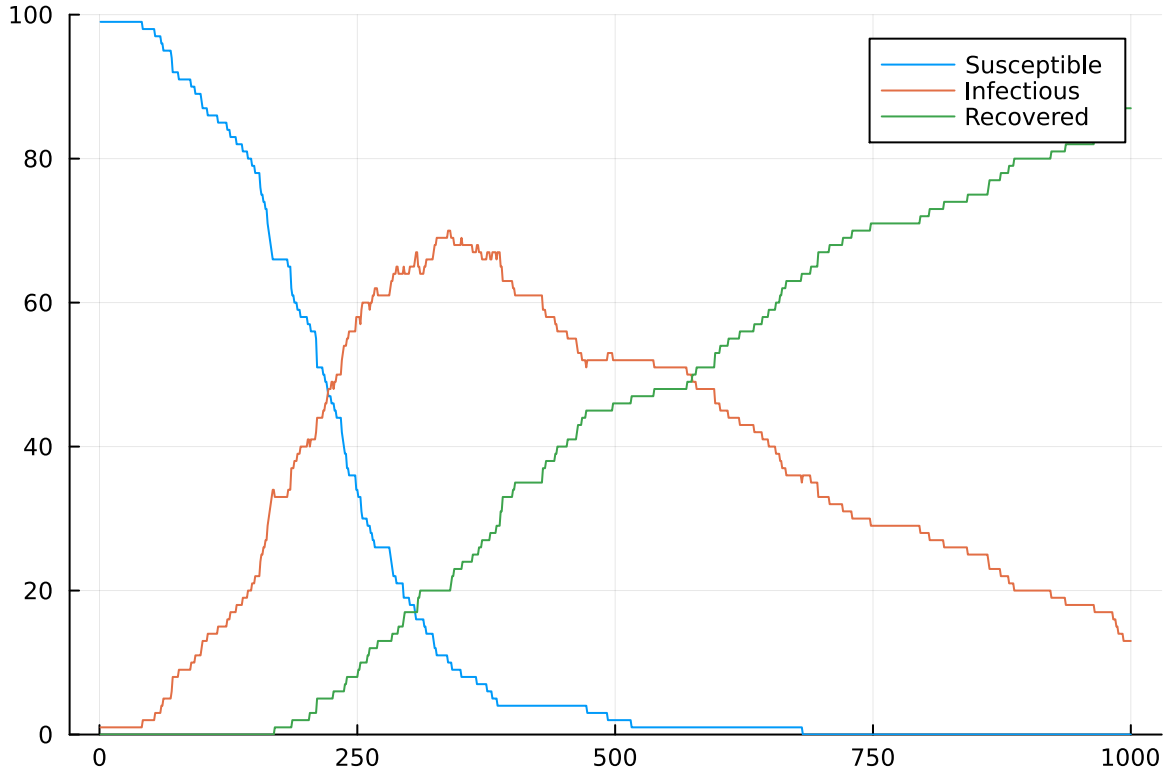
```
function simulation(N::Integer, T::Integer, infection::AbstractInfection)
    agents=generate_agents(N)
    S_counts=[]
    I_counts=[]
    R_counts=[]
    for i in 1:T
        sweep!(agents,infection)

        S_curr=sum([agent.status==S for agent in agents])
        I_curr=sum([agent.status==I for agent in agents])
        R_curr=sum([agent.status==R for agent in agents])

        push!(S_counts,S_curr)
        push!(I_counts,I_curr)
        push!(R_counts,R_curr)
    end
    return (S=S_counts, I=I_counts, R=R_counts)
end
```

```
(S = [2, 2, 1, 1, 0, 0, 0, 0, 0,    more ,0], I = [1, 1, 2, 1, 1, 1, 0, 0, 0,    more ,0], R = [0,
```

```
simulation(3, 20, InfectionRecovery(0.9, 0.2))
```

```
(S = [99, 99, 99, 99, 99, 99, 99, 99, 99,    more ,99], I = [0, 0, 0, 0, 0, 0, 0, 0, 0,    more ,
```

```
simulation(100, 1000, InfectionRecovery(0.005, 0.2))
```




```
let
    run_basic_sir

    N = 100
    T = 1000
    sim = simulation(N, T, InfectionRecovery(0.02, 0.002))

    result = plot(1:T, sim.S, ylim=(0, N), label="Susceptible")
    plot!(result, 1:T, sim.I, ylim=(0, N), label="Infectious")
    plot!(result, 1:T, sim.R, ylim=(0, N), label="Recovered")
end
```

Run simulation again!

Exercise 2.2

Alright! Every time that we run the simulation, we get slightly different results, because it is based on randomness. By running the simulation a number of times, you start to get an idea of the *mean behaviour* of our model. This is the essence of a Monte Carlo method! You use computer-generated randomness to generate samples.

Instead of pressing the button many times, let's have the computer repeat the simulation. In the next cells, we run your simulation `num_simulations=20` times with $N = 100$, $p_{\text{infection}} = 0.02$, $p_{\text{infection}} = 0.002$ and $T = 1000$.

Every single simulation returns a named tuple with the status counts, so the result of multiple simulations will be an array of those. Have a look inside the result, `simulations`, and make sure that its structure is clear.

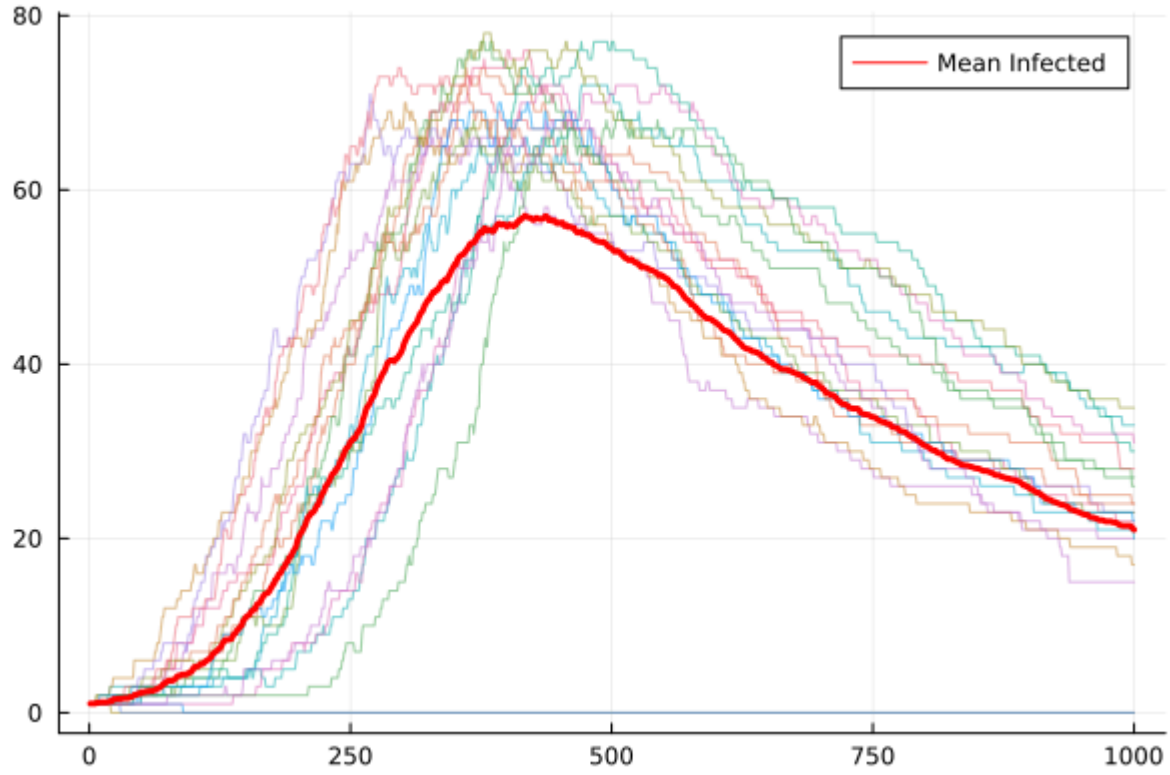
```
repeat_simulations (generic function with 1 method)
```

```
function repeat_simulations(N, T, infection, num_simulations)
    N = 100
    T = 1000

    map(1:num_simulations) do _
        simulation(N, T, infection)
    end
end
```

```
simulations =
 [(S = [99, 99, 99, 99, 99,    more ,98], I = [1, 1, 1, 1, 1,    more ,0], R = [0, 0, 0, 0, 0,
```

```
simulations = repeat_simulations(100, 1000, InfectionRecovery(0.02, 0.002), 20)
```

In the cell below, we plot the evolution of the number of I individuals as a function of time for each of the simulations on the same plot using transparency (`alpha=0.5` inside the plot command).




```
let
    p = plot()
    for sim in simulations
        plot!(p, 1:1000, sim.I, alpha=.5, label=nothing)
    end

    sum_at_T = ones(Float32, length(first(simulations).S))

    for sim in simulations
        for i in 1:length(sim.I)
            sum_at_T[i] += (sim.I[i])
        end
    end
    mean_at_T = sum_at_T ./ length(simulations)

    plot!(p, 1:length(mean_at_T), mean_at_T,
        lw=3,
        label="Mean Infected",
        color=:red
    )
    p
end
```

👉 Calculate the **mean number of infectious agents** of our simulations for each time step. Add it to the plot using a heavier line (`lw=3` for "linewidth") by modifying the cell above.

Check the answer yourself: does your curve follow the average trend?

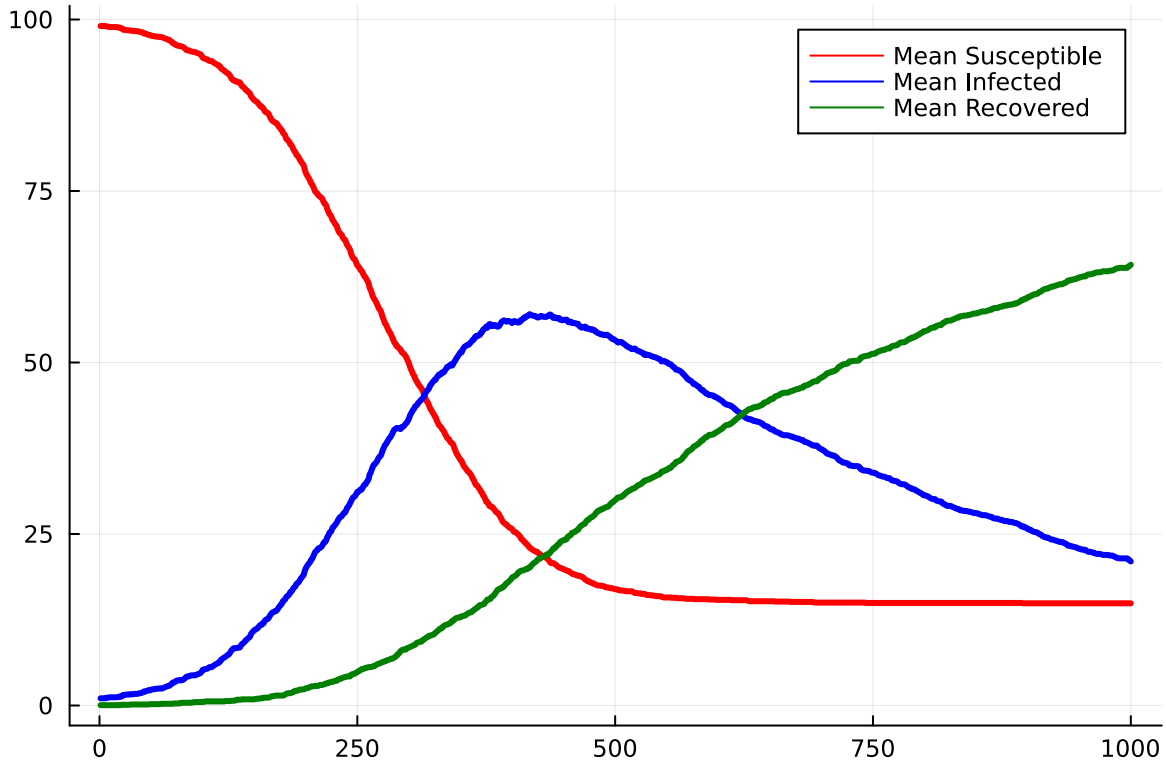
Hint

This exercise requires some creative juggling with arrays, anonymous functions, `map`s, or whatever you see fit!

👉 Write a function `sir_mean_plot` that returns a plot of the means of S, I and R as a function of time on a single graph.

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

```
function sir_mean_plot(simulations::Vector{<:NamedTuple})
    # you might need T for this function, here's a trick to get it:
    T = length(first(simulations).S)
    sum_at_T_S = ones(Float32, T)
    sum_at_T_I = ones(Float32, T)
    sum_at_T_R = ones(Float32,T)
    for sim in simulations
        for i in 1:length(sim.I)
            sum_at_T_S[i] += (sim.S[i])
            sum_at_T_I[i] += (sim.I[i])
            sum_at_T_R[i] += (sim.R[i])
        end
    end
    mean_at_T_S = sum_at_T_S ./ length(simulations)
    mean_at_T_I = sum_at_T_I ./ length(simulations)
    mean_at_T_R = sum_at_T_R ./ length(simulations)

    p=plot()

    plot!(p, 1:length(mean_at_T_S), mean_at_T_S,
        lw=3,
        label="Mean Susceptible",
        color=:red
    )
    plot!(p, 1:length(mean_at_T_I), mean_at_T_I,
        lw=3,
        label="Mean Infected",
        color=:blue
    )
    plot!(p, 1:length(mean_at_T_R), mean_at_T_R,
        lw=3,
        label="Mean Recovered",
        color=:green
    )
    p

    return p
end
```




```
sir_mean_plot(simulations)
```

```
(S = [0.99, 0.99, 0.99, 0.99, 0.99, 0.99, 0.989, 0.989, 0.989,    more ,0.1485], I = [0.01, 0.
```

```
let
    T = length(first(simulations).S)

    all_S_counts = map(result -> result.S, simulations)
    all_I_counts = map(result -> result.I, simulations)
    all_R_counts = map(result -> result.R, simulations)

    (S=round.(sum(all_S_counts) ./ length(simulations) ./ 100, digits=4),
    I=round.(sum(all_I_counts) ./ length(simulations) ./ 100, digits=4),
    R=round.(sum(all_R_counts) ./ length(simulations) ./ 100, digits=4))

end
```

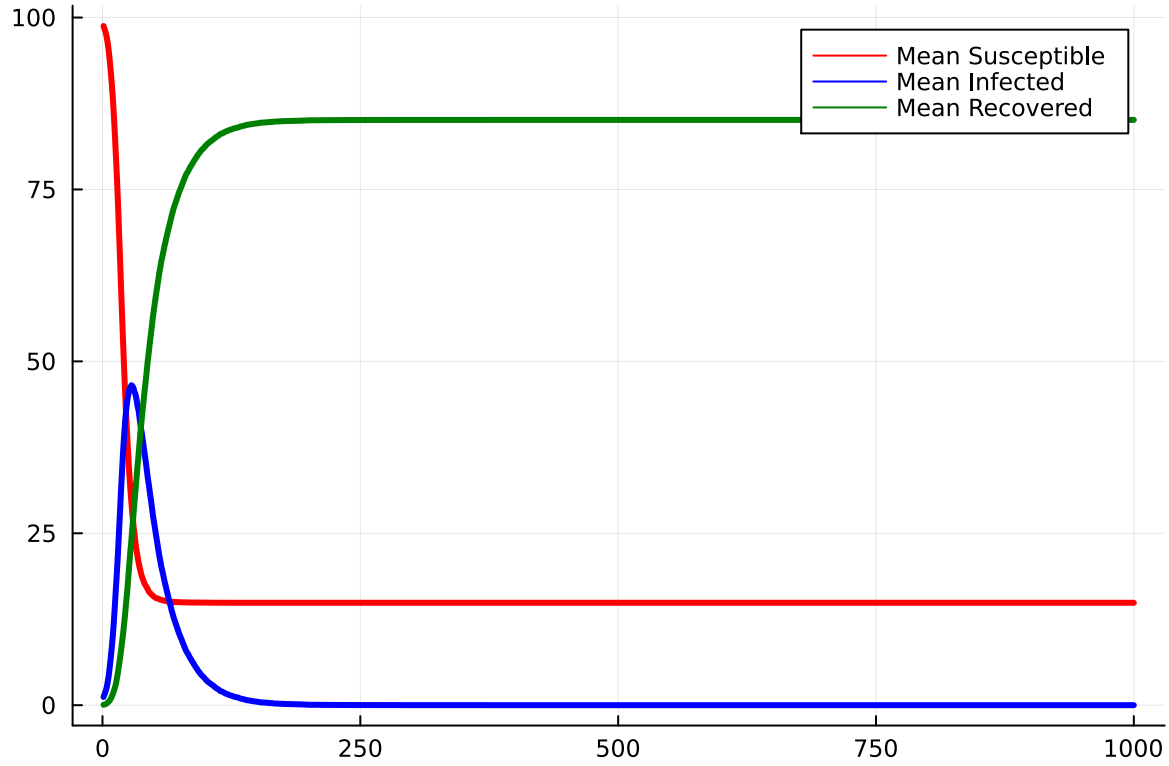
👉 Allow $p_{\text{infection}}$ and p_{recovery} to be changed interactively and find parameter values for which you observe an epidemic outbreak.

0.3

```
@bind p_inf Slider(0.0:0.01:1, default=0.02, show_value=true)
```

0.04

```
@bind p_rec Slider(0.0:0.01:1, default=0.02, show_value=true)
```




```
let
    new_simulations = repeat_simulations(100, 1000, InfectionRecovery(p_inf, p_rec), 200)
    sir_mean_plot(new_simulations)
end
```

```
Answer_1 = "If mean infected>50% -> epidemic"
```

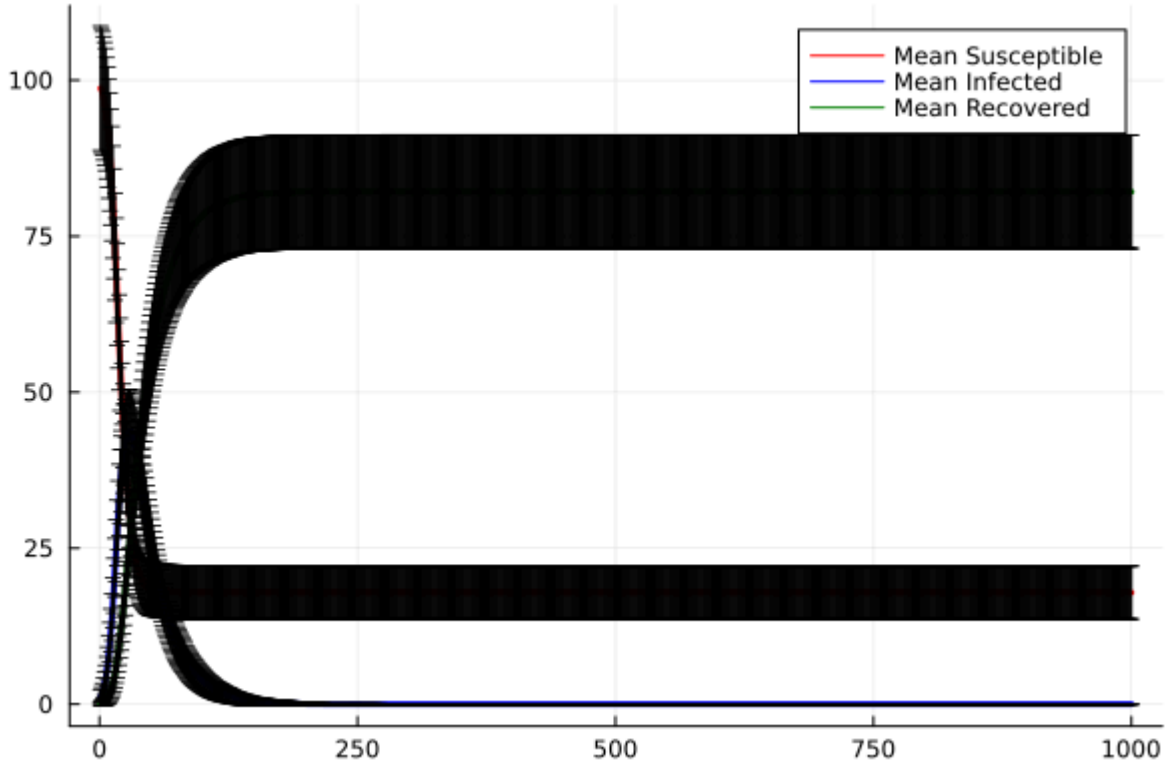
```
Answer_1="""If mean infected>50% -> epidemic"""
```

👉 Write a function `sir_mean_error_plot` that does the same as `sir_mean_plot`, which also computes the **standard deviation** σ of S, I, R at each step. Add this to the plot using **error bars**, using the option `yerr=σ` in the plot command; use transparency.

This should confirm that the distribution of I at each step is pretty wide!

sir_mean_error_plot (generic function with 1 method)

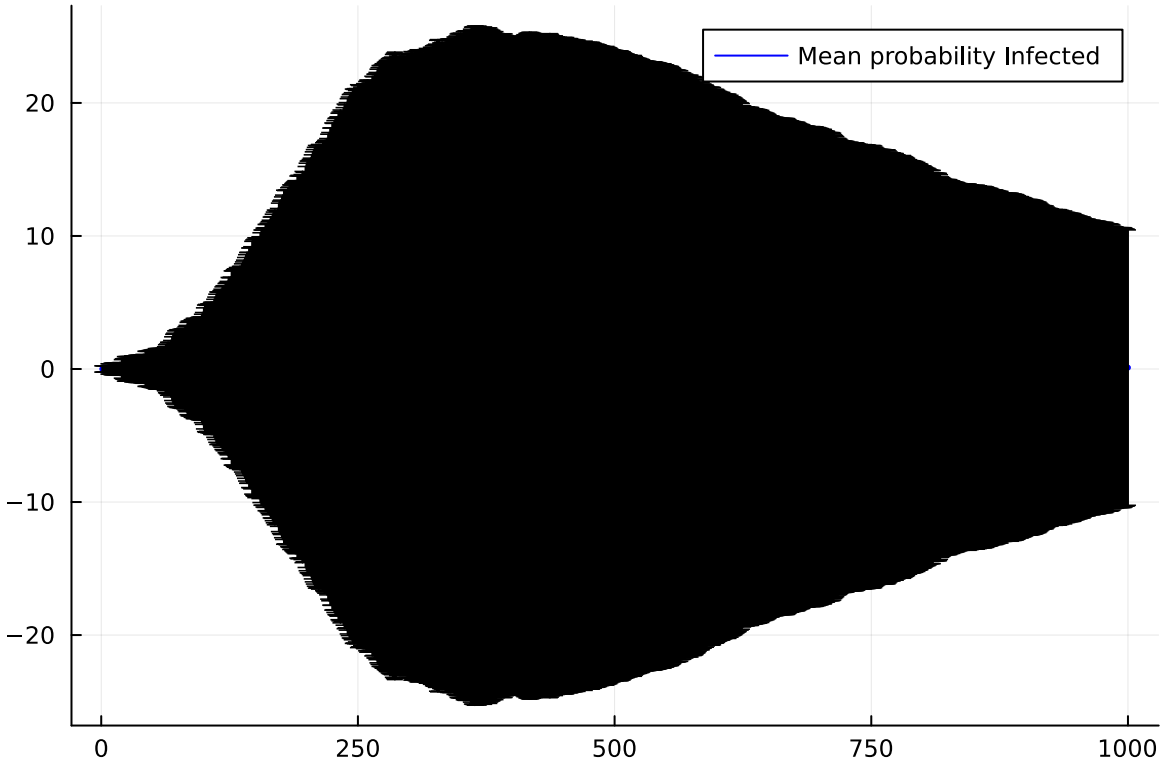
```
function sir_mean_error_plot(simulations::Vector{<:NamedTuple})
    # you might need T for this function, here's a trick to get it:
    T = length(first(simulations).S)
    sum_at_T_S = ones(Float32, T)
    sum_at_T_I = ones(Float32, T)
    sum_at_T_R = ones(Float32,T)
    std_at_T_S = ones(Float32,T)
    std_at_T_I = ones(Float32,T)
    std_at_T_R = ones(Float32,T)
    for sim in simulations
        for i in 1:length(sim.I)
            sum_at_T_S[i] += (sim.S[i])
            sum_at_T_I[i] += (sim.I[i])
            sum_at_T_R[i] += (sim.R[i])
        end
    end
    mean_at_T_S = sum_at_T_S ./ length(simulations)
    mean_at_T_I = sum_at_T_I ./ length(simulations)
    mean_at_T_R = sum_at_T_R ./ length(simulations)

    for sim in simulations
        for i in 1:length(sim.I)
            std_at_T_S[i] += (sim.S[i] - mean_at_T_S[i])^2
            std_at_T_I[i] += (sim.I[i] - mean_at_T_I[i])^2
            std_at_T_R[i] += (sim.R[i] - mean_at_T_R[i])^2
        end
    end
    std_at_T_S = sqrt.(sum_at_T_S ./ (length(simulations)-1))
    std_at_T_I = sqrt.(sum_at_T_I ./ (length(simulations)-1))
    std_at_T_R = sqrt.(sum_at_T_R ./ (length(simulations)-1))

    p = plot()

    plot!(p, 1:length(mean_at_T_S), mean_at_T_S,
        lw=3,
        label="Mean Susceptible",
        color=:red,
        yerr=std_at_T_S
    )
    plot!(p, 1:length(mean_at_T_I), mean_at_T_I,
        lw=3,
        label="Mean Infected",
        color=:blue,
        yerr=std_at_T_I
    )
    plot!(p, 1:length(mean_at_T_R), mean_at_T_R,
        lw=3,
        label="Mean Recovered",
        color=:green,
        yerr=std_at_T_R
    )

    return p
end
```




```
let
    new_simulations = repeat_simulations(100, 1000, InfectionRecovery(p_inf, p_rec), 200)
    sir_mean_error_plot(new_simulations)
end
```

Exercise 2.3

👉 Plot the probability distribution of `num_infected`. Does it have a recognisable shape? (Feel free to increase the number of agents in order to get better statistics.)




```
let
    N=100
    p= plot()
    new_simulations = repeat_simulations(N, 1000, InfectionRecovery(0.02, 0.002), 200)
    T = length(first(simulations).S)
    sum_at_T_I = ones(Float32, T)
    std_at_T_I = ones(Float32,T)
    for sim in simulations
        for i in 1:length(sim.I)
            sum_at_T_I[i] += (sim.I[i])
        end
    end
    mean_at_T_I = sum_at_T_I ./ length(simulations)
    for sim in simulations
        for i in 1:length(sim.I)
            std_at_T_I[i] += (sim.I[i] - mean_at_T_I[i])^2
        end
    end
    std_at_T_I = sqrt.(std_at_T_I ./ (length(simulations)-1))
    prob_at_T_I = std_at_T_I ./ N

    plot!(p, 1:length(mean_at_T_I), prob_at_T_I,
        lw=3,
        label="Mean probability Infected",
        color=:blue,
        yerr=std_at_T_I
    )
end

```

Exercse 2.4

👉 What are three *simple* ways in which you could characterise the magnitude (size) of the epidemic outbreak? Find approximate values of these quantities for one of the runs of your simulation.

```
Answer_2 =
"1.Percentage of max number of infected people at any given time=52%\n2.Percentage of peopl
```

```
Answer_2="""1.Percentage of max number of infected people at any given time=52%
2.Percentage of people that never got infected=13%
"""
```

Exercise 3: *Reinfection*

In this exercise we will *re-use* our simulation infrastructure to study the dynamics of a different type of infection: there is no immunity, and hence no "recovery" rather, susceptible individuals may now be **re-infected**

Exercise 3.1

👉 Make a new infection type `Reinfection`. This has the *same* two fields as `InfectionRecovery` (`p_infection` and `p_recovery`). However, "recovery" now means "becomes susceptible again", instead of "moves to the `R` class.

This new type `Reinfection` should also be a **subtype** of `AbstractInfection`. This allows us to reuse our previous functions, which are defined for the abstract supertype.

```
struct Reinfection <: AbstractInfection
    p_infection
    p_recovery
end
```

👉 Make a *new method* for the `interact!` function that accepts the new infection type as argument, reusing as much functionality as possible from the previous version.

```
interact! (generic function with 2 methods)
```

```
function interact!(agent::Agent, source::Agent, infection::Reinfection)
    if agent.status==S && source.status==I && rand()<infection.p_infection
        set_status!(agent,I)
        source.num_infected += 1
    elseif agent.status==I && rand()<infection.p_recovery
        set_status!(agent,S)
    end
end
```

```
step! (generic function with 2 methods)
```

```
function step!(agents::Vector{Agent}, infection::Reinfection)
    vec_agents= rand(agents,2)
    interact!(vec_agents[1],vec_agents[2],infection)
    return agents

end
```

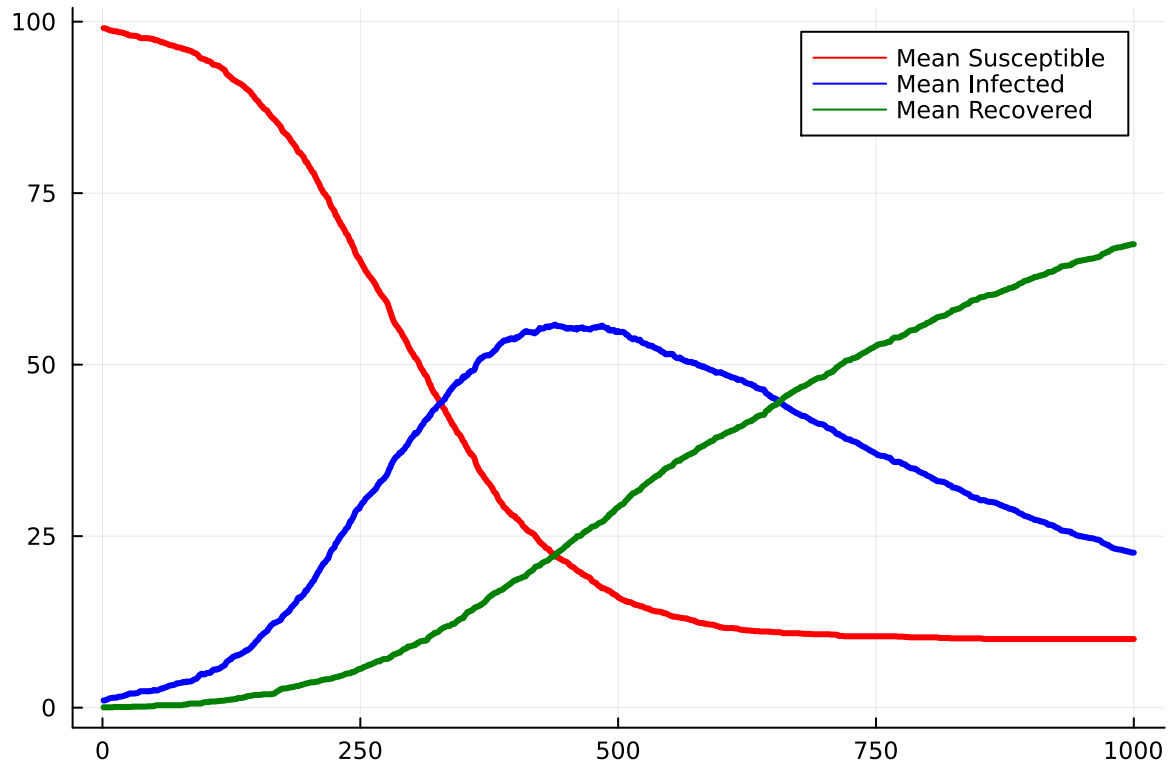
Exercise 3.2

👉 Run the simulation 20 times and plot I as a function of time for each one, together with the mean over the 20 simulations (as you did in the previous exercises).

Note that you should be able to re-use the `sweep!` and `simulation` functions , since those should be sufficiently **generic** to work with the new `step!` function! (Modify them if they are not.)

```
Last_simulations =
  [(S = [99, 99, 99, 99, 99,    more ,0], I = [1, 1, 1, 1, 1,    more ,19], R = [0, 0, 0, 0, 0,
```

```
1 Last_simulations = repeat_simulations(100, 1000, InfectionRecovery(0.02, 0.002), 20)
```




```
1 sir_mean_plot(Last_simulations)
```

👉 Run the new simulation and draw I (averaged over runs) as a function of time. Is the behaviour qualitatively the same or different? Describe what you see.

```
Answer_3 = "The graph of I is qualitatively identical."
```

```
1 Answer_3="""The graph of I is qualitatively identical."""
```

Function library

Just some helper functions used in the notebook.

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

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

```
still_missing (generic function with 2 methods)

keep_working (generic function with 2 methods)

yays =
  [Fantastic!, Splendid!, Great!, Yay ♥, Great! 🎉, Well done!, Keep it up!, Good job!, Awesome!,
```

```
correct (generic function with 2 methods)

not_defined (generic function with 1 method)
```