Ctrl + S

homework 6, version 5

Homework 6: *Probability distributions*

`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 PlutoUI, Plots
```

Exercise 1: *Calculating frequencies*

In this exercise we practise using dictionaries in Julia by writing our own version of the `countmap` function. Recall that that function counts the number of times that a given (discrete) value occurs in an input data set.

A suitable data structure for this is a **dictionary**, since it allows us to store data that is very sparse (i.e. for which many input values do not occur).

Exercise 1.1

👉 Write a function `counts` that accepts a vector `data` and calculates the number of times each value in `data` occurs.

The input will be an array of integers, **with duplicates**, and the result will be a dictionary that maps each occurred value to its count in the data.

For example,

```
counts([7, 8, 9, 7])
```

should give

```
Dict(
    7 => 2,
```

```
8 => 1,
9 => 1.
```

To do so, use a **dictionary** called `counts`. [We can create a local variable with the same name as the function.]

> **Hint**
>
> Do you remember how we worked with dictionaries in Homework 2? You can create an empty dictionary using `Dict()`. You may want to use either the function `haskey` or the function `get` on your dictionary – check the documentation for how to use these functions.

The function should return the dictionary.

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

```
function counts(data::Vector)
    counts = Dict{Int,Int}()

    # your code here
    for num in data
        if haskey(counts,num) == true
            counts[num] += 1
        else
            counts[num] = 1
        end
    end
    return counts
end
```

```
Dict(7 => 2, 9 => 1, 8 => 1)
```

```
counts([7, 8, 9, 7])
```

Test that your code is correct by applying it to obtain the counts of the data vector `test_data` defined below. What should the result be? Test that you do get the correct result and call the result `test_counts`.

```
test_data =  [1, 0, 1, 0, 1000, 1, 1, 1000]
```

```
test_data = [1, 0, 1, 0, 1000, 1, 1, 1000]
```

```
test_counts =  Dict(0 => 2, 1000 => 2, 1 => 4)
```

```
test_counts = counts(test_data)
```

> **Got it!**
>
> Well done!

Got it!

Let's move on to the next question.

Exercise 1.2

The dictionary contains the information as a sequence of **pairs** mapping keys to values. This is not a particularly useful form for us. Instead, we would prefer a vector of the keys and a vector of the values, sorted in order of the key.

We are going to make a new version `counts2` where you do the following (below). Start off by just running the following commands each in their own cell on the dictionary `test_counts` you got by running the previous `counts` function on the vector `test_data` so that you see the result of running each command. Once you have understood what's happening at *each* step, add them to the `counts2` function in a new cell.

👉 Extract vectors `ks` of keys and `vs` of values using the `keys()` and `values()` functions and convert the results into a vector using the `collect` function.

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

```
function keys(data::Dict)
    return collect(t[1] for t in data)
end
```

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

```
function values(data::Dict)

        return collect(t[2] for t in data)
end


```

👉 Define a variable `perm` as the result of running the function `sortperm` on the keys. This gives a **permutation** that tells you in which order you need to take the keys to give a sorted version.

```
perm =  [1, 3, 2]
```

```
perm=sortperm(keys(test_counts))
```

👉 Use indexing `ks[perm]` to find the sorted keys and values vectors.

[Here we are passing in a *vector* as the index. Julia extracts the values at the indices given in that vector]

```
[0, 1, 1000]
```

```
begin
    ks=keys(test_counts)
    ks.=ks[perm]
end
```

```
[2, 4, 2]
```

```
begin
    vs=values(test_counts)
    vs.=vs[perm]
end
```

Verify that your new `counts2` function gives the correct result for the vector `v` by comparing it to the true result (that you get by doing the counting by hand!)

👉 Create the function `counts2` that performs these steps.

```
counts2 (generic function with 1 method)
```

```
function counts2(data::Vector)
        ks=keys(counts(data))
        vs=values(counts(data))
        perm=sortperm(ks)
        ks.=ks[perm]
        vs.=vs[perm]
    return tuple(ks,vs)
end
```

```
([0, 1, 1000], [2, 4, 2])
```

```
counts2(test_data)
```

Got it!

You got the right answer!

Exercise 1.3

👉 Make a function `probability_distribution` that normalizes the result of `counts2` to calculate the relative frequencies of each value, i.e. to give a probability distribution (i.e. such that the sum of the resulting vector is 1).

The function should return the keys (the unique data that was in the original data set, as calculated in `counts2`, and the probabilities (relative frequencies).

Test that it gives the correct result for the vector `vv`.

We will use this function in the rest of the exercises.

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

```
function probability_distribution(data::Vector)
    cum_freq=sum((counts2(data))[2])
    prob_vec=(counts2(data))[2]./cum_freq
    return tuple(counts2(data)[1],prob_vec)
end
```

```
([0, 1, 1000], [0.25, 0.5, 0.25])
```

```
1 probability_distribution(test_data)
```

> **Got it!**
>
> Let's move on to the next question.

Intermezzo: *function* vs. *begin* vs. *let*

In our lecture materials, we sometimes use a `let` block in this cell to group multiple expressions together, but how is it different from `begin` or `function`?

function

Writing functions is a way to group multiple expressions (i.e. lines of code) together into a mini-program. Note the following about functions:

- A function always returns **one object**.[1] This object can be given explicitly by writing `return x`, or implicitly: Julia functions always return the result of the last expression by default. So `f(x) = x+2` is the same as `f(x) = return x+2`.
- Variables defined inside a function are *not accessible outside the function*. We say that function bodies have a **local scope**. This helps to keep your program easy to read and write: if you define a local variable, then you don't need to worry about it in the rest of the notebook.

There are two other ways to group expressions together that you might have seen before: `begin` and `let`.

begin

`begin` will group expressions together, and it takes the value of its last subexpression.

We use it in this notebook when we want multiple expressions to always run together.

let

`let` also groups multiple expressions together into one, but variables defined inside of it are **local**: they don't affect code outside of the block. So like `begin`, it is just a block of code, but like `function`, it has a local variable scope.

We use it when we want to define some local (temporary) variables to produce a complicated result, without interfering with other cells. Pluto allows only one definition per *global* variable of the same name, but you can define *local* variables with the same names whenever you wish!

[1]:

Even a function like

```
f(x) = return
```

returns **one object**: the object `nothing` — try it out!

Example of a scope problem with `begin`

The following will not work, because `fruits` has multiple definitions:

Error message

Multiple definitions for fruits.
Combine all definitions into a single reactive cell using a `begin ... end` block.

```
begin
    fruits = ["🍒", "🍐", "🍋"]
    length(fruits)
end
```

Error message

Multiple definitions for fruits.
Combine all definitions into a single reactive cell using a `begin ... end` block.

```
begin
    fruits = ["🍉"]
    length(fruits)
end
```

Solved using `let`

3

```
let
    vegetables = ["🥦", "🥔", "🥒"]
    length(vegetables)
end
```

1

```
let
    vegetables = ["🪚"]
    length(vegetables)
end
```

This works, because `vegetables` is only defined as a *local variable inside the cell*, not as a global:

Error message

```
UndefVarError: `vegetables` not defined in `Main.var"workspace#3"`
Suggestion: check for spelling errors or missing imports.
```

Show stack trace...

```
vegetables
```

Exercise 2: *Modelling component failure with the geometric distribution*

In this exercise, we will investigate the simple model of failure of mechanical components (or light bulbs, or radioactive decay, or recovery from an infection, or...) that we saw in lectures. Let's call τ the time to failure.

We will use a simple model, in which each component has probability p to fail each day. If it fails on day n, then $\tau = n$. We see that τ is a random variable, so we need to study its **probability distribution**.

Exercise 2.1

👉 Define the function `bernoulli(p)` from lectures. Recall that this generates `true` with probability p and `false` with probability $(1-p)$.

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

```
function bernoulli(p::Real)
    return rand() < p ? true : false
end
```

Exercise 2.2

👉 Write a function `geometric(p)`. This should run a simulation with probability p to recover and wait *until* the individual recovers, at which point it returns the time taken to recover. The resulting failure time is known as a **geometric random variable**, or a random variable whose distribution is the **geometric distribution**.

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

```
function geometric(p::Real)
    count=0
    while bernoulli(p)
            count +=1
    end
    return count
end
```

```
0
```

```
geometric(0.25)
```

Hint

[redacted]

We should always be aware of special cases (sometimes called "boundary conditions"). Make sure *not* to run the code with $p = 0$! What would happen in that case? Your code should check for this and throw an `ArgumentError` as follows:

```
throw(ArgumentError("..."))
```

with a suitable error message.

👉 What happens for $p = 1$?

interpretation_of_p_equals_one =
blablabla

Loop would go on forever

```
interpretation_of_p_equals_one = md"""
blablabla

Loop would go on forever
"""
```

Exercise 2.3

👉 Write a function `experiment(p, N)` that runs the `geometric` function `N` times and collects the results into a vector.

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

```
function experiment(p::Real, N::Integer)

    return [geometric(p) for i in 1:N ]
end
```

small_experiment = [1, 0, 0, 1, 2, 2, 1, 0, 0, 1, 0, 1, 1, 2, 1, 0, 0, 1, 1, 0]

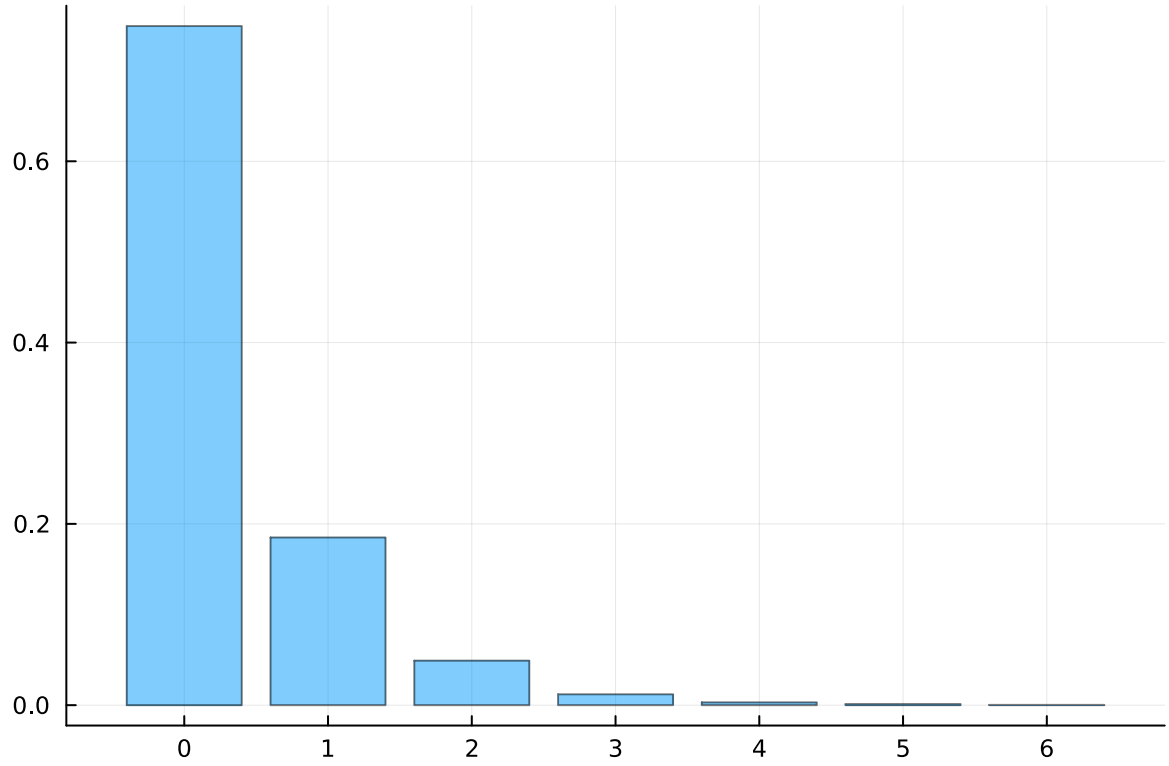
```
small_experiment = experiment(0.5, 20)
```

Exercise 2.4

Let's run an experiment with $p = 0.25$ and $N = 10,000$. We will plot the resulting probability distribution, i.e. plot $P(\tau = n)$ against n, where n is the recovery time.

large_experiment =
[0, 2, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 1, 0, more ,0, 0, 1, 1, 0, 1, 0, 0, 0, 1

```
large_experiment = experiment(0.25, 10000)
```




```
let
    xs, ps = probability_distribution(large_experiment)

    bar(xs, ps, alpha=0.5, leg=false)
end
```

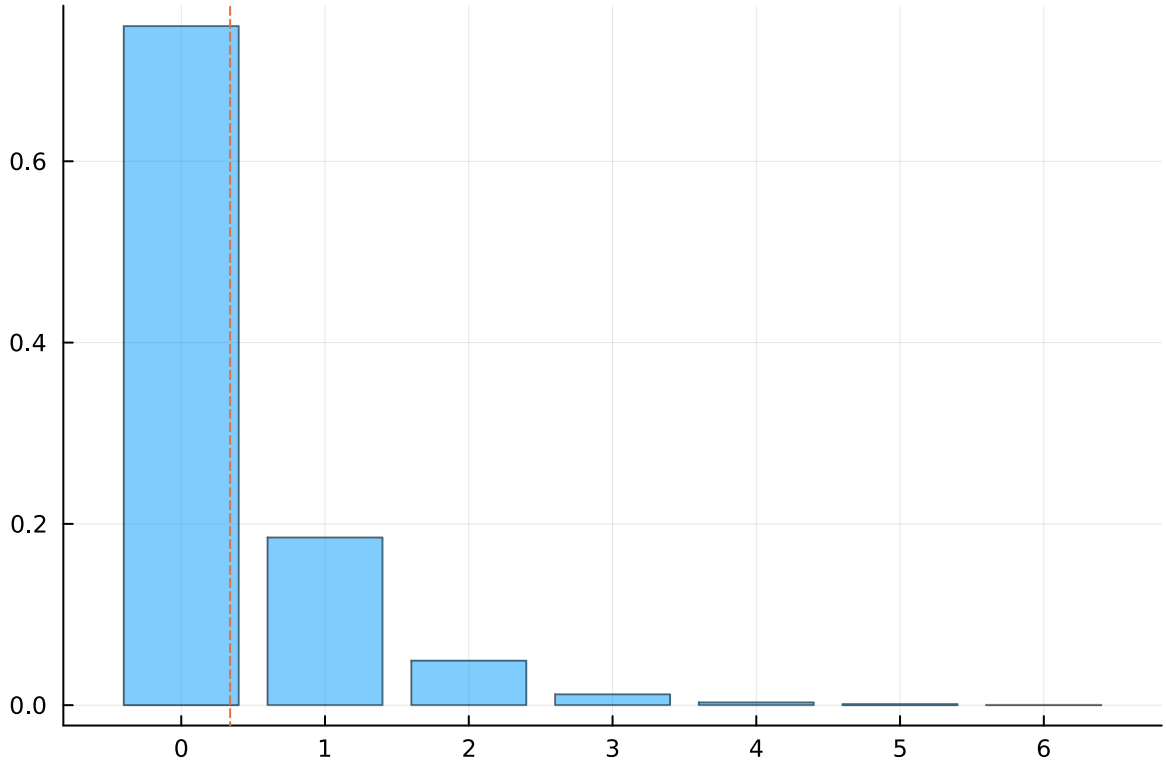
👉 Calculate the mean recovery time.

```
0.3399000000000001
```

```
begin
    xs,ps = probability_distribution(large_experiment)
    mean = sum([xs[i]*ps[i] for i in 1:size(xs)[1]])
end
```

👉 Create the same plot as above, and add the mean recovery time to the plot using the `vline!()` function and the `ls=:dash` argument to make a dashed line.

Note that `vline!` requires a *vector* of values where you wish to draw vertical lines.




```
let

    # your code here
    xs, ps = probability_distribution(large_experiment)

    bar(xs, ps, alpha=0.5, leg=false)
    vline!([mean], ls=:dash)

end
```

Note about plotting

Plots.jl has an interesting property: a plot is an object, not an action. Functions like `plot`, `bar`, `histogram` don't draw anything on your screen - they just return a `Plots.Plot`. This is a struct that contains the *description* of a plot (what data should be plotted in what way?), not the *picture*.

So a Pluto cell with a single line, `plot(1:10)`, will show a plot, because the *result* of the function `plot` is a `Plot` object, and Pluto just shows the result of a cell.

Modifying plots

Nice plots are often formed by overlaying multiple plots. In Plots.jl, this is done using the **modifying functions**: `plot!`, `bar!`, `vline!`, etc. These take an extra (first) argument: a previous plot to modify.

For example, to plot the `sin`, `cos` and `tan` functions in the same view, we do:

```
function sin_cos_plot()
    T = -1.0:0.01:1.0

    result = plot(T, sin.(T))
```

```
    plot!(result, T, cos.(T))
    plot!(result, T, tan.(T))

    return result
```

💡 This example demonstrates a useful pattern to combine plots:

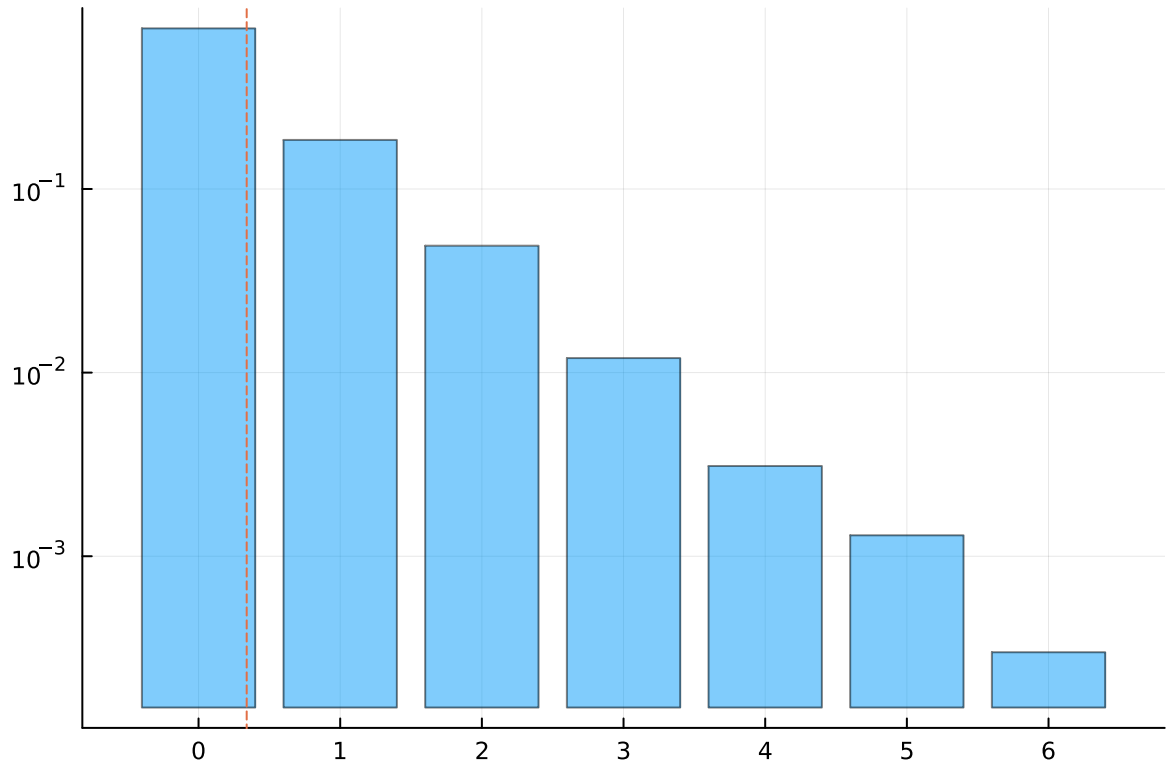
1. Create a **new** plot and store it in a variable
2. **Modify** that plot to add more elements
3. Return the plot

Grouping expressions

It is highly recommended that these 3 steps happen **within a single cell**. This can prevent some strange glitches when re-running cells. There are three ways to group expressions together into a single cell: `begin`, `let` and `function`. More on this here!

Exercise 2.5

👉 What shape does the distribution seem to have? Can you verify that by using one or more log scales in a new plot?




```
let

    # your code here
    xs, ps = probability_distribution(large_experiment)

    bar(xs, ps, alpha=0.5, leg=false,yaxis=:log10)
    vline!([mean], ls=:dash)

end
```

Use the widgets from PlutoUI to write an interactive visualization that performs Exercise 2.3 for p varying between 0 and 1 and N between 0 and $100,000$.

You might want to go back to Exercise 2.3 to turn your code into a *function* that can be called again.

As you vary p, what do you observe? Does that make sense?

```
@bind hello Slider( 2 : 0.5 : 10 )
```

5.5

```
hello
```

Exercise 2.6

👉 For fixed $N = 10,000$, write a function that calculates the *mean* time to recover, $\langle \tau(p) \rangle$, as a function of p.

τ (generic function with 1 method)

```
function τ(p)
    large=experiment(p,10000)
    as,probs=probability_distribution(large)
    mean_τ = sum([as[i]*probs[i] for i in 1:size(as)[1]])
    return mean_τ
end
```

👉 Use plots of your function to find the relationship between $\langle \tau(p) \rangle$ and p.

x_vec = 0.0:0.1:1.0

```
x_vec= 0:0.1:1
```

Error message

> **InterruptException:**

```
bar(x_vec,τ.(x_vec),alpha=0.5)
```

WARNING: Force throwing a SIGINT

Based on my observations, it looks like we have the following relationship:

$$\langle \tau(p) \rangle = my \cdot answer \cdot here$$

```
md"""
Based on my observations, it looks like we have the following relationship:

```math
\langle \tau(p) \rangle = my \cdot answer \cdot here
```
"""
```

Exercise 3: *More efficient geometric distributions*

Let's use the notation $P_n := \mathbb{P}(\tau = n)$ for the probability to fail on the nth step.

Probability theory tells us that in the limit of an infinite number of trials, we have the following exact results: $P_1 = p$; $P_2 = p(1-p)$, and in general $P_n = p(1-p)^{n-1}$.

Exercise 3.1

👉 Fix $p = 0.25$. Make a vector of the values P_n for $n = 1, \ldots, 50$. You must (of course!) use a loop or similar construction; do *not* do this by hand!

```
Ps =
  [0.25, 0.1875, 0.140625, 0.105469, 0.0791016, 0.0593262, 0.0444946, 0.033371, 0.0250282, 0.0
```

```
Ps = let

    # your code here
    P=0.25
    [P*(1-P)^i for i in 0:50]
end
```

👉 Do they sum to 1?

```
0.9999995752587576
```

```
sum(Ps)
```

Exercise 3.2

👉 Check analytically that the probabilities sum to 1 when you include *all* (infinitely many) of them.

$$\sum_{k=1}^{\infty} P_k = \dots your \cdot answer \cdot here \cdots = 1$$

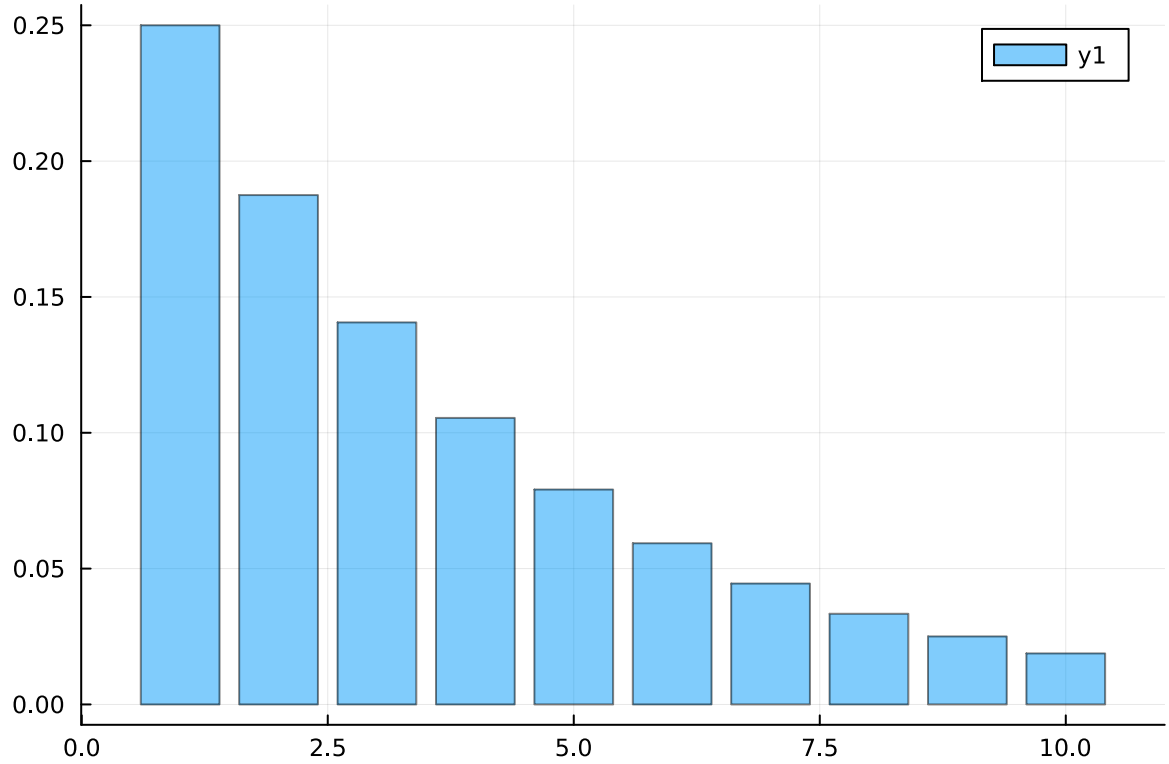
```
md"""

```math
\sum_{k=1}^{\infty} P_k = \dots your \cdot answer \cdot here \dots = 1

```
"""
```

Exercise 3.3: Sum of a geometric series

👉 Plot P_n as a function of n. Compare it to the corresponding result from the previous exercise (i.e. plot them both on the same graph).




```
begin
    P=0.25
    N = 1:1:10
    prs = [P*(1-P)^(i-1) for i in N]

    bar(N,prs, alpha=0.5)
end
```

👉 How could we measure the *error*, i.e. the distance between the two graphs? What do you think determines it?

```
Enter cell code...
```

Exercise 3.4

If p is *small*, say $p = 0.001$, then the algorithm we used in Exercise 2 to sample from geometric distribution will be very slow, since it just sits there calculating a lot of `false`s! (The average amount of time taken is what you found in [1.8].)

Let's make a better algorithm. Think of each probability P_n as a "bin", or interval, of length P_n. If we lay those bins next to each other starting from P_1 on the left, then P_2, etc., there will be an *infinite* number of bins that fill up the interval between 0 and 1. (In principle there is no upper limit on how many days it will take to recover, although the probability becomes *very* small.)

Now suppose we take a uniform random number r between 0 and 1. That will fall into one of the bins. If it falls into the bin corresponding to P_n, then we return n as the recovery time!

👉 To draw this picture, we need to add up the lengths of the lines from 1 to n for each n, i.e. calculate the **cumulative sum**. Write a function `cumulative_sum`, which returns a new vector.

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

```
function cumulative_sum(xs::Vector)

    return cumsum(xs)
end
```

```
[1, 4, 9, 16, 25]
```

```
cumulative_sum([1, 3, 5, 7, 9])
```

Got it!

Keep it up!

```
if !@isdefined(cumulative_sum)
    not_defined(:cumulative_sum)
else
    let
        result = cumulative_sum([1,2,3,4])
        if result isa Missing
            still_missing()
        elseif !(result isa AbstractVector)
            keep_working(md"Make sure that you return an Array: the cumulative sum!")
        elseif length(result) != 4
            keep_working(md"You should return an array of the same size a `xs`.")
        else
            if isapprox(result, [1, 3, 6, 10])
                correct()
            else
                keep_working()
            end
        end
    end
end
```

👉 Plot the resulting values on a horizontal line. Generate a few random points and plot those. Convince yourself that the probability that a point hits a bin is equal to the length of that bin.

```
cumulative =
```

```
 [0.25, 0.4375, 0.578125, 0.683594, 0.762695, 0.822021, 0.866516, 0.899887, 0.924915, 0.94368
```

```
cumulative = cumulative_sum(Ps)
```

Exercise 3.5

👉 Calculate the sum of P_1 up to P_n analytically.

$$1 - (1 - P)^(n - 1)$$

```
md"""
```math
1-(1-P)^(n-1)
```
"""
```

👉 Use the previous result to find (analytically) which bin n a given value of $r \in [0, 1]$ falls into, using the inequality $P_{n+1} \leq r \leq P_n$.

$$n(r, p) = my \cdot answer \cdot here$$

```
md"""
```math
n(r,p) = p(1-p)^r
```
"""
```

Exercise 3.6

👉 Implement this as a function `geomtric_bin(r, p)`, use the `floor` function.

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

```
function geometric_bin(u::Real, p::Real)

    return return floor(Int, log(u) / log(1 - p)) + 1
end
```

Got it!

Keep it up!

We can use this to define a **fast** version of the `geomtric` function:

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

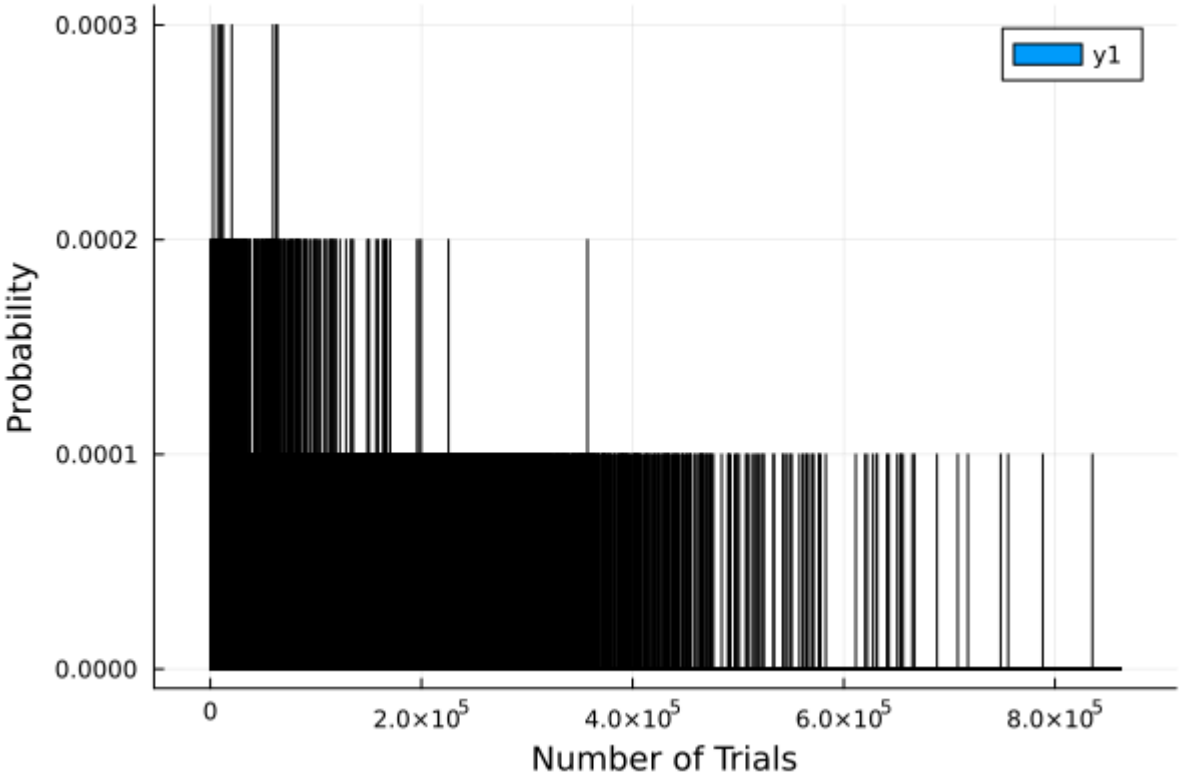
```
geometric_fast(p) = geometric_bin(rand(), p)
```

11

```
geometric_fast(0.25)
```

Exercise 3.7

👉 Generate `10_000` samples from `geometric_fast` with $p = 10^{-10}$ and plot a histogram of them. This would have taken a very long time with the previous method!




```
begin
    p = 10^-5
    samples_fast = [geometric_bin(rand(), p) for i in 1:10_000]

    # 3. Create a histogram of the generated samples.
    histogram(samples_fast,
        bins=1:maximum(samples_fast),
        normalize=:pdf,
        xlabel="Number of Trials",
        ylabel="Probability"
    )
end
```

```
Enter cell code...
```

Exercise 4: *Distribution of the "atmosphere"*

In this question we will implement a (very!) simple model of the density of the atmosphere, using a **random walk**, i.e. a particle that undergoes random *motion*. (We will see more about random walks in lectures.)

We can think of a very light dust particle being moved around by the wind. We are only interested in its vertical position, y, and we will suppose for simplicity that y is an integer. The particle jumps up and down randomly as it is moved by the wind; however, due to gravity it has a higher probability p of moving downwards (to $y-1$) than upwards (to $y+1$), which happens with probability $1-p$.

At $y=1$ there is a **boundary condition**: it hits a reflective boundary (the "surface of the Earth"). We can model this using "bounce-back": if the particle tries to jump downwards from $y=1$ then it hits a reflective boundary and bounces back to $y=1$. (In other words it remains in the same place.)

Exercise 4.1

👉 Write a simulation of this model in a function `atmosphere` that accepts `p`, the initial height `y0`, and the number of steps N as variables.

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

```
function atmosphere(p::Real, y0::Real, N::Integer)
    heights=ones(N+1)
    heights[1]=y0
    for i in 2:N+1
        if rand()<p
            y0 -= 1
        else
            y0 += 1
        end
        heights[i]=y0
    end
    return heights
end
```

```
[10.0, 9.0, 8.0, 7.0, 6.0, 5.0, 4.0, 3.0, 2.0, 3.0, 2.0, 1.0, 0.0, -1.0, -2.0, -1.0, -2.0, -3.
```

```
atmosphere(0.8, 10, 50)
```

Let's simulate it for 10^7 time steps with $x_0 = 10$ and $p = 0.55$.

Exercise 4.2

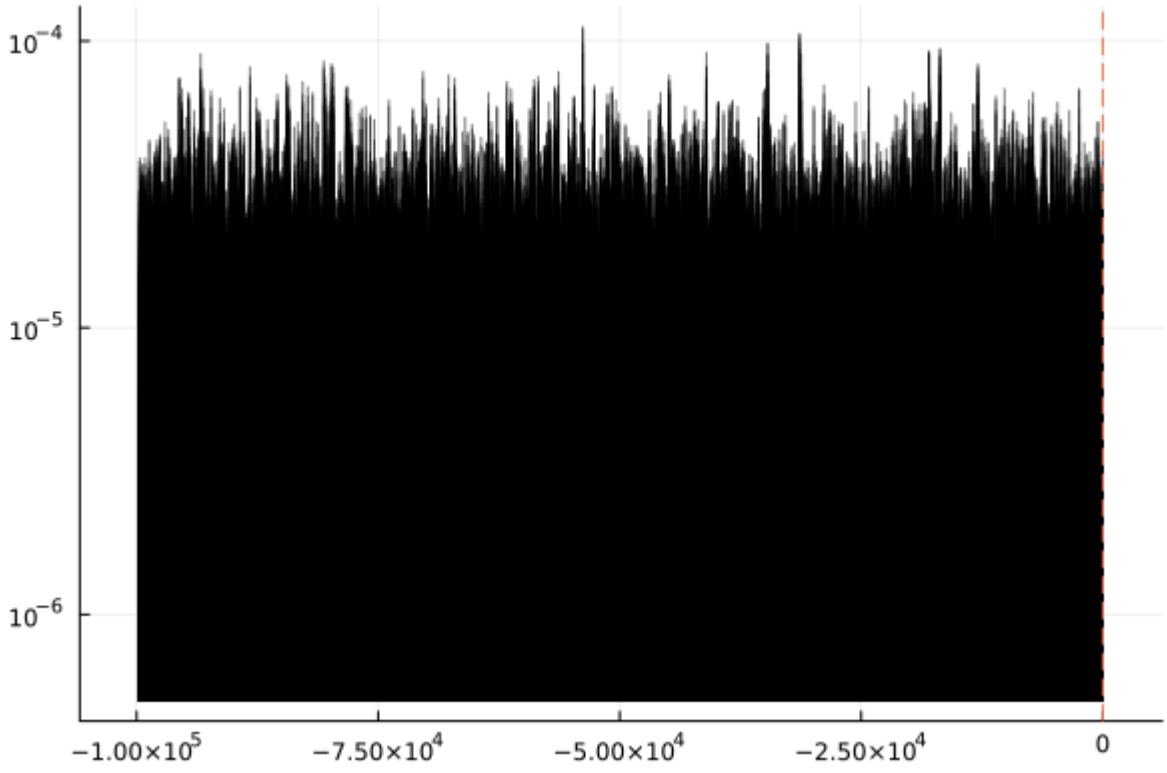
👉 Calculate and plot the probability distribution of the walker's height.

```
new_atm =
 [10.0, 9.0, 8.0, 9.0, 8.0, 9.0, 10.0, 9.0, 10.0, 9.0, 8.0, 7.0, 6.0, 7.0, 8.0, 9.0, 8.0, 7.0,
```

```
new_atm=atmosphere(0.55,10,1000000)
```

```
([-99720, -99719, -99718, -99717, -99716, -99715, -99714, -99713, -99712,    more ,10], [9.99
```

```
probability_distribution(new_atm)
```




```
let
    xs, ps = probability_distribution(new_atm)

    bar(xs, ps, alpha=0.5, leg=false,yaxis=:log10)
    vline!([mean], ls=:dash)

end
```

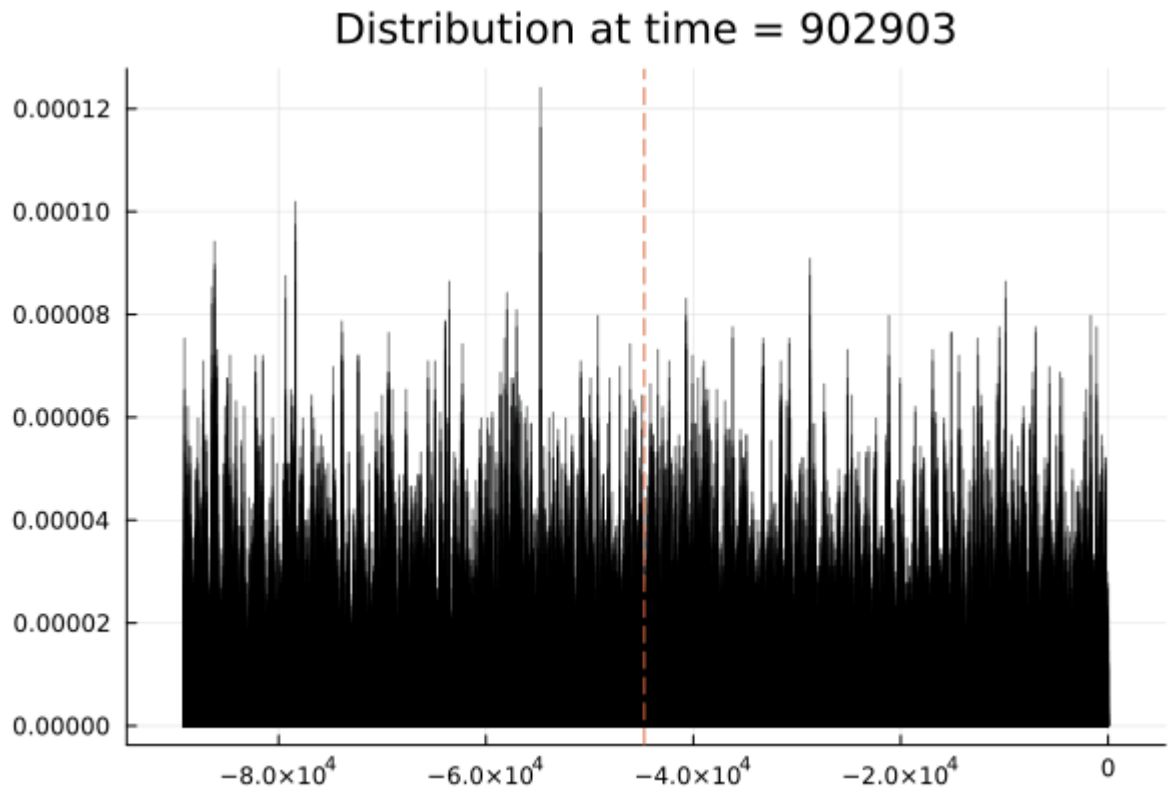
👉 What does the resulting figure look like? What form do you think this distribution has? Verify your hypothesis by plotting the distribution using different scales. You can increase the number of time steps to make the results look nicer.

```
"exponetial distribution"
```

Exercise 4.3

👉 Make an interactive visualization of how the distribution develops over time. What happens for longer and longer times?

```
@bind time_step Slider(1:1:1000000)
```




```
let
    my_atm = atmosphere(0.55, 10, time_step)
    ms, ls = probability_distribution(my_atm)
    mean_val = sum(ms .* ls)
    bar(ms, ls, alpha=0.5, leg=false, title="Distribution at time = $(time_step)")
    vline!([mean_val], ls=:dash)
end
```

👉 Use wikipedia to find a formula for the barometric pressure at a given altitude. Does this result match your expectations?

```
"The result is exponential as expected"
```

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)

todo (generic function with 1 method)
```