Ctrl + S

homework 5, version 3

Homework 5: *Structure*

`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!

Error message from Pkg

```
Format of manifest file at `C:\Users\Dell\AppData\Local\Temp\jl_vmF7hj\Manifest.toml`
already up to date: manifest_format == 2.0.0
```

Show stack trace...

```julia
import Pkg; Pkg.upgrade_manifest()
```

```julia
using PlutoUI
```

Exercise 1: *When is zero not quite zero?*

Students used to pure math are sometimes surprised to see numbers like `1e-15`, `1e-16`, or `1e-17` appearing in computations when `0` might have been expected. At first glance, this behaviour seems 'random' or 'noisy', but in this exercise, we will try to demonstrate some structure behind floating point artihmetic.

Exercise 1.1

👉 Find all integers j with $1 \leq j \leq 1000$ for which Julia's result satisfies `j*(1/j) ≠ 1`.

```julia
v=[j for j in 1:1000 if j*(1/j)!=1]
```

```
[49, 98, 103, 107, 161, 187, 196, 197, 206, 214, 237, 239, 249, 253, 322, 347, 374, 389, 392, 3
```

```julia
begin
    function f(j)
        if j*(1/j) != 1
            return true
        else
            return false
        end
    end
    p=[i for i in 1:1000]
    filter(f,p)
end
```

Hint

Try writing this without a `for` loop. Ideas:

👉 Use `filter`.

👉 Comprehensions can have `if` clauses, as in `[j for j=1:1000 if 3 < j < 17]`.

Notice that when you re-run the computation, the result does not change. Floating-point arithmetic is not random!

Exercise 1.2

👉 Take the smallest number `j` you found above and compute the error, i.e. the distance between `j*(1/j)` and `1`.

Is this an integer power of 2? Which one? (`log2` might help.)

```
0.02040816326530612
```

```julia
1/49
```

```
0.9999999999999999
```

```julia
0.02040816326530612*49
```

```
1.1102230246251565e-16
```

```julia
abs(0.9999999999999999-1)
```

```
-53.0
```

```julia
log2(1.1102230246251565e-16)
```

Exercise 1.3

Caculate all of the following:

- 32 * 23 - 736
- 3.2 * 23 - 73.6
- 3.2 * 2.3 - 7.36

```
0
```

```julia
32*23 - 736
```

```
1.4210854715202004e-14
```

```julia
3.2 * 23 -73.6
```

```
-8.881784197001252e-16
```

```julia
3.2*2.3 - 7.36
```

```
2.220446049250313e-16
```

```julia
eps()
```

We wanted to show you that floating point arithmetic has structure to it. It is not the fuzz or slop that you may have seen as experimental errors in maybe a chemistry or physics class. If you are interested, we may show more later in the semester.

Exercise 2: *Rank-one matrices*

In this exercise we will go into some more detail about how to define types in Julia in order to make a structured-matrix type, similar to the `OneHotVector` type from class.

To begin, we will make a type to represent a rank-1 matrix. A *rank-1 matrix* is a matrix whose columns are all multiples of each other. This already tells us a lot about the matrix; in fact, we can represent any such matrix as the outer product of two vectors `v` and `w`. *Only* the two vectors will be stored; the matrix elements will be *calculated* on demand, i.e. when we *index* into the object.

Exercise 2.1

Let's make a `FirstRankOneMatrix` type that contains two vectors of floats, `v` and `w`. Here `v` represents a column and `w` the multipliers for each column.

We include (in the same cell, due to requirements of Pluto) a constructor that takes a single vector `v` and duplicates it.

```
FirstRankOneMatrix
```

```julia
begin
    struct FirstRankOneMatrix
        # Your code here
        v::Vector{Float64}
        w::Vector{Float64}
    end
    # Add the extra constructor here
    FirstRankOneMatrix(v::Vector{Float64}) = FirstRankOneMatrix(v, v)

end
```

Hint

The extra constructor should have the same name as the type. It should take in a single argument v, and use it for both the column and the multipliers.

👉 Create an object of type `FirstRankOneMatrix` representing the multiplication table of the numbers from 1 to 10 and the numbers from 1 to 12. Call it `ten_twelve`.

```
v=[10,12]
```

```
ten_twelve =
  FirstRankOneMatrix([1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0], [1.0, 2.0, 3.0, 4.0, 5.
```

```
ten_twelve = FirstRankOneMatrix([i for i in 1.0:10.0],[i for i in 1.0:12.0]) # Your
code here
```

Got it!

Good job!

Exercise 2.2 - *extending Base methods*

Right now, our `FirstRankOneMatrix` is just a container for two arrays. To make it *behave* like a matrix, we add methods to functions in `Base`, Julia's standard set of functions. Most of the functions you have used so far come from base, for example:

```
true
```

```
sqrt === Base.sqrt
```

```
true
```

```
size === Base.size
```

```
true
```

```
filter === Base.filter
```

These are built-in functions, and each function comes with a set of methods pre-defined (in Julia's source code *(which is mostly written in Julia)*). Uncomment the next cell to see the full list of methods for `size`. We will add a method to this list!

90 methods for generic function **size** from [90mBase [39m:

- size(itr::**Base.AsyncGenerator**) in Base at asyncmap.jl:389
- size(r::**Core.Compiler.StmtRange**) in Base.IRShow at show.jl:2820
- size(M::**Main.var"workspace#46".LowRankMatrix**) in Main.var"workspace#56" at C:\Users\Dell\.julia\pluto_notebooks\hw5_29fb5b50.jl#==#dd27f508-8503-11eb-36b9-33f5f99f78b0:1
- size(s::**Base.ExceptionStack**) in Base at errorshow.jl:1066
- size(M::**Main.var"workspace#3".FirstRankOneMatrix**) in Main.var"workspace#3" at C:\Users\Dell\.julia\pluto_notebooks\hw5_29fb5b50.jl#==#fada4734-8505-11eb-3f2b-d1f1ef391ba4:1
- size(B::**BitVector**) in Base at bitarray.jl:104
- size(B::**BitVector**, d::**Integer**) in Base at bitarray.jl:107
- size(m::**Base.MethodList**) in Base at reflection.jl:1193
- size(A::**LinearAlgebra.LU**, i::**Integer**) in LinearAlgebra at C:\Users\Dell\.julia\juliaup\julia-1.11.6+0.x64.w64.mingw32\share\julia\stdlib\v1.11\LinearAlgebra\src\lu.jl:371
- size(A::**LinearAlgebra.LU**) in LinearAlgebra at C:\Users\Dell\.julia\juliaup\julia-1.11.6+0.x64.w64.mingw32\share\julia\stdlib\v1.11\LinearAlgebra\src\lu.jl:370
- size(iter::**Base.SCartesianIndices2{K}**) *where K* in Base at reinterpretarray.jl:278
- size(F::**LinearAlgebra.Hessenberg**) in LinearAlgebra at C:\Users\Dell\.julia\juliaup\julia-1.11.6+0.x64.w64.mingw32\share\julia\stdlib\v1.11\LinearAlgebra\src\hessenberg.jl:395
- size(F::**LinearAlgebra.Hessenberg**, d::**Integer**) in LinearAlgebra at C:\Users\Dell\.julia\juliaup\julia-1.11.6+0.x64.w64.mingw32\share\julia\stdlib\v1.11\LinearAlgebra\src\hessenberg.jl:394
- size(adjQ::**LinearAlgebra.AdjointQ**) in LinearAlgebra at C:\Users\Dell\.julia\juliaup\julia-1.11.6+0.x64.w64.mingw32\share\julia\stdlib\v1.11\LinearAlgebra\src\abstractq.jl:63
- size(F::**LinearAlgebra.AdjointFactorization**) in LinearAlgebra at C:\Users\Dell\.julia\juliaup\julia-1.11.6+0.x64.w64.mingw32\share\julia\stdlib\v1.11\LinearAlgebra\src\factorization.jl:41
- size(S::**Base.IdentityUnitRange**) in Base at indices.jl:424
- size(D::**LinearAlgebra.Diagonal**) in LinearAlgebra at C:\Users\Dell\.julia\juliaup\julia-1.11.6+0.x64.w64.mingw32\share\julia\stdlib\v1.11\LinearAlgebra\src\diagonal.jl:140
- size(F::**LinearAlgebra.LQ**) in LinearAlgebra at C:\Users\Dell\.julia\juliaup\julia-1.11.6+0.x64.w64.mingw32\share\julia\stdlib\v1.11\LinearAlgebra\src\lq.jl:160
- size(F::**LinearAlgebra.LQ**, dim::**Integer**) in LinearAlgebra at C:\Users\Dell\.julia\juliaup\julia-1.11.6+0.x64.w64.mingw32\share\julia\stdlib\v1.11\LinearAlgebra\src\lq.jl:159
- size(t::**Tuple**, d::**Integer**) in Base at tuple.jl:29
- size(C::**Base.Experimental.Const**) in Base.Experimental at experimental.jl:29
- size(g::**Base.Generator**) in Base at generator.jl:52
- size(e::**Base.Iterators.Enumerate**) in Base.Iterators at iterators.jl:203
- size(A::**LinearAlgebra.UpperTriangular**) in LinearAlgebra at C:\Users\Dell\.julia\juliaup\julia-1.11.6+0.x64.w64.mingw32\share\julia\stdlib\v1.11\LinearAlgebra\src\triangular.jl:35
- size(a::**GenericMemory**) in Base at genericmemory.jl:67
- size(a::**GenericMemory**, d::**Int64**) in Base at genericmemory.jl:62
- size(a::**GenericMemory**, d::**Integer**) in Base at genericmemory.jl:66
- size(v::**Base.Sort.WithoutMissingVector**) in Base.Sort at sort.jl:599
- size(B::**LinearAlgebra.BunchKaufman**, d::**Integer**) in LinearAlgebra at C:\Users\Dell\.julia\juliaup\julia-1.11.6+0.x64.w64.mingw32\share\julia\stdlib\v1.11\LinearAlgebra\src\bunchkaufman.jl:216
- size(B::**LinearAlgebra.BunchKaufman**) in LinearAlgebra at C:\Users\Dell\.julia\juliaup\julia-1.11.6+0.x64.w64.mingw32\share\julia\stdlib\v1.11\LinearAlgebra\src\bunchkaufman.jl:215
- size(S::**LinearAlgebra.LDLt**, i::**Integer**) in LinearAlgebra at C:\Users\Dell\.julia\juliaup\julia-1.11.6+0.x64.w64.mingw32\share\julia\stdlib\v1.11\LinearAlgebra\src\ldlt.jl:55

- size(S::**LinearAlgebra.LDLt**) in LinearAlgebra at C:\Users\Dell\.julia\juliaup\julia-1.11.6+0.x64.w64.mingw32\share\julia\stdlib\v1.11\LinearAlgebra\src\ldlt.jl:54
- size(A::**LinearAlgebra.SVD**) in LinearAlgebra at C:\Users\Dell\.julia\juliaup\julia-1.11.6+0.x64.w64.mingw32\share\julia\stdlib\v1.11\LinearAlgebra\src\svd.jl:264
- size(A::**LinearAlgebra.SVD**, dim::**Integer**) in LinearAlgebra at C:\Users\Dell\.julia\juliaup\julia-1.11.6+0.x64.w64.mingw32\share\julia\stdlib\v1.11\LinearAlgebra\src\svd.jl:263
- size(Q::**LinearAlgebra.HessenbergQ**, dim::**Integer**) in LinearAlgebra at C:\Users\Dell\.julia\juliaup\julia-1.11.6+0.x64.w64.mingw32\share\julia\stdlib\v1.11\LinearAlgebra\src\abstractq.jl:500
- size(Q::**Union{LinearAlgebra.QRCompactWYQ, LinearAlgebra.QRPackedQ}**, dim::**Integer**) in LinearAlgebra at C:\Users\Dell\.julia\juliaup\julia-1.11.6+0.x64.w64.mingw32\share\julia\stdlib\v1.11\LinearAlgebra\src\abstractq.jl:317
- size(Q::**LinearAlgebra.AbstractQ**, dim::**Integer**) in LinearAlgebra at C:\Users\Dell\.julia\juliaup\julia-1.11.6+0.x64.w64.mingw32\share\julia\stdlib\v1.11\LinearAlgebra\src\abstractq.jl:54
- size(x::**Ref**) in Base at refpointer.jl:97
- size(c::**AbstractChar**, d::**Integer**) in Base at char.jl:195
- size(c::**AbstractChar**) in Base at char.jl:194
- size(H::**LinearAlgebra.UpperHessenberg**) in LinearAlgebra at C:\Users\Dell\.julia\juliaup\julia-1.11.6+0.x64.w64.mingw32\share\julia\stdlib\v1.11\LinearAlgebra\src\hessenberg.jl:58
- size(M::**Main.var"workspace#21".RankOneMatrix**) in Main.var"workspace#23" at C:\Users\Dell\.julia\pluto_notebooks\hw5_29fb5b50.jl#==#f2d8b45c-8501-11eb-1c6a-5f819c240d9d:1
- size(S::**Base.Slice**) in Base at indices.jl:395
- size(r::**AbstractRange**) in Base at range.jl:676
- size(A::**LinearAlgebra.SymTridiagonal**) in LinearAlgebra at C:\Users\Dell\.julia\juliaup\julia-1.11.6+0.x64.w64.mingw32\share\julia\stdlib\v1.11\LinearAlgebra\src\tridiag.jl:150
- size(r::**Base.LogRange**) in Base at range.jl:1632
- size(Q::**LinearAlgebra.LQPackedQ**) in LinearAlgebra at C:\Users\Dell\.julia\juliaup\julia-1.11.6+0.x64.w64.mingw32\share\julia\stdlib\v1.11\LinearAlgebra\src\abstractq.jl:560
- size(bc::**Base.Broadcast.Broadcasted**) in Base.Broadcast at broadcast.jl:262
- size(v::**Base.Pairs**) in Base.Iterators at iterators.jl:291
- size(a::**Array**) in Base at array.jl:194
- size(a::**Array**, d::**Int64**) in Base at array.jl:189
- size(a::**Array**, d::**Integer**) in Base at array.jl:188
- size(iter::**CartesianIndices**) in Base.IteratorsMD at multidimensional.jl:457
- size(A::**LinearAlgebra.LowerTriangular**) in LinearAlgebra at C:\Users\Dell\.julia\juliaup\julia-1.11.6+0.x64.w64.mingw32\share\julia\stdlib\v1.11\LinearAlgebra\src\triangular.jl:35
- size(M::**LinearAlgebra.Bidiagonal**) in LinearAlgebra at C:\Users\Dell\.julia\juliaup\julia-1.11.6+0.x64.w64.mingw32\share\julia\stdlib\v1.11\LinearAlgebra\src\bidiag.jl:275
- size(Q::**LinearAlgebra.HessenbergQ**) in LinearAlgebra at C:\Users\Dell\.julia\juliaup\julia-1.11.6+0.x64.w64.mingw32\share\julia\stdlib\v1.11\LinearAlgebra\src\abstractq.jl:501
- size(iter::**LinearIndices**) in Base at indices.jl:514
- size(A::**PermutedDimsArray{T, N, perm}**) *where {T, N, perm}* in Base.PermutedDimsArrays at permuteddimsarray.jl:49
- size(P::**Base.Iterators.ProductIterator**) in Base.Iterators at iterators.jl:1096
- size(M::**Main.var"workspace#21".RankTwoMatrix**) in Main.var"workspace#43" at C:\Users\Dell\.julia\pluto_notebooks\hw5_29fb5b50.jl#==#0bab818e-8503-11eb-02b3-178098599847:1
- size(x::**Number**) in Base at number.jl:80
- size(x::**Number**, d::**Integer**) in Base at number.jl:81
- size(A::**Base.ReshapedArray**) in Base at reshapedarray.jl:217

- size(B::**BitArray**) in Base at bitarray.jl:105
- size(L::**Base.LogicalIndex**) in Base at multidimensional.jl:768
- size(r::**Base.Iterators.Reverse**) in Base.Iterators at iterators.jl:127
- size(F::**LinearAlgebra.TransposeFactorization**) in LinearAlgebra at C:\Users\Dell\.julia\juliaup\julia-1.11.6+0.x64.w64.mingw32\share\julia\stdlib\v1.11\LinearAlgebra\src\factorization.jl:42
- size(s::**Slices**) in Base at slicearray.jl:230
- size(V::**SubArray**) in Base at subarray.jl:65
- size(a::**Base.ReinterpretArray{T, 0, S, A, false} where {S, A<:AbstractArray{S, 0}}**) *where T* in Base at reinterpretarray.jl:356
- size(a::**Base.ReinterpretArray{T, N, S, A, true} where {N, A<:(AbstractArray{S})}**) *where {T, S}* in Base at reinterpretarray.jl:350
- size(a::**Base.ReinterpretArray{T, N, S, A, false} where {N, A<:AbstractArray{S, N}}**) *where {T, S}* in Base at reinterpretarray.jl:345
- size(A::**LinearAlgebra.UnitUpperTriangular**) in LinearAlgebra at C:\Users\Dell\.julia\juliaup\julia-1.11.6+0.x64.w64.mingw32\share\julia\stdlib\v1.11\LinearAlgebra\src\triangular.jl:35
- size(s::**Base.CodeUnits**) in Base at strings/basic.jl:800
- size(a::**Random.UnsafeView**) in Random at C:\Users\Dell\.julia\juliaup\julia-1.11.6+0.x64.w64.mingw32\share\julia\stdlib\v1.11\Random\src\RNGs.jl:525
- size(M::**LinearAlgebra.Tridiagonal**) in LinearAlgebra at C:\Users\Dell\.julia\juliaup\julia-1.11.6+0.x64.w64.mingw32\share\julia\stdlib\v1.11\LinearAlgebra\src\tridiag.jl:588
- size(A::**LinearAlgebra.UnitLowerTriangular**) in LinearAlgebra at C:\Users\Dell\.julia\juliaup\julia-1.11.6+0.x64.w64.mingw32\share\julia\stdlib\v1.11\LinearAlgebra\src\triangular.jl:35
- size(itr::**Base.Iterators.Accumulate**) in Base.Iterators at iterators.jl:624
- size(t::**AbstractArray{T, N}**, d) *where {T, N}* in Base at abstractarray.jl:42
- size(z::**Base.Iterators.Zip**) in Base.Iterators at iterators.jl:406
- size(Q::**Union{LinearAlgebra.QRCompactWYQ, LinearAlgebra.QRPackedQ}**) in LinearAlgebra at C:\Users\Dell\.julia\juliaup\julia-1.11.6+0.x64.w64.mingw32\share\julia\stdlib\v1.11\LinearAlgebra\src\abstractq.jl:319
- size(F::**Union{LinearAlgebra.QR, LinearAlgebra.QRCompactWY, LinearAlgebra.QRPivoted}**, dim::**Integer**) in LinearAlgebra at C:\Users\Dell\.julia\juliaup\julia-1.11.6+0.x64.w64.mingw32\share\julia\stdlib\v1.11\LinearAlgebra\src\qr.jl:522
- size(F::**Union{LinearAlgebra.QR, LinearAlgebra.QRCompactWY, LinearAlgebra.QRPivoted}**) in LinearAlgebra at C:\Users\Dell\.julia\juliaup\julia-1.11.6+0.x64.w64.mingw32\share\julia\stdlib\v1.11\LinearAlgebra\src\qr.jl:521
- size(C::**Union{LinearAlgebra.Cholesky, LinearAlgebra.CholeskyPivoted}**, d::**Integer**) in LinearAlgebra at C:\Users\Dell\.julia\juliaup\julia-1.11.6+0.x64.w64.mingw32\share\julia\stdlib\v1.11\LinearAlgebra\src\cholesky.jl:516
- size(C::**Union{LinearAlgebra.Cholesky, LinearAlgebra.CholeskyPivoted}**) in LinearAlgebra at C:\Users\Dell\.julia\juliaup\julia-1.11.6+0.x64.w64.mingw32\share\julia\stdlib\v1.11\LinearAlgebra\src\cholesky.jl:515
- size(A::**Union{LinearAlgebra.Hermitian{T, S}, LinearAlgebra.Symmetric{T, S}} where {T, S}**) in LinearAlgebra at C:\Users\Dell\.julia\juliaup\julia-1.11.6+0.x64.w64.mingw32\share\julia\stdlib\v1.11\LinearAlgebra\src\symmetric.jl:225
- size(F::**Union{LinearAlgebra.AdjointFactorization, LinearAlgebra.TransposeFactorization}**, d::**Integer**) in LinearAlgebra at C:\Users\Dell\.julia\juliaup\julia-1.11.6+0.x64.w64.mingw32\share\julia\stdlib\v1.11\LinearAlgebra\src\factorization.jl:43
- size(A::**Union{LinearAlgebra.Adjoint{T, var"#s5052"}, LinearAlgebra.Transpose{T, var"#s5052"}} where {T, var"#s5052"<:(AbstractMatrix)}**) in LinearAlgebra at C:\Users\Dell\.julia\juliaup\julia-1.11.6+0.x64.w64.mingw32\share\julia\stdlib\v1.11\LinearAlgebra\src\adjtrans.jl:326

- size(v::**Union{LinearAlgebra.Adjoint{T, var"#s5052"}, LinearAlgebra.Transpose{T, var"#s5052"}} where {T, var"#s5052"<:(AbstractVector)}**) in LinearAlgebra at C:\Users\Dell\.julia\juliaup\julia-1.11.6+0.x64.w64.mingw32\share\julia\stdlib\v1.11\LinearAlgebra\src\adjtrans.jl:325
- size(A::**Union{LinearAlgebra.Adjoint{T, S}, LinearAlgebra.Transpose{T, S}} where {T, S}**) in LinearAlgebra at C:\Users\Dell\.julia\juliaup\julia-1.11.6+0.x64.w64.mingw32\share\julia\stdlib\v1.11\LinearAlgebra\src\adjtrans.jl:321

```julia
methods(size)
```

Let's extend the `size` and `getindex` methods so that they can also work with our rank-1 matrices. This will allow us to easily access the size and entries of the matrix.

👉 Extend `Base.size` for the type `FirstRankOneMatrix` to return the size as a tuple of values corresponding to each direction.

```julia
function Base.size(M::FirstRankOneMatrix)

    return tuple(size(M.v)[1],size(M.w)[1]) # Your code here
end
```

```
(10, 12)
```

```julia
(Base.size(ten_twelve))
```

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

👉 Extend `Base.getindex` for the type `FirstRankOneMatrix` to return the (i, j)th entry of the outer product.

```julia
function Base.getindex(M::FirstRankOneMatrix, i, j)

    return (M.v)[i]*(M.w)[j] # Your code here
end
```

```
55.0
```

```julia
Base.getindex(ten_twelve, 5, 11)
```

> **Got it!**
>
> Good job!

Exercise 2.3

If you ask Julia to display the value of an object of type `FirstRankOneMatrix`, you will notice that it doesn't do what we are used to, which is probably to display the whole matrix that it corresponds to. Let's see:

```
FirstRankOneMatrix([1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0], [1.0, 2.0, 3.0, 4.0, 5.
```

```julia
ten_twelve
```

This is what matrices normally look like in Julia:

```
10×12 Matrix{Float64}:
 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0
 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0
```

```julia
ones(10, 12) # An example matrix (two-dimensional array)
```

One possible workaround for this is to create a new function specifically to display a matrix of our custom type.

👉 Make a function `print_as_matrix` that prints the entries of a matrix of type `FirstRankOneMatrix` in a grid format. We'll test that it works below with the matrix `R` we already created.

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

```julia
function print_as_matrix(M::FirstRankOneMatrix)

    # Your code here
    return [(M.v)[i]*(M.w)[j] for i in 1:size(M.v)[1], j in 1:size(M.w)[1]]
end
```

We should now be able to see our `FirstRankOneMatrix`-type matrix displayed in the terminal!

```
10×12 Matrix{Float64}:
  1.0   2.0   3.0   4.0   5.0   6.0   7.0   8.0   9.0   10.0   11.0   12.0
  2.0   4.0   6.0   8.0  10.0  12.0  14.0  16.0  18.0   20.0   22.0   24.0
  3.0   6.0   9.0  12.0  15.0  18.0  21.0  24.0  27.0   30.0   33.0   36.0
  4.0   8.0  12.0  16.0  20.0  24.0  28.0  32.0  36.0   40.0   44.0   48.0
  5.0  10.0  15.0  20.0  25.0  30.0  35.0  40.0  45.0   50.0   55.0   60.0
  6.0  12.0  18.0  24.0  30.0  36.0  42.0  48.0  54.0   60.0   66.0   72.0
  7.0  14.0  21.0  28.0  35.0  42.0  49.0  56.0  63.0   70.0   77.0   84.0
  8.0  16.0  24.0  32.0  40.0  48.0  56.0  64.0  72.0   80.0   88.0   96.0
  9.0  18.0  27.0  36.0  45.0  54.0  63.0  72.0  81.0   90.0   99.0  108.0
 10.0  20.0  30.0  40.0  50.0  60.0  70.0  80.0  90.0  100.0  110.0  120.0
```

```julia
print_as_matrix(ten_twelve)
```

Hint

Check the docs for functions like `print`, `println`, `lpad` and `rpad`. Also, you don't have to know exactly how the `as` keyword works for now, but feel free to read about it - it can be useful with iterating!

Exercise 2.4 - *AbstractMatrix*

In fact, Julia (together with Pluto) can do some of this work for us! Julia provides facilities to make our life easier when we tell Julia that our type *behaves like* an array. We do so by making the type a *subtype of* `AbstractMatrix`. This will let our type **inherit** the methods and attributes that come with the `AbstractMatrix` type by default - including some display functions that Pluto notebooks use.

Let's do all this on a new rank-one matrix type: `RankOneMatrix`. As we do this, we will also remove the restriction to `Float64`-type entries by using a *parametrised type* `T`. Parametrised types allow us flexibility to handle different types of entries without repeating a lot of code; we won't go too in-depth about these for now.

```
RankOneMatrix
```

```julia
begin
    struct RankOneMatrix{T} <: AbstractMatrix{T}
        v::AbstractVector{T}
        w::AbstractVector{T}
    end

    # Add the two extra constructors
    # (Should we make these missing by default? if so - remove hint below)
    RankOneMatrix(v) = RankOneMatrix(v, v)
    RankOneMatrix(v::Float64)= RankOneMatrix(v, v)
end
```

In the cell above, we added a second ('outer') constructor that takes a single vector as argument.

👉 Make sure that you can use both constructors by trying them out below.

```
matrix1 = 1×1 RankOneMatrix{Float64}:
           1.0
```

```julia
matrix1 = RankOneMatrix([1.0])
```

```
matrix2 = 2×2 RankOneMatrix{Float64}:
           25.0  30.0
           30.0  36.0
```

```julia
matrix2 = RankOneMatrix([5.0, 6.0])
```

Hint

If you want to specify that your argument is a range, use the `AbstractRange` type.

👉 Add `getindex` and `size` methods for this new type. These will allow us to access entries of our custom matrices with the usual index notation `M[i,j]`, as well as quickly retrieving their dimensions.

```julia
function Base.size(M::RankOneMatrix)

    return tuple(size(M.v)[1],size(M.w)[1]) # Your code here
end
```

```julia
function Base.getindex(M::RankOneMatrix, i, j)

    return M.v[i]*M.w[j] # Your code here
end
```

```
R2 = 2×3 RankOneMatrix{Int64}:
     3  4   5
     6  8  10
```

```julia
R2 = RankOneMatrix([1,2], [3,4,5])
```

```
10
```

```julia
Base.getindex(R2, 2, 3)
```

```
(2, 3)
```

```julia
Base.size(R2)
```

Got it!

Fantastic!

Got it!

Splendid!

👉 Create an object of the new type for the 10×10 multiplication matrix, using a single range.

```
M = 10×10 RankOneMatrix{Int64}:
     1   2   3   4   5   6   7   8   9   10
     2   4   6   8  10  12  14  16  18   20
     3   6   9  12  15  18  21  24  27   30
     4   8  12  16  20  24  28  32  36   40
     5  10  15  20  25  30  35  40  45   50
     6  12  18  24  30  36  42  48  54   60
     7  14  21  28  35  42  49  56  63   70
     8  16  24  32  40  48  56  64  72   80
     9  18  27  36  45  54  63  72  81   90
    10  20  30  40  50  60  70  80  90  100
```

```julia
M = RankOneMatrix(1:10) # missing # Your code here
```

You should see two things: Firstly, the matrix now contains integers, instead of floats (this is thanks to our parametrised type!). And secondly, Julia *automatically* displays the matrix as we wanted, once you have defined `getindex` and `size`, provided you have told Julia that your type is a subtype of `AbstractArray`!

Julia also allows us to automatically convert the matrix to a normal ("dense") matrix, using either `collect` or `Array`. Let's try these out. You may need to re-run the cell below after completing the exercises.

```
10×10 Matrix{Int64}:
  1   2   3   4   5   6   7   8   9   10
  2   4   6   8  10  12  14  16  18   20
  3   6   9  12  15  18  21  24  27   30
  4   8  12  16  20  24  28  32  36   40
  5  10  15  20  25  30  35  40  45   50
  6  12  18  24  30  36  42  48  54   60
  7  14  21  28  35  42  49  56  63   70
  8  16  24  32  40  48  56  64  72   80
  9  18  27  36  45  54  63  72  81   90
 10  20  30  40  50  60  70  80  90  100
```

```julia
collect(M)
```

```
Matrix{Int64} (alias for Array{Int64, 2})
```

```julia
typeof(collect(M))
```

> Fun fact: we did not define a method for `Base.collect`, and yet it works! This is because we told Julia that our `RankOneMatrix` is a subtype of `AbstractMatrix`, for which `Base.collect` already has a method defined in Julia's source code. This fallback method uses `Base.getindex` and `Base.size`, which we did define.

Exercise 2.5

Why do we need a special type to represent special types of structured matrices? One reason is that not only do they give a more efficient representation in space (requiring less memory to store), they can also be more efficient in time, i.e. *faster*.

For example, let's look at **matrix–vector multiplication**. This is a *fundamental* part of many, *many* algorithms in scientific computing, and because of this, we usually want it to be as fast as possible.

For a rank-one matrix given by $M = vw^T$, the matrix–vector product $M \cdot x$ is given by $(w \cdot x)v$. Note that $w \cdot x$ is a number (scalar) which is multiplying the vector element by element. This computation is much faster than the usual matrix-vector multiplication: we are taking advantage of structure!

👉 Define a function `matvec` that takes a `RankOneMatrix M` and a `Vector x` and carries out matrix-vector multiplication. We will be able to compare the result with doing the matrix–vector product using the corresponding dense matrix.

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

```julia
function matvec(M::RankOneMatrix, x)

    return (M.v')*x*(M.w) # Your code here
end
```

Got it!

You got the right answer!

Exercise 3: Low-rank matrices

In this exercise we will combine rank-1 matrices into low-rank matrices, which are sums of rank-1 matrices. Just like in the previous exercise, we will make custom types for these matrices, and we will be able to compute the entries of a matrix on-demand as we request them (instead of storing all of them always).

Exercise 3.1

👉 Make a rank-2 matrix type `RankTwoMatrix` that contains two rank-1 matrices `A` and `B`. For simplicity, you can use the `FirstRankOneMatrix` type.

Include in the same cell a constructor for `RankTwoMatrix` that takes two vectors and makes the rank-one matrices from those vectors.

Note: In principle we should check when constructing the type that the input matrices have the same dimensions, but we will just assume that they do.

RankTwoMatrix

```julia
begin
    struct RankTwoMatrix{T} <: AbstractMatrix{T}
        # Your code here
        A::RankOneMatrix{T}
        B::RankOneMatrix{T}
    end

    # Add a constructor that uses two vectors/ranges
    RankTwoMatrix(v1, v2) = RankTwoMatrix(RankOneMatrix(v1), RankOneMatrix(v2))
end
```

Exercise 3.2

👉 Construct a rank two matrix out of the rank-1 matrix representing the multiplication table of $1.0 : 10.0$, together with the multiplication table of $0.0 : 0.1 : 0.9$.

(Note that both range arguments must contain floats, so that we can add up entries.)

```
10×10 RankTwoMatrix{Float64}:
  1.0   2.0    3.0    4.0    5.0    6.0    7.0    8.0    9.0    10.0
  2.0   4.01   6.02   8.03  10.04  12.05  14.06  16.07  18.08   20.09
  3.0   6.02   9.04  12.06  15.08  18.1   21.12  24.14  27.16   30.18
  4.0   8.03  12.06  16.09  20.12  24.15  28.18  32.21  36.24   40.27
  5.0  10.04  15.08  20.12  25.16  30.2   35.24  40.28  45.32   50.36
  6.0  12.05  18.1   24.15  30.2   36.25  42.3   48.35  54.4    60.45
  7.0  14.06  21.12  28.18  35.24  42.3   49.36  56.42  63.48   70.54
  8.0  16.07  24.14  32.21  40.28  48.35  56.42  64.49  72.56   80.63
  9.0  18.08  27.16  36.24  45.32  54.4   63.48  72.56  81.64   90.72
 10.0  20.09  30.18  40.27  50.36  60.45  70.54  80.63  90.72  100.81
```

```julia
RankTwoMatrix(RankOneMatrix([i for i in 1.0:10.0]),RankOneMatrix([i for i in 0.0:0.1:0.9]))
```

Exercise 3.3

👉 As with last time, extend the `getindex` and `size` methods for the `RankTwoMatrix` type. Keep in mind they are already defined for `RankOneMatrix` .

```julia
function Base.getindex(M::RankTwoMatrix, i, j)

    return (M.A.v[i])*(M.A.w[j])+(M.B.v[i])*(M.B.w[j]) # Your code here
end
```

```julia
function Base.size(M::RankTwoMatrix)

    return tuple(size(M.A.v)[1],size(M.A.w)[1])
end
```

Got it!

Fantastic!

Got it!

Keep it up!

Exercise 3.4

Making a custom type for rank-2 matrices is a step forward from rank-1 matrices: instead of storing two vectors, we store two rank-1 matrices themselves. What if we want to represent a rank-3 matrix? We would need to store *three* rank-1 matrices, instead of just two. What about rank-4, rank-5, and so on?

We can go even further and make a general custom type `LowRankMatrix` for rank-k matrices, for general (ideally low) k. In this case, we should store two main things: the list of rank-1 matrices that our low-rank matrix is made up of, and also the *rank* of the matrix (which is how many rank-1 matrices we are storing).

👉 Complete the definition for the type `LowRankMatrix`. Remember to store both the rank-1 matrices and the rank of the matrix itself.

```
LowRankMatrix
```

```julia
begin
    struct LowRankMatrix <: AbstractMatrix{Float64}
        # Your code here
        Ms::Vector{RankOneMatrix}
        rank::Int
    end
        LowRankMatrix(Vs)=LowRankMatrix(RankOneMatrix(V) for V in Vs )
end
```

Exercise 3.5

👉 Extend the `getindex` and `size` methods to work with the `LowRankMatrix` type. As before, remember that these are already defined for `RankOneMatrix`.

```julia
function Base.getindex(M::LowRankMatrix, i, j)

    return sum((Mat.v[i]*Mat.w[j]) for Mat in M.Ms)
end
```

```julia
function Base.size(M::LowRankMatrix)

    return tuple(size((M.Ms[1]).v)[1],size((M.Ms[1]).w)[1]) # Your code here
end
```

```
10×10 LowRankMatrix:
  1.0   2.0    3.0    4.0    5.0    6.0    7.0    8.0    9.0    10.0
  2.0   4.01   6.02   8.03  10.04  12.05  14.06  16.07  18.08   20.09
  3.0   6.02   9.04  12.06  15.08  18.1   21.12  24.14  27.16   30.18
  4.0   8.03  12.06  16.09  20.12  24.15  28.18  32.21  36.24   40.27
  5.0  10.04  15.08  20.12  25.16  30.2   35.24  40.28  45.32   50.36
  6.0  12.05  18.1   24.15  30.2   36.25  42.3   48.35  54.4    60.45
  7.0  14.06  21.12  28.18  35.24  42.3   49.36  56.42  63.48   70.54
  8.0  16.07  24.14  32.21  40.28  48.35  56.42  64.49  72.56   80.63
  9.0  18.08  27.16  36.24  45.32  54.4   63.48  72.56  81.64   90.72
 10.0  20.09  30.18  40.27  50.36  60.45  70.54  80.63  90.72  100.81
```

```julia
let
    comp1, comp2 = RankOneMatrix(1.0:1.0:10.0), RankOneMatrix(0.0:0.1:0.9)
    ex3 = LowRankMatrix([comp1, comp2], 2)
end
```

Got it!

Awesome!

Got it!

Great! 🎉

Exercise 3.6

👉 Extend the method `matvec` to work with our custom type `LowRankMatrix` (and a `Vector`). Remember that `matvec` is already defined for `RankOneMatrix`.

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

```julia
function matvec(M::LowRankMatrix, x)

    return  sum([matvec(P::RankOneMatrix,x) for P in M.Ms]) # Your code here
end
```

Got it!

Keep it up!

Hint

How would we implement `matvec` for a rank-2 matrix - making use of the two stored rank-1 matrices? Can you generalize this to the rank-k case, when we have many rank-1 matrices?

Exercise 3.7

One of the big advantages of our rank-1 matrices is its space efficiency: to "store" a $n \times n$ matrix, we only need to *actually* store two vectors of n entries, as opposed of n^2 total entries. Rank-2 matrices are a litte less efficient: we store two rank-1 matrices, or four vectors of n vertices. As we increase the rank, we lose space efficiency, at the cost of being able to represent more kinds of matrices.

👉 For what rank will a matrix of rank k need the same amount of storage as the dense version? Explain your answer.

answer =

(n^2-1)÷2

From rank ... greater than (n^2-1)÷2

Because .... a rank k matrix takes 2*k space + 1 space for saving k

```julia
answer = md"""(n^2-1)÷2

From rank ... greater than (n^2-1)÷2

Because .... a rank k matrix takes 2*k space + 1 space for saving k
"""
```

Exercise 4: *The SVD for structured matrices*

In a math class, you may or may not learn about the *singular value decomposition (SVD)*. From a computational thinking point of view, whether you have seen this before or not we hope will not matter. Here we would like you to *get to "know" the SVD*, through experience, rather than through a math lecture. This is how we see "computational thinking" after all. (Definitely do not look up some eigenvalue definition.)

```julia
using LinearAlgebra
```

Exercise 4.1

The `LinearAlgebra` package defines a function `svd` that computes the decomposition for us. Have a look at the result of calling the `svd` function. Try not to get intimidated by the large output (you may need to scroll the cell output), and look at the docs for `svd`.

```
biggie = 100×100 Matrix{Float64}:
         0.93245    0.924268  0.940801  0.110794   …  0.357527   0.568183    0.0383436
         0.246908   0.413248  0.051235  0.595203      0.725744   0.646448    0.529399
         0.15837    0.670304  0.879686  0.997722      0.141982   0.279378    0.771037
         0.355989   0.628355  0.197584  0.985502      0.30396    0.362824    0.354847
         0.63907    0.976263  0.597662  0.0638108     0.885332   0.130899    0.639162
         0.792211   0.898586  0.9351    0.84489    …  0.0702502  0.240562    0.356067
         0.732832   0.33595   0.921646  0.428273      0.768409   0.647821    0.54444
         ⋮                                         ⋱
         0.0174607  0.646451  0.854467  0.651217      0.978402   0.766664    0.210723
         0.76466    0.250359  0.209289  0.88104    …  0.743147   0.36897     0.311578
         0.0916377  0.186718  0.595724  0.443485      0.538931   0.458338    0.302358
         0.507175   0.132976  0.778299  0.58275       0.355076   0.383761    0.0333924
         0.596712   0.40404   0.707319  0.198519      0.437907   0.00217145  0.636171
         0.0505854  0.480191  0.166168  0.271852      0.332433   0.660515    0.97204
```

```julia
biggie = rand(100,100)
```

```
SVD{Float64, Float64, Matrix{Float64}, Vector{Float64}}
U factor:
100×100 Matrix{Float64}:
 -0.0889595  -0.250068    0.0751652   …   0.0433915  -0.0812317   -0.216832
 -0.109288   -0.124826    0.00439088     -0.196819   -0.0096053   -0.0182499
 -0.0908332   0.135965    0.0122522      -0.0464696  -0.00144434  -0.125558
 -0.093441    0.0215209   0.0938596      -0.022907   -0.0589592   -0.133162
 -0.0909441   0.07738     0.152077       -0.238252   -0.0156685   -0.014837
 -0.0952176   0.203173    0.107265    …   0.074795    0.0789822    0.273097
 -0.103443    0.0955181  -0.159739        0.310997   -0.109462    -0.0609661
  ⋮                                   ⋱
 -0.0996758  -0.177366   -0.00338155      0.11517     0.205407     0.0347842
 -0.108657    0.0681983  -0.0410133   …  -0.144933    0.113819    -0.0366819
 -0.10644     0.0124634  -0.137729       -0.148695    0.0596091    0.0797656
 -0.101086   -0.127159   -0.105161       -0.023771    0.197903     0.152677
 -0.0889895   0.0750574  -0.200656       -0.0694747   0.0944506    0.116605
 -0.101605   -0.0719797   0.0659688       0.0824795   0.0457286   -0.00284141
singular values:
100-element Vector{Float64}:
 50.64003805482805
  5.802913831634644
  5.4494122462098735
  5.335222377441876
  5.305844624500972
  5.121692388284296
  5.020304280592681
  ⋮
  0.24841709714111962
  0.21078394511903104
  0.17072073496962334
  0.10000569086452053
  0.04381227151674252
  0.0039474457141041
```

```
svd(biggie)
```

👉 What are the *singular values* of `biggie`?

```
singular_values_of_biggie =
  [50.64, 5.80291, 5.44941, 5.33522, 5.30584, 5.12169, 5.0203, 4.96549, 4.79231, 4.75433, 4.65
```

```
singular_values_of_biggie = σ
```

Got it!

Great!

Exercise 4.2

If we try to run the `svd` function on a `RankOneMatrix`, you will see that it does not work. The error is telling us that **no `svd` method has been defined for our type**. Let's extend the `svd` function to work on objects of our type.

A =

Error message from Main

```
Failed to show value:
MethodError: no method matching iterate(::Missing)
The function `iterate` exists, but no method is defined for this combination of
argument types.

Closest candidates are:
  iterate(::Cmd)
   @ Base process.jl:716
  iterate(::Cmd, ::Any)
   @ Base process.jl:721
  iterate(::Base.EnvDict, ::Tuple{Ptr{UInt16}, Ptr{UInt16}})
   @ Base env.jl:190
  ...
```

Show stack trace...

```julia
A = RankOneMatrix(rand(3),rand(4))
```

```
SVD{Float64, Float64, Matrix{Float64}, Vector{Float64}}
U factor:
3×3 Matrix{Float64}:
 0.106507  -0.00323942  -0.994307
 0.716785  -0.6928       0.0790367
 0.689112   0.721122     0.0714659
singular values:
3-element Vector{Float64}:
 1.0485767249433307
 1.1281147602365134e-16
 1.1995235823703146e-17
Vt factor:
3×4 Matrix{Float64}:
 0.481715    0.158914    0.649465   0.566474
 0.875339   -0.0835411  -0.326007  -0.347162
 0.0414799  -0.160915   -0.643274   0.747385
```

```julia
svd(A)
```

👉 Extend `svd` to work directly on a rank-one matrix, by writing a new method.

Keep things simple. Inside your method, call `LinearAlgebra.svd` on a type that it is *already defined for*.

```julia
function LinearAlgebra.svd(A::RankOneMatrix)

    return LinearAlgebra.svd(A.v * A.w')
end
```

```
SVD{Float64, Float64, Matrix{Float64}, Vector{Float64}}
U factor:
3×3 Matrix{Float64}:
 0.106507  -0.00323942  -0.994307
 0.716785  -0.6928       0.0790367
 0.689112   0.721122     0.0714659
singular values:
3-element Vector{Float64}:
 1.0485767249433307
 1.1281147602365134e-16
 1.1995235823703146e-17
Vt factor:
3×4 Matrix{Float64}:
 0.481715    0.158914    0.649465   0.566474
 0.875339   -0.0835411  -0.326007  -0.347162
 0.0414799  -0.160915   -0.643274   0.747385
```

```julia
LinearAlgebra.svd(A)
```

Got it!

Splendid!

Exercise 4.3

👉 Look at the singular values, how many of them are approximately non-zero? Does that make sense?

```
1
```

```julia
1
```

This number (the number of singular values that are positive) is something you will (or have) learn(ed) in a linear algebra class: it is known as the *rank* of a matrix, and is usually defined through a complicated elimination procedure.

👉 Write a function `numerical_rank` that calculates the singular values of a matrix `A`, and returns how many of them are approximately zero.

To keep things simple, you can assume that "approximately zero" means: less than `tol=1e5`.

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

```julia
function numerical_rank(A::AbstractMatrix; tol=1e-5)
    U,σ,V=svd(A)
    return count(x->x>tol,σ)
end
```

```
1
```

```julia
numerical_rank(A)
```

```
3
```

```julia
numerical_rank(rand(3,3))
```

Got it!

You got the right answer!

Exercise 4.4

👉 Write a function that takes an argument k and generates the sum of k random rank-one matrices of size mxn, and counts the number of essentially non-zero singular values.

Hint: you can do this with a `for` loop, but it can also be done with `sum`, e.g.

`sum(i for i=1:10)`.

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

```julia
function k_rank_ones(k, m, n)
        M=sum(RankOneMatrix(rand(m),rand(n)) for i in 1:k)
    return M
end
```

```
3×3 RankOneMatrix{Float64}:
 0.568648    0.265466    0.0273562
 0.00214909  0.00100327  0.000103387
 0.501409    0.234077    0.0241216
```

```julia
k_rank_ones(1, 3, 3)
```

Got it!

Fantastic!

Exercise 4.5

👉 What is the answer when m and n are both $\geq k$? What if one of m or n is $< k$?

```
"m,n=>k answer is k"
```

```julia
"m,n=>k answer is k"
```

```
"m,n<k answer is min(m,n)"
```

```julia
"m,n<k answer is min(m,n)"
```

```
"m>k,n<k answer is n"
```

```julia
"m>k,n<k answer is n"
```

```
"m<k,n>k answer is m"
```

```julia
"m<k,n>k answer is m"
```

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)
```