Permutations

Objectives

- ***Create a new type*** `Permutation`, representing the mathematical object of the same name
- ***Utilise multiple dispatch to create new methods for inbuilt functions*** with `Permutation` objects as arguments
- Generate the elements of symmetric/alternating groups in `Permutation` form

Declaring a new type with `struct`

I begin by declaring a new type with the keyword `struct` as follows:

```
struct Permutation
    image::Vector{Int64}
end
```

In this case, the new type has a single field `image` which is a vector of integers, representing the images of $1, 2, \ldots, n$ under the permutation (where n is the maximum integer permuted). For example, the permutation $(123)(45)$ has image $(23154)^T$, and can be created as an object by:

```
julia> permutation = Permutation([2,3,1,5,4])
Permutation([2, 3, 1, 5, 4])
```

Although this object doesn't do anything yet, I can access its fields with dot notation:

```
julia> permutation.image
5-element Vector{Int64}:
 2
 3
 1
 5
 4
```

Inner constructors

Not any vector will do as the image. It must consist of exactly the n integers $1, 2, \ldots, n$ (for some n) in any order. To check this, I will use an inner constructor, which is a function inside the `struct` block which runs instead of the automatic constructor `Permutation` that was used earlier.

Suppose that the input is the vector `image`. I want to check two things:

- Is `image` valid for constructing a permutation? To check this, I will reorder the vector to be in increasing order, and then compare it to the vector `[1,2,...,n]`, where `n` is the last entry in the sorted vector[1]. Then, I use the keyword `new`, specific to inner constructors, which allows the creation a new instance of the type `Permutation`

  ```
  sortedimage = sort(image)
  sortedimage == 1:last(sortedimage) && return new(image)
  ```

[1] Actually, instead of the vector `[1,2,...,n]`, I use the expression `1:n`, which isn't actually a vector, but `==` recognises it as meaning the same thing

- Is `image` longer than it needs to be? For example, an `image` of `[2,1,3]` would give a permutation that acts the same as one with an `image` of `[2,1]`, so the extra elements needn't be stored.[2] I find the list of integers that aren't fixed by

  ```
  image[image .!= 1:length(image)]
  ```

  and then find the largest by

  ```
  m = maximum([image[image .!= 1:length(image)]..., 1])
  ```

  with `1` appended on the end since the list of integers that aren't fixed could be empty and `maximum` doesn't work with an empty vector

Combining these together with an error message into an inner constructor gives:

```
struct Permutation
    image::Vector{Int64}
    function Permutation(image::Vector{Int64})
        sortedimage = sort(image)
        m = maximum([image[image .!= 1:length(image)]..., 1])
        sortedimage == 1:last(sortedimage) && return new(image[1:m])
        error("not a valid permutation")
    end
end

julia> Permutation([2,3,4])
ERROR: not a valid permutation

julia> Permutation([2,1,3])
Permutation([2, 1])
```

Outer constructors

Outer constructors utilise multiple dispatch to allow for different syntax to construct types. In this instance, I want to be able to construct a permutation by just giving it a list of integers and not having to wrap them up in vector form myself. The outer constructor I have written to do this is:

```
Permutation(imagevals::Int64...) = Permutation([x for x ∈
imagevals])

julia> σ = Permutation(5,2,1,3,4)
Permutation([5, 2, 1, 3, 4])
```

Evaluating a permutation as a function

The next step is to be able to use a `Permutation` object as the function that it represents, i.e. evaluate it at an integer. I can do this by

```
(σ::Permutation)(x::Int64) = σ.image[x]
```

[2] Whether or not this is necessary is a matter of opinion and implementation. If permutations are defined as bijections of a finite set, then the size of that set does matter, so the permutations would be different. In this implementation, I have decided to ignore this, as if taking permutations to be bijections $\mathbb{N} \to \mathbb{N}$ fixing all but finitely many points, and hence want to minimise calculations by reducing in this way

However, this will give an error for any `x` which isn't an index of `σ.image`. Due to my decision to have permutations be bijections of $\mathbb{N}$ instead of bijections of $\{1, 2, \ldots, n\}$, I would like it to return a value for any positive integer (indeed it will work for all integers with this code, which isn't a problem for me):

```
maxarg(σ::Permutation) = length(σ.image)
(σ::Permutation)(x::Int64) = x ∈ 1:maxarg(σ) ? σ.image[x] : x

julia> σ(4)
3

julia> σ(10)
10
```

I implement the function `maxarg` as a shorthand for `length(σ.image)` since it will be useful in later functions too.

Creating a custom display format for a `Permutation` object

At the moment, whenever a `Permutation` object is returned, it has the form `Permutation(image)`, which isn't very useful. To be more understandable (and more in keeping with standard mathematical notation), I want to customise this displayed form by adding a method to `show` specifically for the `Permutation` type.

The format that I want to display the permutation in is as its disjoint cycle decomposition. To do this, I first need a function to calculate the orbit generated by acting repeatedly on a given element:

```
function orbit(σ::Permutation, x::Int64)
    orb = [x]
    y = σ(x)
    while y != x
        push!(orb,y)
        y = σ(y)
    end
    return orb
end
```

To calculate the disjoint cycle decomposition, I build up the output as follows:

- I start out with an empty `Vector{Vector{Int64}}` (that is, a vector whose elements are vectors of integers)
  ```
  decomp = Vector{Int64}[]
  ```
- A vector `unaccounted` tracks which values in the range `1:maxarg(σ)` are yet to be added to the decomposition. We will iterate until this vector is entirely `false`
  ```
  unaccounted = trues(maxarg(σ))
  while any(unaccounted)
      # code to iterate
  ```

```
end
```

- Inside the loop, I will look for the first value which is unaccounted for in the decomposition so far, calculate its orbit, and then update `unaccounted` accordingly

```
x = findfirst(unaccounted)
xorbit = orbit(σ,x)
unaccounted[xorbit] .= false
```

- Then, if the orbit is non-trivial, I add it to the decomposition

```
length(xorbit) > 1 && push!(decomp,xorbit)
```

Combining this all together gives the function in full:

```
function dcd(σ::Permutation)
    decomp = Vector{Int64}[]
    unaccounted = trues(maxarg(σ))
    while any(unaccounted)
        x = findfirst(unaccounted)
        xorbit = orbit(σ,x)
        unaccounted[xorbit] .= false
        length(xorbit) > 1 && push!(decomp,xorbit)
    end
    return decomp
end
```

I then need to consider how to build up the string from this decomposition

- Each cycle from the decomposition will be expressed as an open parenthesis, followed by the list of integers in order, separated by spaces, and then closed with another parenthesis

```
*( "( ", ["$y " for y ∈ x]... , ")" )
```

- This needs to be repeated for all of the cycles in the decomposition

```
toprint =
*([ *( "( ", ["$y " for y ∈ x]... , ")" ) for x ∈ dcd(σ)]...,
"")
```

- Finally, if (and only if) the permutation is the identity permutation, then the string will be empty at this point. Instead, I want to use the symbol ι

```
toprint == "" && (toprint = "ι")
```

Importing the method `show` so that its inbuilt methods don't get overwritten, I get the new method:

```
import Base.show

function show(io::IO,σ::Permutation)
    toprint =
      *([ *( "( ", ["$y " for y ∈ x]... , ")" ) for x ∈ dcd(σ)]...,
      "")
    toprint == "" && (toprint = "ι")
    print(io, toprint)
end
```

```
julia> Permutation(1,4,5,6,3,2)
( 2 4 6 )( 3 5 )

julia> σ
( 1 5 4 3 )
```

Adding permutation arithmetic

I now want to add some arithmetic for combining permutations, starting with a shortcut for the identity. This will be represented by the constant ι (mirroring the way it is displayed by `show`), and is given by:

```
const ι = Permutation(1)
```

Then, I add two methods to the inbuilt function `one` allowing it to be obtained either by parsing a permutation, or the `Permutation` type. Note that the arguments of the functions are not given names, only their types specified, since the type is the only relevant property about the input which is accessed:

```
import Base.one
one(::Permutation) = ι
one(::Type{Permutation}) = ι
```

Next, I want to be able to compose permutations, for which I will write new methods for the inbuilt operator ∘. In order for an arbitrary number of permutations to be composable, I use an inductive definition which mirrors the definition of composition of functions[3], that is:

```
∘(f) = f
∘(f, g) = ComposedFunction(f, g)
∘(f, g, h...) = ∘(f ∘ g, h...)
```

Hence, my methods for ∘ are:

```
∘(σ::Permutation) = σ
∘(σ::Permutation, τ::Permutation) =
    Permutation([σ(τ(x)) for x ∈ 1:max(maxarg(σ),maxarg(τ))])
∘(σ::Permutation, τ::Permutation, υ::Permutation...) = ∘(σ ∘ τ, υ...)
```

Thirdly, I want an inverse function to be able to find the inverse of a permutation. To do this, I need to find the index of each of $1, 2, \ldots, n$ in the image vector and let that be the image of the new permutation, which is done by:

```
inv(σ::Permutation) =
    Permutation([findfirst(σ.image .== x) for x ∈ 1:maxarg(σ)])
```

[3] This is found in Julia's source code at
https://github.com/JuliaLang/julia/blob/1b93d53fc4bb59350ada898038ed4de2994cce33/base/operators.jl#L940-L942

Finally, I want to be able to exponentiate[4] by any integer (including zero and negative integers), which can be done using some logic combined with the three operations above:

```
function ^(σ::Permutation,n::Int64)
    n == 0 && return one(Permutation)
    n < 0 && ((n,σ) = (-n,inv(σ)))
    return ∘(fill(σ,n)...)
end
```

I can now test these out:

```
julia> ι
ι

julia> σ = Permutation(5,2,1,3,4)
( 1 5 4 3 )

julia> τ = Permutation(4,5,2,3,1)
( 1 4 3 2 5 )

julia> σ ∘ τ
( 1 3 2 4 )

julia> σ^-1
( 1 3 4 5 )

julia> τ^5
ι
```

Constructing symmetric and alternating groups

Before constructing these groups, I will need a quick way of generating transpositions, for which I create a function:

```
function transposition(m::Int64, n::Int64)
    image = collect(1:max(m,n))
    image[m], image[n] = n, m
    return Permutation(image)
end
```

Note that `transposition(m,m)` returns the identity, which turns out to be exactly what I want.

Now, I have all the tools to create the symmetric group S_n as a vector of `Permutation` objects, which I will do recursively using the inductive formula

$$S_1 = \{\iota\}, \qquad S_n = \{\sigma \circ (mn) : \sigma \in S_{n-1}, m \in \{1, 2, ..., n\}\}$$

---

[4] There are several possible options for algorithms to do this, including:

- Repeatedly composing the permutation with itself `n` times (which I have chosen, as it is the simplest)
- Using exponentiation by squaring, which is more efficient for large `n`
- Using the disjoint cycle decomposition and reversing the cycles

- First, I will check that `n` is positive to avoid non-terminating loops (and also because the group doesn't make sense otherwise)
  ```
  n ≥ 1 || error("symmetric group must have a positive
  parameter")
  ```
- Then, I implement the base case of the trivial group when `n == 1`:
  ```
  n == 1 && return [ι]
  ```
- For the recursive step, I copy `symmetricgroup(n-1)` into a vector `n` times, and then set up the corresponding vector of transpositions that I will multiply each by, using array filling and concatenation. Then, I compose the two elementwise
  ```
  permutations = vcat(fill(symmetricgroup(n-1),n)...)
  transpositions =
    vcat([fill(transposition(i,n),factorial(n-1)) for i ∈
  1:n]...)
  return permutations .∘ transpositions
  ```

The entire function is:

```
function symmetricgroup(n::Int64)
    n ≥ 1 || error("symmetric group must have a positive parameter")
    n == 1 && return [ι]

    permutations = vcat(fill(symmetricgroup(n-1),n)...)
    transpositions = vcat([fill(transposition(i,n),factorial(n-1))
for i ∈ 1:n]...)
    return permutations .∘ transpositions
end
```

To find the alternating group, I need to consider the parity of elements. The easiest way to find the parity of a permutation is to consider it as a product of cycles and consider their parities. I already have such a decomposition, given by `dcd`, so using the `iseven` function and `sum` which counts the number of `true` elements of an array:

```
    parity(σ::Permutation) = (-1)^sum([iseven(length(x)) for x ∈
dcd(σ)])
```

Then, finding the alternating group is just a matter of picking out the elements of the symmetric group with parity 1:

```
function alternatinggroup(n::Int64)
    S  = symmetricgroup(n)
    return S [parity.(S ) .== 1]
End
```

Further exercises

- Write a different implementation of exponentiation for permutations (and perhaps compare efficiency)
- Write functions to find conjugacy classes / centralisers of elements in S_n (or trickier, A_n)

- Write a function to generate a group (in this case, a vector of permutations) from a list of generators. Since any finite group is a subgroup of a permutation group, you could represent any finite group in this way
- Rewrite the `Permutation` type to store its data as a list of cycles instead of as a vector representing its image. This will make some operations more efficient and some less