Julia is a scientific programming language that is free and open source.\footnote{Julia may be obtained from http://julialang.org.} It is a relatively new language that borrows inspiration from languages like Python, MATLAB, and R. It was selected for use in this book because it is sufficiently high level\footnote{In contrast with languages like C++, Julia does not require programmers to worry about memory management and other lower-level details, yet it allows low-level control when needed.} so that the algorithms can be compactly expressed and readable while also being fast. This book is compatible with Julia version 1.6. This appendix introduces the concepts necessary for understanding the included code, omitting many of the advanced features of the language.

### G.1 Types

Julia has a variety of basic types that can represent data given as truth values, numbers, strings, arrays, tuples, and dictionaries. Users can also define their own types. This section explains how to use some of the basic types and how to define new types.

#### G.1.1 Booleans

The Boolean type in Julia, written as `Bool`, includes the values `true` and `false`. We can assign these values to variables. Variable names can be any string of characters, including Unicode, with a few restrictions.

```
α = true
done = false
```

The variable name appears on the left side of the equal sign; the value that variable is to be assigned is on the right side.
We can make assignments in the Julia console. The console, or REPL (for read, eval, print, loop), will return a response to the expression being evaluated. The # symbol indicates that the rest of the line is a comment.

```julia
julia> x = true
true
julia> y = false; # semicolon suppresses the console output
julia> typeof(x)
Bool
julia> x == y # test for equality
false
```

The standard Boolean operators are supported:

```julia
julia> !x    # not
false
julia> x && y # and
false
julia> x || y # or
true
```

### G.1.2 Numbers

Julia supports integer and floating-point numbers, as shown here:

```julia
julia> typeof(42)
Int64
julia> typeof(42.0)
Float64
```

Here, `Int64` denotes a 64-bit integer, and `Float64` denotes a 64-bit floating-point value.\(^3\) We can perform the standard mathematical operations:

```julia
julia> x = 4
4
julia> y = 2
2
julia> x + y
6
julia> x - y
2
julia> x * y
8
julia> x / y
2.0
```

\(^3\) On 32-bit machines, an integer literal like 42 is interpreted as an `Int32`. 
### G.1. Types

**Exponentiation**

```julia
julia> x ^ y # exponentiation
16
```

**Remainder from Division**

```julia
julia> x % y # remainder from division
0
```

**Truncated Division Returns an Integer**

```julia
julia> div(x, y) # truncated division returns an integer
2
```

Note that the result of \( x / y \) is a **Float64**, even when \( x \) and \( y \) are integers. We can also perform these operations at the same time as an assignment. For example, \( x += 1 \) is shorthand for \( x = x + 1 \).

We can also make comparisons:

```julia
julia> 3 > 4
false
julia> 3 >= 4
false
julia> 3 ≥ 4 # unicode also works, use \ge[tab] in console
false
julia> 3 < 4
true
julia> 3 <= 4
true
julia> 3 ≤ 4 # unicode also works, use \le[tab] in console
true
julia> 3 == 4
false
julia> 3 < 4 < 5
true
```

### G.1.3 Strings

A string is an array of characters. Strings are not used very much in this textbook except to report certain errors. An object of type `String` can be constructed using " characters. For example:

```julia
julia> x = "optimal"
"optimal"
```

```julia
julia> typeof(x)
String
```
**G.1.4 Symbols**

A *symbol* represents an identifier. It can be written using the `:` operator or constructed from strings:

```
 julia> :A
 :A
 julia> :Battery
 :Battery
 julia> Symbol("Failure")
 :Failure
```

**G.1.5 Vectors**

A *vector* is a one-dimensional array that stores a sequence of values. We can construct a vector using square brackets, separating elements by commas:

```
 julia> x = [];    # empty vector
 julia> x = trues(3);    # Boolean vector containing three trues
 julia> x = ones(3);    # vector of three ones
 julia> x = zeros(3);    # vector of three zeros
 julia> x = rand(3);    # vector of three random numbers between 0 and 1
 julia> x = [3, 1, 4];    # vector of integers
 julia> x = [3.1415, 1.618, 2.7182];    # vector of floats
```

An *array comprehension* can be used to create vectors:

```
 julia> [sin(x) for x in 1:5]
 5-element Vector{Float64}:
 0.8414709848078965
 0.9092974268256817
 0.1411200080598672
 -0.7568024953079282
 -0.9589242746631385
```

We can inspect the type of a vector:

```
 julia> typeof([3, 1, 4])    # 1-dimensional array of Int64s
 Vector{Int64} (alias for Array{Int64, 1})
 julia> typeof([3.1415, 1.618, 2.7182])    # 1-dimensional array of Float64s
 Vector{Float64} (alias for Array{Float64, 1})
 julia> Vector{Float64}    # alias for a 1-dimensional array
 Vector{Float64} (alias for Array{Float64, 1})
```

We index into vectors using square brackets:
We can pull out a range of elements from an array. Ranges are specified using a colon notation:

```julia
julia> x = [1, 2, 5, 3, 1]
5-element Vector{Int64}:
  1
  2
  5
  3
  1
julia> x[1:3]  # pull out the first three elements
3-element Vector{Int64}:
  1
  2
  5
julia> x[1:2:end]  # pull out every other element
3-element Vector{Int64}:
  1
  5
  1
julia> x[end:-1:1]  # pull out all the elements in reverse order
5-element Vector{Int64}:
  1
  3
  5
  2
  1
```

We can perform a variety of operations on arrays. The exclamation mark at the end of function names is used to indicate that the function *mutates* (i.e., changes) the input:

```julia
julia> length(x)
5
julia> [x, x]  # concatenation
2-element Vector{Vector{Int64}}:
```

© 2022 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
appendix g. julia

```
[1, 2, 5, 3, 1]
[1, 2, 5, 3, 1]
julia> push!(x, -1)  # add an element to the end
6-element Vector{Int64}:
  1
  2
  5
  3
  1
  -1
julia> pop!(x)  # remove an element from the end
-1
julia> append!(x, [2, 3])  # append [2, 3] to the end of x
7-element Vector{Int64}:
  1
  2
  5
  3
  1
  2
  3
julia> sort!(x)  # sort the elements, altering the same vector
7-element Vector{Int64}:
  1
  1
  2
  2
  3
  3
  5
julia> sort(x);  # sort the elements as a new vector
julia> x[1] = 2; print(x)  # change the first element to 2
[2, 1, 2, 2, 3, 3, 5]
julia> x = [1, 2];
julia> y = [3, 4];
julia> x + y  # add vectors
2-element Vector{Int64}:
  4
  6
julia> 3x - [1, 2]  # multiply by a scalar and subtract
2-element Vector{Int64}:
  2
  4
julia> using LinearAlgebra
```

© 2022 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2021-11-23 19:25:37-08:00, revision 3499843e, comments to bugs@algorithmsbook.com
It is often useful to apply various functions elementwise to vectors. This is a form of broadcasting. With infix operators (e.g., +, *, and ^), a dot is prefixed to indicate elementwise broadcasting. With functions like \texttt{sqrt} and \texttt{sin}, the dot is postfix:

\begin{verbatim}
 julia> x .* y          # elementwise multiplication
 2-element Vector{Int64}:
    3
    8
 julia> x .^ 2          # elementwise squaring
 2-element Vector{Int64}:
    1
    4
 julia> sin.(x)         # elementwise application of \texttt{sin}
 2-element Vector{Float64}:
    0.8414709848078965
    0.9092974268256817
 julia> sqrt.(x)        # elementwise application of \texttt{sqrt}
 2-element Vector{Float64}:
    1.0
    1.4142135623730951
\end{verbatim}

\textbf{G.1.6 Matrices}

A \textit{matrix} is a two-dimensional array. Like a vector, it is constructed using square brackets. We use spaces to delimit elements in the same row and semicolons to delimit rows. We can also index into the matrix and output submatrices using ranges:

\begin{verbatim}
 julia> X = [1 2 3; 4 5 6; 7 8 9; 10 11 12];
 julia> typeof(X)       # a 2-dimensional array of Int64s
 Matrix{Int64} (alias for Array{Int64, 2})
 julia> X[2]            # second element using column-major ordering
 4
 julia> X[3,2]          # element in third row and second column
 8
\end{verbatim}
We can also construct a variety of special matrices and use array comprehensions:

```
 julia> Matrix(1.0I, 3, 3)     # 3x3 identity matrix
 3x3 Matrix{Float64}:
  1.0  0.0  0.0
  0.0  1.0  0.0
  0.0  0.0  1.0

 julia> Matrix(Diagonal([3, 2, 1]))  # 3x3 diagonal matrix with 3, 2, 1 on diagonal
 3x3 Matrix{Int64}:
  3  0  0
  0  2  0
  0  0  1

 julia> zeros(3, 2)            # 3x2 matrix of zeros
 3x2 Matrix{Float64}:
  0.0  0.0
  0.0  0.0
  0.0  0.0

 julia> rand(3, 2)             # 3x2 random matrix
 3x2 Matrix{Float64}:
  0.801615  0.0769258
```
Matrix operations include the following:

```julia
julia> X'          # complex conjugate transpose
3×4 adjoint(::Matrix{Int64}) with eltype Int64:
   1  4  7  10
   2  5  8  11
   3  6  9  12
julia> 3X .+ 2    # multiplying by scalar and adding scalar
4×3 Matrix{Int64}:
   5  8  11
  14 17 20
  23 26 29
  32 35 38
julia> X = [1 3; 3 1]; # create an invertible matrix
julia> inv(X)       # inversion
2×2 Matrix{Float64}:
  -0.125  0.375
   0.375 -0.125
julia> pinv(X)      # pseudoinverse (requires LinearAlgebra)
2×2 Matrix{Float64}:
  -0.125  0.375
   0.375 -0.125
julia> det(X)       # determinant (requires LinearAlgebra)
-8.0
julia> [X X]        # horizontal concatenation, same as hcat(X, X)
2×4 Matrix{Int64}:
   1  3  1  3
   3  1  3  1
julia> [X; X]       # vertical concatenation, same as vcat(X, X)
4×2 Matrix{Int64}:
   1  3
   3  1
   1  3
   3  1
julia> sin. (X)     # elementwise application of sin
2×2 Matrix{Float64}:
  0.841471  0.14112
```

© 2022 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2021-11-23 19:25:37-08:00, revision 3499843e, comments to bugs@algorithmsbook.com
0.14112  0.841471

`julia> map(sin, X)`  # elementwise application of sin
2×2 Matrix{Float64}:
  0.841471  0.14112  
  0.14112   0.841471

`julia> vec(X)`  # reshape an array as a vector
4-element Vector{Int64}:
  1
  3
  3
  1

### G.1.7 Tuples

A *tuple* is an ordered list of values, potentially of different types. They are constructed with parentheses. They are similar to vectors, but they cannot be mutated:

```julia
julia> x = ()  # the empty tuple
()
julia> isempty(x)
true
julia> x = (1, )  # tuples of one element need the trailing comma
(1,)
julia> typeof(x)
Tuple{Int64}
julia> x = (1, 0, [1, 2], 2.5029, 4.6692)  # third element is a vector
(1, 0, [1, 2], 2.5029, 4.6692)
julia> typeof(x)
Tuple{Int64, Int64, Vector{Int64}, Float64, Float64}
```

```julia
julia> x[2]
0
julia> x[end]
4.6692
julia> x[4:end]
(2.5029, 4.6692)
```

```julia
julia> length(x)
5
julia> x = (1, 2)
(1, 2)
```

```julia
julia> a, b = x;
```

```julia
julia> a
1
julia> b
2
```
G.1.8 Named Tuples

A named tuple is like a tuple, but each entry has its own name:

```julia
julia> x = (a=1, b=-Inf)
(a = 1, b = -Inf)
julia> x isa NamedTuple
true
julia> x.a
1
julia> a, b = x;
julia> a
1
julia> (; :a=>10)
(a = 10,)
julia> (; :a=>10, :b=>11)
(a = 10, b = 11)
julia> merge(x, (d=3, e=10))  # merge two named tuples
(a = 1, b = -Inf, d = 3, e = 10)
```

G.1.9 Dictionaries

A dictionary is a collection of key-value pairs. Key-value pairs are indicated with a double arrow operator `=>`. We can index into a dictionary using square brackets, just as with arrays and tuples:

```julia
julia> x = Dict();  # empty dictionary
julia> x[3] = 4  # associate key 3 with value 4
4
julia> x = Dict(3=>4, 5=>1)  # create a dictionary with two key-value pairs
Dict{Int64, Int64} with 2 entries:
  5 ⇒ 1
  3 ⇒ 4
julia> x[5]  # return the value associated with key 5
1
julia> haskey(x, 3)  # check whether dictionary has key 3
true
julia> haskey(x, 4)  # check whether dictionary has key 4
false
```
G.1.10 Composite Types

A composite type is a collection of named fields. By default, an instance of a composite type is immutable (i.e., it cannot change). We use the struct keyword and then give the new type a name and list the names of the fields:

```julia
struct A
    a
    b
end
```

Adding the keyword mutable makes it so that an instance can change:

```julia
mutable struct B
    a
    b
end
```

Composite types are constructed using parentheses, between which we pass in values for each field:

```julia
x = A(1.414, 1.732)
```

The double-colon operator can be used to specify the type for any field:

```julia
struct A
    a::Int64
    b::Float64
end
```

These type annotations require that we pass in an Int64 for the first field and a Float64 for the second field. For compactness, this book does not use type annotations, but it is at the expense of performance. Type annotations allow Julia to improve runtime performance because the compiler can optimize the underlying code for specific types.

G.1.11 Abstract Types

So far we have discussed concrete types, which are types that we can construct. However, concrete types are only part of the type hierarchy. There are also abstract types, which are supertypes of concrete types and other abstract types.

We can explore the type hierarchy of the Float64 type shown in figure G.1 using the supertype and subtypes functions.

![Figure G.1. The type hierarchy for the Float64 type.](image)
```julia
julia> supertype(Float64)
AbstractFloat
julia> supertype(AbstractFloat)
Real
julia> supertype(Real)
Number
julia> supertype(Number)
Any
julia> supertype(Any)  # Any is at the top of the hierarchy
Any
julia> using InteractiveUtils  # required for using subtypes in scripts
julia> subtypes(AbstractFloat)  # different types of AbstractFloats
4-element Vector{Any}:
  BigFloat
  Float16
  Float32
  Float64
julia> subtypes(Float64)    # Float64 does not have any subtypes
Type[]
```

We can define our own abstract types:

```julia
abstract type C end
abstract type D <: C end  # D is an abstract subtype of C
struct E <: D  # E is a composite type that is a subtype of D
  a
end
```

### G.1.12 Parametric Types

Julia supports *parametric types*, which are types that take parameters. The parameters to a parametric type are given within braces and delimited by commas. We have already seen a parametric type with our dictionary example:

```julia
julia> x = Dict(3=>1.4, 1=>5.9)
Dict{Int64, Float64} with 2 entries:
  3 => 1.4
  1 => 5.9
```

For dictionaries, the first parameter specifies the key type, and the second parameter specifies the value type. The example has `Int64` keys and `Float64` values, making the dictionary of type `Dict{Int64, Float64}`. Julia was able to infer these types based on the input, but we could have specified it explicitly:
While it is possible to define our own parametric types, we do not need to do so in this text.

### G.2 Functions

A function maps its arguments, given as a tuple, to a result that is returned.

#### G.2.1 Named Functions

One way to define a named function is to use the `function` keyword, followed by the name of the function and a tuple of names of arguments:

```julia
function f(x, y)
    return x + y
end
```

We can also define functions compactly using assignment form:

```julia
julia> f(x, y) = x + y;
julia> f(3, 0.1415)
3.1415
```

#### G.2.2 Anonymous Functions

An anonymous function is not given a name, though it can be assigned to a named variable. One way to define an anonymous function is to use the arrow operator:

```julia
julia> h = x -> x^2 + 1 # assign anonymous function with input x to a variable h
#1 (generic function with 1 method)
julia> h(3)
10
julia> g(f, a, b) = [f(a), f(b)]; # applies function f to a and b and returns array
julia> g(h, 5, 10)
2-element Vector{Int64}:
    26
    101
julia> g(x->sin(x)+1, 10, 20)
2-element Vector{Float64}:
    0.4559788891106302
    1.9129452507276277
```
G.2.3 Callable Objects

We can define a type and associate functions with it, allowing objects of that type to be callable:

```julia
julia> (x :: A)() = x.a + x.b  # adding a zero-argument function to the type A defined earlier
julia> (x :: A)(y) = y*x.a + x.b  # adding a single-argument function
julia> x = A(22, 8);
julia> x()
30
julia> x(2)
52
```

G.2.4 Optional Arguments

We can assign a default value to an argument, making the specification of that argument optional:

```julia
julia> f(x=10) = x^2;
julia> f()
100
julia> f(3)
9
julia> f(x, y, z=1) = x*y + z;
julia> f(1, 2, 3)
5
julia> f(1, 2)
3
```

G.2.5 Keyword Arguments

Functions may use keyword arguments, which are arguments that are named when the function is called. Keyword arguments are given after all the positional arguments. A semicolon is placed before any keywords, separating them from the other arguments:

```julia
julia> f(; x = 0) = x + 1;
julia> f()
1
julia> f(x = 10)
11
julia> f(x, y = 10; z = 2) = (x + y)*z;
```

© 2022 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
22

julia> f(2, z = 3)
36
julia> f(2, 3)
10
julia> f(2, 3, z = 1)
5

G.2.6 Dispatch

The types of the arguments passed to a function can be specified using the double colon operator. If multiple methods of the same function are provided, Julia will execute the appropriate method. The mechanism for choosing which method to execute is called dispatch:

julia> f(x::Int64) = x + 10;
julia> f(x::Float64) = x + 3.1415;
julia> f(1)
11
julia> f(1.0)
4.141500000000001
julia> f(1.3)
4.4415000000000004

The method with a type signature that best matches the types of the arguments given will be used:

julia> f(x) = 5;
julia> f(x::Float64) = 3.1415;
julia> f([3, 2, 1])
5
julia> f(0.00787499699)
3.1415

G.2.7 Splatting

It is often useful to *splat* the elements of a vector or a tuple into the arguments to a function using the ... operator:
### G.3 Control Flow

We can control the flow of our programs using conditional evaluation and loops. This section provides some of the syntax used in the book.

#### G.3.1 Conditional Evaluation

Conditional evaluation will check the value of a Boolean expression and then evaluate the appropriate block of code. One of the most common ways to do this is with an `if` statement:

```julia
if x < y
    # run this if x < y
elseif x > y
    # run this if x > y
else
    # run this if x == y
end
```

We can also use the *ternary operator* with its question mark and colon syntax. It checks the Boolean expression before the question mark. If the expression evaluates to true, then it returns what comes before the colon; otherwise, it returns what comes after the colon:

```julia
f(x) = x > 0 ? x : 0;
```

```julia
julia> f(-10)
0
```

```julia
julia> f(10)
10
```
G.3.2 Loops

A loop allows for repeated evaluation of expressions. One type of loop is the while loop, which repeatedly evaluates a block of expressions until the specified condition after the `while` keyword is met. The following example sums the values in the array `x`:

```julia
X = [1, 2, 3, 4, 6, 8, 11, 13, 16, 18]
s = 0
while !isempty(X)
    s += pop!(X)
end
```

Another type of loop is the for loop, which uses the `for` keyword. The following example will also sum over the values in the array `x` but will not modify `x`:

```julia
X = [1, 2, 3, 4, 6, 8, 11, 13, 16, 18]
s = 0
for y in X
    s += y
end
```

The `in` keyword can be replaced by `=` or `∈`. The following code block is equivalent:

```julia
X = [1, 2, 3, 4, 6, 8, 11, 13, 16, 18]
s = 0
for i = 1:length(X)
    s += X[i]
end
```

G.3.3 Iterators

We can iterate over collections in contexts such as for loops and array comprehensions. To demonstrate various iterators, we will use the `collect` function, which returns an array of all items generated by an iterator:

```julia
julia> X = ["feed", "sing", "ignore"];
julia> collect(enumerate(X)) # return the count and the element
3-element Vector{Tuple{Int64, String}}:
 (1, "feed")
 (2, "sing")
 (3, "ignore")
julia> collect(eachindex(X)) # equivalent to 1:length(X)
3-element Vector{Int64}:
```

© 2022 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2021-11-23 19:25:37-08:00, revision 3499843e, comments to bugs@algorithmsbook.com
G.4 Packages

A package is a collection of Julia code and possibly other external libraries that can be imported to provide additional functionality. This section briefly reviews a few of the key packages that we build upon in this book. To add a registered package like Distributions.jl, we can run

```julia
using Pkg
Pkg.add("Distributions")
```

To update packages, we use

```julia
```

```julia
Z = [1 2; 3 4; 5 6];
```

```julia
julia> Y = [-5, -0.5, 0];
julia> collect(zip(X, Y))  # iterate over multiple iterators simultaneously
3-element Vector{Tuple{String, Float64}}:
  ("feed", -5.0)
  ("sing", -0.5)
  ("ignore", 0.0)
```

```julia
julia> import IterTools: subsets
julia> collect(subsets(X))  # iterate over all subsets
8-element Vector{Vector{String}}:
  []
  ["feed"]
  ["sing"]
  ["feed", "sing"]
  ["ignore"]
  ["feed", "ignore"]
  ["sing", "ignore"]
  ["feed", "sing", "ignore"]
```

```julia
julia> collect(eachindex(X))  # iterate over indices into a collection
3-element Vector{Int64}:
  1
  2
  3
```

```julia
Z = [1 2; 3 4; 5 6];
julia> import Base.Iterators: product
julia> collect(product(X, Y))  # iterate over Cartesian product of multiple iterators
3×3 Matrix{Tuple{String, Float64}}:
   ("feed", -5.0)  ("feed", -0.5)  ("feed", 0.0)
   ("sing", -5.0)  ("sing", -0.5)  ("sing", 0.0)
   ("ignore", -5.0) ("ignore", -0.5) ("ignore", 0.0)
```
Appendix G. Julia

To use a package, we use the keyword using as follows:

typedef Distributions

G.4.1 Graphs.jl

We use the Graphs.jl package (version 1.4) to represent graphs and perform operations on them:

```
using Graphs

G = SimpleDiGraph(3);  # create a directed graph with three nodes
add_edge!(G, 1, 3);    # add edge from node 1 to 3
add_edge!(G, 1, 2);    # add edge from node 1 to 2
rem_edge!(G, 1, 3);    # remove edge from node 1 to 3
add_edge!(G, 2, 3);    # add edge from node 2 to 3

typeof(G)  
Graphs.SimpleGraphs.SimpleDiGraph{Int64}

nv(G)  
# number of nodes (also called vertices)

outneighbors(G, 1)  # list of outgoing neighbors for node 1
1-element Vector{Int64}:
  2

inneighbors(G, 1)  # list of incoming neighbors for node 1
Int64[]
```

G.4.2 Distributions.jl

We use the Distributions.jl package (version 0.24) to represent, fit, and sample from probability distributions:

```
using Distributions

dist = Categorical([0.3, 0.5, 0.2])  # create a categorical distribution

data = rand(dist)  # generate a sample

data = rand(dist, 2)  # generate two samples

μ, σ = 5.0, 2.5;  # define parameters of a normal distribution

dist = Normal(μ, σ)  # create a normal distribution
```

© 2022 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2021-11-23 19:25:37-08:00, revision 3499843e, comments to bugs@algorithmsbook.com
julia> rand(dist)  # sample from the distribution
8.257578982345834
julia> data = rand(dist, 3)  # generate three samples
3-element Vector{Float64}:
   6.2618754880077
-1.517157386600685
  1.63114704409583
julia> data = rand(dist, 1000);  # generate many samples
julia> Distributions.fit(Normal, data)  # fit a normal distribution to the samples
Distributions.Normal{Float64}(μ=4.84351796840752, σ=2.411956210019985)

julia> μ = [1.0, 2.0];
julia> Σ = [1.0 0.5; 0.5 2.0];
julia> dist = MvNormal(μ, Σ)  # create a multivariate normal distribution
FullNormal(
  dim: 2
  μ: [1.0, 2.0]
  Σ: [1.0 0.5; 0.5 2.0]
)

julia> rand(dist, 3)  # generate three samples
2×3 Matrix{Float64}:
-0.882292  1.66083  -0.378057
  0.198389  2.42178   0.513822

julia> dist = Dirichlet(ones(3))  # create a Dirichlet distribution Dir(1,1,1)
Distributions.Dirichlet{Float64, Vector{Float64}, Float64}(alpha=[1.0, 1.0, 1.0])

julia> rand(dist)  # sample from the distribution
3-element Vector{Float64}:
  0.2410048432097217
  0.39992304967730224
  0.35907210711297605

G.4.3 JuMP.jl

We use the JuMP.jl package (version 0.21) to specify optimization problems that we can then solve using a variety of solvers, such as those included in GLPK.jl and Ipopt.jl:

julia> using JuMP
julia> using GLPK
julia> model = Model(GLPK.Optimizer)  # create model and use GLPK as solver
A JuMP Model
Feasibility problem with:
  Variables: 0
  Model mode: AUTOMATIC
  CachingOptimizer state: EMPTY_OPTIMIZER
Solver name: GLPK

```julia
julia> @variable(model, x[1:3])  # define variables x[1], x[2], and x[3]
3-element Vector{JuMP.VariableRef}:
x[1]
x[2]
x[3]

julia> @objective(model, Max, sum(x) - x[2])  # define maximization objective

julia> @constraint(model, x[1] + x[2] ≤ 3)  # add constraint

julia> @constraint(model, x[2] + x[3] ≤ 2)  # add another constraint

julia> @constraint(model, x[2] ≥ 0)  # add another constraint
x[2] ≥ 0.0

julia> optimize!(model)  # solve
julia> value.(x)  # extract optimal values for elements in x
3-element Vector{Float64}:
 3.0
 0.0
 2.0
```

### G.5 Convenience Functions

There are a few functions that allow us to specify the algorithms in this book more compactly. Julia 1.7 will support a two-argument version of `findmax`, where we can pass in a function and a collection. It returns the maximum of the function when evaluated on the elements of the collection, along with the first maximizing element. The `argmax` function is similar, but it returns only the first maximizing element. To support this in Julia 1.6, we manually extend these functions:

```julia
function Base.findmax(f::Function, xs)
    f_max = -Inf
    x_max = first(xs)
    for x in xs
        v = f(x)
        if v > f_max
            f_max, x_max = v, x
        end
    end
    return f_max, x_max
end

Base.argmax(f::Function, xs) = findmax(f, xs)[2]
```

© 2022 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2021-11-23 19:25:37-08:00, revision 3499843e, comments to bugs@algorithmsbook.com
The following functions are useful when working with dictionaries and named tuples:

```julia
julia> findmax(x->x^2, [0, -10, 3])
(100, -10)
julia> argmax(abs, [0, -10, 3])
-10
```

We define `SetCategorical` to represent distributions over discrete sets:

```julia
julia> a = Dict{Symbol, Integer}((a=1, b=2, c=3))
Dict{Symbol, Integer} with 3 entries:
 :a => 1
 :b => 2
 :c => 3
julia> isequal(a, (a=1, b=2, c=3))
true
julia> isequal(a, (a=1, c=3, b=2))
true
julia> Dict(Dict{Symbol, Integer}, Float64)(((a=1, b=1)↦0.2, (a=1, b=2)↦0.8))
Dict{Dict{Symbol, Integer}, Float64} with 2 entries:
 Dict(:a=1, :b=1) => 0.2
 Dict(:a=1, :b=2) => 0.8
```

```julia
struct SetCategorical{S}
    elements::Vector{S}  # Set elements (could be repeated)
    distr::Categorical   # Categorical distribution over set elements

    function SetCategorical(elements::AbstractVector{S}) where S
        weights = ones(length(elements))
        return new{S}(elements, Categorical(normalize(weights, 1)))
    end

    function SetCategorical(elements::AbstractVector{S}, weights::AbstractVector{Float64}) where S
        ℓ₁ = norm(weights,1)
        if ℓ₁ < 1e-6 || isinf(ℓ₁)
            return SetCategorical(elements)
        end
```

© 2022 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.
2021-11-23 19:25:37-08:00, revision 3499843e, comments to bugs@algorithmsbook.com
end
distr = Categorical(normalize(weights, 1))
return new{S}(elements, distr)
end
end

Distributions.rand(D::SetCategorical) = D.elements[rand(D.distr)]
Distributions.rand(D::SetCategorical, n::Int) = D.elements[rand(D.distr, n)]
function Distributions.pdf(D::SetCategorical, x)
    sum(e == x ? w : 0.0 for (e, w) in zip(D.elements, D.distr.p))
end

julia> D = SetCategorical(["up", "down", "left", "right"],[0.4, 0.2, 0.3, 0.1]);
julia> rand(D)
"up"

julia> rand(D, 5)
5-element Vector{String}:
  "right"
  "up"
  "up"
  "right"
  "left"

julia> pdf(D, "up")
0.3999999999999999

© 2022 Massachusetts Institute of Technology, shared under a Creative Commons CC-BY-NC-ND license.