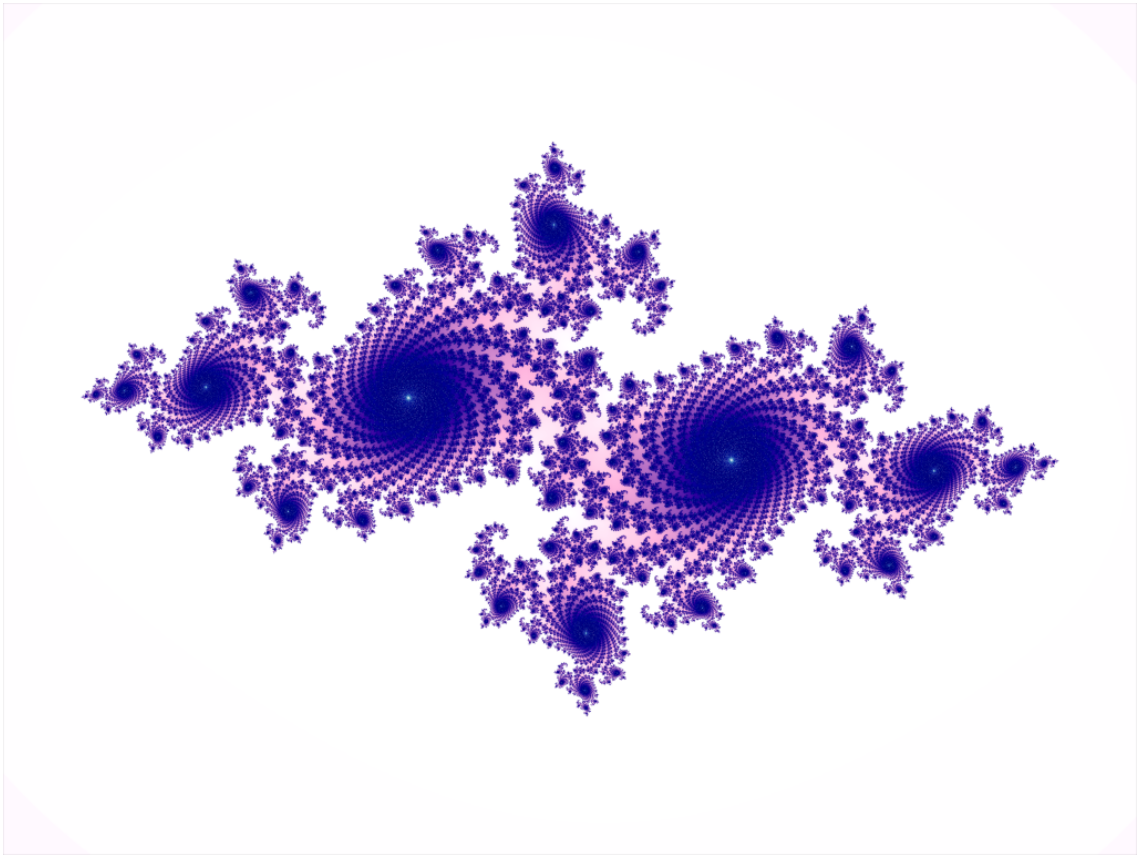
# Train Your Brain

—

# Challenging Yet Elementary Mathematics

*A Julia language companion*

Bogumił Kamiński, Paweł Prałat



# Contents

# 1 Introduction

This document (available on-line at `www.ryerson.ca/train-your-brain/`) is the Julia language companion to the book "Train Your Brain — Challenging Yet Elementary Mathematics". In the book we occasionally used computer programs to help us solve the problem at hand or to get a better understanding of it. Using computer-assisted arguments becomes more and more common in mathematics these days. In particular, within the scope of the book, this kind of support of a computer is most common in problems involving combinatorics and number theory.

In this on-line companion, we give a very brief introduction to the Julia language so that one can run the examples given in the book and experiment with them by herself/himself. Of course, it is not intended to be a comprehensive manual of the Julia language. For that purpose, the reader is directed to resources that are available online; for example, `https://julialang.org/learning/` is a good reference. Moreover, the official Julia manual available at `https://docs.julialang.org/` is an excellent book that can be used to study the language.

On the cover page we have included a plot of the famous Julia set. If you would like to learn more about such mathematical objects and how to plot them using the Julia language visit this website: `https://crucialflow.com/Fatou.jl/`.

# 2 General information about the Julia language

## 2.1 Installing Julia

In order to install Julia and the libraries used in the book please follow the following steps. The Julia installer can be downloaded from `https://julialang.org/downloads/`. We recommend installing the latest version marked as *stable*. The detailed platform-specific installation instructions are provided in `https://julialang.org/downloads/platform.html`.

Next, we need to start a new Julia session by opening up a terminal emulator on your computer and writing `julia`. (One might need to set a path to the Julia executable for this command to work or prepend the `julia` command with the full path to the executable.). Here we show how it should look like (the version number of your Julia installation might be different—1.5.2 is a stable version at the moment of writing of this manual):

```
~$ julia
               _
   _       _ _(_)_     |  Documentation: https://docs.julialang.org
  (_)     | (_) (_)    |
   _ _   _| |_  __ _   |  Type "?" for help, "]?" for Pkg help.
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 1.5.2 (2020-09-23)
 _/ |\__'_|_|_|\__'_|  |  Official https://julialang.org/ release
|__/                   |

julia>
```

In some of our examples we used additional packages, like *Combinatorics.jl* package, that are not automatically installed. In order to add a package, one needs to run the following commands in Julia's terminal (here we add *Combinatorics.jl* package):

```
julia> using Pkg
julia> Pkg.add("Combinatorics")
```

This is a standard way of adding packages to the Julia installation. Let us point out that we used `julia>` prefix. You do *not* need to type it—it is a prompt that Julia displays by default when it is

running. We retain it in the examples in the book to maintain clarity which commands should be written into the terminal. They are always shown after the prompt.

In order to exit Julia, one needs to write `exit()` in the command prompt:

```
julia> exit()
~$
```

## 2.2 Your first Julia session

Let us first define a variable `x` to point to a value of `100`:

```
julia> x = 100
```

Now, we define function `f` that takes one integer argument `n` and checks if it is a prime number:

```
julia> function f(n::Int)
    if n < 2
        return false
    end
    for i in 2:n-1
        if mod(n, i) == 0
            return false
        end
    end
    return true
end
```

If we pass a number that less than `2`, then `false` is returned, as clearly the number is not prime (the smallest prime number is 2). Otherwise, the function traverses all natural numbers from `2` to `n-1` and checks if they divide `n`. If any divisor is found, then `false` is returned; otherwise, `true` is returned.

In this introductory example we see that standard control flow structures start with keywords such as `if` (conditional), `for` (loop), `function` (function definition) and are finished with `end` keyword. In order to return a value from the function, we use `return` keyword followed by a value that we would like to be returned.

Note that the signature of the function is `function f(n::Int)`, which means that its name is `f` and it takes one argument `n` whose type must be `Int`. As we will learn soon, `Int` is a way to indicate that we want it to be an integer number (more details on this type are explained later in the on-line companion).

Let us check how our function works:

```
julia> f(7)
true

julia> f(10)
false

julia> f(x)
false

julia> f(0.5)
ERROR: MethodError: no method matching f(::Float64)
Closest candidates are:
  f(::Int64) at REPL[3]:2
Stacktrace:
 [1] top-level scope at none:0
```

It is worth noting that we could pass variable `x` that we defined earlier as an argument to our function `f`. Also, because we have restricted our function to work only with integers, the call `f(0.5)` produced an error as we tried to pass a floating point number.

It is important to know what variable names are allowed in Julia. Variable names must begin with a letter (A-Z or a-z) or underscore. Subsequent characters may also include ! and digits. In general, Julia allows the usage of a subset of Unicode code points other than ASCII characters in variable names, but do not use them in the book.

Before we finish let us learn how to get help in Julia. When you are in a Julia prompt like this:

```
julia>
```

presess a `?` key (question mark) on your keyboard. The prompt will change to:

```
help?>
```

Now type the name of command you want to investigate. Assume we want to learn more what `Int` means, so we type it and press enter to get:

```
help?> Int
search: Int Int8 Int64 Int32 Int16 Int128 Integer intersect intersect!

  Int64 <: Signed

  64-bit signed integer type.

julia>
```

In this way you can get help about functions, types and constants defined in Julia.

When finishing working with Julia remember to exit it using the `exit` function:

```
julia> exit()
~$
```

In this way you have successfully finished your first session using the Julia language.

## 2.3 Working with Julia using scripts

Instead of using Julia interactively, as described in Subsection 2.2, one may also use it in batch mode by processing script files. Julia scripts are typically named with `.jl` suffix. In order to test how batch processing of scripts works, create an example `test.jl` file containing the following code:

```
println("Hello! Let us calculate some factorials")

for i in 0:5
    println(i, ": ", factorial(i))
end

println("Good bye!")
```

Now, open a terminal and run `julia test.jl` command, making sure that you execute these commands in the directory where `test.jl` file is located. Your results should be similar to this one:

```
~$ julia test.jl
Hello! Let us calculate some factorials
0: 1
1: 1
```

```
2: 2
3: 6
4: 24
5: 120
Good bye!
~$
```

Note that after finishing, Julia terminates and returns control to the shell. Another way to execute this script is to start Julia first and then use the `include` function like this:

```
~$ julia
               _
   _       _ _(_)_     |  Documentation: https://docs.julialang.org
  (_)     | (_) (_)    |
   _ _   _| |_  __ _   |  Type "?" for help, "]?" for Pkg help.
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 1.4.0 (2020-03-21)
 _/ |\__'_|_|_|\__'_|  |  Official https://julialang.org/ release
|__/                   |

julia> include("test.jl")
Hello! Let us calculate some factorials
0: 1
1: 1
2: 2
3: 6
4: 24
5: 120
Good bye!

julia> exit()
~$
```

This time, after executing the `include("test.jl")` command, Julia stays in interactive mode (so that, for example, one could `include` another script). In order to return to the shell, `exit()` call has to be used.

In the book we present all examples in the interactive mode but, alternatively, one could save them to scripts that are run the way we described above. The only difference is that if you run a script you have to use the `println` function (or other functions that print data) to see any result on the screen—exactly as we did in the example above. The reason is that in the interactive mode Julia prints the results of executed code to the terminal whereas in the batch mode nothing is printed unless the programmer explicitly asks for it. Actually, it is possible to suppress printing an outcome of the computations also in the interactive mode by writing `;` at the end of the executed code. Here is an example:

```
julia> sin(100)
-0.5063656411097588

julia> sin(100);

julia>
```

Now, you should be ready to use Julia programs that other people have written. Let us then teach you how to write simple programs by yourself!

## 2.4 Key scalar types

### 2.4.1 Numeric types

Julia defines multiple scalar numeric types. In this introduction, we will focus on most important ones that represent integer numbers, floating point numbers, rational numbers, and Boolean values.

**Integer values**

Integer number literals consist of digits and optionally a `-` character to indicate a negative value. Also, one may use `_` character as a visual separator (this sign is ignored). Here are examples of integer literals:

```
$ julia
julia> 10
10

julia> -10
-10

julia> -1_000_000
-1000000
```

In the book, we assume that you work on 64-bit computer. In such case, the type of all above literals is called `Int64`, or `Int` (note that on 32-bit machine they would be 32-bit instead and have `Int32` type, that would still be called `Int` for short). You can check it using `typeof` function:

```
julia> typeof(10)
Int64
```

The range of values represented by the `Int64` type (or briefly `Int`) can be checked using the `typemax` and `typemin` functions:

```
julia> typemax(Int)
9223372036854775807

julia> typemin(Int)
-9223372036854775808
```

It is important to remember that operations on integer numbers can overflow, which is silently accepted:

```
julia> a = typemax(Int)
9223372036854775807

julia> b = typemin(Int)
-9223372036854775808

julia> a + 1
-9223372036854775808

julia> a + 1 == b
true

julia> b - 1
9223372036854775807

julia> b - 1 == a
true
```

This range of `Int` type should be enough for most practical applications. However, if for some reason it is not enough, then one may use integers having `BigInt` type. The easiest way to define them is by using `big` function on some integer value. Here is an example:

```
julia> x = 10
10

julia> typeof(x)
Int64

julia> y = big(x)
10

julia> typeof(y)
BigInt

julia> x^50
-5376172055173529600

julia> y^50
100000000000000000000000000000000000000000000000000
```

Operations on `BigInt` type do not overflow but, as a trade-off, they are slower. Values having `BigInt` type can be arbitrarily large (their size is limited only by the amount of memory of your computer).

**Floating point values**

Floating point literals are defined in the same way as integer literals except that they require a `.` to indicate a decimal separator. Also scientific notation using `e` is supported:

```
julia> 1.0
1.0

julia> 1.
1.0

julia> 1e0
1.0

julia> 1_234.56
1234.56

julia> 12e3
12000.0

julia> 12e34
1.2e35

julia> typeof(12e34)
Float64
```

The basic type used to represent floating point numbers is `Float64` and it uses CPU in-built floating point arithmetic. There are special values of `Inf`, `-Inf` and `NaN` representing positive infinity, negative infinity, and not-a-number values. Also writing `-0.0` allows to define negative zero. All these conventions follow IEEE 754 standard (see e.g. `https://en.wikipedia.org/wiki/IEEE_754` for a reference).

Floating point numbers have a finite representation. Therefore `eps` function is provided so that you can check, given a floating point value `x`, what is the distance to next larger value representable using the type of `x`. Here are some examples:

```
julia> eps(1.0)
2.220446049250313e-16

julia> eps(10.0)
1.7763568394002505e-15

julia> eps(1000.0)
1.1368683772161603e-13

julia> 10.0 + eps(10.0)
10.000000000000002

julia> 10.0 + eps(10.0) == 10.0
false

julia> 10.0 + eps(10.0) / 2
10.0

julia> 10.0 + eps(10.0) / 2 == 10.0
true
```

Here are maximum and minimum values representable using `Float64` type:

```
julia> prevfloat(Inf)
1.7976931348623157e308

julia> nextfloat(-Inf)
-1.7976931348623157e308
```

Similarly, here are the smallest (in absolute values) representable using `Float64` type:

```
julia> nextfloat(0.0)
5.0e-324

julia> prevfloat(0.0)
-5.0e-324
```

As in the case for integers, if one needs more precision or larger magnitude of values, then `BigFloat` type can be used instead. Again, the simplest way to get a value of this type is to use `big` function on a floating point value.

```
julia> x = 10.0
10.0

julia> typeof(x)
Float64

julia> y = big(x)
10.0

julia> typeof(y)
BigFloat
```

```
julia> x^1000
Inf

julia> y^1000
1.000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000004e+1000
```

One may use the `setprecision` function to change the precision of computations using `BigFloat` values.

**Rational numbers**

For some applications, it might be convenient to use exact rational arithmetic. This can be easily done in Julia using `//` operator. Here is a simple example of performing arithmetic operations on rational numbers:

```
julia> x = 1//2
1//2

julia> y = 1//3
1//3

julia> x + y
5//6

julia> 5x + y/7
107//42
```

One may easily convert a rational number to a floating point number using the `float` function (it also can be used to convert other numeric types to floating point numbers):

```
julia> float(5x + y/7)
2.5476190476190474
```

In order to see the difference between exact and approximate floating point arithmetic consider the following example:

```
julia> float(1//2) + float(1//3) - float(5//6)
-1.1102230246251565e-16

julia> 1//2 + 1//3 - 5//6
0//1
```

In the first case we use floating point numbers, so the result is only close to 0. In the second case we use rational numbers and the result is exactly 0.

Working with rational numbers has a clear advantage but one should keep in mind that operations on rational numbers are much slower than their counterparts on floating point numbers. Also, if you perform intensive computations using rational numbers, then you could easily get a denominator that is too large to be stored as a `Int64` number. For instance,

```
julia> 1//1_000_000_000_001 + 1//1_000_000_000_003
ERROR: OverflowError: 1000000000001 *y overflowed for type Int64
```

In such cases, one should use the `big` version of integer numbers to create the rational numbers:

```
julia> 1//big(1_000_000_000_001) + 1//big(1_000_000_000_003)
2000000000004//1000000000004000000000003
```

Julia automatically keeps track of what kind of rational number (lower precision but faster, or higher precision but slower) you requested.

### Boolean values

Boolean values have type `Bool` and take only two values, namely, `true` and `false`. In arithmetic operations, `true` is treated as `1` whereas `false` is treated as `0`. Here is an example:

```
julia> x = true
true

julia> typeof(x)
Bool

julia> y = false
true

julia> 10 + true
11

julia> 100 * false
0
```

The important usage of `Bool` type are logical conditions, for example, `if` statements or `while` loops. Therefore, the value of standard comparison operators such as `==` (equality), or `<`, `>`, `<=`, `>=` are normally of type `Bool`:

```
julia> 10 == 11
false

julia> 10 < 11
true

julia> if 10 < 11
    "10 less than 11"
else
    "10 is not less than 11"
end
"10 less than 11"

julia> if 1
    "this should not execute"
end
ERROR: TypeError: non-boolean (Int64) used in boolean context
Stacktrace:
 [1] top-level scope at none:0
```

In particular, note that the last example produced an error because we tried to use an `Int64` value in logical test.

Logical values can be combined using standard operators such as *not*, *and*, and *or*. These operations are respectively denoted by `!`, `&&` and `||`:

```
julia> !true
false

julia> true && false
```

```
false

julia> true || false
true
```

It is important to know that when using `&&` and `||`, the following two rules are applied: (a) minimal evaluation, and (b) their last argument does not have to have `Bool` type (but only if the value of the expression is not used later in logical operations). By minimal evaluation (also called short-circuit evaluation, see `https://en.wikipedia.org/wiki/Short-circuit_evaluation`) we mean that the second argument of a boolean operation is executed only if the first argument does not suffice to determine the logical value of the expression. Here are examples of these rules applied:

```
julia> true && println("println was called")
println was called

julia> true || println("println was called")
true

julia> false && println("println was called")
false

julia> false || println("println was called")
println was called
```

(Note that the `println` function prints a value to a standard output.) This, in particular, means that writing `condition && expression` is equivalent to

```
if condition
    expression
end
```

and writing `condition || expression` is equivalent to

```
if !condition
    expression
end
```

This pattern can be often encountered in actual Julia code so it is worth to learn it. Here is a simple example of its application:

```
julia> function count_even(n)
    even_count = 0
    for i in 1:n
        iseven(i) && (even_count += 1)
    end
    even_count
end

julia> count_even(10)
5
```

There are two things in this example that are worth highlighting. The `condition && expression` required us to take `even_count += 1` expression that increases `even_count` variable by `1` in parenthesis (the reason is operator precedence in Julia). Secondly, as the last expression in our function definition was simply `even_count` we did not have to write `return` keyword in front of it (implicitly function returns the value of last expression evaluated inside its body).

Finally, let us note that similarly to `+=` one can use `-=`, `*=`, and `/=` to decrease, multiply and, respectively, divide a variable by some value in place.

### 2.4.2 Strings, characters and symbols

Strings are designed to hold textual information. In the book, we do not work with processing strings so we only provide a minimal information about them. First, let us mention that one can hold any binary stream as a string. By default, Julia interprets strings as UTF-8 encoded. String literals are delimited in quotes ", for example, `"Hello \"World\"\n"`. As you can see Julia uses C-style escape style, in particular, `\n` is used to represent a newline in the string. One can also use triple-quoted string literals, if many newlines or quotation marks are desired in the string, for example:

```
julia> """Hello
          "World",
       """
"Hello\n   \"World\",\n"
```

Julia provides a handful of useful built-in operations on strings, most importantly, concatenation using `*` and variable interpolation using `$`, for example:

```
julia> x = "Alice"
"Alice"

julia> y = "Bob"
"Bob"

julia> x * " met " * y
"Alice met Bob"

julia> "$x met $y"
"Alice met Bob"
```

If one wants to create a string that does not use escaping sequences nor interpolation, then such literals are created by prepending `raw` before the string, for example:

```
julia> raw"$x = $y \n"
"\$x = \$y \\n"
```

One can get access to characters contained in a string using the `collect` function. As a result, one gets an array of them. Note that characters have a distinct type in Julia which is `Char`.

```
julia> x = collect("Alice")
5-element Array{Char,1}:
 'A'
 'l'
 'i'
 'c'
 'e'

julia> x[1]
'A': ASCII/Unicode U+0041 (category Lu: Letter, uppercase)
```

Finally, sometimes one might encounter a value that looks like `:Alice`. Such values are called symbols and have `Symbol` type.

```
julia> typeof(:Alice)
Symbol
```

You can think of them as strings that are very fast to compare for equality, so they are sometimes useful. Actually, Julia internally represents variable names, etc., as symbols, but we will not use this feature in the book. It is easy to convert strings to symbols and symbols to string using the `Symbol` and `String` functions:

```
julia> typeof(:Alice)
Symbol

julia> Symbol("Alice")
:Alice

julia> String(:Alice)
"Alice"

julia> Symbol("Alice met Bob")
Symbol("Alice met Bob")
```

The last example uses a symbol that is not a valid variable name (recall that variable names do not allow spaces in them and symbols are internally used to represent variable names). You can still construct and use such symbol, but the distinction is that it is not displayed with `:` prefix (this notation is reserved for symbols that are valid variable names).

### 2.4.3 Special types `Nothing`, `Missing`, and `Any`

Occasionally, one might encounter the following values that are subtle but important to understand. The first of them is `nothing` that has type `Nothing`. It is used to represent *absence* of value (we know that such value does not exist). To check that some value has this type you can use the `isnothing` function. Here is an example:

```
julia> findfirst([true, false])
1

julia> findfirst([false, false])

julia> isnothing(findfirst([false, false]))
true
```

The `findfirst` function tries to locate the first `true` in a collection. If it succeeds, then it returns the index of the first `true`. However, if such index is *absent*, then it returns `nothing`. Notice, by investigating the message after `findfirst([false, false])` command, that `nothing` does not get printed to the console by default. The second special value is `missing` that has type `Missing`. It is used to represent *unknown* value (we know that such value does exists, but we do not know it). Typically you encounter such value when reading data in from some source, for example, for a collection of points on the map for some of them we might not have an information about their elevation (we know this value exists, but we simply have no information about it). In order to check that some value has this type you can use the `ismissing` function. Additionally, if `missing` is encountered in comparisons, then it produces `missing` value (that is, Julia implements three-valued logic for this type). Here are some examples:

```
julia> ismissing(1)
false

julia> ismissing(missing)
true

julia> 1 == missing
missing

julia> 1 > missing
```

```
missing

julia> 1 > missing
missing
```

The third special type that is important to know is `Any`. No value has this type (it is a so-called abstract type), but this type can be used to indicate that we accept any type in some place of the program.

## 2.5 Functions

The most basic form of function signature is using the `function` keyword. Here is a simple function definition:

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

This function takes three arguments and returns their sum. One can omit `return` keyword indicating a function return value if this is the last executed statement in the function body. Short functions can be alternatively defined using one-line notation using the `=` symbol:

```
julia> f(x, y, z) = x + y + z
f (generic function with 1 method)

julia> f(1, 2, 3)
6
```

One can specify default values for the function arguments as follows:

```
julia> g(x, y=0, z=0) = x + y + z
g (generic function with 3 methods)

julia> g(1, 2, 3)
6

julia> g(1, 2)
3

julia> g(1)
1

julia> g()
ERROR: MethodError: no method matching g()
Closest candidates are:
  g(::Any) at REPL[3]:1
  g(::Any, ::Any) at REPL[3]:1
  g(::Any, ::Any, ::Any) at REPL[3]:1
Stacktrace:
 [1] top-level scope at none:0
```

Note that we have to pass the first argument to the function as we have not specified a default value for it. Additionally, in case of an error, Julia informs us what kind of calls (called methods in Julia) are defined for the function. Alternatively, one can get a complete list of methods using the `methods` function:

```
julia> methods(g)
# 3 methods for generic function "g":
[1] g(x) in Main at REPL[3]:1
[2] g(x, y) in Main at REPL[3]:1
[3] g(x, y, z) in Main at REPL[3]:1
```

Moreover, in the error message above, we are informed that the function accepts arguments of any type, which is indicated by `::Any`. However, if needed, one can easily restrict a function to accept only specific argument types. This can be done by adding `::Type` condition after a variable, where `Type` is a name of type we want to use. Here is a simple example:

```
julia> function h(c::Bool, x::Float64)
           if c
               return sqrt(x)
           else
               return x^2
           end
       end
h (generic function with 1 method)

julia> methods(h)
# 1 method for generic function "h":
[1] h(c::Bool, x::Float64) in Main at REPL[9]:2

julia> h(true, 10.0)
3.1622776601683795

julia> h(true, 10)
ERROR: MethodError: no method matching h(::Bool, ::Int64)
Closest candidates are:
  h(::Bool, ::Float64) at REPL[9]:2
Stacktrace:
 [1] top-level scope at none:0
```

This example shows that using overly specific types in method signature is often not a good idea. Despite the fact that there is no performance penalty in leaving out the type restriction, such restrictions are valuable when one wants to provide a different behaviour of a function conditional on the type of the passed variable. It is best understood with an example:

```
julia> t(x::Int) = "got an integer"
t (generic function with 1 method)

julia> t(x::String) = "got a string"
t (generic function with 2 methods)

julia> t(x) = "got an unsupported value type"
t (generic function with 3 methods)

julia> t(1)
"got an integer"

julia> t("Alice")
"got a string"

julia> t(1.0)
"got an unsupported value type"
```

Julia also allows us to capture the type information using the `where` keyword in the signature of a function. Here is a simple example (note that after `where` the list of variables capturing argument types are enclosed in curly braces):

```
julia> gettype(x::T) where {T} = T
gettype (generic function with 1 method)

julia> gettype(1)
Int64

julia> typeof(1)
Int64
```

The `gettype` function returns a type of an argument passed to it. Essentially, above we have redefined a built-in function `typeof`. Type extraction could be useful if one wants to construct a container that is able to hold values of the type of passed argument (we will discuss the container types in the next subsection in more detail). Here is a simple example—the `getcontainer` function below creates an empty vector that is able to hold values of the type of the argument passed to it.

```
julia> getcontainer(x::T) where {T} = T[]
getcontainer (generic function with 1 method)

julia> getcontainer("Hello!")
0-element Array{String,1}
```

As an additional feature, one can define functions to have keyword arguments. Keyword arguments are most useful in functions that have a large of arguments and most of them have some natural default values. This situation is often encountered in plotting functions when one has a large number of various options that affect how one wants the plot to be shown. They are defined after `;` in the function signature. The difference between positional arguments (that we already learned) and keyword arguments is twofold: (a) keyword arguments need to have the argument name passed when you call a function, and (b) keyword arguments can be passed in any order (as opposed to positional arguments). Let us show this in practice by considering the following example:

```
julia> function circle(x, y; radius=1, color="white", border="black")
           println("plotting $color circle with $border border and " *
                    "$radius radius at location ($x, $y)")
       end
circle (generic function with 1 method)

julia> circle(1,1)
plotting white circle with black border and 1 radius at location (1, 1)

julia> circle(1,1, color="black")
plotting black circle with black border and 1 radius at location (1, 1)

julia> circle(1,1, color="black", radius=100)
plotting black circle with black border and 100 radius at location (1, 1)
```

In this example we used string concatenation operator `*` to split a long string into two lines. Moreover, note that we provided a default value for each keyword argument, for example, `radius=1` (which is a common practice). However, default values for keyword arguments may be omitted when defining the function, for example, we may have function like this: `circle(x, y; radius, color, border)`.

Sometimes one wants to define a function that does not need a name but is directly used or passed around. This can be easily achieved via anonymous functions. Their basic syntax is the following

`arguments -> body`. For example `count` function assumes that its first argument is a function and the second argument is a collection. It returns the number of times the function evaluates to `true` when being consecutively passed the elements of the collection. It is often useful to use anonymous functions in such situations. Consider the following example where we count the number of zeros in the collection:

```
julia> count(x -> x == 0, [0, 1, 2, 3, 2, 1, 0])
2
```

Interestingly, the `x -> x == some_value` pattern is so common in practice that in Julia it has a shorthand syntax `==(some_value)` which expands to the anonymous function defined earlier. Therefore, the example above could alternatively be written as follows:

```
julia> count(==(0), [0, 1, 2, 3, 2, 1, 0])
2
```

Finally, it is good to know that functions can be assigned to variables like any other values. This is especially useful with anonymous functions. Let us illustrate this technique on the above example:

```
julia> eq0 = ==(0)
(::Base.Fix2{typeof(==),Int64}) (generic function with 1 method)

julia> eq0(10)
false

julia> eq0(0)
true

julia> count(eq0, [0, 1, 2, 3, 2, 1, 0])
2
```

As a final note, observe that many functions take a function as their first argument. In such cases, one may conveniently define this function as anonymous using a so called `do` block. For example, function `map` has the following signature `map(eq0, collection)` and applies `eq0` to all elements of the `collection`. Consider the following example to understand how it works:

```
julia> map(x -> x^2, 1:3)
3-element Array{Int64,1}:
 1
 4
 9
```

You can equivalently write it like this using a `do` block. This syntax is particularly useful if you have multiple statements in your function.

```
julia> map(1:3) do x
           x^2
       end
3-element Array{Int64,1}:
 1
 4
 9
```

## 2.6 Most common container types

### 2.6.1 Tuples and NamedTuples

Tuples in Julia are fixed-size, immutable, one dimensional collections that can hold heterogeneous types of data. They are created using parenthesis `(` and `)` one can access their elements using indexing. Below we create zero, one, and two element tuples:

```
julia> ()
()

julia> (1,)
(1,)

julia> (1, "2")
(1, "2")
```

Here is an example how to access a tuple:

```
julia> x = (1,2,3)
(1, 2, 3)

julia> x[1]
1

julia> x[4]
ERROR: BoundsError: attempt to access (1, 2, 3)
  at index [4]
Stacktrace:
 [1] getindex(::Tuple, ::Int64) at .\tuple.jl:24
 [2] top-level scope at none:0

julia> x[1] = 10
ERROR: MethodError: no method matching
setindex!(::Tuple{Int64,Int64,Int64}, ::Int64, ::Int64)
Stacktrace:
 [1] top-level scope at none:0
```

`NamedTuple` can be thought of as a generalization of a tuple to allow assigning names to its elements. Their construction is similar to tuples with the only difference that now each entry of a named tuple has to have a form `name=value`. Here are some examples:

```
julia> nt = (a=1, b="s")
(a = 1, b = "s")

julia> nt[1]
1

julia> nt.a
1
```

### 2.6.2 Arrays

Arrays are multidimensional collections that are mutable (that is, their values can change) and that can hold a specified type of values. Here are some basic examples:

```
julia> vector = [1, 2, 3]
3-element Array{Int64,1}:
 1
 2
 3

julia> matrix = [true false false
                 false true false]
2x3 Array{Bool,2}:
  true  false  false
 false   true  false

julia> vector2 = [1, "a"]
2-element Array{Any,1}:
 1
  "a"
```

Note that each array is described by `Array{element type,dimension}` signature (actually, this is the type of the container). In particular, `vector2` has element type equal to `Any`, which means that you can store any values in it. Here are some more examples of investigating the structure of an array, accessing its elements and changing their values:

```
julia> ndims(vector)
1

julia> size(vector)
(3,)

julia> eltype(vector)
Int64

julia> ndims(matrix)
2

julia> size(matrix)
(2, 3)

julia> eltype(matrix)
Bool

julia> vector[1]
1

julia> matrix[2, 2]
true

julia> vector2[1]
1

julia> vector[1] = 100
100

julia> vector
3-element Array{Int64,1}:
 100
   2
```

```
   3

julia> vector[1] = "a"
ERROR: MethodError: Cannot ‘convert‘ an object of type String
to an object of type Int64
Closest candidates are:
  convert(::Type{T<:Number}, ::T<:Number) where T<:Number at number.jl:6
  convert(::Type{T<:Number}, ::Number) where T<:Number at number.jl:7
  convert(::Type{T<:Integer}, ::Ptr) where T<:Integer at pointer.jl:23
  ...
Stacktrace:
 [1] setindex!(::Array{Int64,1}, ::String, ::Int64) at .\array.jl:767
 [2] top-level scope at none:0

julia> vector2[1] = missing
missing

julia> vector2
2-element Array{Any,1}:
 missing
 "a"
```

In particular, note that we could not assign value `"a"` to a vector `vector` because it can hold only integer values. On the other hand, it was possible to do for `vector2` because it can hold any value (its element type is `Any`).

One can easily generate pseudo random arrays using the `rand` and `randn` functions (the first one generates uniformly at random a number from the $[0, 1)$ interval whereas the second one generates a number from the standard normal distribution):

```
julia> x = rand(3)
3-element Array{Float64,1}:
 0.5005929508120419
 0.9282967466111087
 0.9374989389229635

julia> y = rand(3,2)
3x2 Array{Float64,2}:
 0.408122   0.794773
 0.0583201  0.006015
 0.738905   0.25786
```

You can expect get different numbers than what we present here. It is also easy to create arrays containing only zeros or ones:

```
julia> zeros(4)
4-element Array{Float64,1}:
 0.0
 0.0
 0.0
 0.0

julia> ones(3,2)
3x2 Array{Float64,2}:
 1.0  1.0
 1.0  1.0
 1.0  1.0
```

A special type of read-only, one-dimensional arrays are ranges. The most common way to specify them is by using the `start:stop` form or by the `start:step:stop` form. One can convert them to standard arrays using the `collect` function, however, it is usually not needed so use it only if you need to mutate their contents later:

```
julia> 1:4
1:4

julia> 1:2:9
1:2:9

julia> collect(1:4)
4-element Array{Int64,1}:
 1
 2
 3
 4

julia> collect(1:2:7)
5-element Array{Int64,1}:
 1
 3
 5
 7
```

Finally, when dealing with arrays of numbers, one may use standard linear algebra operations; in such operations, vectors are considered to be column vectors:

```
julia> x = [1,2,3]
3-element Array{Int64,1}:
 1
 2
 3

julia> y = ones(2, 3)
2x3 Array{Float64,2}:
 1.0  1.0  1.0
 1.0  1.0  1.0

julia> y * x
2-element Array{Float64,1}:
 6.0
 6.0
```

The way to transpose a vector or a matrix is to use the `permutedims` function:

```
julia> permutedims(x)
1x3 Array{Int64,2}:
 1  2  3

julia> permutedims(y)
3x2 Array{Float64,2}:
 1.0  1.0
 1.0  1.0
 1.0  1.0
```

One may check what kind of elements an array holds by using the `eltype` function. Sometimes it is useful to create a vector or a matrix that is initially empty (the values stored in it are *undefined*) and only fill it afterwards. For this one may use `Array{Type}` function with a special `undef` value passed:

```
julia> Array{String}(undef, 3)
3-element Array{String,1}:
 #undef
 #undef
 #undef

julia> Array{String}(undef, 3, 4)
3×4 Array{String,2}:
 #undef  #undef  #undef  #undef
 #undef  #undef  #undef  #undef
 #undef  #undef  #undef  #undef
```

In the above example, we created a 3-element vector and, respectively, a $3 \times 4$ matrix that can hold strings. The `#undef` text printed out informs us that no string is stored in a given cell of an array yet. One may *write* to such cell but not read from it. Here is an example:

```
julia> x = Array{String}(undef, 3)
3-element Array{String,1}:
 #undef
 #undef
 #undef

julia> x[1]
ERROR: UndefRefError: access to undefined reference
Stacktrace:
 [1] getindex(::Array{String,1}, ::Int64) at .\array.jl:729
 [2] top-level scope at none:0

julia> x[1] = "Hello!"
"Hello!"

julia> x
3-element Array{String,1}:
    "Hello!"
 #undef
 #undef

julia> x[1]
"Hello!"
```

However, one should not rely on Julia returning an error when trying to read an undefined reference. For some types (namely, the so-called *bits types*) Julia actually returns such array with some valid values but they are undefined:

```
julia> Array{Int}(undef, 3)
3-element Array{Int64,1}:
 21728009072
    72027968
    72060464

julia> Array{Float64}(undef, 3, 5)
```

```
3×5 Array{Float64,2}:
 5.29397e-316  5.28906e-316  5.28907e-316  5.28907e-316  5.28907e-316
 7.26835e-316  5.28906e-316  5.28907e-316  5.28907e-316  5.28908e-316
 5.28906e-316  5.28906e-316  1.07372e-313  5.28907e-316  5.28865e-316
```

One may check if a type is bits or not using `isbitstype` function:

```
julia> isbitstype(Int)
true

julia> isbitstype(String)
false
```

### 2.6.3 Sets

Sets are collections of unique values that are implemented in an efficient way so that operations dealing with them run fast. The most common operations on sets are: adding an element using `push!`, deleting an element using `pop!`, and checking if some element is contained in the set using `in`. Here are some basic examples:

```
julia> s = Set([1,2,3])
Set([2, 3, 1])

julia> 1 in s
true

julia> pop!(s, 1)
1

julia> 1 in s
false

julia> push!(s, 1)
Set([2, 3, 1])

julia> in(1, s)
true
```

### 2.6.4 Dictionaries

Dictionaries are similar to sets but they keep key-value pairs. Apart from using `pop!` (like for sets) to remove key-value pairs, one can use indexing to get and set the value given the key and the `haskey` function to check if a dictionary contains some key. In order to directly work only with keys or values stored in the dictionary, one can use `keys` and `values` functions. Also, note that you can initially populate the dictionary using the `key=>value` syntax. Here are some examples:

```
julia> d = Dict("a"=>1, "b"=>2)
Dict{String,Int64} with 2 entries:
  "b" => 2
  "a" => 1

julia> pop!(d, "a")
1

julia> d
```

```
Dict{String,Int64} with 1 entry:
  "b" => 2

julia> d["c"] = 3
3

julia> d
Dict{String,Int64} with 2 entries:
  "c" => 3
  "b" => 2

julia> d["b"] = 12
12

julia> d
Dict{String,Int64} with 2 entries:
  "c" => 3
  "b" => 12

julia> haskey(d, "b")
true

julia> keys(d)
Base.KeySet for a Dict{String,Int64} with 2 entries. Keys:
  "c"
  "b"

julia> values(d)
Base.ValueIterator for a Dict{String,Int64} with 2 entries. Values:
  3
  12
```

## 2.7 Control flow

The basic control flow constructs in Julia are functions (we already discussed them), conditional evaluation, loops, comprehensions, and exception handling. Additionally we will discuss a powerful feature of the Julia language that is broadcasting.

### 2.7.1 Conditional evaluation

Similarly to other languages, one may perform the computations conditionally using the if-else statement. Here is a general specification where `elseif` and `else` parts are optional:

```
if condition_1
    action_1
elseif condition_2
    action_2
else
    action_3
end
```

Here is a specific example:

```
julia> function f(v)
           if v == 1
```

```
            println("got one")
        elseif v == 2
            println("got two")
        else
            println("got something else")
        end
    end
f (generic function with 1 method)

julia> f(1)
got one

julia> f(2)
got two

julia> f(3)
got something else
```

### 2.7.2 Looping

In Julia one can use `for` and `while` loops. Their general specification is as follows:

```
for variable in collection
    action
end
```

and

```
while condition
    action
end
```

Additionally, one may use `break` statement to immediately terminate the loop and `continue` statement to immediately proceed to the next iteration of the loop. Here are two examples how these concepts are applied in practice:

```
julia> for i in 1:10
           i < 8 && continue
           println(i)
       end
8
9
10

julia> while true
           x = rand()
           println(x)
           x < 0.1 && break
       end
0.5482376043057251
0.9483962978432809
0.22282486812911362
0.06418952131181066
```

### 2.7.3 Comprehensions

Comprehensions are used to construct array using a construct similar to a `for` loop. Here is a basic specification of a comprehension syntax:

```
[value_computation for value in collection if condition]
```

In this construction `value` will go through all elements of `collection` and the result of the expression `value_computation` is stored if the `condition` is fulfilled. The `if` part is optional. Here are two examples:

```
julia> [x^2 for x in 1:5]
5-element Array{Int64,1}:
  1
  4
  9
 16
 25

julia> [x^2 for x in 1:5 if isodd(x)]
3-element Array{Int64,1}:
  1
  9
 25
```

### 2.7.4 Exception handling

Some Julia functions can throw `Exception`s when they fail. The try-catch-finally block can be used to gracefully handle such situations. The general structure of these blocks is as follows:

```
try
    action
catch exception_name
    action_on_error
finally
    action_guaranteed
end
```

Here is a simple example how this construction is used:

```
julia> function fun(x)
           try
               y = sqrt(x)
           catch
               println("could not calculate sqrt of $x")
           finally
               println("good bye")
           end
       end
fun (generic function with 1 method)

julia> fun(10)
good bye
3.1622776601683795

julia> fun(-10)
could not calculate sqrt of -10
good bye
```

## 2.8 Broadcasting

Sometimes there is a need to perform the same operation on all elements of some collection or collections. In Julia, one can easily achieve this using broadcasting. Let us explain the idea behind it by considering a few examples. We start with a scalar operation:

```
julia> x = 1
1

julia> sqrt(3 + x)
2.0
```

Now, what would happen if `x` were not a scalar (a number) but, instead, a vector of numbers?

```
julia> x = [1, 6]
2-element Array{Int64,1}:
 1
 6

julia> sqrt(3 + x)
ERROR: MethodError: no method matching +(::Int64, ::Array{Int64,1})
Closest candidates are:
  +(::Any, ::Any, ::Any, ::Any...) at operators.jl:502
  +(::T<:Union{Int128, Int16, Int32, Int64, Int8, UInt128, UInt16,
   UInt32, UInt64, UInt8}, ::T<:Union{Int128, Int16, Int32, Int64,
   Int8, UInt128, UInt16, UInt32, UInt64, UInt8}) where
   T<:Union{Int128, Int16, Int32, Int64, Int8, UInt128, UInt16,
   UInt32, UInt64, UInt8} at int.jl:53
  +(::Integer, ::Integer) at int.jl:797
  ...
Stacktrace:
 [1] top-level scope at none:0
```

We get an error as Julia does not know how to add a number to a vector. This should not be surprising since such operation does not make sense mathematically and so Julia tries to protect you from making a mistake. However, often in the code, one needs to perform such operations. We already know that we may use `map` function or a comprehension to perform this calculation:

```
julia> x = [1, 6]
2-element Array{Int64,1}:
 1
 6

julia> map(v -> sqrt(3 + v), x)
2-element Array{Float64,1}:
 2.0
 3.0

julia> [sqrt(3 + v) for v in x]
2-element Array{Float64,1}:
 2.0
 3.0
```

However, there is an easier way to achieve the same outcome. One may simply add a `.` (dot) to the names of functions to make the corresponding operation start working on vectors.

```
julia> x = [1, 6]
2-element Array{Int64,1}:
 1
 6

julia> sqrt.(3 .+ x)
2-element Array{Float64,1}:
 2.0
 3.0
```

This works for any function and any operator. The only thing to remember is that the dot `.` goes *after* a function name but *before* an operator name. Let us consider then a slightly more complex example where we broadcast over two objects having different dimensions:

```
julia> x = [1, 2, 3]
3-element Array{Int64,1}:
 1
 2
 3

julia> y = ["a" "b"]
1x2 Array{String,2}:
 "a"  "b"

julia> string.(x, y)
3x2 Array{String,2}:
 "1a"  "1b"
 "2a"  "2b"
 "3a"  "3b"

julia> string.(y, x)
3x2 Array{String,2}:
 "a1"  "b1"
 "a2"  "b2"
 "a3"  "b3"
```

First, let us point out that the `string` function takes its arguments and concatenates their string representation. Now, because `x` is a column vector and `y` is a one-row matrix, we see that their dimensions of size one get expanded to match the dimensionality of the second argument. However, this applies only to dimensions of size one and other sizes must match exactly so the following code raises an error:

```
julia> x = [1, 2, 3]
3-element Array{Int64,1}:
 1
 2
 3

julia> y = ["a", "b"]
2-element Array{String,1}:
 "a"
 "b"

julia> string.(x, y)
ERROR: DimensionMismatch("arrays could not be broadcast to a common size")
Stacktrace:
```

```
 [1] _bcs1 at .\broadcast.jl:438 [inlined]
 [2] _bcs at .\broadcast.jl:432 [inlined]
 [3] broadcast_shape at .\broadcast.jl:426 [inlined]
 [4] combine_axes at .\broadcast.jl:421 [inlined]
 [5] instantiate at .\broadcast.jl:255 [inlined]
 [6] materialize(::Base.Broadcast.Broadcasted{Base.Broadcast.
     DefaultArrayStyle{1},Nothing,typeof(string),Tuple{Array{Int64,1},
     Array{String,1}}}) at .\broadcast.jl:753
 [7] top-level scope at none:0
```

Broadcasting is a much broader and involved topic but the content introduced above should be enough to let you efficiently perform most basic data processing.

# 3 Computer support in solving mathematical problems

In this section we present selected applications of computer support in solving mathematical problems. All examples are related to some problems presented in the book to highlight cases when using a computer would be useful. In the section that follows this one, we will discuss examples in which we actually used computer aided arguments to solve them.

## 3.1 Analyzing of function shapes

In Problem 1.9.2, we analyze the function

$$f(x) \quad = \quad \frac{x}{2}\sqrt{1-x^2}+\frac{1}{16}\sqrt{16x^2-1}$$

and our goal is to show that it is less than $\frac{4}{9}$ for any $x \in [0.25, 1]$. In order to get an intuition about the problem and the function involved, we may first plot the function and then numerically find its maximum in the provided interval. For plotting, we may use *PyPlot.jl* package but before we begin we need to install it, provided that it is not installed yet[1].

```
julia> using Pkg

julia> Pkg.add("PyPlot")
```

Now we are ready. First, let us define the function we want to analyze and the range for variable `x` over which we will want to plot it. We fix the step to be equal to `0.0001`.

```
julia> f(x) = x*sqrt(1-x^2)/2+sqrt(16*x^2-1)/16
f (generic function with 1 method)

julia> x = 0.25:0.0001:1.0
0.25:0.0001:1.0
```

Now, we can plot it using the following commands:

```
julia> using PyPlot

julia> plot(x, f.(x))
```

[1]Note that PyPlot.jl is a wrapper around *matplotlib* package from Python. A good starting point if you want to investigate the alternatives is to visit `http://docs.juliaplots.org/` website.

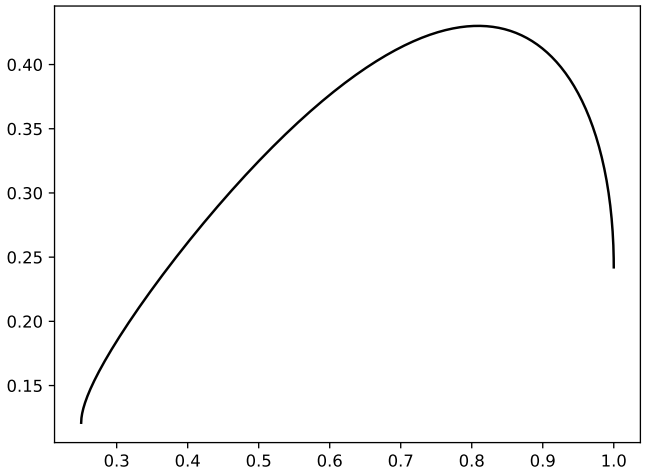



Figure 1: Plot of function $f(x) = \frac{x}{2}\sqrt{1-x^2} + \frac{1}{16}\sqrt{16x^2-1}$.

A plot like the one presented in Figure 1 should appear in a separate window. Note that in order to apply the function `f` to all elements of the sequence `x` we appended `.` (a dot) to the function name. From the plot we immediately see that (most probably) function $f$ has a single maximum. We can find its approximated value and the value of $x$ for which it is attained using the following commands:

```
julia> maximum(f.(x))
0.43011934965590826

julia> x[argmax(f.(x))]
0.809
```

With some luck we might guess that the optimal `x` is actually $(1+\sqrt{5})/4$, which might give us a hint how to solve the problem. If we wanted to approximate the optimal solution with a higher precision, then we could use a search algorithm. Below we provide a Julia implementation of ternary search, which we may apply as the function has only one extremum. The idea of this search is the following. Suppose that we know that a function $f$, over an interval $[a, b]$ with $a < b$, has a unique extremum. Then, we may consider two points, namely, $m_1 = (2a + b)/3$ and $m_2 = (a + 2b)/3$. If $f(m_1) > f(m_2)$, then we know that it is impossible that the extremum lies in the interval $(m_2, b]$ and so we can restrict our search to $[a, m_2]$. On the other hand, if $f(m_1) \leq f(m_2)$, then we know that the extremum does not lie in the interval $[a, m_1)$ and so we can restrict the search to $[m_1, b]$. Note that in one iteration we make the interval shorter by a multiplicative factor of $2/3$, and so the length of the interval diminishes geometrically. In the implementation below, we iterate the procedure as long as the length of the search interval is greater than the `tol` parameter.

```
julia> function ternary(lo, hi, tol)
           while hi-lo > tol
               m1, m2 = (2*lo+hi)/3, (lo+2*hi)/3
               if f(m1) > f(m2)
                  hi = m2
               else
```

```
                lo = m1
            end
        end
        return (lo+hi)/2
    end
ternary (generic function with 1 method)

julia> res = ternary(0.25, 1.0, 1e-8)
0.8090169980999418
```

Now we may check how far the result is from the exact solution:

```
julia> res - (1+sqrt(5))/4
3.724994313003549e-9
```

We see that it is within a set tolerance level of $10^{-8}$. Standard floating point numbers in Julia (and, in general, in computer representation) have a limited precision. If one wanted to use more precise computations, then high-precision arithmetic can be used. These computations are slower but, in this case it does not matter as the algorithm is not computationally involved. In Julia adjusting the code is very simple; the only thing that needs to be done is to wrap numbers in `big` function. Here is an example of calculation that is precise up to 30 decimal digits:

```
julia> res = ternary(big(0.25), big(1.0), big(10)^(-30))
0.809016994374947424102293417182725257934927034660960515325816868542775580148497 5

julia> res - (1+sqrt(big(5)))/4
-9.380092522755524192091574190744280985465126095733495374588364528259526212842508e-32
```

Let us mention that we could have achieved similar results using the standard package *Optim.jl* that provides the `optimize` function that can perform the optimization for us. First, we need to install the package, if it is not installed yet:

```
julia> using Pkg

julia> Pkg.add("Optim")
```

Now, we can run the computations:

```
julia> using Optim

julia> res = optimize(x -> -f(x), 0.25, 1.0)
Results of Optimization Algorithm
 * Algorithm: Brent's Method
 * Search Interval: [0.250000, 1.000000]
 * Minimizer: 8.090170e-01
 * Minimum: -4.301194e-01
 * Iterations: 11
 * Convergence: max(|x - x_upper|, |x - x_lower|) <=
   2*(1.5e-08*|x|+2.2e-16): true
 * Objective Function Calls: 12

julia> res.minimizer - (1+sqrt(5.0))/4
5.752353593457826e-9

julia> res = optimize(x -> -f(x), big(0.25), big(1.0))
Results of Optimization Algorithm
```

```
 * Algorithm: Brent's Method
 * Search Interval: [0.250000, 1.000000]
 * Minimizer: 8.090170e-01
 * Minimum: -4.301194e-01
 * Iterations: 17
 * Convergence: max(|x - x_upper|, |x - x_lower|) <=
   2*(4.2e-39*|x|+1.7e-77): true
 * Objective Function Calls: 18

julia> res.minimizer - (1+sqrt(big(5.0)))/4
-3.12761133980583041287203931701505023846100414136920369104352916343787484163841 2e-45
```

In the above scenario, the `optimize` function takes three arguments: a function to *minimize*, a lower and an upper bound for the argument that determines the interval where the minimum should be searched for. Because `optimize` minimizes a passed function and our function `f` should be maximized, we created an anonymous function `x -> -f(x)` that negates function `f`.

Observe that, similarly to our example using ternary search, we can use double precision computations with `Float64` numbers or we could pass `BigFloat` values to the function, which were obtained by using `big` function. As can be seen from the output of the `optimize` function, the algorithm used *Brent's Method* to find the minimum. An interested reader can find its description here `https://en.wikipedia.org/wiki/Brent%27s_method`.

## 3.2 Analyzing sequences

In Problem 2.6.1, we are concerned with the following sequence that is defined recursively: $x_1$ is any positive real number and for any $n \in \mathbf{N}$, $x_{n+1} = x_n + 1/x_n^2$. The following function computes the first $10^7$ values of $x_n$ and prints the value of $x_n^3/n$ for every $n \equiv 0 \pmod{10^6}$.

```
julia> function gen_x(x)
           for n in 2:10^7
               x = x + 1 / (x^2)
               if mod(n, 10^6) == 0
                   println(n, ":\t", x^3 / n)
               end
           end
       end
gen_x (generic function with 1 method)
```

Let us check the behaviour of the sequence for a few initial values of $x_1$:

```
julia> gen_x(2.0)
1000000:    3.0000179507720985
2000000:    3.00000932195842
3000000:    3.000006349793709
4000000:    3.0000048342657473
5000000:    3.0000039120411643
6000000:    3.0000032904212253
7000000:    3.0000028423824268
8000000:    3.000002503775446
9000000:    3.000002238665366
10000000:   3.000002025334477

julia> gen_x(100.0)
1000000:    3.9999983862944894
2000000:    3.4999994729552606
```

```
3000000:    3.3333331008612297
4000000:    3.2499998912372425
5000000:    3.1999999545175353
6000000:    3.1666666574067888
7000000:    3.1428571558626635
8000000:    3.1250000273591345
9000000:    3.111111148022531
10000000:   3.1000000433986594

julia> gen_x(0.05)
1000000:    67.02399704592145
2000000:    35.011998544846115
3000000:    24.341332377214286
4000000:    19.005999292975577
5000000:    15.804799442121608
6000000:    13.670666207978416
7000000:    12.146285326255782
8000000:    11.002999664808259
9000000:    10.113777483556166
10000000:   9.402399738443076
```

One can observe that for initial values that are of the order of magnitude of units, the sequence seems to converge fast to 3. However, for very large or very small values of $x_1$, even after $10^7$ iterations we are still nowhere near to the limit. We note, though, that the sequence $x_n^3/n$ seems to be eventually decreasing in all the cases. Therefore, as the sequence is non-negative, it would be enough to prove its convergence. Using this observation we note that the condition:

$$\frac{(x+1/x^2)^3}{n+1} \quad < \quad \frac{x^3}{n}$$

can be rewritten as follows:

$$n(x^2+1)^3 \quad < \quad (n+1)x^9$$

The above inequality holds, provided the following condition is satisfied:

$$1 \quad < \quad x^2(x-1).$$

Now, we note that for $x > 1$ both $x^2$ and $x-1$ are increasing functions and for $x = 1.5$ this condition is satisfied. Therefore, the original inequality holds if $x \geq 1.5$. It follows that for $x_1 \geq 1.5$ the sequence is always decreasing. On the other hand, if $x_1 < 1.5$, then arithmetic-geometric inequality implies that

$$x_2 \quad = \quad x_1 + \frac{1}{x_1^2} \quad \geq \quad 2\sqrt{1/x_1} \quad = \quad 2/\sqrt{1.5} \quad > \quad 1.5,$$

so in this case the sequence will be decreasing starting from $x_2$. We conclude that the sequence $x_n/\sqrt[3]{n}$ has a limit.

Let us highlight one thing. With a computer support, we gained some intuition on how one may approach the fist part of the problem, namely, the existence of the solution. On the other hand, the performed experiments provide us with little understanding about the actual limiting value, as the convergence is very slow. Hence, this example can serve as a warning that one should not blindly rely exclusively on computational results, especially when dealing with asymptotic properties.

Note, however, that it is relatively simple to observe that the limit must be greater than or equal to 3. Using the arithmetic-geometric inequality one more time, we get

$$x_2 \quad = \quad x_1 + 1/x_1^2 \quad = \quad x_1/2 + x_1/2 + 1/x_1^2 \quad \geq \quad 3/\sqrt[3]{4},$$

and, as a consequence, using a computer we may verify that $x_2^3/2 > 3.375 > 3$ for any $x_1$. Now, we can show that if $x_n^3/n > 3$, then $x_{n+1}^3/(n+1) > 3$ by observing that

$$\frac{x_{n+1}^3}{n+1} = \frac{\left(x_n + 1/x_n^2\right)^3}{n+1} > \frac{x_n^3+3}{n+1} > \frac{3n+3}{n+1} = 3.$$

Hence, by induction, we get that $x_n > 3$ for all $n \in \mathbf{N}$.

## 3.3 Polynomial root finding

In Problem 3.1.3, we are asked to find the value of $\sum_{i=1}^{3} 1/x_i^4$, where $x_i$'s are roots of the equation $3x^3 + 6x^2 - 1 = 0$. The challenge in this problem is that finding the exact roots of this polynomial is possible but it is challenging. On the other hand, using Julia we can easily approximate them. First, let us install *Polynomials.jl* package, provided that it is not installed yet.

```
julia> using Pkg

julia> Pkg.add("Polynomials")
```

Now, we can compute the approximated values of the roots themselves and then compute the sum that the problem asked us to evaluate.

```
julia> using Polynomials

julia> p = Polynomial([-1, 0, 6, 3])
Polynomial(-1 + 6*x^2 + 3*x^3)

julia> x = roots(p)
3-element Array{Float64,1}:
 -1.908482911767036
 -0.46617814806823543
  0.3746610598352708

julia> sum(1 ./ x .^ 4)
71.99999999999982
```

The function `Poly` allows us to define a polynomial. Its argument is a vector providing the coefficients of the polynomial. Then, using the `roots` function we may find the roots of the polynomial. Finally, we calculate $\sum_{i=1}^{3} 1/x_i^4$. Note that we use `./` and `.^` as we need to broadcast the division and the exponentiation over all elements of `x` before summing the results. Let us observe that the exact solution to our problem is 72, but we have obtained 71.99999999999982 which is not exact. This is a typical situation in numerical computations which are usually only approximations of the true values. As a final note, it is worth to point out that in a case like this one we may think of the result obtained using computer assistance as a way to double check that the analytical result we have obtained is correct. They are not meant to replace rigorous arguments.

## 3.4 Calculating probabilities

Let us start with Problem 4.6 in which we are asked to break the stick into $n+1$ pieces uniformly at random. Our goal is to compute the probability that one can form a polygon using these pieces. Denote the lengths of pieces as $x_1, x_2, \ldots, x_{n+1}$. In the analysis of the problem, we note that it is possible if and only if the longest piece in this sequence is shorter than 0.5. This condition can be easy checked computationally:

```
julia> using Statistics

julia> gensplit(n) = diff([0; sort(rand(n)); 1])
gensplit (generic function with 1 method)

julia> simulate(n, reps) =
    mean([maximum(gensplit(n)) < 0.5 for i in 1:reps])
simulate (generic function with 1 method)

julia> for n in 2:8
           println(n, ":    expected: ", rpad(1-(n+1)/2^n, 10),
                   "    sampled: ", simulate(n, 10^6))
       end
2:    expected: 0.25          sampled: 0.249933
3:    expected: 0.5           sampled: 0.499662
4:    expected: 0.6875        sampled: 0.687294
5:    expected: 0.8125        sampled: 0.812829
6:    expected: 0.890625      sampled: 0.890557
7:    expected: 0.9375        sampled: 0.937341
8:    expected: 0.96484375    sampled: 0.964884
```

In the example above, we first load `Statistics` module that defines `mean` function that we use in the code. Next, `gensplit` function randomly generates a split of a stick of unit length into `n+1` pieces. Note that we identify `n` splitting points at random using the `rand` function and then we sort them. Next, we concatenate this list with `0` in front and `1` at the end, and to get the actual lengths of the sticks we calculate the corresponding differences using the `diff` function. The `simulate` function takes two parameters, number of splitting points `n` and number of iterations of the simulation `reps`. The `maximum(gensplit(n)) < 0.5` formula checks if a single randomly generated split is valid (that is, as required, the longest stick is shorter than `0.5`). Then, we calculate the `mean` of the results of evaluation of this formula to get an estimate of the probability that the desired property holds. In the code, we have used `rpad` function to add trailing spaces to numbers so that they all have width 10 when printed.

The final `for` loop runs a simulation for `n` ranging from `2` to `8`. First, we calculate the exact result that we have derived analytically in Problem 4.6. Next, we produce an estimation obtained via a simulation executed one million times. (Of course, if you actually run the code, then expect to get slightly different results). As you can observe the exact and simulated results are similar, which make us more confident that the exact results are correct.

Similarly, we may design a simulation to find the approximation for Problem 4.6.1. Let us briefly remind the setup and the question. Consider an urn that initially contains one white and one black ball. We repeatedly perform the following process. In a given round, one ball is drawn randomly from the urn and its colour is observed. The ball is then returned to the urn, and an additional ball of the same colour is added to the urn. We repeat this selection process for 50 rounds so that the urn contains 52 balls. What number of white balls is the most probable?

First, let us define a function running a single simulation of the process:

```
julia> function urnsim()
           white = 1
           black = 1
           for i in 1:50
               if rand() < white / (white + black)
                   white += 1
               else
                   black += 1
```

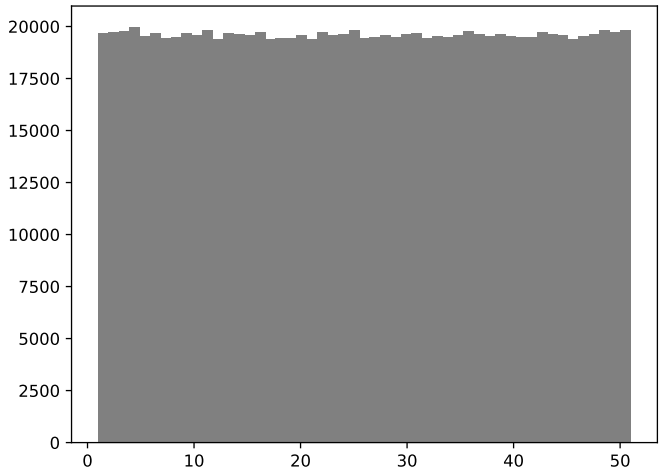



Figure 2: A histogram of white ball count in Problem 4.6.1.

```
            end
        end
        return white
    end
urnsim (generic function with 1 method)
```

The condition `rand() < white / (white + black)` reflects the fact that with probability equal to `white / (white + black)` we pick a white ball and so in the end there is one more white ball in the urn. Otherwise, a black ball is picked and we have to increase the number of black balls by one. The following code runs the simulation one million times and plots a histogram divided into 51 bins representing white ball counts from 1 to 51. In order to run the code make sure you have `PyPlot` package installed (if you followed the examples presented earlier in this on-line companion, then you already should have it installed).

```
julia> using PyPlot

julia> hist([urnsim() for i in 1:10^6], 51)
```

Figure 2 presents the resulting plot (as always, let us warn you that your plot might be slightly different due to randomness in the process of generating the data). One may observe that the histogram is approximately flat that is consistent with the theoretical result we have obtained.

## 3.5 Working with large integers

It is sometimes the case that a problem that is potentially challenging without access to a computer becomes trivial when we are allowed to use it. A good example is Problem 5.4.3. where we are asked to find the two last digits of $7^{123}$. Using a computer we can solve it in one line of code and we obtain the solution instantly:

```
julia> mod(big(7)^123, 100)
43
```

Here, the only thing to remember is to make sure we are using arbitrary precision numbers, as `big(7)` in this case. If we us an ordinary `Int`, then we would get a wrong result on a 64-bit machine:

```
julia> mod(7^123, 100)
15
```

The reason we got a wrong answer is that operations on standard (not arbitrary precision) integers overflow if they exceed the range of values that can be represented in 64 bits. We clearly have this problem here, which can be quickly checked as follows:

```
julia> big(7)^123
885235703693468016844358113727181275856700611147021449335692452600932537289998 80
9814218814737093654963 43

julia> typemax(Int)
9223372036854775807
```

We see that $7^{123}$ is much larger than the value that fits 64-bit representation of integers (which actually is equal to $2^{63} - 1$).

Now, let us ask a more challenging question. What are the last two digits of $123456789^{123456789}$? This number has almost one billion digits, which can be checked by computing the numerical approximation of $123456789 \cdot \log(123456789)/\log(10)$:

```
julia> 123456789 * log(123456789) / log(10)
9.989524572267263e8
```

While these computations are still potentially doable on a large machine, there is a faster way to solve this problem instead of direct computation (if we insist to use a computer support). First, note that

$$123456789^{123456789} \equiv 89^{123456789} \pmod{100}.$$

Next, observe that $89^{10} \equiv 1 \pmod{100}$, which can be quickly checked using the computer:

```
julia> mod(big(89)^10, 100)
1
```

Therefore,

$$89^{123456789} \equiv 89^{9} \pmod{100}.$$

The last value can be easily verified to have `09` as the last two digits:

```
julia> mod(big(89)^9, 100)
9
```

Another approach to reach the same conclusion would be to directly check that the cycle of last two digits of $123456789^n$ is 10 using the following code:

```
julia> for n in 1:11
           println(lpad(n, 2), ": ", lpad(mod(big(123456789)^n, 100), 2))
       end
 1: 89
 2: 21
 3: 69
 4: 41
 5: 49
 6: 61
 7: 29
```

```
 8: 81
 9:  9
10:  1
11: 89
```

We immediately see that if the exponent's last digit is 9, then the last two digits we are looking for are `09`. Note that in the code we used the `lpad` function to print all numbers using two characters, prepending them with a whitespace, if needed.

# 4 Understanding the Examples from the Book

In this section, we briefly explain the codes presented in the book in order to give users that are less familiar with coding an understanding how one can program in Julia. These examples are sorted in order of increasing complexity. As a result, some examples from earlier chapters might be discussed after examples from later chapters.

Please note that in the on-line companion we do not repeat the whole solutions, and only concentrate on explaining how the codes presented in the book work. Please refer to the book to understand the context in which these codes are used.

## 4.1 Finding Reminders—Problem 5.4

Start a fresh Julia session and write the following:

```
julia> Set(mod(i ^ 4, 17) for i in 0:16)
```

As a result, Julia produces:

```
Set([4, 16, 0, 13, 1])
```

Let us now discuss carefully what has happened:

- when we write `i in 0:16`, we tell Julia that we want variable `i` to traverse integers from `0` to `16` with step `1`;
- when we write `mod(i ^ 4, 17)`, we tell Julia that we want variable `i` to be raised to the power of `4`, and then we want to calculate the reminder of division of the result by `17`;
- as repeating this process for values of `i` from `0` to `16` may produce duplicates, we use a `Set` function to create a set (in a mathematical sense) from these values; in particular, the output does not display duplicates (but the numbers presented are not ordered, as sets in mathematics are not ordered).

Alternatively, we could create a *vector* consisting of the resulting values as follows:

```
julia> [mod(i ^ 4, 17) for i in 0:16]
17-element Array{Int64,1}:
  0
  1
 16
 13
  1
 13
  4
  4
 16
 16
```

```
 4
 4
13
 1
13
16
 1
```

We used square brackets instead of `Set` to tell Julia that we want to get a vector as a result. This time, of course, we see duplicates in the output.

## 4.2 Rolling of three dice—Problem 4.9.3

Here is another example of code from the book:

```
julia> count(x -> 10<=x<=11, i+j+k
            for i in 1:6, j in 1:6, k in 1:6) / 6^3
0.25
```

Some parts of this expression should be intuitively clear. We consider all possibilities for variables `i`, `j`, and `k`, each of them is an integer value from `1` to `6`. In other words, we consider all triples $(i, j, k) \in [6] \times [6] \times [6]$. Hence, in `i+j+k for i in 1:6, j in 1:6, k in 1:6` part of the code, we traverse all possible sums of these three variables over $6^3$ possibilities for the involved variables.

A new part in this code is the `count` function. It takes as an input a collection (in our case, all the sums `i+j+k` we traverse) and counts how many times a given predicate is true when applied to them. A *predicate* is a function of some variable that checks if some property is satisfied for a given variable and, depending on that, it returns `true` or `false`. In our case, the predicate is `x -> 10<=x<=11`, a function that takes value `x` as an input (these values will be the sums of the form `i+j+k`) and checks if it is greater than or equal to `10` and less than or equal to `11`. As a result, the whole expression calculates how many times (as a fraction of the total number of triples) the sum `i+j+k` is equal to `10` or `11`.

In order to get more comfortable with `count` function, let us consider the following, slightly simpler, example:

```
julia> count(x -> mod(x, 2) == 0, 1:11)
5
```

It counts even integers in the range from `1` to `11`, as `mod(x, 2) == 0` checks if a given number gives the reminder of `0` when divided by `2`.

If one is not sure what some function does, typing `?` before a function name and then hitting enter provides some help. Here is an example with `count` function:

```
help?> count
search: count countlines count_ones count_zeros fieldcount

  count(p, itr) -> Integer
  count(itr) -> Integer

  Count the number of elements in itr for which predicate p
  returns true. If p is omitted, counts the number of true
  elements in itr (which should be a collection of boolean
  values).

  Examples
```

```
==========

julia> count(i->(4<=i<=6), [2,3,4,5,6])
3

julia> count([true, false, true, true])
3
```

## 4.3 Sum of Four-digit Numbers—Problem 4.3.3

Here is our next example:

```
julia> sum(x for x in 1000:9999 if !(0 in digits(x)))
36446355
```

Let us start with understanding what the `digits` function does. It takes a number as an input and returns a vector that consists of its digits in decimal representation. Here is a simple example:

```
julia> digits(1234)
4-element Array{Int64,1}:
 4
 3
 2
 1
```

Let us come back to our example. The part `0 in digits(x)` checks if `x` contains at least one digit `0`. A sign `!` in front of it is a negation, that is, `if !(0 in digits(x))` indicates that we only process values of `x` that do *not* contain `0` in their decimal representation. Finally, we calculate their sum using `sum` function.

## 4.4 Legendre's Formula—Problem 5.2

In our earlier examples, we used a short syntax for defining functions without a name. For example, the function `x -> x^2` squares a value that is passed to it (namely, variable `x`) . Such functions are usually called anonymous, and are typically very short. On the other hand, we might want to give a name for longer and more sophisticated functions. We will now learn how to do this.

In the book, we defined a function that calculates the value of the Legendre formula:

```
julia> function legendre(n, d)
           s = 0
           q = 1
           while true
               q *= d
               a = div(n, q)
               if a == 0
                   return s
               end
               s += a
           end
       end
```

The body of the function is enclosed between `function` and `end` keywords. The part of the code in line `function legendre(n, d)` specifies the name of a function, `legendre`. The function requires two arguments as an input, `n` and `d`, and should return $\sum_{i=1}^{\infty} \left\lfloor \frac{n}{d^i} \right\rfloor$.

At the very beginning, we define two auxiliary variables, `s` and `q`, that initially take values `0` and `1`, respectively. The variable `s` keeps track of the partial sum and at the end is equal to the value

of the calculated Legendre's formula. On the other hand, the variable `q` keeps track of the power of `d` by which we divide `n`. This repeated changing of value of `q` to consecutive powers of `d` is achieved in line `q *= d` which indicates that the value of `q` is replaced by `q * d`. The block from `while` to `end` is performed repeatedly as long as condition after `while` is `true`. In this case, it will iterate infinitely; that is, the only way that the function terminates is when the line `return s` is reached. This is similar to the Legendre's formula, where the upper limit of summation is equal to $\infty$. Of course, once the value of $\left\lfloor \frac{n}{d^i} \right\rfloor$ is equal to zero, we should terminate the loop as the partial sum is equal to the final value and no further terms are needed. This happens when `q` becomes larger than `n`. In fact, we calculate the floor of `n` divided by `q` and store its value in `a`. If the value of `a` is `0`, this means that all remaining terms in the Legendre formula are equal to zero so we may return `s` and break the infinite loop. Otherwise, in line `s += a` we replace `s` by a new value equal to `s + a`; that is, we update the partial sum. As a result, when the function terminates, `s` indeed is equal to the value of the Legendre's formula.

## 4.5 Subset Selection—Problem 5.7.2

In the next example, we will use an additional, non-standard, library we installed before that is called *Combinatorics.jl*. As its name indicates, it provides various useful functions and concepts in the area of combinatorics. Our goal is to find two disjoint sets in $[k]$, $S_1$ and $S_2$, such that the sum of elements in $S_1$ is equal to the sum of elements in $S_2$, and the same applies to the corresponding products. We call this function `f` and we require two arguments: `k` is the size of the original set which we take subsets from and `n` is the size of each set. Here is the complete code:

```
using Combinatorics

function f(n, k)
    for x in combinations(1:k, n),
        y in combinations(setdiff(1:k, x), n)
        if x < y # avoid printing duplicates
            if sum(x) == sum(y) && prod(x) == prod(y)
                println((x, y))
            end
        end
    end
end
```

The code is rather straightforward but let us discuss it briefly. In line `using Combinatorics`, we load the extension library. It is needed as we need the `combinations` function that generates subsets of a given size from a given set.

In the code, `x` is the first subset, and `y` is the second subset. The variable `x` traverses all `n` element subsets of set `1:k`, that is, set $[k]$. Then, for a given set `x`, `y` traverses all `n` element subset of the set `setdiff(1:k, x)`, that is, $[k] \setminus x$. Of course, this approach considers a given pair of two sets $x$ and $y$ twice. In order to avoid it, we first check if $x < y$ which in Julia compares if `x` is lexicographically less than `y`. In particular, only one pair passes the test. Then, if both the sum and the product of the corresponding elements in both sets are equal, then we print both sets. Here, the `&&` denotes the logical operator *and*.

## 4.6 Sicherman Dice—Problem 4.9

The next example is the Sicherman dice solver presented below.

```
using Base.Iterators

function listdies()
```

```
    ref = [1,2,3,4,5,6,5,4,3,2,1] # reference distribution
    for d1 in product(1, 2:8, 2:8, 2:8, 2:8, 2:8)
        for d2 in product(1, 2:8, 2:8, 2:8, 2:8, 2:8)
            if issorted(d1) && issorted(d2) && d1 <= d2
                x = [a1 + a2 for x in d1, y in d2]
                if [count(v -> v==s, x) for s in 2:12] == ref
                    println((d1, d2))
                end
            end
        end
    end
end
```

The `ref` variable keeps the reference vector `[1,2,3,4,5,6,5,4,3,2,1]` which corresponds to the distribution of the sum of two standard dice rolled. For simplicity, in order to deal with integer values, we normalize it such that the sum is equal to 36 (not 1, as in the probability distribution). For example, 1, the first number in the vector indicates that there is one way to get the sum of 2 (probability 1/36), 2, the second number captures the fact that there are two ways to get the sum of 3, and so on.

The line `product(1, 2:8, 2:8, 2:8, 2:8, 2:8)` makes both `d1` and `d2` to traverse all six-element sequences with the property that the first element is always 1 and the other elements have values from 2 to 8. Such sequences represent the numbers occurring on six sides of a given die. Here we used the fact that each die has to have precisely one 1 and the largest number that can occur is 8. Since there are multiple vectors that represent the same pair of dice (up to a permutation), we use the condition `issorted(d1) && issorted(d2) && d1 <= d2` that ensures that only sequences of non-decreasing numbers are considered and only pairs in which the first die is not greater than the second one lexicographically.

The key line is `[a1 + a2 for x in d1, y in d2]`. In order to understand what it does, let us give an example of a specific pair of dice:

```
julia> d1 = (1,2,3,4,5,6)
(1, 2, 3, 4, 5, 6)

julia> d2 = (1,3,4,5,6,7)
(1, 3, 4, 5, 6, 7)

julia> [a1 + a2 for a1 in d1, a2 in d2]
6x6 Array{Int64,2}:
 2  4   5   6   7   8
 3  5   6   7   8   9
 4  6   7   8   9  10
 5  7   8   9  10  11
 6  8   9  10  11  12
 7  9  10  11  12  13
```

We see that the crucial line takes both collections and produces a *matrix* with all combinations of sums of elements from `d1` (varying in rows) and `d2` (varying in columns). As a result, this matrix has $6 \cdot 6 = 36$ entries, each entry corresponding to a specific outcome of our random experiment that happens with probability 1/36. In the following line `[count(v -> v==s, x)  for s in 2:12]` we count the number of 2, 3, ..., 12 entries in the matrix and compare against the reference vector. If there is a match, then we print out the result.

## 4.7 Painting 13-gon—Problem 4.7.2

Now let us move to the problem with finding isosceles triangles in a regular 13-gon. Recall that our goal was to verify that for any selection of 5 vertices from the 13 sided regular polygon, there always exists an isosceles triangle. Here is our solution:

```
using Combinatorics

function isisosceles(points)
    sort(points)
    d1 = points[2] - points[1]
    d2 = points[3] - points[2]
    d3 = 13 - d1 - d2
    return d1 == d2 || d2 == d3 || d3 == d1
end

function test13gon()
    for p5 in combinations(1:13, 5)
        if !any(isisosceles(p3) for p3 in combinations(p5, 3))
            return false
        end
    end
    return true
end
```

In the first part of this example, we define two functions. The first function, `isisosceles`, takes a collection of three points and checks if they form an isosceles triangle. We first sort the points in an increasing order using `sort` function. We then calculate the distances between the points using the fact that they are sorted (assuming that by distance we mean *number of hops one has to make*), and finally check if *any* of the three pairs of distances is equal. The `||` operator is the *or* operation.

The second function, `test13gon`, investigates whether there exists a 5-element subset of the vertices of a 13-gon that does *not* form an isosceles triangle. The only new part of the definition is `any` function. It takes a collection (in our case all 3-element subsets of `p5`) and returns `true` if some element of the collection is `true`; otherwise, it returns `false`. Finally, this time the `!` sign is significant and it negates the result of the operation following it.

```
function isosceles_count(i)
    c = string(i, base=3, pad=13)
    count(t -> c[t[1]] == c[t[2]] == c[t[3]] && isisosceles(t),
          combinations(1:13, 3)), c
end

function best13gon()
    mapreduce(isosceles_count, min, 2*3^11:3^12-1)
end
```

Our goal is to investigate all colourings of the vertices of the 13-gon with one of three colours denoted `0`, `1`, or `2`. We may independently consider colourings with a single colour that clearly produce a lot of monochromatic triangles. For the remaining colourings with at least two colours, without loss of generality, we may assume that vertex 13 has colour 0 and vertex 12 has colour 2. As a result, we may associate each colouring with a 13-element sequence starting with `"02"` and then having eleven values taken from the set $\{0, 1, 2\}$ (that is associated with some non-monochromatic sequence) or a 13-element sequence consisting of only zeros (that is associated with one of the three monochromatic sequences). Hence, these numbers in base 3 representation are either 0 or between $2 \cdot 3^{11}$ and $3^{12} - 1$.

In function `isosceles_count`, given a colouring represented by an integer `i`, we first convert `i` to a 13 element string `c` using `string(i, base=3, pad=13)`. We then consider all 3-element subsets of the set [13], examine if they form an isosceles triangle (by checking `isisosceles(t)`) and have the same colour (by checking `c[t[1]] == c[t[2]] == c[t[3]]`), and count the number of isosceles triangles in a given colouring. The tricky part in this function is "`, c`" which appears after the `count` function call. It makes the function return a tuple consisting of the number of triangles followed by the colouring itself. This will be useful in our second step.

In function `best13gon`, all possible colourings from $2 \cdot 3^{11}$ to $3^{12}-1$ are traversed. The `mapreduce` function applies first `isosceles_count` function to a given colouring and then applies `min` function to the returned values. As a result, a colouring that produces the minimum number of isosceles triangles is selected.

Since `mapreduce` function is a really powerful tool that allows the user to many useful things in one call, let us see the excerpt form its help (recall that we can obtain it by pressing `?` and then typig `mapreduce`):

```
help?> mapreduce
search: mapreduce

  mapreduce(f, op, itr; [init])

  Apply function f to each element in itr, and then reduce the
  result using the binary function op. If provided, init must
  be a neutral element for op that will be returned for empty
  collections.
```

## 4.8 Ski Jumping Tournament—Problem 4.6.2

The next example we explain is the tournament simulator. Let us first describe the exact solution method. Here is the relevant fragment of the code:

```
function probs65()
    probs = Dict{Tuple{Int, Int}, Float64}()

    function prob(j,k)
        if !haskey(probs, (j,k))
            if j == 0
                probs[(j,k)] = 1/k
            else
                probs[(j,k)] = 1/k*sum(prob(j-1,i) for i in j:(k-1))
            end
        end
        return probs[(j,k)]
    end
    [prob(j, 65) for j in 0:64]
end
```

A new element in this code is a definition of the `probs` variable. It is a dictionary—a data type that works similarly to functions in mathematics. In particular, it allows the user to associate arguments into given values. In our example, `probs[(i,n)]` is the equivalent of $p_{i,n}$ introduced in mathematical solution of this problem. Similarly to the definition of $p_{i,n}$, `probs[(i,n)]` is defined recursively. In line `haskey(probs, (j,k))`, we check if the value of $p_{j,k}$ has been previously computed and stored in memory. If not, then we compute it (possibly recursively), assign to `probs` variable so that it can be used in the future and then we simply return it. The last line `[prob(j, 65) for j in 0:64]` computes the vector of $p_{j,65}$ for $j$ ranging from 0 to 64.

In order to check our reasoning we also provided an independent simulation code that experimentally tests the correctness of computation of $p_{i,n}$ values. The core simulation function is defined as follows:

```
function sim_tournament()
    best_length = rand()
    best_changes = 0
    for i in 2:65
        jump_length = rand()
        if jump_length > best_length
            best_length = jump_length
            best_changes += 1
        end
    end
    return best_changes
end
```

It simulates one instance of execution of the tournament. The key new function here is `rand()`. It generates one pseudo random value from the interval $[0, 1)$, uniformly and independently. These values will represent the lengths of the jumps of particular jumpers. Note that it does not actually depend on which distribution we draw the jump lengths from as long as it is continuous and the draws are independent. The variable `best_length` remembers the length of the currently longest jump. it is initially set to a random number that corresponds to the jump of the first jumper. The second important variable is `best_changes` that counts how many times the leader was changed during the tournament. In order to approximate the probabilities $p_{i,65}$, we have to run this simulation many times. In our code we executed it 10,000,000 times, which yields an accurate estimate.

```
function run_simulation()
    simprobs65 = zeros(65)
    sim_runs = 10_000_000
    for i in 1:sim_runs
        simprobs65[sim_tournament() + 1] += 1
    end
    simprobs65 / sim_runs
end
```

Let us point out a few important things. Variable `simprobs65` keeps how many times a given number of changes in leadership occurred. It has 65 entries, as we can have from 0 to 64 changes. For each independent run of the simulation we obtain `sim_tournament()` and then increase the corresponding entry of `simprobs65` variable. Remember that the indexing of vectors in Julia starts from *one*, not from *zero* as in some other programming languages. Hence the additional `+ 1` in line `simprobs65[sim_tournament() + 1] += 1`. In particular, `simprobs65[1]` is equal to the number of simulation outcomes with no change of leadership whereas `simprobs65[65]` counts the outcomes with constant changes. Finally, line `simprobs65 / sim_runs` normalizes our results so that we return the estimate of the corresponding probabilities.

## 4.9 Number of Solutions—Problem 2.5

This time we want to verify the solution we derived analytically. Let us first define the following function in Julia:

```
function find_sequence(s)
    x1 = sum((s[i]+1)/4*prod(s[i+1:end]) for i in 1:length(s))
    x = Int[x1]
    for si in s
```

```
        push!(x, si*x[end] + (si+1)/2)
    end
    println(x)
end
```

It takes one argument `s`, which is a control sequence, and prints the calculated values of $x_i$ for this sequence. If the analytical solution is correct, then the first and the last value of `x` must be equal. In the presented solution, `sum` function calculates the sum of values as `i` ranges from `1` to `length(s)`, while `prod` takes the product of a sub-vector of `s` starting from element `i+1` until its last element. Note that, in particular, the product of the empty collection of integers is defined to be equal to `1`, exactly as one would expect in mathematical formulas. The syntax `Int[x1]` creates a vector with one element `x1` that is restricted to hold only integers (recall that we have shown that the sequence of $x_i$ in the solution must consist of integers). Finally the `push!` function iteratively appends one element to the end of vector `x`.

## 4.10 The Maximum GCD—Problem 5.1.3

Here is our last example. As above, our goal is to verify the solution we have found without the use of computer. Here is the code we used:

```
julia> function check(n)
           s = unique(digits(n))
           if length(s) != 2 || minimum(s) == 0
               return 0
           end
           return gcd(1111*sum(s) - n, n)
       end
check (generic function with 1 method)

julia> gcds = [check(n) for n in 1000:9999];

julia> max_gcd = maximum(gcds)
1212

julia> [n for n in 1000:9999 if check(n) == max_gcd]
2-element Array{Int64,1}:
 4848
 8484
```

Function `check` takes an integer `n` as an input and returns the greatest common divisor of $n$ and $f(n)$, provided that $n$ satisfies the required condition. In order to check this condition, we first we set the value of variable `s` to be a vector of unique digits of `n`. The condition requires that we should only consider numbers that have exactly two distinct digits, neither of which is equal to `0`. If `n` does not satisfy this condition, then the function `check` returns `0` which is clearly less than the greatest common divisor we are looking for and so it will be ignored once we search over all values of $n$. If `n` satisfies the required condition, we return the greatest common divisor of $n$ and $f(n)$ applying the closed-form for $f(n)$ derived in the solution to this problem.

Next, we create a vector `gcds` which consists of greatest common divisors of all pairs $n$ and $f(n)$ we are interested in, or 0s if $n$ does not satisfy the required condition, where $n$ ranges from 1000 to 9999. We then find the maximum element in this vector and store it in `max_gcd` variable. Finally, the line `[n for n in 1000:9999 if check(n) == max_gcd]` investigates all values of `n` from `1000` to `9999` and retains only those for which the corresponding GCD matches the maximum value we found. Alternatively, instead of writing `check(n)` we could have used `gcds[n-999]` as the value of `check(n)` is stored in the `n-999`th entry of `gcds` vector. Note that we subtracted `999` from `n` because vector indexing in the Julia language is 1-based.

This concludes our brief overview of the Julia language. We hope that the presented examples show that computer assisted arguments written in this language have nice syntax and many built-in functionality in this language support mathematical reasoning. It is also worth mentioning that computations using Julia language scale very well, as it has been designed not only to be elegant but also fast.