PROGRAMMING IN *MATHEMATICA*, A PROBLEM-CENTRED APPROACH

Contents

1. Introduction 4
1.1. *Mathematica* as a calculator 4
1.2. Numbers 6
1.3. Algebraic computations 7
1.4. Variables 7
1.5. Equalities, =, :=, == 7
2. Defining functions 10
2.1. Formulas as functions 10
2.2. Anonymous functions 11
3. Lists 12
3.1. Functions producing lists 13
3.2. Listable functions 15
4. Changing heads! 22
5. A bit of Logic and Set Theory 25
5.1. Being logical 25
5.2. Handling sets 27
5.3. Decision making, If statement 29
6. Sums and products 30
7. Loops 34
7.1. Nested loops 40
7.2. Nest, NestList and more 42
7.3. Fold and FoldList 46
7.4. Inner and Outer 48
8. Substitution, *Mathematica* rules! 50
9. Pattern matching 51
10. Functions with multiple definitions 57
10.1. Functions with local variables 62
10.2. Functions with conditions 63
11. Recursive functions 64
12. Matrices; Multilinear algebra 66
References 69

Date: March 8, 2007.

> Teaching the mechanical performance of routine mathematical operations and nothing else is well under the level of the cookbook because kitchen recipes do leave something to the imagination and judgment of the cook but the mathematical recipes do not.
>
> G. Pólya

This note grew out of a module I gave at Queen's University Belfast in the Winter semester 2004, Spring semester 2005 and Winter semester 2006. Although there are many books already written about how to use *Mathematica*, I noticed they fall into two categories: either they provide an explanation about the commands, in the style of: enter the command, push the button and see the result; or books which study some problems and write several-paragraph codes in *Mathematica*. The books in the first category did not inspire me (nor my imagination) and the second category were too difficult to understand and not suitable for learning (or teaching) *Mathematica*'s abilities to do programming and solve problems.

I could not find a book that I could follow to teach this module. In class one cannot go on forever showing students just how commands in *Mathematica* work; on the other hand it would be very difficult to follow the codes if one writes a program having more than five lines in class (especially as *Mathematica*'s style of programming provides a condensed code). Thus this note. This note promotes *Mathematica*'s style of programming. I tried to show when we adopt this approach, how naturally one can solve (nice) problems with (*Mathematica*) style.

Here is an example: Does the formula $n^2 + n + 41$ produce a prime number for $n = 1$ to 39?

Solution.

```
(#^2 +#+ 1) & /@ Range[39] ∈ Primes
True
```

Or in another Problem we tried to show how one can effectively use pattern matching to check that for two matrices A and B, $(ABA^{-1})^5 = AB^5A^{-1}$. One only needs to introduce the fact that $AA^{-1} = 1$ and then *Mathematica* will check the problem by cancelling the inverse elements instead of direct calculation.

Although the above code might look like Dutch now, the reader will observe as we proceed how the codes start making sense, as if this is the most natural way to approach the problems. (People who approach the problems with a procedural style of programming will experience that this style replace their way of thinking!) We have tried to let the reader learn from the codes and avoid long and exhausting explanations, as the codes will speak for themselves. Also we have tried to show that in *Mathematica* (like in the real world) there are many ways to approach a problem and solve it. We have tried to inspire the imagination!

Someone once rightly said the *Mathematica* programming language is rather a "Swiss army knife" containing a vast array of features. *Mathematica* provides us with a powerful mathematical functions. Along with this, one can freely mix different styles of programming, functional, list-base and procedural to achieve a lot. This mélange of programming styles is what we promote in this note.

I mostly choose problems having something to do with numbers as they do not need any particular background.

Thus this note could be considered for a course in *Mathematica*, or for self study. It mostly concentrates on programming and problem solving in *Mathematica*. There are excellent books

written about *Mathematica* for example Ilan Vardi [3], Stan Wagon [4] or Shaw-Tigg [2] to name a few. The reader is encouraged to have a look at them as well.

Thanks: Ilan Vardi for his input, Brian McMaster for polishing my English

1. Introduction

In this section we give a quick introduction to the very basic things one can perform with *Mathematica*.

1.1. *Mathematica* as a calculator.

Mathematica can be used as a calculator with the basic arithmetic operations $+, -, *, /$ and ˆ for powers.

```
2682440^4 + 15365639^4 + 18796760^4
```

```
180630077292169281088848499041
```

```
20615673^4
```

```
180630077292169281088848499041
```

This shows that $2682440^4 + 15365639^4 + 18796760^4 = 20615673^4$, disproving a conjecture by Euler that three fourth powers can never sum to a fourth power. (This conjecture remained open for almost 200 years, until Noam Elkies at Harvard came up with the above counterexample in 1988)

Mathematica can handle large calculations:

```
2^9941-1
```

```
346088282490851215242960395767413316722628668900238547790489283445006220809834
114464364375544153707533664486747635050186414707093323739706083766904042292657
896479937097603584695523190454849100503041498098185402835071596835622329419680
597622813345447397208492609048551927706260549117935903890607959811638387214329
942787636330953774381948448664711249676857988881722120330008214696844649561469
971941269212843362064633138595375772004624420290646813260875582574884704893842
439892702368849786430630930044229396033700105465953863020090730439444822025590
974067005973305707995078329631309387398850801984162586351945229130425629366798
595874957210311737477964188950607019417175060019371524300323636319342657985162
360474512090898647074307803622983070381934454864937566479918042587755749738339
033157350828910293923593527586171850199425548346718610745487724398807296062449
119400666801128238240958164582617618617466040348020564668231437182554927847793
809917495802552633233265364577438941508489539699028185300578708762293298033382
857354192282590221696026655322108347896020516865460114667379813060562474800550
717182503337375022673073441785129507385943306843408026982289639865627325971753
720872956490728302897497713583308679515087108592167432185229188116706374484964
985490944305412774440794079895398574694527721321665808857543604774088429133272
929486968974961416149197398454328358943244736013876096437505146992150326837445
270717186840918321709483693962800611845937461435890688111902531018735953191561
073191960711505984880700270887058427496052030631941911669221061761576093672419
481606259890321279847480810753243826320939137964446657006013912783603230022674
342951943256072806612601193787194051514975551875492521342643946459638539649133
096977765333294018221580031828892780723686021289827103066181151189641318936578
454002968600124203913769646701839835949541124845655973124607377987770920717067
108245037074572201550158995917662449577680068024829766739203929954101642247764
456712221498036579277084129255555428170455724308463899881299605192273139872912
009020608820607337620758922994736664058974270358117868798756943150786544200556
034696253093996539559323104664300391464658054529650140400194238975526755347682
486246319514314931881709059725887801118502811905590736777711874328140886786742
863021082751492584771012964518336519797173751709005056736459646963553313698192
960002673895832892991267383457269803259989559975011766642010428885460856994464
428341952329487874884105957501974387863531192042108558046924605825338329677719
469114599019213249849688100211899682849413315731640563047254808689218234425381
```

995903838524127868408334796114199701017929783556536507553291382986542462253468
272075036067407459569581273837487178259185274731649705820951813129055192427102
805730231455547936284990105092960558497123779789849218399970374158976741548307
086291454847245367245726224501314799926816843104644494390222505048592508347618
947888895525278984009881962000148685756402331365091456281271913548582750839078

914699790194262248837894635 51

If a number of the form $2^n - 1$ happens to be prime, it is called a Mersenne prime. Recall that a prime number is a number which is divisible only by 1 and itself. It is easy to see $2^2 - 1$ and $2^3 - 1$ and $2^5 - 1$ are Mersenne primes. The list continues. In 1963, Gillies found that the above number, $2^{9941} - 1$, is a Mersenne prime. With my laptop it takes 16 seconds for *Mathematica* to check that this is a prime number.[1]

```
PrimeQ[2^9941-1]

True
```

Back to easier calculations:

```
24/17
```
$\frac{24}{17}$

Mathematica always tries to give a precise value, thus gives back $\frac{24}{17}$ instead of attempting to evaluate the fraction.

```
Sin[Pi/5]
```
$\frac{1}{2}\sqrt{\frac{1}{2}(5-\sqrt{5})}$

In order to get the numerical value, one can use the function `N[]`.

```
N[24/17]
1.41176

N[24/17, 20]
1.4117647058823529412

?N

N[expr] gives the numerical value of expr. N[expr, n] attempts to
give a result with n-digit precision.
```

All elementary mathematical functions are available here, Log, Exp, Sqrt, Sin, Cos, Tan, ArcSin, ... For a complete list have a look at MATHEMATICAL FUNCTIONS:ELEMENTARY FUNCTIONS in the *Mathematica*'s help.

[1] The largest Mersenne prime found so far is $2^{30402457} - 1$ which is discovered in December 2005

1.2. **Numbers.** Recall that one can decompose any number n as a product of powers of primes and this decomposition is unique, i.e, $n = p_1^{k_1} \cdots p_t^{k_t}$ where p_i's are prime. Thus $37534 = 2 \times 7^2 \times 383$. *Mathematica* can do all these:

```
FactorInteger[37534]
{{2,1},{7,2},{383,1}}
```

```
FactorInteger[6473434456376432]
{{2,4},{3239053,1},{124909859,1}}

PrimeQ[124909859]
True

Prime[8]
19
```

`Prime[n]` produces the n-th prime number. `PrimeQ[n]` determines whether n is a prime number.

In 1640 Fermat conjectured that the formula $2^{2^n} + 1$ always produces a prime number. Almost a hundred years later the first counterexample was found.

```
PrimeQ[2^(2^1)+1]
True

PrimeQ[2^(2^2)+1]
True

PrimeQ[2^(2^3)+1]
True

PrimeQ[2^(2^4)+1]
True

PrimeQ[2^(2^5)+1]
False

2^(2^5)+1
4294967297

FactorInteger[2^(2^5)+1]
{{641,1},{6700417,1}}
```

1.3. **Algebraic computations.** One of the abilities of *Mathematica* is to handle symbolic computations. Consider the expression $(x+1)^2$. One can use *Mathematica* to expand this expression:

```
Expand[(x+1)^2]
```
$1+2x+x^2$

Mathematica can also do the inverse of this task, namely to factorize an expression:

```
Factor[1 + 2x + x^2]
```
$(1+x)^2$

My favorite example is this one. Try to factorize the expression $x^{10}+x^5+1$. Here is one way to do that:

$$x^{10}+x^5+1=$$
$$x^{10}+x^9-x^9+x^8-x^8+\cdots+x^5-x^5+x^5+x^4-x^4+\cdots+x-x+1=$$
$$x^{10}+x^9+x^8-x^9-x^8-x^7+x^7+x^6+x^5-x^6-x^5-x^4+x^5+x^4+x^3-x^3-x^2-x+x^2+x+1=$$
$$x^8(x^2+x+1)-x^7(x^2+x+1)+x^5(x^2+x+1)-x^4(x^2+x+1)+x^3(x^2+x+1)-x(x^2+x+1)+x^2+x+1=$$
$$(x^2+x+1)(x^8-x^7+x^5-x^4+x^3-x+1).$$

Mathematica can easily come up with the factorization:

```
Factor[x^10 + x^5 + 1]
```
$(x^2+x+1)(1-x+x^3-x^4+x^5-x^7+x^8)$.

It is a fact that the product of four consecutive numbers plus one is always a squared number:

```
Factor[n*(n+1)*(n+2)*(n+3)+1]
```
$(1+3n+n^2)^2$

1.4. **Variables.** In order to feed data to a computer program one needs to define variables to be able to assign data to them. As long as you use common sense, any names you choose for variables are valid in *Mathematica*. Names like `x, y, x3, myfunc, xQuaternion, ...` are all fine. Do not use underscore _ to define a variable [2]. The underscore is reserved and will be used in the definition of functions in Section 2.

1.5. **Equalities, =, :=, ==.** Primarily there are three equalities in *Mathematica*. There is a fundamental differences between = and :=. Study the following example:

```
x=5;y=x+2;

y
7

x=10
10
```

[2]This is quite common in Pascal or C, to define variables such as `x_printer, com_graph, ...`

```
y
7

x=15
15

y
7
```

Now compare it with the following one, when we replace = with :=

```
x=5;y:=x+2;

y
7

x=10
10

y
12

x=15
15

y
17
```

From the example it is clear that when we define y=x+2 then y takes the value of x+2 and this will be assigned to y. No matter if x changes its value, the value of y remains the same. In other words, y is independent of x. But in y:=x+2, y is dependent on x, and when x changes, the value of y changes too. Namely using := then y is a function with variable x. The following is an excellent example to show the difference between = and :=.

```
?Random

Random[ ] gives a uniformly distributed pseudorandom Real in the
range 0 to 1.

x=Random[]
0.246748

x
0.246748
```

```
x
0.246748

x:=Random[]

x
0.60373

x
0.289076

x
0.564378
```

We will examine this difference between `=` and `:=` again in Example 3.1.
Finally the equality == is used to compare:

```
5==5
True

3==5
False
```

We will discuss more on this in Section 5.1.

2. Defining functions

2.1. **Formulas as functions.** Defining functions is one of the strong features of *Mathematica*. One can define a function in several different ways in *Mathematica* as we will see in the course of this lecture.

Let us start with a simple example of defining the formula $f(n) = n^2 + 4$ as a function and calculate $f(-2)$:

```
f[n_]:= n^2 +4
```

First notice that in defining a function we use :=. The symbol `n` is a dummy variable and as expected one plugs in the data in place of `n`.

```
f[-2]
8
```

In fact as we will see later, one can plug "anything" in place of `n` and that's why functions in *Mathematica* are far superior to those in Pascal or `C`.

One more note about the extra underscore _ in the definition of the function. The underscore which will be called blank here determines the "pattern" of x. We shall talk about patterns and pattern matching in Section 9 and leave it as it is for the moment.

We proceed by defining the function $g(x) = x + sin(x)$.

```
g[x_]:= x+Sin[x]
g[Pi]
```

π

One can define functions of several variables. Here is a simple example defining $f(x, y) = \sqrt{x^2 + y^2}$

```
f[x_,y_]:=Sqrt[x^2+y^2]
f[3,4]
5
```

It is very easy to compose functions in *Mathematica*, i.e., applying functions one after the other on data. Here is an example of this:

```
f[x_]:=x^2+1
g[x_]:=Sin[x]+Cos[x]
f[g[x]]
1+(Cos[x]+Sin[x])^2
g[f[x]]
Cos[1+x^2]+Sin[1+x^2]
```

And this is a little function to find out if the $n-th$ Fibonacci number is divisible by 5.

```
remain[n_]:=Mod[Fibonacci[n],5]
remain[14]
2
remain[15]
0
```

Thus the 15th Fibonacci number is divisible by 5. Note that the function `remain` is itself a composition of two functions, namely the functions `Fibonacci` and `Mod`.

Besides the traditional way of `remain[x]`, there are two other ways to apply a function to an argument as follows:

```
15//remain
0
remain@5
0
```

Later we will define functions with conditions, functions with several definitions and functions containing several lines of code (a procedure).

2.2. **Anonymous functions.** Sometimes we need to "define a function as we go" and use it on the spot. *Mathematica* enables us to define a function without giving it a name, use it, and then move on! Obviously if we need to use a specific function frequently, then the best way is to give it a name and define it as we did in Section 2.1. Here is an anonymous function equivalent to $f(x) = x^2 + 4$:

```
(#^2+4)&[5]
29
```

The expression `(#^2+4)&` defines a nameless function. As usual we can plug in data in place of `#`. The symbol `&` determines where the definition of the function is completed. Using anonymous functions, here is another way to find out if the 15th Fibonacci number is divisible by 5:

```
Fibonacci[15]//Mod[#,5]&
0
```

Here is an example of an anonymous function for $f(x, y) = \sqrt{x^2 + y^2}$

```
Sqrt[#1^2+#2^2]&[3,4]
5
```

As you might guess, `#1` and `#2` refer to the first and second variables in the functions.

3. Lists

One can think of a computer program as a function which accepts some (crude) data or information and gives back the data we would like to obtain. *Lists* are the way *Mathematica* handles information. Roughly speaking, a list is a collection of objects. The objects could be of any type and pattern. Let us start with an example of a list:

```
{1,-4/7,stuff,1-2x+x^2}
```

This looks like a mathematical set. One difference is that lists respect order:

```
{1,2}=={2,1}
False
```

The other difference is that a list can contain a copy of the same object several times:

```
{1,2,1}=={1,2}
False
```

The natural thing here is to be able to access the elements of a list.

```
p={x,1,-8/3,a,b,{c,d,e},radio}
p[[1]]
x
p[[5]]
{c,d,e}
p[[-1]]
radio
p[[{2,4}]]
{1,a}
p[[{-2,5}]]
{{c,d,e},b}
p[[-2,{2,3}]]
{d,e}
```

Examining the above examples reveals that p[[i]] gives the $i-th$ member of the list. There are other commands to access the elements of a list as follows:

```
First[p]
x
Last[p]
radio
Rest[p]
{1,-8/3,a,b,{c,d,e},radio}
Rest[%]
```

```
{-8/3,a,b,{c,d,e},radio}
Drop[p,3]
{a,b,{c,d,e},radio}
Take[p,2]
{x,1}
```

Most of these commands are self-explanatory and a close look at the above examples shows what each of them will do. All these commands and more are listed in the *Mathematica* Help under Element Extraction.

One of the secret of writing codes comfortably is if one would be able to manipulate lists easily. Often times in applications situations like the following arise:

- Given $\{x_1, x_2, x_3, \cdots, x_n\}$ and $\{y_1, y_2, y_3, \cdots, y_n\}$, produce $\{x_1, y_1, x_2, y_2, x_3, y_3, \cdots, x_n, y_n\}$.
- Given $\{x_1, x_2, \cdots, x_n\}$ produce $\{x, x_1 + x_2, \cdots, x_1 + x_2 + \cdots + x_n\}$

Mathematica provides us with commands to obtain the above arrangement easily. We will look at these commands in Section 7.3 and Section 7.4.

3.1. **Functions producing lists.** *Mathematica* provides us with commands of which the output is a list. These commands have a nature of repetition and replace loops in procedural programming. Let us look at some of them here before starting to write more serious codes.

```
Range[10]
{1,2,3,4,5,6,7,8,9,10}
Range[2,17,4]
{2,6,10,14}
?Range
Range[imax] generates the list {1, 2,..., imax}.  Range[imin, imax] generates the
list {imin,..., imax}.  Range[imin, imax, di] uses step di
```

Another command is `Table`.

```
In[9]:= Table[n^2+1,{n,1,13}]

Out[9]= {2,5,10,17,26,37,50,65,82,101,122,145,170}
```

```
Table[x^i + y^i, {i, 2, 17, 4}]
{x^2 + y^2, x^6 + y^6, x^10 +y^10, x^14 + y^14}
```

In `Table[n^2+1,{x,1,13}]`, `n` runs from 1 to 13 and each time the function `n^2+1` is evaluated. The second example shows how easily we can work symbolically in *Mathematica*. As in `Range`, in the second `Table`, `i` starts from 2 with steps 4. Here one more example how beautifully *Mathematica* can handle symbols

Table[x_i,{i,1,10}]
$\{x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9, x_{10}\}$

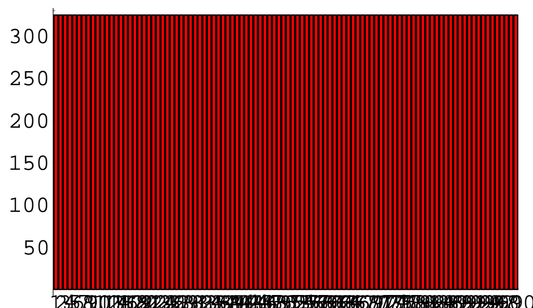
Here is a nice example showing the difference between two equality `=` and `:=`.

Example 3.1. This example uses `BarChart` which is available in the package `Graphics`. In order to make this command available, one needs to use `Needs["Graphics‘"]`. This example is the continuation of the discussion in Subsection 1.5.

```
Needs["Graphics‘"]
```

```
x=Random[Integer,{1,1000}];
```

```
BarChart[Table[x,{1000}]]
```




```
x:=Random[Integer,{1,1000}]
```

```
BarChart[Table[x,{1000}]]
```

In order to understand this example better, get the list generated by `Table[x,1000]` for each of the definitions of `x` individually and compare them.

3.2. **Listable functions.** There are times when we would like to apply a function to all the elements of a list. Suppose `f` is a function and `{a,b,c}` a list. We want to be able to "push" the function `f` inside the list and get `{f[a],f[b],f[c]}`. Many of *Mathematica*'s built-in functions have this property that they simply "go inside" a list. This property of a function is called *listable*. For example `Sqrt` is a listable function. We will use this function in the following, to show that the product of four consecutive numbers plus one is always a squared number.

```
Table[n(n+1)(n+2)(n+3)+1,{n,1,10}]
{25, 121, 361, 841, 1681, 3025, 5041, 7921, 11881, 17161}
Sqrt[%]
{5, 11, 19, 29, 41, 55, 71, 89, 109, 131}
```

The equivalent shorthand to apply a function to a list is `/@` as follows:

```
Sqrt /@ {a,b,c}
```
$\{\sqrt{a}, \sqrt{c}, \sqrt{c}\}$

Here is another example:

```
Table[(1+x)^i,{i,5}]
{1 + x, (1 + x)^2, (1 + x)^3, (1 + x)^4, (1 + x)^5}
Expand /@ %
{1 + x, 1 + 2 x + x^2, 1 + 3 x + 3 x^2 + x^3, 1 + 4 x + 6 x^2 + 4 x^3 + x^4, 1 +
5 x + 10 x^2 + 10 x^3 + 5 x^4 + x^5}
```

Problem 3.2. *The formula $n^2 + n + 41$ has a very interesting property. Observe that this formula produces prime numbers for n from 0 to 39.*

Solution.
First we produce the numbers :

```
In[7]:= Table[n^2+n+41,{n,1,40}]

Out[7]=
{43,47,53,61,71,83,97,113,131,151,173,197,223,251,281,313,347,383,421,461,503,
547,593,641,691,743,797,853,911,971,1033,1097,1163,1231,1301,1373,1447,1523,
1601,1681}
```

Then we apply `PrimeQ` to this list. This function is listable.

```
In[8]:=
PrimeQ[%]

Out[8]=
{True,True,True,True,True,True,True,True,True,True,True,True,True,True,True,
True,True,True,True,True,True,True,True,True,True,True,True,True,True,True,
True,True,True,True,True,True,True,True,True,False}
```

One wonders if one changes 41 to another number in the formula $n^2 + n + 41$ whether one gets more consecutive prime numbers. We will examine this in Problem 7.2.

We are ready to write the first serious code. In this problem we use the function `PrimeQ`. This is a listable function.

Problem 3.3. *How many numbers of the form* $3n^5 + 11$*, when n varies from* 1 *to* 2000*, are prime?*

Solution. First, let us produce the first 20 numbers of this form.

```
plist=Table[3n ^5+11,{n,1,20}]
{14, 107, 740, 3083, 9386, 23339, 50432, 98315, 177158, 300011, 483164, 746507, 1113890,
1613483, 2278136, 3145739, 4259582, 5668715, 7428308, 9600011}
```

The next step would be to apply `PrimeQ` to all the numbers and find out which ones are prime. Since this is a listable function this is enough:

```
PrimeQ[plist]
{False, True, False, True, False, True, False, False, False, False, False, True,
False, True, False, True, False, False, False, False}
```

Now we are left to count the number of `True` ones. This is do-able here, 6 of these numbers are prime.

But *Mathematica* gives us the ability to select, from the elements of a list, the desired ones. The command `Select` is the one which selects the elements which satisfy a desired property (or a desired pattern, more about this later).

```
Select[plist,PrimeQ]
{107, 3083, 23339, 746507, 1613483, 3145739}
```

These are prime numbers in the list `plist`. The command `Length` gives the length of a list. If we assemble all the steps in one line we have

```
Length[Select[Table[3n ^5+11,{n,1,20}],PrimeQ]]
6
```

Thus to find out how many numbers of the form $3n^5 + 11$ is prime when n runs from 1 to 2000, all we have to do is to change 20 to 2000:

```
Length[Select[Table[3n ^5+11,{n,1,2000}],PrimeQ]]
97
```

Let us look at another example with the same nature. The following example shows that anonymous functions fits very well with `Select`.

Problem 3.4. *For which $1 \leq n \leq 1000$ does the formula $2^n + 1$ produce a prime number?*

Solution. Here is the solution:

```
Select[Range[1000],PrimeQ[2^(#)+1]&]
{1, 2, 4, 8, 16}
```

Let us take a deep breath and go through this one-liner code slowly. The function `PrimeQ[2^(#)+1]&` is an anonymous function which gives `True` if the number $2^n + 1$ is prime and `False` otherwise.

```
PrimeQ[2^(#)+1]&[2]
True
```

`Range[1000]` creates a list containing the numbers from 1 to 1000. The command `Select` applies the anonymous function above to each element of this list and in case the result is true, the element will be selected. Thus `{1, 2, 4, 8, 16}` are the only numbers that make $2^n + 1$ a prime number.

Problem 3.5. *Notice that $12^2 = 144$ and $21^2 = 441$, namely the numbers and their squares are reverses of each other. Find all the numbers up to $10,000$ with this property.*

Solution. We need to introduce some new built-in functions. `IntegerDigits[n]` gives a list of the decimal digits in the integer n. We also need `Reverse` and `FromDigits`:

```
IntegerDigits[80972]
{8,0,9,7,2}
Reverse[%]
{2,7,9,0,8}
FromDigits[%]
27908
```

Thus the above shows we can easily produce the reverse of a number:

```
re[n_]:=FromDigits[Reverse[IntegereDigits[n]]]
re[634554]
455436
```

Having this function under our belt, the solution to the problem is just one line. Notice that the problem is asking for the numbers n such that re[n^2]=re[n]^2 .

```
Select[Range[10000],re[#^2]==re[#]^2&]
{1, 2, 3, 10, 11, 12, 13, 20, 21, 22, 30, 31, 100, 101, 102, 103, 110, 111, 112,
113, 120, 121, 122, 130, 200, 201, 202, 210, 211, 212, 220, 221, 300, 301, 310, 311,
1000, 1001, 1002, 1003, 1010, 1011, 1012, 1013, 1020, 1021, 1022, 1030, 1031, 1100,
1101, 1102, 1103, 1110, 1111, 1112, 1113, 1120, 1121, 1122, 1130, 1200, 1201, 1202,
1210, 1211, 1212, 1220, 1300, 1301, 2000, 2001, 2002, 2010, 2011, 2012, 2020, 2021,
```

```
2022, 2100, 2101, 2102, 2110, 2111, 2120, 2121, 2200, 2201, 2202, 2210, 2211, 3000,
3001, 3010, 3011, 3100, 3101, 3110, 3111, 10000}
```

Here is one more example using the command `FromDigits`. We know that 11 is a prime number. One wonders what is the next prime number consisting only of ones. A wild guess, a number with 23 digits all one? All we have to do is to produce this number then, with `PrimeQ`, test whether this is prime. Here is one way to generate this number.

```
Table[1, {23}]
{1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1}

FromDigits[%]
11111111111111111111111

PrimeQ[%]
True
```

Here is the code to find out which numbers of this kind up to 500 digits are prime.

```
Select[Range[500], PrimeQ[FromDigits[Table[1, {#}]]] &]

{2, 19, 23, 317}
```

The idea of sending a function into a list, i.e., applying a function to each element of a list, seems to be a good one. We already mentioned that the listable built-in functions are able to go inside a list, like `PrimeQ` or `Prime`. Have a look at the Attributions of Prime in the following:

```
??Prime
Prime[n] gives the nth prime number.
Attributes[Prime] = {Listable, Protected}
```

The command `Map` enables us to force any function, including user-defined functions, to go inside a list.

Here is the first example. Without defining the symbol `s`, we will map it to a list:

```
Map[s,Range[10]]
{s[1], s[2], s[3], s[4], s[5], s[6], s[7], s[8], s[9], s[10]}
```

The equivalent (and shorthand) way to write the code above is:

```
s/@ Range[10]
{s[1], s[2], s[3], s[4], s[5], s[6], s[7], s[8], s[9], s[10]}
```

`Map` fits well with pure functions:

```
Map[1+#^2&,{x,y,x}]
```

```
{1 + x^2, 1 + y^2, 1 + z^2}
```

Problem 3.6. *What digit does not appear as the last digit of the first 20 Fibonacci numbers?*

Solution. This one-liner code collects all the digits which appears as the last digit:

```
Union[Last /@ (IntegerDigits /@ (Fibonacci /@ Range[20]))]
{0, 1, 2, 3, 4, 5, 7, 8, 9}
```

Thus 6 is the only digit which is not present. Let us understand this code. Recall that /@ applies a function to a list. `Fibonacci /@ Range[20]` produces the first 20 Fibonacci numbers.

```
Fibonacci /@ Range[20]
{1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584,  4181,
6765, 10946, 17711, 28657, 46368, 75025}
```

Then `IntegerDigits` would go inside this list and get the digits of each number and then the function `Last` will get the last digits as required. `Union` will get rid of any repetition in the list.

Since `Fibonacci` and `IntegerDigits` are listable functions, one can also write the above code as follows:

```
Union[Last /@ IntegerDigits[Fibonacci[Range[20]]]]
```

If one does not want to use `IntegerDigits` then one can use the `Mod` function to get access to the last digit of a number.

```
?Mod

Mod[m, n] gives the remainder on division of m by n.

Mod[264,10]

4

?Quotient

Quotient[m, n] gives the integer quotient of m and n.

Quotient[264,10]

26

26*10+4

264
```

Thus another way to write the code is as follows. Note that `Mod` is also a listable function.

```
Union[Mod[Fibonacci[Range[20]],10]]
```

In Section 5.2 we will see how to use *Mathematica* to get the digit 6, namely to handle sets.

Recall that one can decompose any number n as a product of powers of primes and this decomposition is unique, i.e., $n = p_1^{k_1} \cdots p_t^{k_t}$ where p_i's are prime. Let us call a number a *square free* number if, in its decomposition to primes, all the k_i's are 1. Namely, no power of primes can divide this number. Thus $15 = 3 \times 5$ is a square free number but $16 = 2^4$ is not.

Recall that `Select[list,f]`, will apply the function `f` (which returns `True` or `False`) to all the elements say x of the `list` and return those elements for which `f[x]` is true. There is an option in `Select` which make it possible to get only the first n elements of `Select` that `f` returns `True` as follows: `Select[list,f,n]`. This comes in very handy, as in many problems, we want to test the elements until something goes wrong or some desirable element comes up. The following example demonstrates this.

Problem 3.7. *Write a function* `squareFreeQ[n]` *that returns* `True` *if the number n is a square free number, and* `False` *otherwise.*

Solution. Here is the code:

```
squareFreeQ[n_]:=Select[Last/@FactorInteger[n],#≠1 &,1]=={}
```

We need to introduce some new built-in functions. `FactorInteger` gives the decomposition of a number into its prime factors, for example $12 = 2^2 \times 3^1$:

```
FactorInteger[12]
{{2, 2}, {3, 1}}
```

So it is clear that if $n = p_1^{k_1} \cdots p_t^{k_t}$, then

```
FactorInteger[n]
```
$\{\{p_1, k_1\}, \{p_2, k_2\}, \cdots, \{p_t, k_t\}\}$

Now all we have to do is to go through this list and see if all k_i's are one. So the first step is to apply `Last` to each list to discard p_i's and get k_i's.

```
Last /@ FactorInteger[n]
```
$\{k_1, k_2, \cdots, k_t\}$

Having this list, we shall go through the list one by one and examine if k_i's are one. The anonymous function `#≠ 1&` does exactly this. So `Select[{`$k_1, k_2, \cdots, k_t$`},#≠ 1&]` gives the list of k_i's which are not one. But in our case, looking for square free primes, it is enough if only one k_i is not one. Then the number is not square free. Thus we use an option of `Select` which goes through the list until it finds an element such that k_i is not one. So we need to modify

the code as `Select[`$\{k_1, k_2, \cdots, k_t\}$`,#`$\neq$ `1&,1]`. We are almost done. All we have to do is to see if this list is empty or not (namely is there any k_i not equal to one). And for this we compare `Select[`$\{k_1, k_2, \cdots, k_t\}$`,#`$\neq$ `1&,1]=={}`.

✠

We can solve the above problem later with a slightly different method (See Problem 4.1).

Problem 3.8. *Find out how many primes bigger than n and smaller than $2n$ exist, when n goes from 1 to 30.*

Solution. We define an anonymous function which finds all the primes bigger than n and smaller than $2n$ and then gets the size of this list. Once we are done with this, we apply this function to a list of numbers from 1 to 30. Our anonymous function looks like this: `Length[Select[Range[# + 1, 2 # - 1], PrimeQ]] &`.
Analyzing this, `Range[# + 1, 2 # - 1]` produces all the numbers between n and $2n$. Then `Select` finds out which of them are in fact prime. Then we use the command `Length` to get the number of elements of this list. One can optimize this a bit, as we don't need to look at the whole range of n to $2n$, as clearly even numbers are not prime so we can ignore them right from the beginning. But we leave it to the reader to do this. All we have to do now is to apply this function with `Map` or `/@` to numbers from to 1 to 30.

```
Length[Select[Range[# + 1, 2 # - 1], PrimeQ]] & /@ Range[30]

{0, 1, 1, 2, 1, 2, 2, 2, 3, 4, 3, 4, 3, 3, 4, 5, 4, 4, 4, 4, 5, 6,
5, 6, 6,  6, 7, 7, 6, 7}
```

This seems not to be the most efficient way to write this problem as each time we test the same numbers again and again whether they are primes. We solve this problem using another approach in Problem 7.1 using a `Do Loop`. Also the reader is encouraged to look at the built-in function `PrimePi`.

✠

4. Changing heads!

Let us for the moment be a bit abstract. *Mathematica* has a very consistent way of dealing with any expression. Any expression in *Mathematica* has the following presentation `head[arg1,arg2,...,argn]` where `head` and `arg` could be expressions themselves. To make this point clear let us use the command `FullForm` which shows how *Mathematica* considers an expression.

```
FullForm[a + b + c]
Plus[a, b, c]

FullForm[a*b*c]
Times[a, b, c]

FullForm[{a, b, c}]
List[a, b, c]
```

Notice that the expressions `a+b+c` and `{a,b,c}` which present very different things have such close presentations. Here `Plus` is a function and `a,b,c` are plugged into this function. `Plus` is the head of the expression `a+b+c`. One can see from the `FullForm` that the only difference between `a+b+c` and `{a,b,c}` is their heads! We can get the head of any expression:

```
Head[{a, b, c}]
List

Head[a + b + c]
Plus

{a, b, c}[[0]]
List
```

Mathematica gives us the ability to replace the head of an expression with another head. The consequence of this is simply mind-blowing!

This can be done with the command `Apply`. Here is the traditional example:

```
Apply[Plus,{a,b,c}]
a+b+c
```

It is not difficult to explain this. The full form of `{a,b,c}` is `List[a,b,c]` with the head `List`. All *Mathematica* does is to change the head `List` to `Plus`, thus we have `Plus[a,b,c]` which is `a+b+c`.

The shorthand for `Apply` is `@@`, as the following example shows:

```
Plus @@ Fibonacci[Range[10]]/10.
14.3
```

This gives the average of the first 10 Fibonacci numbers.

Here are two more examples. The first one defines a function $ep(n) = 1 + \frac{1}{1} + \frac{1}{2!} + \cdots + \frac{1}{n!}$ and the other $p(n) = (1 + x)(1 + x^2) \cdots (1 + x^n)$.

```
ep[n_]:=1.+Plus @@ (1/Range[n]!)

p[n_]:=Times @@ (1+x^Range[n])
```

Let us look at the first example closely. `Range[n]` produces a list $\{1, 2, 3, \cdots, n\}$. Note that the factorial function `!` is listable, thus `Range[n]!` would produce $\{1!, 2!, 3!, \cdots, n!\}$. We should also note that all the arithmetic operations are listable, including `/` . Thus `1/Range[n]!` produces $\{\frac{1}{1!}, \frac{1}{2!}, \frac{1}{3!}, \cdots, \frac{1}{n!}\}$. We are almost there, all we have to do is to replace the head of $\{\frac{1}{1!}, \frac{1}{2!}, \frac{1}{3!}, \cdots, \frac{1}{n!}\}$ which is a `List` with `Plus` and as a result get $\frac{1}{1!} + \frac{1}{2!} + \cdots + \frac{1}{n!}$.

Both these are classical examples of using `Sum` and `Product` which are available in *Mathematica*. We will see these commands later.

Let's look at Problem 3.7 again.

Problem 4.1. *Write a function* `squareFreeQ[n]` *that returns* `True` *if the number n is a square free number, and* `False` *otherwise.*

Solution.

```
squareFree1Q[n_] := Times @@ Last /@ FactorInteger[n] == 1

squareFree1Q /@ {12, 13, 14, 25, 26}

{False, True, True, False, True}
```

If $n = p_1^{k_1} \cdots p_t^{k_t}$, then `FactorInteger[n]` will produce $\{\{p_1, k_1\}, \{p_2, k_2\}, \cdots, \{p_t, k_t\}\}$. We are after numbers such that all the k_i are 1 in the decomposition. Thus we can get all the k_i, multiply them and if we get anything other than 1, then this would be a non-square free number. Thus the first step is to apply `Last` to the list `Last /@ FactorInteger[n]` to get $\{k_1, k_2, \cdots, k_r\}$. Then all we have to do is to multiply them all together, and here comes the `Times @@` to change the head of $\{k_1, k_2, \cdots, k_r\}$ from `List` to `Times`.

✠

Problem 4.2. *Find all the numbers up to one million which have the following property: if $n = d_1 d_2 \cdots d_k$ then $n = d_1! + d_2! + \cdots + d_k!$ (e.g. $145 = 1! + 4! + 5!$).*

Solution

```
Select[Range[1000000], Plus @@ Factorial /@ IntegerDigits[#] == #
&]
{1, 2, 145, 40585}
```

The code consists of an anonymous function which for any n checks whether it has the desired property of the problem. Then by using `Select`, we check the list of all the numbers from 1 to one million, `Range[1000000]`. Our anonymous function is `Plus @@ Factorial /@ IntegerDigits[#] == # &`. Let's look at the left hand side of `==`. The built-in function `IntegerDigits[#]` applying to $n = d_1 d_2 \cdots d_k$ produces the list of digits of n, namely $\{d_1, d_2, \cdots, d_k\}$. Next applying `Factorial /@` to this list, we get $\{d_1!, d_2!, \cdots, d_k!\}$. Now all we need is to get the sum of elements of this list, and this is possible by changing the head from `List` to `Plus` by `Plus @@`. Once this is done, we compare the left hand side of `==` with the right hand side which is the original number `#`.

Problem 4.3. *A number is perfect if it is equal to the sum of its proper divisors, e.g.,* $6 = 1+2+3$ *but* $18 \neq 1+2+3+6+9$*. Write a program to find all the perfect numbers up to 10000 (Hint, have a look at the command* `Divisors`*).*

Solution. Here is a step-by-step approach to the solution.

```
Divisors[6]

{1,2,3,6}

Most[Divisors[6]]

{1,2,3}

Apply[Plus,Most[Divisors[6]]]

6

Select[Range[10000], # == Apply[Plus, Most[Divisors[#]]] &]

{6, 28, 496, 8128}
```

The numbers 6, 28 and 496 were already known as perfect numbers 2000 years before Christ. A glance at the list shows that all the perfect numbers we have found are even. It is still unknown whether there is an odd perfect number. Probably this is the oldest unsolved question in mathematics!

Problem 4.4. *Among the first one million numbers, what is the largest number* n *which is divisible by all positive integers* $\leq \sqrt{n}$*?*

Solution

```
Select[Range[100000],(Mod[#,LCM @@ Range[Floor[Sqrt[#]]]]==0)&]

{1, 2, 3, 4, 6, 8, 12, 24}
```

Exercise 4.5. *Decipher what the following codes do:*

```
g[n_] := Times @@ Apply[Plus , Inner[List, x^Range[n],
1/x^Range[n], List], 1]

t[n_] := Times @@ Apply[Plus, Thread[List[x^Range[n],
1/x^Range[n]]], 1]
```

Exercise 4.6. *Find all the numbers up to one million which have the following property: if* $n = p_1^{k_1} \cdots p_t^{k_t}$ *is the prime decomposition of* n *then* $n = k_1 \times p_1 + k_2 \times p_2 + \cdots + k_t \times p_t$*.*

5. A bit of Logic and Set Theory

5.1. **Being logical.** In mathematical logic statements can have a value of `True, False` or undefined. (We don't want to go into detail here mainly because I don't know the detail!) This helps us to "make a decision" and write programs based on the value of a statement (I am thinking of the classical `If-Then` statement). We have seen `==` which compares the left hand side and the right hand side. Studying the following examples carefully will tell us how *Mathematica* approaches logical statements:

```
3^2+4^2==5^2
True

3^2+4^2>5^2
False

9Sqrt[10!] < 10Sqrt[9!]
False

(x-1)(x+1)==x^2-1
(x-1)(x+1)==x^2-1

Simplify[%]
True

x==5
x==5

{1,2}=={2,1}
False

{a,b}=={b,a}
{a,b}=={b,a}
```

As one notices, *Mathematica* echoes back the expressions that it can't evaluate (e.g., x==5). Among them `{a,b}=={b,a}`, although one expect to get `False` as lists respect order. This is because *Mathematica* does not know about the values of a and b, and in case a and b are the same then `{a,b}=={b,a}` is `True`, and `False` otherwise.

One can combine logical statements with usual operations `And, Or, Not,...` or the equivalent &&, ||, ! as the following examples show:

```
2 > 3 && 3 > 2
False

And[2 > 3, 3 > 2]
False
```

```
1 < 2 < 3
True

2 > 3 || 3 < 2
True

Or[2 > 3, 3 > 2]
True


3^2+4^2>= 5^2
True
```

In general $A\&\&B$ is false if one of A or B is false and $A||B$ is true if one of them is true. In order to produce all possible combinations of true and false we use the command `Outer` as the following example shows

```
Outer[f, {a, b}, {x, y}]

{{f[a, x], f[a, y]}, {f[b, x], f[b, y]}}
```

Thus if in the above we replace `f` with `And` or `Or` we will get all the possible combinations of true and false.

```
Outer[And, {True, False}, {True, False}]
{{True, False},
{False,False}}

Outer[Or, {True, False}, {True, False}]
{{True, True}, {True,
False}}
```

One can specify the domains `Algebraics, Booleans, Complexes, Integers, Primes, Rationals` and `Reals`, for a variable. Look at the following examples:

```
Pi ∈ Rationals
False

Plus @@ Sqrt[Range[1, 7, 2]] ∈ Algebraics
True
```

The last example shows that $1+\sqrt{3}+\sqrt{5}+\sqrt{7}$ is an algebraic number (i.e. is a solution of a polynomial equation with integer coefficients).

One can use membership ($\in$) to approach some problems.

Problem 5.1. *Is the formula* $(n!)^2+1$ *a prime number for* $n=1$ *to* 6*?*

Solution.

```
(#!^2 + 1) & /@ Range[6] ∈ Primes
False
```

Here we first apply the anonymous function `(#!^2 + 1)` which is the formula $(n!)^2 + 1$ to the list containing 1 to 6. Then we ask *Mathematica* if the elements of this list belong to the domain `Primes`. The answer is `False`. The following code shows that the above formula does not produce a prime number for $n = 6$:

```
PrimeQ /@ ( (#!^2 + 1) & /@ Range[1, 6])
{True, True, True, True, True, False}
```

✠

One should be careful that *Mathematica* cannot (yet) perform miracles. For example, one can prove that $\sqrt[3]{2+\sqrt{5}} + \sqrt[3]{2-\sqrt{5}}$ is an integer, but

```
(2 + 5^(1/2))^(1/3) + (2 - 5^(1/2))^(1/3) ∈ Integers
False
```

Mathematica provides the logical quantifiers $\forall$, $\exists$ and $\Rightarrow$ with `ForAll`, `Exits` and `Implies` commands. But these seem to be not that powerful. For example one cannot prove Fermat's little theorem which says $2^{p-1} \equiv 1 (\text{mod } p)$ where $p > 2$ is a prime number with them!.

```
ForAll[p, p ∈ Primes, Mod[2^(p - 1), p] == 1]
```

Or even an easy fact that the product of four consecutive numbers plus one is a squared number.

```
Implies[n ∈ Integers && n > 0, Sqrt[n(n + 1)(n + 2)(n + 3) + 1] ∈ Integers]
```

In both cases *Mathematica* gives back the same expression, indicating she cannot decide on them.

5.2. **Handling sets.** Now it has been agreed that *any* mathematics starts by considering a set, i.e., a collection of objects. As we mentioned, the difference between mathematical sets and lists in *Mathematica* is the fact that lists respect order and repetition, which is to say one can have several copies of one object in a list. Sets are not sensitive about repeated objects, e.g., the set $\{a, b\}$ is the same as the set $\{a, b, b, a\}$. There is no concept of sets in *Mathematica* and if necessary one considers a list as a set.

If one wants to get rid of duplications in a list, one can use

```
Union[{a,b,b,a}]

{a,b}
```

Considering two sets, the natural operations between them are union and intersection. *Mathematica* provides `Union` to collects all elements from different lists in one list (after removing all the duplications) and `Intersection` for collecting common elements (again discarding repeated elements) . The following examples show how these commands work.

```
u= {1, 2, 3, 4, 5, 2, 4, 7, 4}; a = {1, 4, 7, 3}; b = {5, 4, 3, 2};
```

```
Union[u]
{1, 2, 3, 4, 5, 7}

Complement[u, a]
{2, 5}

?Complement
Complement[eall, e1, e2, ... ] gives the elements in eall which are not in any
of the ei.

Complement[u, a ∩ b] == Complement[u, a] ∪ Complement[u, b]
True
```

The first example shows `Union[list]` will get rid of repetition in a list. The command `Complement[u,a]` will give the elements of `u` which are not in `a`. From the example one can see that `a ∩ b` is acceptable in *Mathematica* and is a shorthand for `Intersection[a,b]`. In the last example we checked a theorem of set theory namely $(A \cap B)^c = A^c \cup B^c$ where ^{c} stands for complement.

Example 5.2. The following trick will be used later (inside a loop) to collect data.

```
A={}
A=A ⋃ {x}
{x}
A=A ⋃ {y}
{x,y}
A=A ⋃ {z}
{x,y,z}
```

This is the same traditional trick as `sum=sum+i`. Each time `sum=sum+i` is performed, `i` will be added to `sum` and this result will be the value of `sum`.

There are other ways to add an element to a list.

```
Append[{a,b,c},d]

{a,b,c,d}

A={};A=Append[A,x]

{x}

A=Append[A,y]

{x,y}

A=Append[A,z]

{x,y,z}
```

`AppendTo[s, elem]` is equivalent to `s = Append[s, elem]`

5.3. Decision making, If statement.

6. Sums and products

In the previous section we could write a code to calculate the series $ep(n) = 1 + \frac{1}{1} + \frac{1}{2!} + \cdots + \frac{1}{n!}$. *Mathematica* offers us two commands, namely `Sum` and `Product` to easily handle these problems as the following examples show:

```
Sum[s[i], {i, 1, 7}]
s[1] + s[2] + s[3] + s[4] + s[5] + s[6] + s[7]

Sum[s[i], {i, 1, k}]
```
$\sum_{i=1}^{k} s[i]$

The second example shows again that *Mathematica* can handle things symbolically.

Problem 6.1. *Write a function* $ep(n) = 1 + \frac{1}{1} + \frac{1}{2!} + \cdots + \frac{1}{n!}$

Solution Here is the code:

```
ep[n_] := 1 + Sum[1/k!, {k, 1, n}]

N[ep[100]]
2.71828

N[E]
2.71828
```

Sometimes *Mathematica* can do great things!

```
ep[Infinity]
```
$\mathbb{E}$

This shows that the above sequence converges to exp number.

Problem 6.2. *Prove that*

$$(1 + 2 + 3 + \cdots + n)^2 = (1^3 + 2^3 + 3^3 + \cdots + n^3).$$

Solution. Writing the above equality symbolically, we want to show $\sum_{i=1}^{n} i^3 = (\sum_{i=1}^{n} i)^2$. This example shows that *Mathematica* is aware of formulas for some certain sums, including the ones above:

```
p[n_] := Sum[i, {i, 1, n}]

p[n]

(1/2) n (1 + n)

p3[n_] := Sum[i^3, {i, 1, n}]

p3[n]

(1/4) n^2 (1+n)^2
```

```
p[n]^2==p3[n]
```

```
True
```

The above example shows *Mathematica* knows that $1+2+\cdots+n = \frac{n(n+1)}{2}$ and $1^3+2^3+\cdots+n^3 = (\frac{n(n+1)}{2})^2$. The first formula was known to Gauss at the age of seven. In fact he proved the formula as follows:

$$\begin{array}{cccc c} 1 & 2 & \cdots & n & + \\ n & n-1 & \cdots & 1 & \\ \hline n+1 & n+1 & \cdots & n+1 & \end{array}$$

Thus twice the sum of the series is $n(n+1)$ and thus the formula. The second formula follows by an easy induction.

✠

Problem 6.3. *Write a function to calculate the following sequence*

$$p(n) = \frac{1}{1} + \frac{1}{1+2} + ... + \frac{1}{1+2+...+n}$$

Solution. A glance at the sequence shows that there are in fact two sequences involved. Thus one needs two `Sum`, one to take care of $1+2+\cdots+i$ and the other the sum of these expressions.

```
s[n_] := Sum[1/Sum[j, {j, 1, i}], {i, 1, n}]
```

One of the advantages of the front-end in *Mathematica* is to provide the ability of writing mathematics. Writing the above sequence using mathematical symbols, one has $\sum_{i=1}^{n}(\sum_{j=1}^{i} j)$.

Using the palette provided by *Mathematica*, one can enter exactly the same expression in the front-end and define the function `s` this way.

$s[n_] = \sum_{i=1}^{n}(\sum_{j=1}^{i} j)$

✠

Mathematica can easily handle complicated symbolic calculations as the following example demonstrates. Recall that the binomial coefficient $\binom{n}{k}$ stands for $\frac{n!}{k!(n-k)!}$. The command `Binomial[n, k]` is available.

Problem 6.4. *Define*

$$p(n) = \sum_{k=0}^{n} \binom{n}{k}^2 (1+x)^{2n-2k}(1-x)^{2k}$$

and show that, for any chosen n, the coefficients of x are positive.

Solution. We shall first translate the above formula into *Mathematica.*

```
p[n_] := Sum[Binomial[n, k]^2(1 + x)^(2n - 2k)(1 - x)^(2k), {k, 0,
n}]
```

```
p[3] (1 - x)^6 + 9(1 - x)^4(1 + x)^2 + 9(1 - x)^2(1 + x)^4 + (1
+x)^6
```

```
Expand[p[3]]
70 + 40x^2 + 36x^4 + 40x^6 + 70x^8
```

As one sees, all the coefficients are positive. One can gather these coefficients in a list

```
CoefficientList[Expand[p[7]], x]

{3432, 0, 1848, 0, 1512, 0, 1400, 0, 1400, 0, 1512, 0, 1848, 0,
3432}
```

This is one of my favourite examples of using `Sum`.

Problem 6.5. *Define* $S(k, n) = \sum_{i=1}^{n} i^k$. *Prove that*

$$\sum_{a=0}^{n} \frac{S(2, 3a+1)}{S(1, 3a+1)}$$

is always a square number.

Solution.

```
s[k_, n_] := Sum[i^k, {i, 1, n}]

Sum[s[2, 3a + 1]/s[1, 3a +1],{a,0, n}]

1 + n + n (1 + n)

Factor[%]
(1 + n)^2
```

The command `Product` performs in the same way.

```
Product[s[i], {i, 1, 7}]
s[1]s[2]s[3]s[4]s[5]s[6]s[7]

Product[s[i], {i, 1, k}]
```

$\prod_{i=1}^{k} s[i]$

Here is a code to produce $(x + 1/x)(x^2 + 1/x^2) \cdots (x^n + 1/x^n)$.

```
p[n_] := Product[(x^k + 1/x^k), {k, 1, n}]
```

Problem 6.6. *Define a function* $t(n)$, *which is the sum of all the remainders of division by* n *into the numbers* 1 *to* n.

Solution. Here are two ways to write this function, one using `Sum` and the other using the list-based programming of the previous section.

```
t[n_] := Sum[Mod[n, k], {k, 1, n}]

tt[n_] := Plus @@ (Mod[n, #] &/@ Range[n])
```

7. Loops

If we agree that the primary ability that a computer language provides is the ability to repeat a certain code "fast" then *Mathematica* provides three *loops* that enable us to repeat part of our codes. These are quite similar to the loops that exist in any procedural language like Pascal or C. The first and the simplest one is the `Do` loop. Here is the traditional example. The structure of the `Do` loop reminds one of the commands like `Sum` or `Table`.

```
Do[Print[i],{i,1,7}]

1

2

3

4

5

6

7
```

The code:

```
Do[f[i],
{i,1,1000000}]
```

repeats the expression `f[i]` one million times where `i` runs from 1 to 1000000. In fact this is equivalent to

```
f/@ Range[10000000]
```

Here is a little comparison.

```
Timing[Do[
    f[i],
    {i,1,10000000}]]

{6.93 Second,Null}

Timing[f/@ Range[10000000];]

{5.008 Second,Null}
```

Apart from writing a code which is faster, one needs to try to write codes in a way in which they are also readable.

We will write Problem 3.8 using a `Do Loop`.

Problem 7.1. *Find out how many primes bigger than n and smaller than $2n$ exist, when n goes from 1 to 30.*

Solution. First we find all prime numbers up to 60.

```
prime60=Select[Range[60],PrimeQ]

{2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59}
```

Now for any n we check how many prime numbers lie between n and $2n$. To do this, as in Problem 3.8, we create a list of numbers between n and $2n$, `Range[n+1,2n-1]` then using `Intersection` we find out how many prime numbers are in this interval `Range[i+1,2i-1]` $\bigcap$ `prime60`. Using `Length` we can find the number of the primes that lie in this interval. Once we have this, then using a `Do Loop` we run this code for n from 1 to 30.

```
p={}; Do[
AppendTo[p,Length[Range[i+1,2i-1] ⋂ prime60]], {i,1,30}
];p
```

✠

For our next application of a `Do` loop, recall Problem 3.2. The formula $n^2 + n + 41$ produces prime numbers when n runs from 0 to 39. This was noticed by Euler some 300 years ago. One wonders whether one gets more consecutive prime numbers for a different constant in the above formula. The next Problem examines this:

Problem 7.2. *Consider the formula n^2+n+i. Find out the number of consecutive primes (starting from $n = 0$) that one gets when i runs from 1 to 10,000.*

Solution.

One way to approach the problem is to write a code to find out how many consecutive primes one gets (starting from $n = 0$) for a fixed i in the formula $x^2 + x + i$. Once this is done, then one can use a `Do` loop to change the value of i from 1 up to 10000. The code which finds out the number of consecutive primes is the same in nature to Problem 3.7.

The code

```
Select[Range[100], (PrimeQ[#^2 + # + 41] == False &), 1]

{40}
```

returns the first number in the range of $\{0, \cdots, 100\}$ such that the formula $n^2 + n + 41$ does not return a prime number. All we have to do now is to assemble this code in a loop as follows:

```
A={}
Do[
  A=Union[A,Select[Range[100],(PrimeQ[#^2+#+i]==False&),1]],
  {i,1,10000}];A

{1,2,3,4,5,6,7,10,12,16,40}
```

A line such as

```
A=Union[A,Select[Range[100],(PrimeQ[#^2+#+i]==False&),1]]
```

which is equivalent to

```
A= A ⋃ Select[Range[100],(PrimeQ[#^2+#+i]==False&),1]
```

collects all new results in the list `A`. We have seen this trick in Example 5.2.

A glance at the result shows that $n^2 + n + 41$ produces the maximum number of consecutive primes as was noticed by Euler. As a matter of fact, the formula which produces 16 consecutive prime numbers is $n^2 + n + 17$ which was also found by Euler.

We shall see more examples of the `Do` loop later. The next loop is the `While` loop. This one operates on a boolean (`True` or `False`) statement and gives you the ability to repeat a block until the boolean statement becomes `False`.

Problem 7.3. *Find the first prime number consisting only of ones and bigger than* 11*.*

Solution. Here is the mystery code:

```
n=111;
While[!PrimeQ[n],
    n=10n+1];
Print[n]

1111111111111111111
```

Here `!PrimeQ[n]` is our boolean statement and `n=10n+1` is the code we want to repeat. The code `n=10n+1` simply gets the number `n` and places 1 at the far right of the number (right?). So the aim is to put as many 1s in front of the original `n` which is 111 here to get a prime number. The `While` loop does exactly this. It is going to repeat the above code until `!PrimeQ[n]` becomes `False`. That is until `PrimeQ[n]` becomes `True`, that is until `n` becomes prime. And that's what we are looking for.

As you can see the `While` loop has the form `While[condition,body]`. The body of the loop can consist of several lines separated by `;`.

Here is a little test to see that the result of the above code is consistent with the code we wrote on Page 18.

```
IntegerDigits[n]

{1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1}

Length[%]

19
```

Problem 7.4. *Find the closest prime number below a given number n.*

Solution. Here we still have an example which can be "naturally" written by `While`. Notice that the body of the loop contains one line.

```
n = Input["enter a number"]
While[! PrimeQ[n],
    n--];
Print[n]
```

`Input` opens a box and asks for a value. This is a good way if one wants to ask a user for data. Again `!PrimeQ[n]` returns `True` and keeps the loop repeating until `n` is prime. That's what the question asks.

Problem 7.5. *Find all prime numbers less than a given n.*

Solution. We will use the loop `While` to find one by one all the prime numbers smaller than n starting from the smallest prime number 2. Notice that here the body of `While` has two sentences.

```
i = 1; n = Input["enter a number?"]; pset = {};
While[Prime[i] ≤ n,
 pset = pset ∪ {Prime[i]};
 i++];
pset
```

Ok, for $n = 321$ we get all the prime numbers up to 321.

```
{2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59,
61, 67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113, 127, 131,
137, 139, 149, 151, 157, 163, 167, 173, 179, 181, 191, 193, 197,
199, 211, 223, 227, 229, 233, 239, 241, 251, 257, 263, 269, 271,
277, 281, 283, 293, 307, 311, 313, 317}
```

Here until `Prime[i]` is smaller than `n` the loop keeps collecting `Prime[i]` in a list `pset` which at the beginning we define empty (see Example 5.2). After each step we go a bit forward by adding one to i, that is `i++`, and repeat the same procedure again until `Prime[i]` is bigger than `n`.

The last loop in *Mathematica* is the `For` loop. Here is the easiest example:

```
For[i=5,i<10,i++,Print[i]]

5

6

7

8

9
```

The loop `For` consists of different parts as follows `For[init,condition,steps,body]`. The `init` part is where we initialize the variables we need to use in the body of the loop. in the above example this was `i=5`. The second part is where the loop checks whether a boolean expression

appears and is where we decide when to terminate the loop. Each of these parts can have several sentences which should be separated by `;`. Let us look at another example.

Problem 7.6. *Find the sum of the sequence*

$$\frac{1}{1+2}+\frac{2}{2+3}+\cdots+\frac{10}{10+11}.$$

Solution.

```
For[i = 1; sum = 0, i < 11, i++, sum += i/(i + i + 1)];
sum

64157087/14549535
```

Notice that the `init` part of the loop consist of two lines. Also notice that `sum+=i/(i + i + 1)` is a shorthand for `sum=sum+i/(i + i + 1)` as i++ is a shorthand for `i=i+1`. In the same way `i*=n` is a shorthand for `i=i*n`.

To refresh the memory, here are the other approaches to get the sum of the above sequence

```
Sum[i/(2i + 1), {i, 1, 10}]

64157087/14549535

Plus @@ (#/(2# + 1) & /@ Range[10])

64157087/14549535
```

One can leave out any part of a `For` loop. For example

```
For[ ,False , , Print["Never see the light of day"]
```

produces nothing. One can also see that `While[test,body]` is the same as `For[ ,test, , body]` and `Do[body,x,xmin,xmas,xinc]` is the same as `For[x=xmin,x≤xmax,x+=inc,body]`. But again, there are times when `While` makes the code more readable and there are times when `For` is a better choice.

Let us do some experiments:

```
Timing[Do[,{10^6}]]

{0.02 Second,Null}

Timing[Do[,{1000000}]]

{0.1 Second,Null}

Timing[i=1;While[i<10^6,i++]]

{2.614 Second,Null}
```

```
Timing[i=1;While[i<1000000,i++]]

{1.932 Second,Null}

Timing[For[i=1,i<10^6,i++]]

{2.654 Second,Null}

Timing[For[i=1,i<1000000,i++]]

{1.973 Second,Null}
```

Here is one more example.

Problem 7.7. *An integer $d_n d_{n-1} d_{n-2} \dots d_1$ is palindromic if $d_n d_{n-1} d_{n-2} \dots d_1 = d_1 d_2 \dots d_{n-1} d_n$ (for example 15651). Write a code to ask for a number $d_n d_{n-1} d_{n-2} \dots d_1$ and find out if it is palindromic. Enhance the code further such that if the number is not palindromic then the code tests whether $d_n d_{n-1} d_{n-2} \dots d_1 + d_1 d_2 \dots d_{n-1} d_n$ is (for example, 108+801=909). Furthermore write a code to give the number of times it is needed to repeat this procedure till one gets a palindromic number starting with $d_n d_{n-1} d_{n-2} \dots d_1$ (if it takes more than 150 times, let the function return infinity).*

Solution We start with an example. Let $n = 98$. We need to systematically check whether n is palindromic. If not, then produce 89, add this to $n = 98$ and check whether this is palindromic. We have seen how to produce the reverse of a number using `IntegerDigits, Reverse` and `FromDigits` (see Problem 3.5). Here is the first step

```
n=98;nlist=IntegerDigits[n]

{9,8}

If[ nlist != Reverse[nlist],n=n+FromDigits[Reverse[nlist]]]

187
```

If the result is not palindromic, one has to do the same procedure again. Thus we use a `While loop` to do this for us.

```
n = 98; nlist = IntegerDigits[n];
While[nlist != Reverse[nlist],
  n = n + FromDigits[Reverse[nlist]];
  nlist = IntegerDigits[n]
  ]; n

8813200023188
```

One can enhance the code:

```
n = Input["Enter a number"]; i = 1; nlist = IntegerDigits[n];
safetyNet = True;

While[nlist != Reverse[nlist] && safetyNet,
  Print[i, " ", n]; n = n + FromDigits[Reverse[nlist]]; i++;
  nlist = IntegerDigits[n];
  If[i > 150, safetyNet = False]
  ]
If[i > 150, Print["........Aborted"], Print[i, " ", n]
  ]
```

7.1. **Nested loops.** In many applications there are several factors (variables) which change simultaneously, and this calls for what we call a *nested loop.* Instead of trying to describe the situation, let us see some examples.

```
Do[
  Do[
    Print[i, " ", j],
    {j, 1, 2}
    ],
  {i, 1, 3}
  ]

1 1

1 2

2 1

2 2

3 1

3 2
```

The code contains two `Do` loops. In the inner one, the counter `j` runs from 1 to 2 and once this is done, the outer loop performs and the counter `i` goes one further and again the inner loop starts to run.

Problem 7.8. *Find all the pairs* (n, m) *such that* $n^2 + m^2$ *is a square number (e.g.* $3^2 + 4^2 = 5^2$*).*

Solution.

```
Do[
 Do[
 If[Sqrt[i^2 + j^2] ∈ Integers, Print[i, " ", j]],
 {j, i, 10}
 ],
```

```
{i, 1, 10}
]
```

Here is the result

```
3  4

6  8
```

Here the outer loop starts with the counter `i` getting the value 1. Then it is the turn of the block inside this loop, which is again another loop run. In the inner loop `{j, i, 10}` makes the counter `j` run from `i` to 10. This done, in the outer loop `i` takes 2 and then `j` runs from 2 to 10 and so on. The reader should see that this is enough to find all the pairs up to 10 with the desired property. Can you say how many times the `If` line is going to be performed?

We have already seen the command `Table` which provides a sort of loop. In fact `Table` can provide us with a nested loop.

```
Table[{i, j}, {i, 1, 3}, {j, 1, 2}]

{{{1, 1}, {1, 2}}, {{2, 1}, {2, 2}}, {{3, 1}, {3, 2}}}
```

One should compare this with the example of the nested `Do` loop. As the result shows, here `j` is the counter for the inner loop.

One of the issues that might arise here is that the output is a nested list (i.e. too many {). Sometimes we really do not need the nested list answer to our question. For example we want to come up with a code to solve Problem 7.8 by using `Table`. In order to get rid of extra "{", one can use the command `Flatten`.

```
Flatten[Table[{i, j}, {i, 1, 3}, {j, 1, 4}]]

{1, 1, 1, 2, 1, 3, 1, 4, 2, 1, 2, 2, 2, 3, 2, 4, 3, 1, 3, 2, 3, 3,
3, 4}
```

`Flatten` gets rid of all the lists inside a list, i.e., removes all the "{". In our problem we want a list of pairs. In this case we need

```
Flatten[Table[{i, j}, {i, 1, 3}, {j, 1, 4}],1]

{{1, 1}, {1, 2}, {1, 3}, {1, 4}, {2, 1}, {2, 2}, {2, 3}, {2, 4},
{3, 1}, {3, 2}, {3, 3}, {3, 4}}
```

Now we have all the pairs. But some of them are repeated. For us `{1,3}` is the same as `{3,1}`. So as in Problem 7.8, we need the inner counter to depend on the outer one as follows:

```
Flatten[Table[{i, j}, {i, 1, 3}, {j, i, 4}], 1]
```

```
{{1, 1}, {1, 2}, {1, 3}, {1, 4}, {2, 2}, {2, 3}, {2, 4}, {3, 3},
{3, 4}}
```

All we have to do now is to select the pairs (m, n) such that $\sqrt{m^2 + n^2} \in \mathbb{N}$.

```
Select[Flatten[Table[{i, j}, {i, 1, 10}, {j, i, 10}], 1],
(Sqrt[#[[1]]^2 + #[[2]]^2] ∈ Integers) &]
```

```
{{3, 4}, {6, 8} }
```

7.2. **Nest, NestList and more.** Let $f(x)$ be a function defined on a variable x. There are times when one needs to apply the function f on itself several times, i.e., $f(\cdots f(f(x))\cdots)$ (recall composition of functions). *Mathematica* provides a command to do exactly this:

```
Nest[f, x, 4]
```

```
f[f[f[f[x]]]]
```

If one wants to keep track of each step, the command `NestList` is available

```
NestList[f, x, 4]
```

```
{x, f[x], f[f[x]], f[f[f[x]]], f[f[f[f[x]]]]}
```

Here are some examples:

```
f[x_]:=1/(1+x)
Nest[f,x,4]
```

$$\frac{1}{1+\frac{1}{1+\frac{1}{1+\frac{1}{1+x}}}}$$

```
NestList[f,x,4]
```

$$\left\{\frac{1}{1+x}, \frac{1}{1+\frac{1}{1+x}}, \frac{1}{1+\frac{1}{1+\frac{1}{1+x}}}, \frac{1}{1+\frac{1}{1+\frac{1}{1+\frac{1}{1+x}}}}\right\}$$

```
NestList[Sqrt[#+6]&,Sqrt[6],4]
```

$$\{\sqrt{6}, \sqrt{6+\sqrt{6}}, \sqrt{6+\sqrt{6+\sqrt{6}}}, \sqrt{6+\sqrt{6+\sqrt{6+\sqrt{6}}}}, \sqrt{6+\sqrt{6+\sqrt{6+\sqrt{6+\sqrt{6}}}}}\}$$

There are two more commands of this type, `NestWhile` and `NestWhileList`.

```
?NestWhile
NestWhile[f, expr, test] starts with expr, then repeatedly applies f until  applying
test to the result no longer yields True.
```

The following problem uses `NestWhile`.

Problem 7.9. *A Happy number is a number such that if one squares its digits and adds them together, and then takes the result and squares its digits and add them together again and keep doing this process, one comes down to the number 1. Find all the Happy ages, i.e., happy numbers up to 100.*

Solution.

```
Select[Range[100],
  NestWhile[
        Plus @@ (IntegerDigits[#]^2)&,#,(!#==4) & (!#==1)&]==1&]

{1,7,10,13,19,23,28,31,32,44,49,68,70,79,82,86,91,94,97,100}
```

There is a more elegant approach to this problem using recursive functions in Problem 11.1. In any case one can observe that happy ages are mostly before one gets a job or after the retirement!

Ok, we are going to make up a problem and use `NestList` to get some answers.

Problem 7.10. *A number $a_1a_2\cdots a_n$ is called pure prime if $a_1a_2\cdots a_n$, $a_1a_2\cdots a_{n-1}, \cdots, a_1a_2$ and a_1 are all prime. Prove that pure prime numbers are finite in number and find all of them.*

Solution. First we have to find a way to drop the last digit of a number. The function `Quotient` might help

```
?Quotient

Quotient[m, n] gives the integer quotient of m and n.

Quotient[5937, 10]
593

Quotient[593, 10]
59

Quotient[59 , 10]
5
```

The above example shows that applying `Quotient` to a number repeatedly drops the last digit of the number one by one. Thus

```
NestList[Quotient[#, 10] &, 5937, 3]

{5937,593,59,5}
```

Now we have all the numbers. We only need to test whether all of them are prime.

```
PrimeQ /@ NestList[Quotient[#,10]&,5937,3]

{False, True, True, True}
```

Thus 5937 just misses being a pure prime. If we want to define this as a function, a little problem might arise. In the case of 5937 we have to apply `Quotient` three times to this number. But for a number n with arbitrary digits, we need to use `FixedPointList`.

```
?FixedPointList

FixedPointList[f, expr] generates a list giving the results of
```

```
applying f  repeatedly, starting with expr, until the results no
longer change.
```

```
FixedPointList[Quotient[#,10]&,5937]
```

```
{5937, 593, 59, 5, 0, 0}
```

```
FixedPointList[Quotient[#,10]&,7647653]
```

```
{7647653, 764765, 76476, 7647, 764, 76, 7, 0, 0}
```

It is clear that we have to drop the last two elements from the list.

```
Drop[FixedPointList[Quotient[#,10]&,5937],-2]
```

```
{5937, 593, 59, 5}
```

Now it is the time to apply `PrimeQ` to the list to check whether all these numbers are prime.

```
PrimeQ[Drop[FixedPointList[Quotient[#,10]&,5937],-2]]
```

```
{False,True,True,True}
```

What we need is a list containing of only `True`'s. Thus if only one of the numbers happens to be not prime, the whole number is not pure prime as is the case with 5937. We can combine all the boolean in the list with `And` and the result would make it clear whether the number is pure prime. Here is the code

```
Apply[And,{False,True,True,True}]
```

```
False
```

Thus putting all these together we have

```
purePrime[n_]:=Apply[And,PrimeQ[Drop[FixedPointList[Quotient[#,10]&,n],-2]]]
```

```
Select[Range[10, 99], purePrime]
```

```
{23,29,31,37,53,59,71,73,79}
```

```
Select[Range[100,999],purePrime]
```

```
{233, 239, 293, 311, 313, 317, 373, 379, 593, 599, 719, 733, 739,
797}
```

```
Select[Range[1000, 9999], purePrime]
```

```
{2333,2339,2393,2399,2939,3119,3137,3733,3739,3793,3797,5939,7193,7331,7333,
7393}
```

This seems not to be a good algorithm to find all pure prime numbers as it already takes some time to find all the 6-digit pure primes. In fact the problem here is that in order to find, say, all the 4-digit pure primes, the above algorithm has to check all the numbers from 1000 to 9999. But this is not necessary. The following example demonstrates this. If we know that 719 is pure prime then all we have to check to find the pure primes which have four digits and whose last three digits are 719, are the numbers $\{7190, 7191, ..., 7199\}$.

```
Range[719*10, 719*10 + 9]

{7190,7191,7192,7193,7194,7195,7196,7197,7198,7199}
```

We do not need to consider even numbers.

```
Range[719*10+1, 719*10 + 9,2]

{7191,7193,7195,7197,7199}
```

Now we need to find out which of these numbers are prime.

```
Select[%, PrimeQ]

{7193}
```

This shows that 7193 is a pure prime. Thus we start up with all one-digit primes and find all the two-digit primes as above.

```
purelist={2,3,5,7}

{2, 3, 5, 7}

Range[10#+1,10#+9,2]&[purelist]


{{21, 23, 25, 27, 29}, {31, 33, 35, 37, 39}, {51, 53, 55, 57, 59},
{71, 73, 75, 77, 79}}

Flatten[Range[10#+1,10#+9,2]&[purelist]]

{21, 23, 25, 27, 29, 31, 33, 35, 37, 39, 51, 53, 55, 57, 59, 71,
73, 75, 77, 79}

purelist=Select[Flatten[Range[10#+1,10#+9,2]&[purelist]],PrimeQ]

{23, 29, 31, 37, 53, 59, 71, 73, 79}
```

Thus these are all two-digit pure primes. Having them, we can immediately find all three-digit primes.

```
purelist=Select[Flatten[Range[10#+1,10#+9,2]&[purelist]],PrimeQ]
```

```
{233,239,293,311,313,317,373,379,593,599,719,733,739,797}
```

Thus our clever code to find all the pure primes is as follows:

```
purelist={2,3,5,7};
While[purelist != {},
  purelist=Select[Flatten[Range[10#+1,10#+9,2]&[purelist]],PrimeQ];
  Print[purelist]
  ]

{23,29,31,37,53,59,71,73,79}

{233,239,293,311,313,317,373,379,593,599,719,733,739,797}

{2333,2339,2393,2399,2939,3119,3137,3733,3739,3793,3797,5939,7193,7331,7333,
7393}

{23333,23339,23399,23993,29399,31193,31379,37337,37339,37397,59393,59399,
71933,73331,73939}

{233993,239933,293999,373379,373393,593933,593993,719333,739391,739393,739397,
739399}

{2339933,2399333,2939999,3733799,5939333,7393913,7393931,7393933}

{23399339,29399999,37337999,59393339,73939133}

{}
```

7.3. **Fold and FoldList.** Recall one of the questions we asked in Section 3, namely: Given $\{x_1, x_2, \cdots, x_n\}$ how one can produce $\{x, x_1 + x_2, \cdots, x_1 + x_2 + \cdots + x_n\}$?

Let us look at the commands `Fold` and `FoldList`.

```
Fold[f,x,{a,b,c}]

f[f[f[x,a],b],c]

FoldList[f,x,{a,b,c}]

{x,f[x,a],f[f[x,a],b],f[f[f[x,a],b],c]}
```

Replace the function `f` with `Plus` and `x` with `0` and observe what happens (see the following Problem for the answer).

Here is a use of `FoldList` to write another code for Problem 6.3.

Problem 7.11. *Write a function to calculate the sum of the following sequence?*

$$p(n) = \frac{1}{1} + \frac{1}{1+2} + ... + \frac{1}{1+2+...+n}$$

Solution. Here is the code:

```
p[n_]:= Plus @@ (1/Rest[FoldList[Plus, 0, Range[n]]])
```

In order to decipher this code, let us look at the standard example of `FoldList`.

```
FoldList[Plus,0,{a,b,c}]

{0,a,a+b,a+b+c}
```

Thus dropping the annoying 0 from the list:

```
Rest[FoldList[Plus,0,{a,b,c}]]

{a,a+b,a+b+c}
```

and

```
1/Rest[FoldList[Plus,0,{a,b,c}]]
```

$\{\frac{1}{a}, \frac{1}{a+b}, \frac{1}{a+b+c}\}$

makes the orignal code clear.

Problem 7.12. *For which natural numbers n is it possible to choose signs + and − in the expression*

$$1^2 \pm 2^2 \pm 3^2 \pm \cdots \pm n^2$$

so that the result is 0?

Solution.

One can find the following code in Vardi [3].

```
Fold[(#1/.x→x+#2)(#1/.x→x-#2)&,x,{a,b,c}]/.x→1

(1-a-b-c) (1+a-b-c) (1-a+b-c) (1+a+b-c) (1-a-b+c) (1+a-b+c) (1-a+b+c) (1+a+b+c)
```

Motivating with this, one can approach the problem.

```
Do[ If[(Fold[(#1/.x→x+#2)(#1/.x→x-#2)&,x, Range[n]^2]/.x→0) = 0,Print[n] ],
{n,1,40}]
7
8
11
12
15
16
19
20
23
$ Aborted
```

However, this seems to take time and there might be a better way to approach this problem.
Another approach:

```
 t[n_]:=Flatten[Outer[List,Sequence @@Table[{I
k,k}^2,{k,2,n}]],n-2]

Do[
  If[Select[t[n],Plus @@ #=-1&,1]\[!={},Print[n]],
  {n,3,40}]

7

8

11

12

15

16

19

20

Hold[Abort[],Abort[]]
```

7.4. **Inner and Outer.** Recall the follow question from Section 3: Given $\{x_1, x_2, x_3, \cdots, x_n\}$ and $\{y_1, y_2, y_3, \cdots, y_n\}$, how can one produce $\{x_1, y_1, x_2, y_2, x_3, y_3, \cdots, x_n, y_n\}$?

Let us look at two more commands `Inner` and `Outer`:

```
Inner[f,{a,b},{x,y},g]

g[f[a,x],f[b,y]]
```

If we replace the functions `f` and `g` with `List` we get:

```
Inner[List,{a,b},{x,y},List]

{{a,x},{b,y}}

Flatten[%]

{a,x,b,y}
```

Or this one to get rid of `Flatten`:

```
Inner[Sequence,{a,b},{x,y},List]
```

```
{a,x,b,y}
```

8. Substitution, *Mathematica* rules!

In *Mathematica* one can substitute an expression with another using *rules*. In particular one can substitute a variable with a value without assigning the value to the variable. Here is how it goes:

```
x+y/.x→2
2+y
```

If we ask for the value of x, we see

```
x
x
FullForm[x + y /.  x →2]
Plus[2, y]
```

The following examples show the variety of things one can do with rules.

x+y/.$\{x \to a, y \to b\}$
a+b

$x^2 + y/.x \to y/.y \to x$
$x + x^2$

Solve[$x^3 - 4x^2 + 4x^2 + 1$== 0]
$\{\{x\to -1\},\{x\to (-1)^{\frac{1}{3}}\}\ \{x\to -(-1)^{\frac{2}{3}}\}\}$

x/.%
$\{-1, (-1)^{\frac{1}{3}}, -(-1)^{\frac{2}{3}}\}$

$x + 2y/.\{x \to y, y \to a\}$
$2a + y$

The last example reveals that *Mathematica* goes through the expression only once and replaces the rules. If we need *Mathematica* to go through the expression again and replace any expression which is possible until no substitution is possible, one uses `//.` as follows:

$x + 2y//.\{x \to y, y \to a\}$
$3a$

In fact `/.` and `//.` are shorthand for `Replace` and `ReplaceRepeated` respectively.

```
ReplaceRepeated[x+2y, { x → y, y → x }]
ReplaceRepeated::rrlim:  Exiting after x + 2y scanned 65536 times.
x+2y

ReplaceRepeated[1/(1+x), x→ 1/(1+x), MaxIterations− >4]
```

$\frac{1}{1+\frac{1}{1+\frac{1}{1+\frac{1}{1+x}}}}$

In Section 9 we will use the rules effectively with the pattern matching facility of *Mathematica* (see, for example, Problem 9.2).

9. Pattern matching

Mathematica provides us with the ability to decide whether an expression matches a specific pattern. Following R. Gaylord's exposition [1], consider the expression x^2. This expression is precisely of the following form or *pattern*, "x raised to the power of two":

```
MatchQ[x^2, x^2]

True
```

But x^2 will be matched by the following loosely description, "something" or "an expression"

```
MatchQ[x^2, _]

True
```

Here _ stands (or rather sits) for an expression (_ is called *a blank* here). x^2 will also match "x to the power of something"

```
MatchQ[x^2, x^_]

True
```

Before we go further, we need to mention that one can give a name to a blank expression as follows `n_`. Here the expression _ is labelled `n`. (We have already seen `n_` in defining a function. In fact when defining a function, we label an expression that we plug into the function). Also one can restrict the expression by limiting its head! Namely, `_head` matches an expression with the head `head`. Look:

```
FullForm[x^2]

Power[x, 2]

Head[x^2]

Power

MatchQ[x^2, _Power]

True

Head[4]

Integer

MatchQ[4,_Integer]

True

Head[4/3]
```

```
Rational
```

```
MatchQ[4/3,_Integer]
```

```
False
```

Putting these together `n_Plus` means an expression which is labelled `n` and has the head `Plus`. Continuing with our example, x^2 matches "x to the power of an integer number"

```
MatchQ[x^2, x^_Integer]
```

```
True
```

```
MatchQ[x^2, x^_Real]
```

```
False
```

The same way x^2 matches "something or an expression to the power of 2"

```
MatchQ[x^2, _^2]
```

```
True
```

```
MatchQ[x^2, _^5]
```

```
False
```

Finally, x^2 matches "something to the power of something"

```
MatchQ[x^2,_^_]
```

```
True
```

One can define a condition on a pattern, namely to test whether an expression satisfies a certain condition. Here is an example:

```
MatchQ[5, _Integer?(# > 3 &)]
```

```
True
```

```
MatchQ[2, _Integer?(# > 3 &)]
```

```
False
```

The pattern `_Integer?(# > 3 &)` stands for an expression which has `Integer` as its head, that is an integer number, which is bigger than three.

Here are more examples:

```
MatchQ[x^2, _^_?OddQ]
```

```
False
```

```
MatchQ[x^2, _^_?EvenQ]
```

```
True
```

Here is how all these concepts help. One can single out an expression of specific pattern and once this is done then change the expression. It is all about accessing and then manipulating!

Here are some examples:

```
MatchQ[{a,b},{_,_}]

True

MatchQ[{a,b},{x_,y_}]

True
```

Study the following examples carefully!

```
{{a,b},{c,d}}/.{x_,y_}→ x y
{a c,b d}
```

`{{a,b},{c,d}}/.{x_,y_}→` x^y

$\{a^c, b^d\}$

```
{{a,b},{c,d}}/.{x_,y_}→ y
{c , d}
```

Here is the third approach to Problem 3.7 and 4.1.

Problem 9.1. *Write a function* `squareFreeQ[n]` *that returns* `True` *if the number n is a square free number, and* `False` *otherwise.*

Solution.

```
t=FactorInteger[234090]

{{2,1},{3,4},{5,1},{17,2}}
```

We are after those numbers that when decomposed into powers of prime, say, $\{\{p_1, k_1\}, \{p_2, k_2\}, \cdots, \{p_t, k_t\}$ then all k_i are 1.

The pattern `{_,y_?(#>1&)}` describes those lists with the second element (a number) bigger than 1.

```
MatchQ[{3,4},{_,y_?(#>1&)}]

True
```

Here is the time to introduce `Cases`.

```
? Cases
```

```
Cases[{e1, e2, ... }, pattern] gives a list of the ei that match
the pattern.
```

```
Cases[{6,test,20,5.3,35,5/3},_Integer]

{6,20,35}
```

We use `Cases` to get all the pairs with k_i's bigger than 1. If this list is not empty then the number is not square free.

```
Cases[{{2,1},{3,4},{5,1},{17,2}},{_,y_?(#>1&)}]

{{3,4},{17,2}}

Cases[{{2,1},{3,4},{5,1},{17,2}},{_,y_?(#>1&)}] != {}

False
```

We are ready to put all these together and write a function for finding square free numbers.

```
squareFree3[n_]:=Cases[FactorInteger[n],{_,y_?(#>1&)}] != {}

squareFree3[234090]

False

squareFree3[3 * 5*13*17]

True
```

So far we have been dealing with one expression. What if instead of one expression we would be dealing with a bunch of them?

```
MatchQ[{x^2},{_}]

True

MatchQ[{x^2, x^3, x^5}, {_}]

False

MatchQ[{x^2, x^3, x^5}, {__}]

True
```

As one can see from the above example, _ _ stands for a sequence of data as _ is for just one expression. In fact _ _ is for a sequence of nonempty expressions, and _ _ _ is for a sequence of empty or more data. The following examples show this clearly.

```
MatchQ[{},{_}]

False

MatchQ[{},{__}]

False

MatchQ[{},{___}]

True
```

Here is one more example to show the difference between _ _ and _ _ _:

```
MatchQ[{3,5,2,2,stuff,7},{__,3,___}]

False

MatchQ[{3,5,2,2,stuff,7},{___,3,___}]

True

MatchQ[{3,5,2,2,7,us},{___,2,2,___}]

True
```

We are ready to write a little game.

Problem 9.2. *Write a game as follows. A player gets randomly* 7 *cards between* 1 *and* 10*. He would be able to drop any two cards between* 4 *and* 10 *that are similar. Then the sum of the cards that remain in the hand is what a player scores. A player with minimum score wins.*

Solution. First, we generate a list containing 7 random numbers between 1 and 10.

```
s=Table[Random[Integer,{1,9}],{7}]

{2,5,2,3,4,7,4}
```

Now we shall write a code to discard any two numbers which are the same. The trick we use here is, we look inside the list and recognise the same numbers (which have the same pattern), mark them and with a rule send the list to a new list containing all the elements except the similar ones. Here is the code:

```
s/.{m___,x_,y___,x_,n___}->{m,y,n}
{5,3,4,7,4}
```

Here *Mathematica* looks for similar expressions `x_` and `x_`. To the left of `x_` is `m___` which means any sequence of empty or more expressions. Similarly in between `x`'s we place `y___`. That is, in between similar numbers could be empty (i.e., the similar numbers are next to each other) or a bunch of other expressions. Finally to the right hand side of the second `x_` is `n___`. In the example, our original list is `{2,5,2,3,4,7,4}`. *Mathematica* recognises that there are two 2 in the list, so will assign them to `x_`. To the left hand side of the first 2 there is no data, thus `m___` would get an empty value, `y___` would be 5 and `n___` would be the whole sequence of $3, 4, 7, 4$ in the right hand side of the second 2. Thus the rule`{m___,x_,y___,x_,n___}->{m,y,n}` will discard the `x`'s and give us `{5,3,4,7,4}` .

Still there are two 4's in the list but as we have seen in Section 8, `/.->` would go through the list only once. Thus if we run the same code again, this time with our new list, we shall get rid of double 4's.

```
{5,3,4,7,4}/.{m___,x_,y___,x_,n___}->{m,y,n}
{5,3,7}
```

Remember that `//.->` was designed exactly for this job.

```
s//.{m___,x_,y___,x_,n___}->{m,y,n}
{5,3,7}
```

But in the game we are allowed to drop the cards between 4 and 10. Thus we shall put in a little test to find similar numbers bigger than 3. Here is the enhanced code:

```
s//.{m___,x_?(3<#<10&),y___,x_,n___}->{m,y,n}
{2,5,2,3,7}
```

Notice that it is enough to put a test for one of the `x_`'s.

Just to make sure we understood this, let's try the code for

```
s={7, 6, 2, 7, 1, 2, 1, 6, 7}
s//.{m___,x_?(3<#<10&),y___,x_,n___}->{m,y,n}
{2, 1, 2, 1, 7}
```

The rest is to sum the numbers in the list. For example

```
Plus @@ %
13
```

10. Functions with multiple definitions

In this section we will talk about the ability of *Mathematica* to handle a function with multiple definitions. Plus we will see how a function can contain more than one line, namely contain a block of codes (a sort of mini program or a procedure).

Recall the very first function that we defined in Section 2.1.

```
f[n_]:= n^3 +11
f[-2]
3
```

In the light of the previous section, one can see what this code exactly means. One can send any expression with any pattern into `f[n_]`. The expression is labelled `n`. Now we can easily restrict the sort of data we want to send into a function, by simply describing the sort of pattern we desire. For example if in the above function, we would like the function only to perform on positive integers, then

```
f[n_Integer?Positive] := n^3 + 11

f[4]

75

f[-2]

f[-2]
```

Here are some more examples:

```
g[n_Integer?(0 < # < 5 &)] := Sqrt[5 - n]
g[2]
```
$\sqrt{3}$
```
g[6]
g[6]

e[p__?(PolynomialQ[#, x] &)] := Expand[p,x]
e /@ {4, (1 + x)^2, Sin[x] + Cos[x]}
{4, 1 + 2 x + x^2, e[Cos[x] + Sin[x]]})
```

One can even be carried away with this ability. Here is a function that gives the prime factors of a number which consist of only odd digits (e.g. 3715).

```
myfunc[n_Integer?(Select[IntegerDigits[#], EvenQ, 1] == {}&)]:=
Map[First, FactorInteger[n]]

myfunc[3715]

{5, 743}
```

```
myfunc[593183]

myfunc[593183]
```

One of the great features of *Mathematica* is that one can define a function with multiple definitions. Here is a harmless example

```
oddeven[(n_?EvenQ)?Positive] := Print[n, " even and positive"]
oddeven[(n_?EvenQ)?Negative] := Print[n, " even and negative"]
oddeven[(n_?OddQ)?Positive] := Print[n, " odd and positive"]
oddeven[(n_?OddQ)?Negative] := Print[n, " odd and negative"]

Map[oddeven, {-2, 5, -3, -4}];

-2  even and negative

5 odd and positive

-3  odd and negative

4  even and positive
```

Here we have the function `oddeven` with four definitions. An integer falls into one of the cases above, and *Mathematica* has no problem going through all the definitions of the function and applying the appropriate one to the given number. If one asks for the definition of `oddeven`, one can see *Mathematica* has all four definitions in memory, in the same order that one has defined the function.

```
?oddeven

Global`oddeven

oddeven[(n_?EvenQ)?Positive] := Print[n,  even and positive]

oddeven[(n_?EvenQ)?Negative] := Print[n,  even and negative]

oddeven[(n_?OddQ)?Positive] := Print[n,  odd and positive]

oddeven[(n_?OddQ)?Negative] := Print[n,  odd and negative]
```

Problem 10.1. *Define the Collatz function as follows:*

$$f(x) = \begin{cases} x/2 & \text{if } x \text{ is even} \\ 3x+1 & \text{if } x \text{ is odd.} \end{cases}$$

It is a conjecture that if one applies f repeatedly to any number, one arrives at 1. Find out how many times one needs to apply f to numbers 1 to 200 to get 1.

Solution.
Here is the Collatz function:

```
f[x_Integer?EvenQ] := x/2

f[x_Integer] := 3x + 1
```

One can write the following one-liner for the rest of the code.

```
Length /@ ( NestWhileList[f, #, ! # == 1 &] & /@ Range[200])

{1, 2, 8, 3, 6, 9, 17, 4, 20, 7, 15, 10, 10, 18, 18, 5, 13, 21,
21, 8, 8, 16,  16, 11, 24, 11, 112, 19, 19, 19, 107, 6, 27, 14,
14, 22, 22, 22, 35, 9, 110,  9, 30, 17, 17, 17, 105, 12, 25, 25,
25, 12, 12, 113, 113, 20, 33, 20, 33, 20,  20, 108, 108, 7, 28,
28, 28, 15, 15, 15, 103, 23, 116, 23, 15, 23, 23, 36,  36, 10, 23,
111, 111, 10, 10, 31, 31, 18, 31, 18, 93, 18, 18, 106, 106, 13,
119, 26, 26, 26, 26, 26, 88, 13, 39, 13, 101, 114, 114, 114, 70,
21, 13, 34,  34, 21, 21, 34, 34, 21, 96, 21, 47, 109, 109, 109,
47, 8, 122, 29, 29, 29,  29, 29, 42, 16, 91, 16, 42, 16, 16, 104,
104, 24, 117, 117, 117, 24, 24, 16,  16, 24, 37, 24, 86, 37, 37,
37, 55, 11, 99, 24, 24, 112, 112, 112, 68, 11, 50, 11, 125, 32,
32, 32, 81, 19, 32, 32, 32, 19, 19, 94, 94, 19, 45, 19, 45,  107,
107, 107, 45, 14, 120, 120, 120, 27, 27, 27, 120, 27}

Max[%]

125
```

Problem 10.2. *Define the function*

$$f(x) = \begin{cases} \sqrt{x} & \text{if } x \geq 0 \\ \sqrt{-x} & \text{if } x < 0 \end{cases}$$

and plot the graph of the function for $-1 \leq x \leq 1$.

Solution. One can define $f(x)$ in *Mathematica*as a function with two definitions as follows:

```
f[x_?Positive] := Sqrt[x]

f[x_?Negative] := Sqrt[-x]

Plot[f[x],{x,-1,1}]
```

As you might have noticed, so far, there has been no confusion regarding the multiple definitions of a function. Namely, the data that is sent to the function satisfied only one of the patterns in the definition of the function. In `oddeven`, a number could be only one of the cases of positive/negative and odd/even and in the previous example a number is either positive or negative. But imagine we define a function as follows:

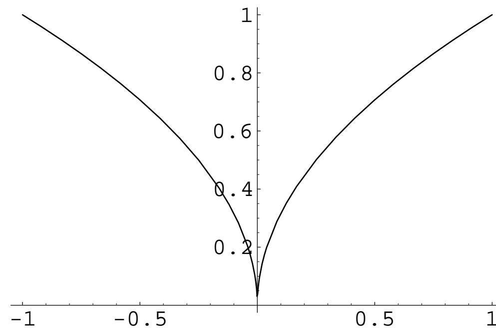


FIGURE 1. A function with multiple definitions

```
f[x_] := x
f[x_Integer] := x!
```

Now one might ask what would be `f[4]`. There are two definitions for `f` and 4 can match both patterns, namely `x_ or x_Integer`.

```
f[4]

24

f[5]

120

f[2.3]

2.3

f[test]

test
```

Thus for any integer the definition which is the factorial of a number is performed and for other data the other definition (obviously). If we find out in what order *Mathematica* saves the definitions of functions, we can justify this action.

```
?f

Global‘f

f[x_Integer] := x
f[x_] := x
```

Thus in principle, *Mathematica* stores the definitions from the one with more precise pattern matching (here the one with `x_Integer`). If she cannot define which definition has the more precise

pattern matching, then she stores the definition in the order in which it has been entered in the system. Here is an example of this type:

```
cic[n_?(# > 1 &)] :=
  Show[Graphics[{{RGBColor[1, 0, 0], Disk[{0, 0}, 1]}, {RGBColor[0, 0, 0],
          Disk[{1, 0}, 1]}}], AspectRatio -> Automatic]

cic[n_?(# > 0 &)] :=
  Show[Graphics[{{RGBColor[0, 0, 0], Disk[{0, 0}, 1]}, {RGBColor[1, 0, 0],
          Disk[{1, 0}, 1]}}], AspectRatio -> Automatic]
```

```
cic[5]
```

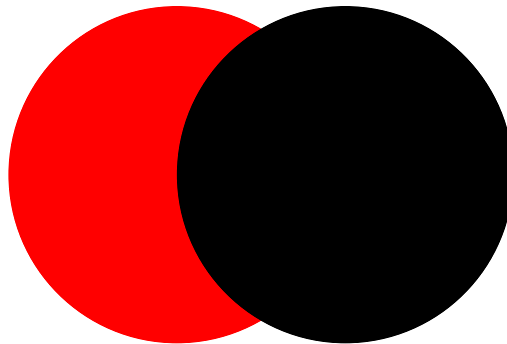


FIGURE 2. `cic[5]`

```
cic[1]
```

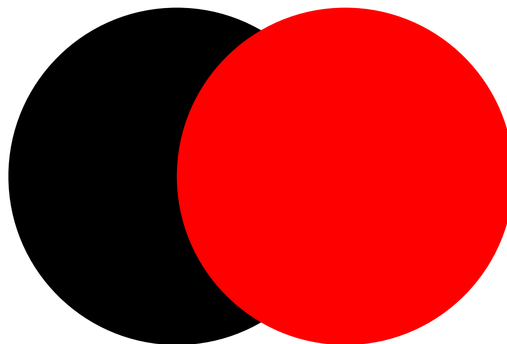


FIGURE 3. `cic[1]`

In this example, *Mathematica* stores the definition of the functions in the same order that we entered it, as there is no preference in the patterns that have been defined.

Problem 10.3. *Define a function f(x) in Mathematica which satisfies*

$$f(xy) = f(x) + f(y)$$
$$f(x^n) = nf(x)$$
$$f(n) = 0$$

where n is an integer and show that

$$f(\prod_{i=1}^{20} i(x_i)^i) = \sum_{i=1}^{20} if(x_i)$$

```
f[x_ y_] := f[x] + f[y]
f[x_^n_Integer] := n f[x]
f[n_Integer] = 0

f[Product[i!(x_i)^i, {i, 1, 20}]]
f[x_1] + 2 f[x_2] + 3 f[x_3] + 4 f[x_4] + 5 f[x_5] + 6 f[x_6] + 7 f[x_7] + 8 f[x_8]
+ 9 f[x_9] + 10 f[x_10] + 11 f[x_11] + 12 f[x_12] + 13 f[x_13] + 14 f[x_14] + 15 f[x_15]
+ 16 f[x_16] + 17 f[x_17] + 18 f[x_18] + 19 f[x_19] + 20 f[x_20]

∑_i^20 i f[x_i]== f[Product[i!(x_i)^i, {i, 1, 20}]]
True
```

✠

10.1. **Functions with local variables.** One of the approaches of procedural languages to programming is to break the program into "mini-programs" or *procedures* and then put them together to get the code we need. These procedures have their own variables called *local* variables, that is, variables which have been defined only inside the procedure. So far all the functions that we have defined consist of only one line. *Mathematica*'s functions can be also used as procedures, namely can contain several lines of code and their own local variables. Let us look at a simple example. Recall Problem 7.5, which finds all the prime numbers less than n. Let us write this as a function `lPrimes[n]` to produce a list of all such primes.

```
lPrimes[n_] := Module[{pset = {}, i = 1},
While[Prime[i] <= n,
pset = pset ∪ {Prime[i]};
i++];
pset]

lPrimes[8]
{2, 3, 5, 7}
```

A function with several lines of codes in *Mathematica* are wrapped by `Module`. The structure looks like `Module[{local variables},body]`. In the above example the variables `pset` and `i` are variables defined only inside the function `lPrimes`. Here to check that these are undefined outside the function:

```
  pset
  pset
  i
  i
```

10.2. **Functions with conditions.** Consider the following code

```
f[n_] := Sqrt[n] /; n > 0

f[4]

2

f[-4]

f[-4]
```

Here `/;` is a shorthand for an `If`. We have seen we can restrict the pattern of the data we pass into a function. The equivalent ways to define the above function are

```
f1[n_?Positive]:=Sqrt[n]

f2[n_]:=If[n>0,Sqrt[n]]
```

Sometimes using `/;` helps to make the code much more readable than using other ways to put conditions.

Here is another version of the Game 9.2.

Problem 10.4. *Write a game as follows. A player gets randomly* 7 *cards between* 1 *and* 10*. He would be able to drop any two cards with the sum* 5*. Then the sum of the cards that remain in the hand is what a player scores. A player with minimum score wins.*

Solution. Let us first design a function that accepts a sequence of numbers and deletes any two numbers of which the sum is 5. Having an eye on the code of Problem 9.2:

```
aHandD[n___, y_, t___, z_, m___] :=
  aHandD[n, t, m] /; y + z == 5

aHandD[2,3,5,4,1,3,7]

aHandD[5,3,7]
```

Then, we can simply change the head of this expression to `Plus` to get the sum of the cards.

```
Apply[Plus,%]

15
```

Now we produce a list of 7 random numbers and write a little function, call it `aHand`, to put all these lines together:

```
Table[Random[Integer,{1,9}],{7}]
```

```
{5,2,7,3,1,6,3}

Apply[Sequence,%]

Sequence[5,2,7,3,1,6,3]

aHandD[%]

aHandD[5,7,1,6,3]

aHand=Module[{},
aHandD[Apply[Sequence, Table[Random[Integer, {1,
9}], {7}]]];
  Print[Apply[Plus, %]]
  ]
```

We shall see a similar approach to a problem involving matrices in Problem 12.4.

11. Recursive functions

Imagine two mirrors setting parallel to each other with an apple sitting in between. Then one can see infinite number of apples in the mirrors. This might give an impression of what a recursive function is. The classic example is Fibonacci numbers. Consider the sequence of numbers starting with 1 and 1 and continue with sum of the two previous numbers as the next number in the sequence. Following this rule, one obtains the sequence $1, 1, 2, 3, 5, 8, 13, 21, \cdots$. To define this sequence mathematically, one writes $F_1 = F_2 = 1$ and $F_n = F_{n-1} + F_{n_2}$.

One can use *Mathematica* to define Fibonacci numbers in the exact same way recursively:

```
f[1] = 1; f[2] = 1;
f[n_] := f[n-1] + f[n-2]

f /@ Range[10]
{1, 1, 2, 3, 5, 8, 13, 21, 34, 55}
```

Now try to compute the 50th Fibonacci number. This will take ridiculously a long time. What is the problem? The problem will show itself if you try to calculate, say, $f[5]$ by hand. By definition, $f[5] = f[4] + f[3]$ thus one needs to calculate $f[4]$ and $f[3]$. Again by definition $f[4] = f[3] + f[2]$ and $f[3] = f[2]+f[1]$. Thus in order to find the value if $f[4]$ one needs to find out $f[3]$ and $f[2]$ and for $f[3]$ one needs to calculate $f[2]$ and $f[1]$. Thus *Mathematica* is trying to calculate $f[3]$ twice unnecessarily. This shows that in order to save time, one needs to save the values of the functions in the memory. This has been done in the following codes. Compare this with the above.

```
Clear[f]

f[1] = 1; f[2] = 1;
f[n_] := f[n] = f[n - 1] + f[n - 2]
```

```
f[50]
12586269025
```

Here we use recursive programming to solve Problem 7.9.

Problem 11.1. *A Happy number is a number that if one squares its digits and add them together, and then take the result and square its digits and add them together again and keep doing this process, one comes down to the number 1. Find all the Happy ages, i.e., happy numbers up to 100.*

Solution.

```
f[1] = 1
f[4] = 4

f[n_] := f[Plus @@ ( IntegerDigits[n]^2)]

Select[Range[100], f[#] == 1 &]

{1, 7, 10, 13, 19, 23, 28, 31, 32, 44, 49, 68, 70, 79, 82, 86, 91, 94, 97, 100}
```

One can re-write many codes which have a repetition nature in the form of recursive. Recall Collatz function from Problem 10.1. One can write the function as follows

```
f[1]=1
f[n_Integer?EvenQ] := f[n/2]
f[n_Integer] := f[3n + 1]
```

Then if one applies f to any number one should get 1 (If not, then one has solved the Collatz conjecture in negative!).

12. Matrices; Multilinear algebra

It is known that matrix calculation is a tedious job. It will take well over 10 minutes to multiply

$$\begin{pmatrix} 2 & -3 & 13 & -4 & 8 & -10 \\ 12 & 1 & -18 & -4 & 2 & 5 \\ 18 & 21 & 10 & 0 & 9 & 7 \\ 8 & -12 & -4 & 0 & -3 & -11 \\ 15 & -7 & 2 & 4 & 2 & 12 \\ -2 & -5 & 12 & 3 & -9 & -4 \end{pmatrix} \times \begin{pmatrix} 11 & 34 & -21 & 0 & -43 & 12 \\ 12 & -33 & 9 & -12 & 7 & 2 \\ 16 & -7 & -43 & 84 & 3 & -6 \\ 4 & 9 & 12 & -1 & -54 & -2 \\ 7 & 22 & -5 & 23 & 0 & 10 \\ -2 & -10 & 33 & 2 & 11 & 12 \end{pmatrix}$$

only to obtain a wrong answer!.

One can easily enter a matrix into *Mathematica* by using Input:Create Table/Matrix. If we assign `A` to the first matrix and `B` to the second then

```
A.B //MatrixForm
```

$$\begin{pmatrix} 254 & 316 & -1046 & 1296 & 38 & -92 \\ -156 & 459 & 638 & -1464 & -292 & 342 \\ 659 & -23 & -433 & 809 & -520 & 372 \\ 277 & -349 & -155 & -679 & -330 & 0 \\ 119 & 687 & -30 & 318 & -772 & 310 \\ 67 & -118 & -570 & 850 & -119 & -250 \end{pmatrix}$$

```
Det[A]
12327530
```

One uses the function `MatrixForm` to obtain the result in matrix form!. Otherwise one gets a list of vectors.

Even more impressive is how easily *Mathematica* computes the inverse of this matrix. Try

```
Inverse[A]//MatrixForm
```

One can generate a matrix by using `Array`.

Problem 12.1. *Write a function to check that for any n, the following identity hold.*

$$\begin{vmatrix} 1 & 1 & 1 & \cdots & 1 & 1 \\ b_1 & a_1 & a_1 & \cdots & a_1 & a_1 \\ b_1 & b_2 & a_2 & \cdots & a_2 & a_2 \\ \vdots & \vdots & \vdots & & \vdots & \vdots \\ b_1 & b_2 & b_3 & \cdots & b_n & a_n \end{vmatrix} = (a_1 - b_1)(a_2 - b_2) \cdots (a_n - b_n)$$

Solution. Here the defintion of the matrix using `Array`.

```
m[n_]:=Array[
Which[#1==1,1,
#1 ≤ #2,a_{#1-1},
True,b_{#2} ]&,
{n+1,n+1}]
```

Check that this in fact produces matrices of the above form.

```
m2[n_] := Product[a_i - b_i, {i, 1, n}]
```

```
Simplify[Det[m[7]] == m2[7]]

True
```

Problem 12.2. *Write a function to check that for any n, the following identity hold.*

$$\begin{vmatrix} x & a_1 & a_2 & \cdots & a_n \\ a_1 & x & a_2 & \cdots & a_n \\ a_1 & a_2 & x & \cdots & a_n \\ \vdots & \vdots & \vdots & & \vdots \\ a_1 & a_2 & a_3 & \cdots & x \end{vmatrix} = (x + a_1 + a_2 + \cdots + a_n)(x - a_1)(x - a_2) \cdots (x - a_n)$$

Solution. The solution is left to the reader this time!

Problem 12.3. *Write a function to accept a matrix A_{nn} and produce*

$$B_{n^2n^2} = \begin{pmatrix} \begin{pmatrix} a_{11} & 0 & 0 \\ 0 & \ddots & 0 \\ 0 & 0 & a_{11} \end{pmatrix} & \begin{pmatrix} a_{12} & 0 & 0 \\ 0 & \ddots & 0 \\ 0 & 0 & a_{12} \end{pmatrix} & \cdots & \begin{pmatrix} a_{1n} & 0 & 0 \\ 0 & \ddots & 0 \\ 0 & 0 & a_{1n} \end{pmatrix} \\ \cdots & \cdots & \cdots & \cdots \\ \vdots & \vdots & \vdots & \vdots \\ \begin{pmatrix} a_{n1} & 0 & 0 \\ 0 & \ddots & 0 \\ 0 & 0 & a_{n1} \end{pmatrix} & \cdots & \cdots & \begin{pmatrix} a_{nn} & 0 & 0 \\ 0 & \ddots & 0 \\ 0 & 0 & a_{nn} \end{pmatrix} \end{pmatrix}.$$

Then show that $\det(A)^n = \det(B)$.

The following is a nice problem demonstrating the use of pattern matching in *Mathematica* for solving problems.

Problem 12.4. *Let A and B be 3×3 matrices. Show that* $(ABA^{-1})^5 = AB^5A^{-1}$.

Solution. Let us first take the naive approach. We define two arbitrary matrices and, using *Mathematica*, we will multiply them and check whether both sides give the same result.

```
(A=Array[x_{#1,#2}&, {3,3}])//MatrixForm
IA=Inverse[A];
```

$$\begin{pmatrix} x_{1,1} & x_{1,2} & x_{1,3} \\ x_{2,1} & x_{2,2} & x_{2,3} \\ x_{3,1} & x_{3,2} & x_{3,3} \end{pmatrix}$$

```
(B=Array[y_{#1,#2}&, {3,3}])//MatrixForm
IB=Inverse[B];
```

$$\begin{pmatrix} y_{1,1} & y_{1,2} & y_{1,3} \\ y_{2,1} & y_{2,2} & y_{2,3} \\ y_{3,1} & y_{3,2} & y_{3,3} \end{pmatrix}$$

Now we check the equality for n=3 and take the time:

```
Timing[d=A.B.IA.A.B.IA.A.B.IA;
d1=A.B.B.B.IA;
Simplify[d==d1]]

{70.963 Second, True}
```

This already takes long. One can easily prove by induction that $(ABA^{-1})^n = AB^nA^{-1}$ for any positive integer n. For example for $n = 2$ we have $(ABA^{-1})^2 = ABA^{-1}ABA^{-1} = ABBA^{-1} = AB^2A^{-1}$. This shows a pattern here. Namely we can easily cancel A with A^{-1} if they are adjacent to each other. We introduce this to *Mathematica* and try to check the equality this way. This is very similar to Problem 10.4 in nature.

```
matmul[x___,y_,z_,t___]:=
matmal[x,t]/;Simplify[y.z==IdentityMatrix[3]]

matmal[r___]:=Apply[Dot,{r}]

matmal[]=1;

Timing[Simplify[
matmul[A,B,IA,A,B,IA,A,B,IA,A,B,IA,A,B,IA]==A.B.B.B.B.B.IA]]

{0.01 Second, True]
```

Assorted Exercises

Write the following functions:

$$\frac{2}{\pi} = \frac{\sqrt{2}}{2}\frac{\sqrt{2+\sqrt{2}}}{2}\frac{\sqrt{2+\sqrt{2+\sqrt{2}}}}{2}\cdots$$

$$\frac{2}{\pi} = \frac{1\times 3}{2\times 2}\frac{3\times 5}{4\times 4}\frac{5\times 7}{6\times 6}\cdots\frac{(2n-1)(2n+1)}{2n\times 2n}$$

$$\frac{\exp}{2} = (\frac{2}{1})^{\frac{1}{2}}(\frac{2}{3}\frac{4}{3})^{\frac{1}{4}}(\frac{4}{5}\frac{6}{5}\frac{6}{7}\frac{8}{7})^{\frac{1}{8}}(\frac{8}{9}\frac{10}{9}\frac{10}{11}\frac{12}{11}\frac{12}{13}\frac{14}{13}\frac{14}{15}\frac{16}{15})^{\frac{1}{16}}\cdots$$

References

[1] R. Gaylord, *Mathematica* Programming Fundamentals, Lecture Notes, Available in MathSource
[2] W. Shaw, J. Tigg, Applied *Mathematica*, Addison-Wesley Publishing, 1994
[3] I. Vardi, Computational Recreations in *Mathematica*, Addison-Wesley Publishing, 1991
[4] S. Wagon, *Mathematica* in Action, Springer-Verlag, 1999

R. Hazrat, Department of Pure Mathematics, Queen's University, Belfast BT7 1NN, U.K.
E-mail address: r.hazrat@qub.ac.uk