

# 9. Concluding Techniques

---



## 9.1 Introduction

What are these concluding techniques?

Objectives

## 9.2 Timing

Timing a computation

Exercises

## 9.3 Printing

Printing an intermediate result

Exercises

## 9.4 Advanced Patterns

Patterns of sequences of elements

Exercises

## 9.5 Timing

Recursive functions

Recursion Limits

## 9.6 Dynamic Programming

The concept of a dynamic program

Looking at the rule base for a recursively defined function

## 9.5 Procedural Programming

Traditional programming constructs

**Do**

**For**

**While**

## 9.5 Problems

Problem 1: Exploring timing

Problem 2: Exploring advanced patterns

Problem 3: Exploring dynamic programming

## 9.1 Introduction

---

### What are these concluding techniques?

---

The concluding techniques that you will learn in this module encompass three types

- Techniques to make your programs faster
- Techniques to help you debug your programs
- Techniques that are still in use from older type languages

### Objectives

---

- To be able to determine the time taken for a computation
- To be able to print diagnostics
- To be able to use advanced patterns
- To understand the concepts of recursion and dynamic programming
- To be aware of the procedural programming style

## 9.2 Timing

---

### Timing a computation

---

You can find out how long a computation takes by using **Timing**. For example, suppose you want to find out how long it takes to compute  $\pi$  to 100 000 decimal places, you would enter

```
Timing[N[ $\pi$ , 100000];]
{13.0333 Second, Null}
```

**Timing** always returns a *pair* of outputs, the first of the pair being the time in seconds, and the second the output of the calculation being timed.

Sometimes it is convenient to assign this pair to a pair of symbols for later use.

```
{t, p} = Timing[Plot[Sin[x], {x, 0, 2  $\pi$ },
  DisplayFunction -> Identity, PlotPoints -> 100]]
{0.0166667 Second, - Graphics -}
```

Now **p** is the plot expression alone (without the timing information).

```
p
- Graphics -
```

Mathematica will sometimes remember parts of the calculation you ask it to do, so that the next time you ask it to do the same calculation in that session, it may be faster.

Quite often you will get a time of **0. Second**. This means that the calculation went faster than the minimum time registrable by the timing function.

## Exercises

---

### ◆ Exercise: Timing a computation

Determine the time it takes to calculate  $e$  to 10000 decimal places. *Hint:* You may want to suppress the output.

### ◆ Exercise: Timing a computation

Determine the time it takes to calculate **1000!**.

### ◆ Exercise: Timing a computation

Determine the time it takes to find the **Eigenvalues** of the matrix  $\{\{a,b\},\{c,d\}\}$ .

◆

## 9.3 Print

---

### Printing an intermediate result

---

Sometimes you have a long calculation and you would like to get an idea of Mathematica's progress. You can do this by inserting **Print** statements into your code at various stages.

For example, suppose you were calculating a table of factorials, and you wanted to keep track of the intermediate results as they were being generated, that is, before the final complete output is assembled and returned. Without a **Print** statement you would enter:

```
Table[n!, {n, 10, 15}]  
{3628800, 39916800, 479001600,  
 6227020800, 87178291200, 1307674368000}
```

Now include a **Print** statement as the *first* component of a compound expression (that is, one whose components are separated by a semi-colon ;)

```

Table[(Print[n!]; n!), {n, 10, 15}]

3628800
39916800
479001600
6227020800
87178291200
1307674368000
{3628800, 39916800, 479001600,
6227020800, 87178291200, 1307674368000}

```

Here you get the intermediate results *printed* as they are calculated. The *output* is given as the last line.

The result of a **Print** statement is a *side-effect*, just like the display of a graphic. It is *not* an output, and you have no access to it other than by copying and pasting. You should *not* use print statements for anything other than temporary information. All outputs of interest should be generated as outputs, not as side-effects.

**The result of a **Print** statement is a *side-effect*.  
Use them *only* for the *temporary display* of information.**

Remember, it is only the *last* statement in a compound expression that is given as output. So that if you had reversed the order above to **(n!;Print[n!])** you would have suppressed the actual output you had wanted.

```

Table[(n!; Print[n!]), {n, 10, 15}]

3628800
39916800
479001600
6227020800
87178291200
1307674368000
{Null, Null, Null, Null, Null, Null}

```

## Exercises

---

### ◆ Exercise: Using Print

Generate a table of **Prime**[n] from n equals 100000 to 100005 together with a **Print** of the intermediate results.

◆ **Exercise: Using Print with Timing**

Generate a table of **Prime[n]** from **n** equals **100000** to **100005** together with a **Print** of the intermediate results *and* the time it took for each intermediate result to be calculated.

◆

## 9.4 Advanced Patterns

---

### Patterns of sequences of elements

---

You are, by now, used to using patterns in rules. For example we could use a rule to put all the integers in a list of data to zero.

```
{x, 3.2, y, 1, 2, 3, z, 4, -7.9} /. x_Integer -> 0
{x, 3.2, y, 0, 0, z, 0, -7.9}
```

We can also use conditions in the rule. For example, to put only those integers less than or equal to 2 to zero.

```
{x, 3.2, y, 1, 2, 3, z, 4, -7.9} /. x_Integer /; (x <= 2) -> 0
{x, 3.2, y, 0, 0, 3, z, 4, -7.9}
```

We can also use patterns to represent whole *sequences* of elements. There are three different types

<b>x_</b>	Represents just <i>one</i> element. (One underscore)
<b>x__</b>	Represents a sequence of one <i>or more</i> elements. (Two underscores)
<b>x___</b>	Represents a sequence of <i>zero</i> , one or more elements. (Three underscores)

With these we can then make up rules which operate *on the whole data list at once* to transform it into what we want.

◆ **Example**

For example, suppose we wanted to create a new list identical to the original list except that any integers in it *are replaced by their sum and put at the end of the list*. We first need to write a rule with a pattern which can represent *any* list of the form we are manipulating, but identify specifically just two arbitrary elements to be added.

For brevity we name our rule **R**.

```
R = {z1___, x_Integer, z2___, y_Integer, z3___} ->
    {z1, z2, z3, x + y};
```

Here, the **z1**\_\_\_\_, **z2**\_\_\_\_, and **z3**\_\_\_\_ simply say that you can have *any* number (perhaps zero) of elements surrounding or between the two integers to be added.

Applying the rule **R** adds the first two integers found as the list is scanned from left to right (**1, 2**) and puts **3** at the end of the list.

$$\{\mathbf{x}, 3.2, \mathbf{y}, 1, 2, 3, \mathbf{z}, 4, -7.9\} /. \mathbf{R}$$

$$\{\mathbf{x}, 3.2, \mathbf{y}, 3, \mathbf{z}, 4, -7.9, 3\}$$

The rule has only been applied once because `/.`  only applies a rule once. But here, the resulting list still has integers that could be added. We could repeat the replacement by applying the rule **R** again to the previous result. This adds the **3** and the **4**, and puts their sum **7** at the end of the list.

$$\{\mathbf{x}, 3.2, \mathbf{y}, 3, \mathbf{z}, 4, -7.9, 3\} /. \mathbf{R}$$

$$\{\mathbf{x}, 3.2, \mathbf{y}, \mathbf{z}, -7.9, 3, 7\}$$

But still we have the numbers 3 and 7 at the end of the list to add. Applying the rule again gives the final result.

$$\{\mathbf{x}, 3.2, \mathbf{y}, \mathbf{z}, -7.9, 3, 7\} /. \mathbf{R}$$

$$\{\mathbf{x}, 3.2, \mathbf{y}, \mathbf{z}, -7.9, 10\}$$

There is a function called **ReplaceRepeated** (`//.`) which continues applying a rule until the expression does not change any more. This will do the whole manipulation in one operation.

$$\{\mathbf{x}, 3.2, \mathbf{y}, 1, 2, 3, \mathbf{z}, 4, -7.9\} //. \mathbf{R}$$

$$\{\mathbf{x}, 3.2, \mathbf{y}, \mathbf{z}, -7.9, 10\}$$

## Exercises

---

### ◆ Exercise: Exploring advanced patterns

Take the list  $\{\mathbf{x}, 3.2, \mathbf{y}, 1, 2.679, 3, \mathbf{z}, 4, -7.9\}$  and write a rule to multiply all its real numbers, and place it at the beginning of the list.

### ◆ Exercise: Exploring advanced patterns

Take the list  $\{\mathbf{x}, 3.2, \mathbf{y}, 1, 2.679, 3, \mathbf{z}, 4, -7.9\}$  and write a rule to add any *adjacent* numbers.

### ◆ Exercise: Exploring advanced patterns

Write a rule to find the minimum number in a list of numbers, *without* using the **Min** function.

◆

## 9.5 Recursion

---

### Recursive functions

---

A *recursive function* is one whose value depends on other *values* of the function.

For example, we can attempt to define the factorial function with

```
Clear[factorial]
factorial[n_] := n factorial[n-1]
```

Hence, if we enter `factorial[5]`, Mathematica looks at its rules for the function `factorial`, and finds that it can replace `factorial[5]` by `5*factorial[5-1]` which it simplifies to `5 factorial[4]`. It looks at its rules again and sees that it can replace `factorial[4]` by `4*factorial[3]` giving `5*4 factorial[3]`. It looks at its rules again and sees that it can replace `factorial[3]` by `3*factorial[2]` giving `5*4*3 factorial[2]`, and so on.

But where does it stop? Let's see what happens:

```
factorial[5]
$RecursionLimit::reclim : Recursion depth of 256 exceeded.
0
```

It doesn't stop, because we haven't told it when it should!. What we need is another rule to give the stopping value. For the factorial function this is `factorial[0] := 1`. Our new definition becomes

```
Clear[factorial]
factorial[0] := 1;
factorial[n_] := n factorial[n - 1];
```

Now when we enter `factorial[5]` again, the function stops at `1`, and we get  $5 \times 4 \times 3 \times 2 \times 1 = 120$ .

```
factorial[5]
120
```

But what happens if we enter a non-integer argument.

```
factorial[5.0]
```

```
$RecursionLimit::reclim : Recursion depth of 256 exceeded.
```

```
0. Hold[factorial[-249. - 1]]
```

We would like the function to work for any positive integers, so we have to constrain the rules to deal only with positive integers.

```
Clear[factorial]

factorial[0] := 1;

factorial[n_Integer] /; Positive[n] := n factorial[n - 1];
```

```
factorial[5.0]
```

```
factorial[5.]
```

**Make sure recursive functions know where to stop.**

#### ◆ Example

Write a function to reverse the elements of a list without using the function **Reverse**.

Suppose we take a test list:

```
A = {a, b, c, d, e, f};
```

The strategy is to divide the list into *two* parts, the **First** element and the **Rest** of the elements.

```
First[A]
```

```
a
```

```
Rest[A]
```

```
{b, c, d, e, f}
```

Reverse these two to get **{Rest[A], First[A]}**. Then focus attention on **Rest[A]** as the list to be reversed, and continue until there is nothing left in the list to reverse. This is the stopping point and must be included as an extra rule. Finally get rid of all the extra list braces by using **Flatten**.

```
Clear[reverse]

reverse[{}] := {};

reverse[x_List] := Flatten[{reverse[Rest[x]], First[x]}];
```

```
reverse[{a, b, c, d, e, f}]
{f, e, d, c, b, a}
```

## Recursion Limits

---

Because most recursions are inadvertent, and would bring the computer to a screaming halt if allowed to proceed unregulated, Mathematica automatically counts the number of recursive steps in a calculation and stops the calculation if the number of steps exceeds the default value of 256. We will try this out on the factorial function. 250 steps is okay:

```
factorial[250];
```

However, 260 steps gets aborted.

```
factorial[260];
```

```
$RecursionLimit::reclim : Recursion depth of 256 exceeded.
```

We can reset the default value of the system variable `$RecursionLimit` to any number we want, (say **1024**) by entering

```
$RecursionLimit = 1024;
```

Now we can calculate `factorial[260]` without a complaint.

```
factorial[260]
```

```
3830195860836169235117497985604491875279556752309096960191 :
300817480651475135399533485285838275429773913773383359294 :
010103333339344249624060099745511339849626153802980398232 :
848965472622820196848860832049579523313702327662760125732 :
592551956622024712475139889122106940319324041688318583612 :
166708334763727216738353107304842707002261430265483385206 :
376839110078156900663427220806900528365808580136352143713 :
956803295894115605151395493267411709188354023557693440000 :
0000000000000000000000000000000000000000000000000000000000 :
000
```

## 9.6 Dynamic Programming

---

### The concept of a dynamic program

---

*Dynamic programming* is defining functions in such a way that they *remember* the values they have calculated, so that when these values are required next time, they will be immediately available (instead of having to be recalculated). Dynamic programs are usually significantly faster than non-dynamic implementations of the same algorithm.

Mathematica keeps each value calculated as another rule in the program, and hence dynamically writes the extended program for you. Here, you are effectively writing a program which intelligently modifies itself.

#### ◆ Example

Let us time the function **factorial** that we defined in the section above for **factorial[1000]**. We still have the recursion limit set at **1024**, so this should work.

```
Table[Timing[factorial[n]; n], {n, 200, 1000, 200}]
{{0.05 Second, 200},
 {0.0833333 Second, 400}, {0.133333 Second, 600},
 {0.2 Second, 800}, {0.283333 Second, 1000}}
```

The computation time is roughly linear with number of recursive steps.

We now modify the function to have the right hand side not only *calculate* the value, but *set* the value of the **factorial** function to be that value.

```
ClearAll[factorial]
factorial[0] := 1;
factorial[n_Integer] /; Positive[n] :=
  (factorial[n] = n factorial[n - 1]);
```

The *first* time we enter the dynamic function, it takes about the same time as the non-dynamic one.

```
Table[Timing[factorial[n]; n], {n, 200, 1000, 200}]
{{0.133333 Second, 200},
 {0.133333 Second, 400}, {0.133333 Second, 600},
 {0.166667 Second, 800}, {0.2 Second, 1000}}
```

The second time we enter it, it is too fast to register.

```
Table[Timing[factorial[n]; n], {n, 200, 1000, 200}]
{{0. Second, 200}, {0. Second, 400},
 {0. Second, 600}, {0. Second, 800}, {0. Second, 1000}}
```

If we ask for a different set of values, it is still just as fast since it has them all in memory. Of course, when you end the session and quit Mathematica, you will lose this information.

```
Table[Timing[factorial[n]; n], {n, 170, 1000, 200}]
{{0. Second, 170}, {0. Second, 370},
 {0. Second, 570}, {0. Second, 770}, {0. Second, 970}}
```

We should here reset **\$RecursionLimit** back to the default

```
$RecursionLimit = 256;
```

## Looking at the rule base for a recursively defined function

We would like to look and see what *Mathematica* knows about this new definition, but unless we want a very long printout we need to clear it (here we use **ClearAll**, to get rid of any values as well), reenter the function definition, then do a smaller calculation :

```
ClearAll[factorial]

factorial[0] := 1;

factorial[n_Integer] /; Positive[n] :=
  (factorial[n] = n factorial[n - 1]);
```

First we enter the function without having used it. The rules in the system are just as we have given them. We can find out the rule base for a function **f** by entering **??f**:

```
?? factorial
```

```
Global`factorial

factorial[0] := 1

factorial[n_Integer] /; Positive[n] := factorial[n] = n factorial[n - 1]
```

Now we use the function once on an argument of **5**. The new dynamically generated rules are now in the Kernel's memory, ready to be instantly returned if asked for.

```
factorial[5]
```

```
120
```

```
?? factorial
```

```
Global`factorial

factorial[0] := 1

factorial[1] = 1

factorial[2] = 2

factorial[3] = 6

factorial[4] = 24

factorial[5] = 120

factorial[n_Integer] /; Positive[n] := factorial[n] = n factorial[n - 1]
```

It can thus be seen that this dynamic mode of programming (using **Set** on the right hand side) actually modifies the program or function itself to augment its list of definitional rules.

Use dynamic programming for *fast* recursion.

## 9.7 Procedural Programming

---

### Traditional programming constructs

---

In this section we briefly discuss *procedural programming*. Traditional languages like *C*, *Basic*, *Fortran* and *Pascal* permit you only to program procedurally. Mathematica contains all the procedural programming constructs. However, it also has more efficient and effective ways of programming (which you have already learned).

We introduce three of the main procedural programming functions **Do**, **For**, and **While**, so that you can read other people's procedural code and relate it to what you have already learned. We introduce them at this stage as the last topic, not because they are the most advanced, but because we recommend, wherever possible, *not* using them.

#### **Do**

---

**Do** acts very much like **Table**, with the same iterator syntax. (The iterators are the arguments that tells the function which values to run through, like **{x, 3, 11}** below).

Here is a simple table

```
Table[x, {x, 3, 11}]
{3, 4, 5, 6, 7, 8, 9, 10, 11}
```

Here, the **Do** function does the same thing, but *does not give any output* (hence this effort is wasted).

```
Do[x, {x, 3, 11}]
```

To get an output from **Do**, you must get it to *do* something which produces an output.

To get the same output as the **Table** gives us, we need to build up the list inside the **Do** by starting out with an empty list **{}**, which we name **x** here, and then successively *build* the list we want by appending (adding a new element at the end) new elements.

We can **Append** a new element **x** to the end of a list **{a, b, c}** by writing

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

To **Do** this to **{}** from **x** equals **3** to **11** we enter

```
X = {};
Do[X = AppendTo[X, x], {x, 3, 11}]
```

There is still no output, but **X** will now have been modified:

```
X
{3, 4, 5, 6, 7, 8, 9, 10, 11}
```

Let us take a larger example and compare the timing of the two approaches

```
Timing[Table[x, {x, 3, 11000}];]
{0.05 Second, Null}

Timing[X = {};
Do[X = AppendTo[X, x], {x, 3, 11000}]]
{78.95 Second, Null}
```

You can see that in Mathematica, using **Do** and **AppendTo** takes about 1500 times as long as using **Table**. **AppendTo** is a particularly inefficient operation since it has to build a *new* list each time it is applied.

---

### For

Here is the same example using **For**.

```
X = {};
For[x = 3, x ≤ 11, x = x + 1, X = AppendTo[X, x]]
```

Like **Do**, **For** returns nothing. We have to ask for the value of **X**.

```
X
{3, 4, 5, 6, 7, 8, 9, 10, 11}
```

We see how long this takes on our larger example

```
Timing[X = {};
For[x = 3, x ≤ 11000, x = x + 1, X = AppendTo[X, x]]]
{80.4167 Second, Null}
```

---

### While

Here is the same example (we think!) using **While**.

```
X = {}; x = 3;
While[x ≤ 11, (x = x + 1; X = AppendTo[X, x])]
```

Like **Do** and **For**, **While** returns nothing. We have to ask for the value of **X**.

**x**

```
{4, 5, 6, 7, 8, 9, 10, 11, 12}
```

But this is not the same output. Our *initial value*  $x = 3$  and *test*  $x \leq 11$  have a different influence to that of the **Do** and **For** loops. We need to modify them to  $x = 2$  and  $x \leq 10$ .

```
X = {}; x = 2;
While[x ≤ 10, (x = x + 1; X = AppendTo[X, x])]
```

**x**

```
{3, 4, 5, 6, 7, 8, 9, 10, 11}
```

Again we see how long this takes on our larger example

```
Timing[X = {}; x = 2;
While[x ≤ 10999, (x = x + 1; X = AppendTo[X, x])]]
{81.0833 Second, Null}
```

**In Mathematica, avoid using Do, For, and While wherever possible.**

## 9.8 Problems

---

### Problem 1: Exploring timing

---

1. Write a function called **plotTiming** which takes the number of points in the **PlotPoints** option and returns the time it takes to do the plot:

```
Plot3D[Sin[x y], {x, 0, 2  $\pi$ }, {y, 0, 2  $\pi$ },
DisplayFunction → Identity, PlotPoints → n]
```

*Hint:* To get a *number* for the time, you will need to divide by the symbol **Second**.

2. Make a **ListPlot** of the times it takes to make the plot above as a function of the number of plot divisions **n**, from **n** equals **10** to **100** plot divisions. Calculate the time it takes to make this list plot. What unexpected behaviour do you observe as the number of plot divisions increases?

◆

### Problem 2: Exploring advanced patterns

---

Take the list  $\{x, 3.2, y, 1, 2.679, 3, z, 4, -7.9\}$  and write a rule to multiply all its symbols, and place it at the beginning of the list.



### **Problem 3: Exploring dynamic programming**

---

Turn the example function **reverse**, discussed above into a dynamic program. Compute how long it takes to reverse a table of integers from 1 to 200 with both the original and dynamic versions. Compare the results. (Remember, it is only the second time the dynamic function is applied that gives the time savings.)

