Getting started in APL
by solving Project Euler problems 1 to 50

Revision 1

Revisions

Revision	Date	Comments
1	11/11/2022	Initial release

Contents

Preface

If you are reading this, I guess it's fair to assume that you are into solving Project Euler problems. The soltution threads on PE were the place where I first encountered array programming languages like J, K or – much less frequently – APL.

My main language at that point was C, and I still enjoy using it. But seeing those cryptic one-liners (after I finally had my 100 line solution running) quickly aroused my curiosity. I first went with J, but I soon realized that this wasn't my cup of tea. It's a great language for sure and it offers some very useful builtin functions (e.g. regarding prime numbers), but I don't like the fact that you can't use inline functions like in APL or K. You either need to define the functions seperately, or use hard to read tacit style. K, on the other hand, was a better fit for my taste, but I didn't make friends with it for various reasons.

I avoided APL up to that point, because I was deterred by the special symbols it uses. But when you finally bring yourself to actually try it, you soon realize that it's just a matter of a few hours (at most) before working with them feels just as natural as using the equivalent ASCII characters in J or K.

This book follows my process of learning APL by solving Project Euler problems. I went with Dyalog APL, but you should be able to apply most of it to other dialects. If you want to give it a try, you don't need to install anything. Just visit Dyalog's online interpreter at TryAPL and you are good to go. My solutions are all tested to work there, but the ones using file input won't.

Disclaimer

I am by no means an expert in APL, nor math. While my solutions work, they'll probably show my lack of experience. So please don't take this book as an Hitchhiker's guide (I refer to Mastering Dyalog APL for this), but as a tool to to get you started and to overcome any reservations you may have towards APL.

This is thought to be a "follow along with me learning APL", so I won't optimize the solutions any further. But I appreciate you pointing out typos, errors or sections that need further clarification. Feel invited to contact me via mgeiss@mgeiss.de.

If you found this book to be helpful and want to support my work, you can do so by making a donation.

Problem 1 – Multiples of 3 or 5

So let's dive right into it with problem 1:

> If we list all the natural numbers below 10 that are multiples of 3 or 5, we get 3, 5, 6 and 9. The sum of these multiples is 23.
>
> Find the sum of all the multiples of 3 or 5 below 1000.

I have two solutions to offer, and during the course of this book, I will usually show the complete solutions before dissecting them. In my opinion, that's just the easier way to get going in languages like APL.

Anyway, here they come:

```
+/⍸∨⌿0=3 5∘.|⍳999   ⍝Solution 1 (Comments start with '⍝')

+/∪(3×⍳333),5×⍳199  ⍝Solution 2
```

Solution 1 uses modulo calculation to find all dumbers which are evenly divisible by 3 or 5. In solution 2, those multiples are directly created as lists.

Solution 1

Because array languages like APL usually evaluate expressions from right to left, we will also start on the right side. The first operator we find there uses **Iota** (⍳) in its monadic function (i.e. with only a right argument) as an index generator, with 999 as the argument.

Used in this most basic form, ⍳ will return a list from 1 up to its right argument:

```
    ⍳10
1 2 3 4 5 6 7 8 9 10
```

If you are familiar with J or K, you will notice that APL uses 1 as its index origin by default. This is not an issue as far as this book or solving PE problems is concerned, and I always have it set to 1.

If you absolutely want or need to, you can change the origin to zero with **⎕IO←0**:

```
    ⎕IO←0

    ⍳10
0 1 2 3 4 5 6 7 8 9
```

Iota is one of the operators you will see me use in pretty much all solutions, because we need lists of numbers all the time. Usually, it's the input on which we need to work on, but sometimes we also need lists that depend on the current value of a variable. And of course we can also use that as the argument instead of a fixed value:

```
    n←15

    ⍳n
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
```

The contents of the list can be further changed by prepending expressions as needed. Let's say we want a list to cover a specific range or to contain only even or odd numbers, squares, powers of 2 etc. This can all be done with just a few keystrokes:

```
    10+⍳10
11 12 13 14 15 16 17 18 19 20

    2×⍳10
2 4 6 8 10 12 14 16 18 20

    1+2×⍳10
3 5 7 9 11 13 15 17 19 21

    2*⍨⍳10
1 4 9 16 25 36 49 64 81 100

    2*⍳10
2 4 8 16 32 64 128 256 512 1024
```

I used **Switch** (⍨) for the squares example, which swaps the arguments of the affected operator, and can oftentimes help saving parenteses.

In this case, it also keeps the right-to-left principle intact because *Power* (*****) is one of those that evaluate left to right:

```
    2*3
8

    2*¨3
9
```

But now let's get back to the solution.

After having defined our input list, the next operator is **Residue** (**|**), which works just like the modulo function in any other language, except for the fact that it evaluates right to left again:

```
    7|3
3

    3|7
1
```

We can use lists on both sides of an operator, which allows us to calculate the results for modulo 3 and modulo 5 for the complete list with a single expression. But in order for this to work properly, we also need to bind the **Outer Product** expression (**∘.**) to the Residue operator. The result is a table containing two lists of results, the first one for mod 3, the second one for mod 5:

```
    3 5∘.|⍳9
1 2 0 1 2 0 1 2 0
1 2 3 4 0 1 2 3 4
```

Without *Outer Product* we would have gotten an error message, because of non matching list lengths:

```
    3 5|⍳9
LENGTH ERROR: Mismatched left and right argument shapes
    3 5|⍳9
       ^
```

Here are some examples for when *Outer Product* isn't needed:

```
⍝Single element left, list right
    5|⍳9
1 2 3 4 0 1 2 3 4

⍝List left, single element right
    (⍳9)|10
0 0 1 2 0 4 3 2 1

⍝Matching lengths result in pairwise operation
    3 5|7 8
1 3          ⍝3|7 , 5|8
```

Think of *Outer Product* as a method to apply an operator to all combinations of left and right elements. For example, you can use it to build a multiplication table:

```
    (⍳4)∘.×⍳5
1 2 3 4 5
2 4 6 8 10
3 6 9 12 15
4 8 12 16 20
```

We now have the results for mod 3 and mod 5, but we are only interested in those cases where the result is 0, i.e. the number being divisible by 3 or 5 without remainder.

To identify those results, we can add a comparison using **Equal** (`=`), which returns a table of two boolean lists, having 1s on all positions where the results were 0. To make our life easier, and because we want to find the numbers which are divisible by 3 OR 5, we can then merge both lists into one using the **Or** operator (`∨`).

Let's see what happens when we do all of this in order:

```
    3 5∘.|⍳9
1 2 0 1 2 0 1 2 0
1 2 3 4 0 1 2 3 4

    0=3 5∘.|⍳9
0 0 1 0 0 1 0 0 1
0 0 0 0 1 0 0 0 0

    ∨⌿0=3 5∘.|⍳9
0 0 1 0 1 1 0 0 1
```

We now have a 1 on every position where there was a 1 in the first or second list (or both). But to get the desired result, we needed to apply **Reduce First** (⌿) to the *Or* operation. Because we will also use **Reduce** (/) in a minute, we should take a look at these two operators first.

Given a list as the input, *Reduce* will insert the operator between all elements of the list. For example, if we want to get the sum of all numbers in a list (which we constantly do), we can use + with *Reduce*:

```
    +/1 2 3 4   ⍝Same as 1+2+3+4
10
```

Reduce is also known as *Fold*. If the input is a table, *Reduce* will insert the operator in all rows, returing a list of the results. *Reduce First* will insert the operator in all columns:

```
    ⎕←mat←3 3⍴⍳9
1 2 3
4 5 6
7 8 9

    +/mat
6 15 24     ⍝(1+2+3),(4+5+6),(7+8+9)

    +⌿mat
12 15 18    ⍝(1+4+7),(2+5+8),(3+6+9)
```

And that's why I had to use *Reduce First* instead of *Reduce* for the solution. Every column of the Table represents the results for 3 mod N and 5 mod N, and to join both with *Or*, the operator needs to be inserted between the elements of the columns.

We now have our list of joined boolean results. But what we need are the numbers that correspond to those results, and we can get them by using **Where** (⍸) . Given a list of 1s and 0s, it will return the indices of all 1s like so:

```
    ⍸1 1 0 0 1 0 1
1 2 5 7

    ⍸∨⌿0=3 5∘.|⍳9
3 5 6 9
```

As you can see, conveniently the indices of the 1s in our resulting boolean list are just the numbers we are interested in. This wouldn't have worked if the input didn't start at 1 and/or if the numbers in the list weren't spaced by 1. There are other methods to filter out the relevant results in these cases, and we'll see them in later problems. All that's left to do is to sum them all up using **+/**, and we are done:

```
    +/⍸⍳∨≠0=3 5∘.|⍳9
23
```

Solution 2

As I said in the beginning, solution 2 doesn't calculate the multiples. Instead, I set up two lists with the multiples under 1000 using **5×⍳199** and **3×⍳333**:

```
+/∪(3×⍳333),5×⍳199
```

Those two lists can then be joined using **Catenate** (**,**). But we shouldn't forget to put the second list (the left argument) in parentheses, or else the interpreter just joins 333 with the first list and then applies **⍳** to the result. Let's see how the result of *Catenate* looks like with a shorter range of numbers up to 20:

```
    (3×⍳6),5×⍳4
3 6 9 12 15 18 5 10 15 20
```

Now we face the problem that 15 (and all multiples of it in the actual solution) is in there twice, because it's a multiple of 3 and also of 5. To resolve this issue, we can use **Unique** (**∪**) which removes all multiples from a list:

```
    ∪1 2 2 3 4 5 5 5 6
1 2 3 4 5 6

    ∪(3×⍳6),5×⍳4
3 6 9 12 15 18 5 10 20
```

All that's left to do now is to use **+/** again to sum up all numbers.

Comparing the performance

With an input range that small, both solutions return the result pretty much instantly. But if we increase the range to all numbers under one million, we start to see a difference.

Solution 2 finishes in 8 milliseconds, solution 1 needs 25 milliseconds, thus taking three times as long, probably because of the modulo calculations. That's really not a big issue, but when we start dealing with complex problems and large number ranges, optimizing the solution is sometimes necessary. That being said, you will be facing memory issues sooner than runtime issues, at least if you aren't a millisecond junkie.

```
    ]runtime "+/⍳⍨∨≠0=3 5∘.|⍳999999"
* Benchmarking "+/⍳⍨∨≠0=3 5∘.|⍳999999"
            (ms)
CPU (avg):    25
Elapsed:      25


    ]runtime "+/∪(3×⍳333333),5×⍳199999"
* Benchmarking "+/∪(3×⍳333333),5×⍳199999"
            (ms)
CPU (avg):     8
Elapsed:       8
```

And that concludes chapter 1. In the next problem, we will start using functions and learn about two different methods to implement loops.

Problem 2 – Even Fibonacci numbers

Problem 2 deals with the Fibonacci sequence, and we need to find the sum of all even members of the sequence below four million:

> Each new term in the Fibonacci sequence is generated by adding the previous two terms. By starting with 1 and 2, the first 10 terms will be:
>
> 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...
>
> By considering the terms in the Fibonacci sequence whose values do not exceed four million, find the sum of the even-valued terms.

I can again offer two solutions, the first one using recursion to implement a loop, the second one working with the **Power Operator** (⍣):

```
{4E6>s←+/¯2↑⍵:∇⍵,s◇+/⍵[⍳⍨2|⍵]}1 2        ⍝Solution 1

{+/⍵[⍳⍨2|⍵]}{⍵,+/¯2↑⍵}⍣{4e6<+/¯2↑⍺}1 2  ⍝Solution 2
```

Solution 1

Here we see our first use of an anonymous inline function. Everything enclosed by the curly braces is the function body, and **Omega** (**⍵**) is APL's placeholder for the right argument. Let's see how that works with some easier to digest examples:

```
    {⍵+2}3
5

    {⍵+2}1 2 3
3 4 5

    {⍵×⍵}⍳10
1 4 9 16 25 36 49 64 81 100
```

The function will take the argument, replace **⍵** with it, and return the result. Just think of it as a more or less complex operator, whose function you can design yourself.

If needed, we can use **Alpha** (**⍺**) as the placeholder for a left argument, and also assign the function to a name:

```
    3{⍺+⍵}5
8

    plus←{⍺+⍵}
    3 plus 5
8
```

Furthermore, we can allow the function to be used modadically or dyadically by defining a default value for **⍺**. For example, we could design an increase function that increases its input by 1 if no left argument is given, or by the value of the left argument if one is used (essentially making it a useless plus function with benefits):

```
    increase←{⍺←1⋄⍵+⍺}
    increase 3
4

    5 increase 3
8
```

The **Statement Seperator** (**⋄**) does just what its name implies. Think of it as a replacement for a new line, when using inline functions instead of writing an APL script or using the editor of your APL distribution. It's similar to **;** in languages like C or Java.

Going back to the solution, we see that the function gets the list **1 2** passed as its right argument, which is the beginning of the Fibonacci sequence as defined in the problem. The function body consists of the following:

```
4E6>s←+/¯2↑⍵:∇⍵,s⋄+/⍵[⍳⍴⍵-2|⍵]
```

This is a so called **Guard** which works according to the principle **C:T⋄F**. In the first step, the condition C is evaluated. If this returns true, the expression T gets executed, else F. You can also chain multiple guards in an if...then...elseif...then...else...manner with **C1:T1⋄C2:T2⋄C3...⋄F**.

Let's see how that works with a few simpler examples:

```
    {⍵>3:1⋄0}5
1

    {⍵>3:1⋄0}2
0

    {⍵<3:3⋄⍵>5:5⋄0}2
3

    {⍵<3:3⋄⍵>5:5⋄0}6
5

    {⍵<3:3⋄⍵>5:5⋄0}4
0
```

In our case, the condition to check is `4E6>s←+/¯2↑⍵`, which evaluates if the sum of the last two elements of the current sequence is less than four million. `4e6` stands for $4x10^6$ and is just a shorter alternative to typing `4000000`, but that would also work just fine. `¯2↑⍵` uses **Take** (`↑`) to get the last two elements. Given a positive number N as its left argument, *Take* will return the first N elements. Using a negative N (denoted by `¯`), the result will be the last N elements:

```
    2↑3 4 5 6 7
3 4

    ¯2↑3 4 5 6 7
6 7
```

The sum is calculated by prepending our good friend `+/`. I store this value in `s` because I need it again in the next expression an this saves us from typing (and, more importantly, APL from calculating) `+/¯2↑⍵` again. This is a good example for the fact that you can define and initialize a variable within an expression.

If this condition is true (thus, the next term of the sequence will be less than four million), the next statement `∇⍵,s` is executed. This uses **Self Reference** (`∇`), which indicates the recursion. `∇` is the placeholder for the current function block, so `∇⍵,s` means "Append `s` to the current `⍵` and call yourself with that as the new right argument".

Hence, the new input will be `1 2 3`, the sequence extended by one term.

It is important to realize that this will acutally change the contents of **⍵**, so in the next iteration **⍵** is **1 2 3**, then **1 2 3 5** etc.

This will go on until we reach the point where the next term would be larger than four million and the "false" statement gets executed. Before we discuss the actual expression, let's quickly see what happens when we just put **⍵** there:

```
    {4E6>s←+/¯2↑⍵:∇⍵,s◇⍵}1 2
1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765
10946 17711 28657 46368 75025 121393 196418 317811 514229 832040
131346269 2178309 3524578
```

As expected, the result is the "finished" sequence. It also shows that the function works properly, so we just need to take care of filtering out the even terms and summing them. The list of even terms is found with **⍵[⍳~2|⍵]**. First, **2|⍵** returns a list of 1s and 0s, with 0s for all numbers that are divisible by 2. And because mod 2 has this special property of only resulting in 1 or 0, we can treat it as a boolean list and use **Not** (**~**) to switch the states:

```
    2|1 2 3 4 5 6 7 8 9 10
1 0 1 0 1 0 1 0 1 0

    ~2|1 2 3 4 5 6 7 8 9 10
0 1 0 1 0 1 0 1 0 1
```

The result is the same as if we had used **0=2|⍵** but with less to type. Just like in problem 1, we can now use *Where* again to get the indices of all even numbers in the list:

```
    {4E6>s←+/¯2↑⍵:∇⍵,s◇~2|⍵}1 2
0 1 0 0 1 0 0 1 0 0 1 0 0 1 0 0 1 0 0 1 0 0 1 0 0 1 0 0 1 0 0 1

    {4E6>s←+/¯2↑⍵:∇⍵,s◇⍳~2|⍵}1 2
2 5 8 11 14 17 20 23 26 29 32
```

To get from this to the corresponding terms of the sequence, we can use a notation similar to the array indexing in C/C++/Java etc. with **⍵[⍳~2|⍵]**. If you put a single number in the brackets, it works just like in those languages and will return the element at that index (keep in mind that APL's index origin is 1 by default).

A list of numbers returns the elements at multiple positions accordingly:

```
    fib←1 2 3 5 8 13 21 34 55 89
    fib[4]
5

    fib[2 5 8]
2 8 34

    {4E6>s←+/¯2↑⍵:∇⍵,s⋄⍵[⍳~2|⍵]}1 2
2 8 34 144 610 2584 10946 46368 196418 832040 3524578
```

And finally, you guessed it, **+/** will once again take care of the sum and we are done.

I will, however, add another way to select items from a list when you have a boolean list of the same length to use as a filter, and this is **Replicate** (**/**). I will discuss it in depth in later problems, but I guess you can figure it out by yourself with the function adjusted accordingly (note the use of *Switch* to swap the arguments and save parentheses):

```
    {4E6>s←+/¯2↑⍵:∇⍵,s⋄(~2|⍵)/⍵}1 2
2 8 34 144 610 2584 10946 46368 196418 832040 3524578

    {4E6>s←+/¯2↑⍵:∇⍵,s⋄⍵/⍨~2|⍵}1 2
2 8 34 144 610 2584 10946 46368 196418 832040 3524578
```

Solution 2

This solution uses the *Power Operator* (⍣, not to be confused with the exponentiation/nth power operator *) to build the Fibonacci sequence:

```
{+/⍵[⍳~2|⍵]}{⍵,+/¯2↑⍵}⍣{4e6<+/¯2↑⍺}1 2
```

You can also see another new aspect here: We can chain function blocks together just like we did with operators before. It's working in a $f(g(x))$ fashion, where the outermost function represents the leftmost function (or operator) in APL.

In this case, **{+/⍵[⍳~2|⍵]}** is the last function to be evaluated, and it is used to sum the even terms of the sequence in the exact same manner as in solution 1.

Which also means that the sequence itself is built using this part of the solution:

```
{⍵,+/¯2↑⍵}⍣{4e6<+/¯2↑⍺}1 2
```

The *Power Operator* is a very useful (powerful, indeed) tool to implement loops. The most simple form to use it is with a number as its right argument. In that case, it defines the number of iterations:

```
    {⍵×2}⍣10⊢1
1024
```

Here, 1 is the initial input, and it needed to be seperated from right the argument of the Power Operator, to prevent APL from interpreting **`10 1`** as a list. That's why I put **Right** (⊢) in between. Ignore that for now, we will discuss *Right* and *Left* at some later point. This use case would be roughly equivalent to a for loop.

To implement a while loop, we can use a comparison function as the right argument instead of a fixed number, with **⍺** being the current result of the function to the left of the *Power Operator*. In this case, the function will iterate until the comparison returns true:

```
    {⍵×2}⍣{⍺>4000}1
4096
```

Another interesting application is using **`⍣=`** to let a function run until a fixed point is reached. That is, until the function returns its input unchanged. Because the APL interpreter has a limited precision for floating point numbers, we can also make use of that when a function converges to some number. Here are three examples for this use case:

```
    {⍵<1000:⍵+1⋄⍵}⍣=1   ⍝starting with 1, add 1 until we reach 1000
1000

    1+{1÷2+⍵}⍣=1    ⍝continued fraction of the square root of 2
1.414213562

    1+{1÷1+⍵}⍣=1    ⍝continued fraction of the golden ratio
1.618033989
```

You should definitely play around with ⍣⍨ a bit to get a feel for it. Performance-wise, I couldn't see much of a difference compared to using ∇, but that's probably to be expected within the small scope of PE problems.

If we go back to the solution, we see that ⍣⍨ is used with a comparison as its stop condition, and that is basically the same as in solution 1 – we append the sum of the last two terms until that sum would become larger than four million:

```
    {⍵,+/¯2↑⍵}⍣⍨{4e6<+/¯2↑⍺}1 2
1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765
10946 17711 28657 46368 75025 121393 196418 317811 514229 832040
1346269 2178309 3524578
```

And that's it – the sequence is calculated as intended, and that result is passed to **{+/⍵[⍳~2|⍵]}**, taking care of the sum of even terms.

Problem 3 – Largest prime factor

To solve problem 3, we need to calculate the largest prime factor of 600851475143.

> The prime factors of 13195 are 5, 7, 13 and 29.
>
> What is the largest prime factor of the number 600851475143?

There are various methods for prime factorization, from simple trial division to highly complex algorithms. My solution is based on a simple form of trial division, not even using a prime sieve to make sure that we only divide by prime numbers. But you will see that this is more than sufficient for numbers that "small".

It takes just a millisecond to find the largest factor, which is only a 4-digit number in this case. But even if you remove the last digit from 600851475143, resulting in a largest prime factor of 10976461, the runtime is still barely measurable. Now let's take a look at the solution:

```
{α←3⋄0=2|⊃ω:∇(÷∘2@1)ω,2⋄0=α|⊃ω:α∇(÷∘α@1)ω,α⋄(α×α)<⊃ω:(α+2)∇ω⋄⊃ω}6008...
```

The first thing that happens is **α←3**, setting the initial value for **α** to 3. I do this because apart from 2, we only need to divide by odd numbers. If we handle that case seperately, we can then increase **α** by 2 in every iteration. We could in fact omit the "2-case" completely for this particular problem, because the number is odd. But in order to keep the solution suited for any number, I left it in there.

Then follows a chain of three guards. The first one handles the division by 2, hence it checks if **0=2|⊃ω** and then recursively calls the function with **(÷∘2@1)ω,2** as the new right argument. **⊃ω** uses **First** (**⊃**) to return the first element of **ω**. *First* does the same as **1↑ω**, it's just shorter.

Why do I use *First* if the input is just a single number? Because in the process of doing the factorization, I will append each factor to the current **ω** in order to have a complete list of the prime factors in the end. It's not needed for this question, but we can make use of that in later problems.

Now let's talk about **At** (**@**). It can be used to apply a function (or to set a value) only at defined indices or ranges of a list. The indices or the desired range go to the right of **@**, the value or function goes to the left.

I'll show some simpler examples:

```
    (-@2 4 6)6 4 8 3 7 1 2 5 9
6 ¯4 8 ¯3 7 ¯1 2 5 9

    (0@6 7 8)6 4 8 3 7 1 2 5 9
6 4 8 3 7 0 0 0 9

    (0@(>∘3))6 4 8 3 7 1 2 5 9
0 0 0 3 0 1 2 0 0
```

In our case, the list on which *At* should work on is `ω,2`, which is 2 appended to the current `ω`. For the next iteration, we need to divide the first element of this list (which is always the current "state" of our number to factor) by 2 with `(÷∘2@1)ω,2`.

We can't just use `÷2`, but need to combine this to a single expression with **Bind** (`∘`). You can think of *Bind* as a glue, making a monadic function out of `÷` and `2` which can be applied to numbers or lists as if we were using `{ω÷2}`:

```
    (÷∘2) 1 2 3 4 5 6
0.5 1 1.5 2 2.5 3
```

In fact, you can use this so called "tacit" style instead of `{...}` functions for a lot of things, but I usually try to avoid it if I can for readability reasons. But that's just me...

When we arrive at an odd number (or if the number to factor is odd), the next guard takes effect, which checks if `0=α|⊃ω`, being `0=3|⊃ω` initially. If that evaluates to true, the recursion continues with `α` unchanged and `(÷∘α@1)ω,α` as the new right argument. This does just the same as discussed before, only using `α` instead of 2.

If `⊃ω` is not divisible by `α`, the last guard comes into play, calling the function again with `α+2` as the new left argument and `ω` unchanged.

The *if* condition for this is `(α×α)<⊃ω`, meaning "If `α` is smaller than the square root of `⊃ω`". We can limit the loop to the square root, because it can be shown that every composite number has at least one prime factor which is smaller that its square root. Or in other words, if no prime factor smaller than the square root of `⊃ω` is found, then `⊃ω` is the last (and largest) prime factor.

That's why the final "false" case just returns `⊃ω`, and that's also the solution to problem 3.

But let's quickly check if the function works as intended, by replacing that last statement with just **⍵** and give it a number to factor:

```
    {α←3◇0=2|⊃⍵:∇(÷∘2@1)⍵,2◇0=α|⊃⍵:α∇(÷∘α@1)⍵,α◇(α×α)<⊃⍵:(α+2)∇⍵◇⍵}10080
7 2 2 2 2 2 3 3 5
```

And that indeed are all prime factors of 10080. Because of the way the function is designed, the largest factor is the first one in the list, while the others were appended in ascending order. If we needed the whole list sorted in ascending order, we could take care of this easily by adjusting the last statement to **⍵[⍋⍵]**, but I'll save that topic for another problem.

If you followed the explanations for Problem 1 and 2, you now know enough APL to solve many of the easier PE problems. You will get to learn some more operators and functionalities in the following chapters, but the hardest part is over!

Problem 4 – Largest palindrome product

Problem 4 wants us to find the largest palindromic number (i.e. a number that stays the same written forwards or backwards) that can be formed by the product of two 3-digit numbers.

> A palindromic number reads the same both ways. The largest palindrome made from the product of two 2-digit numbers is 9009 = 91 × 99.
>
> Find the largest palindrome made from the product of two 3-digit numbers.

My solution is based on converting the numbers to strings, as strings are just lists of characters in APL (just like in C) and can be reversed easily:

```
{⌈/ω[⍸{ω≡⌽ω}¨⍷¨ω]}∪∊∘.×⍨100+⍳899

{⌈/ω/⍨{ω≡⌽ω}¨⍷¨ω}∪∊∘.×⍨100+⍳899    ⍝same, using Replicate
```

As already mentioned in problem 1, *Outer Product* (`∘.`) can be used to create a multiplication table like so:

```
    (⍳4)∘.×⍳5
1 2  3  4  5
2 4  6  8 10
3 6  9 12 15
4 8 12 16 20
```

To get all possible products of 3-digit numbers, we can use the same technique. First, let's build a list of all 3-digit numbers with `100+⍳899`. If we want to use the same input on both sides of an operator, we can use **Commute/Switch** (`⍨`) to copy the right argument over to the left side:

```
    +⍨3
6

    ∘.×⍨⍳3
1 2 3
2 4 6
3 6 9
```

As you can already see in the little multiplication table using only `⍳3`, we get a lot of redundant results which we can filter out using *Unique*, as we know. But to do so, we

first need to convert the matrix back to a list, otherwise *Unique* will only look for unique rows or columns, depending on how you apply it to the matrix.

To get the elements as a list, we prepend **Enlist** (∈), after which it's "safe" to apply *Unique*:

```
    ∈∘.×⍨⍳5
1 2 3 4 5 2 4 6 8 10 3 6 9 12 15 4 8 12 16 20 5 10 15 20 25

    ∪∈∘.×⍨⍳5
1 2 3 4 5 6 8 10 9 12 15 16 20 25
```

This list of unique products in now the input to my function `{⌈/⍵[⍸⍵≡ϕ⍵⍤⍕⍤⍵]}`.

The first action there is `⍕⍤⍵`, which applies **Format** (⍕) to **Each** (⍤) element of the list seperately. Wait, what?

Sometimes, when you need a list of seperate results, you don't want to apply an operator or a function to an input list as a whole. In this case, I use *Each* because otherwise *Format* – which converts a number to a string – would just make a single string out of the whole list (like a sentence with the numbers as words). But we need to be able to reverse each number seperately, and that's why we need them as seperate strings.

If you turn boxing on with `]box on` (should be on by default if you use TryAPL), you can clearly see the difference:

```
    ⍕123 456 789
123 456 789

    ⍕⍤123 456 789
┌───┬───┬───┐
│123│456│789│
└───┴───┴───┘
```

For the same reason, the inner function is also applied with *Each*. If we use **Reverse** (ϕ) on this result (which, you guessed it, returns the reverse of its input) without using *Each*, it will just reverse the order, even if the strings are now seperate items. I'll show the difference again:

```
    ϕ⍳10
10 9 8 7 6 5 4 3 2 1
```

```
    ⌽⍤¨123 456 789
┌───┬───┬───┐
│789│456│123│
└───┴───┴───┘

    ⌽¨⍤¨123 456 789
┌───┬───┬───┐
│321│654│987│
└───┴───┴───┘
```

But the inner function doesn't only reverse each string, it also checks with **Match** (**≡**) if the original matches its reverse. We can't use **=** in this case, becasue applied to strings it compares every character and returns a list of results instead of a single boolean value:

```
    {⍵≡⌽⍵}'3243'
0

    {⍵≡⌽⍵}'3223'
1

    {⍵=⌽⍵}'3243'
1 0 0 1
```

The result up to this point is a boolean list with 1s on all positions where we have a palindromic number:

```
    {{⍵≡⌽⍵}¨⍤¨⍵}∪∈∘.×⍨100+⍳899
1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 ...
```

We can now use *Where* again to get the indices of the 1s, and then use that to filter our input with **⍵[⍸⍵≡⌽⍵¨⍤¨⍵]** (or by using *Replicate* as shown in the alternative solution):

```
    {⍵[⍸{⍵≡⌽⍵}¨⍤¨⍵]}∪∈∘.×⍨100+⍳899
10201 11211 12221 13231 14241 15251 16261 17271 ...
```

There is only one thing left to do, and that is to find the largest number. For this, we apply **Maximum** (**⌈**) with *Reduce*.

Given numbers to its left and right, *Maximum* will return the larger of both. Using it on a list of numbers with *Reduce*, *Maximum* will return the largest member:

```
    3⌈6
6

    6⌈3
6

    ⌈/1 8 4 5 3 7 2
8
```

And that already concludes this chapter, but we met quite a few new operators to make of in the following problems!

Problem 5 – Smallest multiple

The chapter about problem 5 will be a short one, because of Dyalog APL's built function to find the least common multiple (LCM).

> 2520 is the smallest number that can be divided by each of the numbers from 1 to 10 without any remainder.
>
> What is the smallest positive number that is evenly divisible by all of the numbers from 1 to 20?

This is just another wording for the question "Which is the LCM of the numbers from 1 to 20?". As I said, Dyalog APL has a built in function to calculate the LCM of two numbers, and it shares the logical **And** operator (**∧**). All we need to do now, is to apply this with *Reduce* to the list of numbers from 1 to 20, and we are done:

```
∧/⍳20
```

But in order to not let this end so soon, and because it fits here, I'll mention that the *Or* operator (**∨**) is also a GCD (greatest common divisor) function:

```
     24∨512
8
```

And that's really it for this problem. No, wait, there's more!

For those of you who use an APL dialect without LCM function, I'll also provide a "manual" solution which is just using a well knwon GCD algorithm as the inner function to calculate the LCM in the outer one:

```
{(⍺×⍵)÷⍺{⍵=0:⍺⋄(⍺|⍵)∇⍺}⍵}/⍳20
```

I'll levae that one uncommented, and now it's really the end of this chapter.

Problem 6 – Sum square difference

In problem 6 we need to calculate the difference between the square of the sum and the sum of the squares of the first 100 natural numbers.

> The sum of the squares of the first ten natural numbers is
>
> $1^2 + 2^2 + \cdots + 10^2 = 385$
>
> The square of the sum of the first ten natural numbers is
>
> $(1 + 2 + \cdots + 10)^2 = 55^2 = 3052$
>
> Hence the difference between the sum of the squares of the first ten natural numbers and the square of the sum is
>
> $3052 - 385 = 2640$
>
> Find the difference between the sum of the squares of the first one hundred natural numbers and the square of the sum.

A 1:1 translation of this to APL is my solution and it looks as follows:

```
{(2*⍨+/⍵)-+/⍵*2}⍳100
```

There's really not much to explain here if you read the previous chapters. I use *Switch* in the expression for the square of the sum in order to save a pair of parentheses, but we absolutely need parentheses before the minus sign to let the interpreter know that we want the expression to the left evaluated before the result of the right one gets subtracted.

The next problem is going to be much more interesting, I promise!

Problem 7 – 10001st prime

Problem 7 deals with prime numbers, and that's the perfect opportunity to build a prime sieve function, because we'll need that over and over again for different PE problems.

> By listing the first six prime numbers: 2, 3, 5, 7, 11, and 13, we can see that the 6th prime is 13.
>
> What is the 10001st prime number?

I use the "classic" Sieve of Eratosthenes to solve this, because despite its simplicity it's still one of the fastest algorithms for small prime numbers. Even solving Problem 10 with it (the sum of the first two million primes) takes just 65 milliseconds on my laptop.

And this is my Implementation of it in the following solution (barely counts as a one-liner):

```
{n←⍵⋄3{n≥⍺×⍺:(⍺+2)∇(0@((⍺-1)↓⍺×⍳⌊n÷⍺))⍵⋄(⍸⍵)[10001]}0 1,(⍵-2)⍴1 0}115000
```

I set the limit for the sieve to 115000, because it can be shown that the nth prime is guaranteed to be less than $n(\log n + \log\log n)$, which calculates to 114319 for n=10001.

I think a good way to start discussing this is reducing the function to the actual prime sieve. I'll also lower the limit to 100:

```
    {n←⍵⋄3{n≥⍺×⍺:(⍺+2)∇(0@((⍺-1)↓⍺×⍳⌊n÷⍺))⍵⋄⍸⍵}0 1,(⍵-2)⍴1 0}100
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
```

You can see that I use two nested functions. The outer one gets the upper limit as its right argument, then it uses `0 1,(⍵-2)⍴1 0` to create the initial boolean list for the sieve. In the first expression the limit is stored in n to make it accessible in the inner function without needing to pass is directly as a left or right argument.

Hint: When you use nested functions, it is important to know that **⍺** and **⍵** are completely independent in both, although they share the same name/symbol.

`0 1` are the first elements of the list (because 1 isn't prime, but 2 is), and to that I append `(⍵-2)⍴1 0`. This uses **Shape** (**⍴**) to create a list of `⍵-2` copies of `1 0` like so:

```
    10⍴1 0
1 0 1 0 1 0 1 0 1 0
```

I do this to save one run of the sieve where it marks all even numbers except 2 as non-prime, so we can limit the inner function to odd numbers. This speeds up the sieve function by 40-50% in my experience. If the input of *Shape* is a list, and the left argument is longer than this, it will continue appending to the list until the shape is full. Or vice versa, if the left argument is smaller, the list will be truncated accordingly:

```
    10⍴⍳3
1 2 3 1 2 3 1 2 3 1

    4⍴⍳10
1 2 3 4
```

You can also use Shape to create n-dimensional arrays. The same rules apply regarding fill and truncate:

```
    3 5⍴⍳20
 1  2  3  4  5
 6  7  8  9 10
11 12 13 14 15

    3 5⍴⍳3
1 2 3 1 2
3 1 2 3 1
2 3 1 2 3
```

The result in our case is a list of n elements representing every number up to the limit, with the first one being 0 (for non-prime) while all odd numbers except 2 are currently assumed to be prime:

```
    {0 1,(⍵-2)⍴1 0}100
0 1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 ...
```

The inner function gets this as its right argument and 3 as its left argument, being the first number for the sieving algorithm to work on. The function body looks like this:

```
{n≥⍺×⍺:(⍺+2)∇(0@((⍺-1)↓⍺×⍳⌊n÷⍺))⍵⋄⍸⍵}
```

The function consists of an if...then...else guard which follows the SoE procedure apart from checking if the new value for **⍺** has already been cancelled, but is limited to odd numbers, which allows us to increase **⍺** by 2 in every iteration.

We first check with **n≥α×α** if **α** is smaller than the square root of **n**. While that evaluates to true, the following recursive block gets executed:

```
(α+2)∇(0@((α-1)↓α×⍳⌊n÷α))ω
```

The expression to the right of **∇**, which will be the input for the next iteration, is the one responsible for setting all multiples of the current value of **α** to zero. It uses *At* to do this, and **(α-1)↓α×⍳⌊n÷α** to build the list of multiples, starting with the square of **α**, which is sufficient since all multiples below that have already been cancelled in previous iterations.

This one took a bit of fiddling around with the numbers, but you can see that it works as intended:

```
    3{(α-1)↓α×⍳⌊ω÷α}100
9 12 18 21 24 27 30 33 36 ... 93 96 99

    5{(α-1)↓α×⍳⌊ω÷α}100
25 30 35 40 45 50 55 60 65 70 75 80 85 90 95 100

    7{(α-1)↓α×⍳⌊ω÷α}100
49 56 63 70 77 84 91 98
```

We have to use **Floor** (**⌊**) here to get the floor of **n÷α**, because clearly **n** ist'n evenly divisible by every value of **α**, but **⍳** can only create a list given an integer number.

Furthermore, I use **Drop** (**↓**) to drop the leading **α-1** elements of the list. This takes care of removing the numbers up to the square root of **α**. In general, given a positive number N to its left, *Drop* will remove the leading N elements. A negative N will drop the trailing N elements:

```
    3↓1 2 3 4 5
4 5

    ¯2↓1 2 3 4 5
1 2 3
```

The indices for *At* are delivered by **(α-1)↓α×⍳⌊n÷α** and we want *At* to put a 0 there. This is now the left argument for the new iteration, the right one is just **α+2**.

When **⍺** reaches the square root of **⍵**, the recursion stops and *Where* is used to return the indices of the remaining 1s, being the prime numbers. We now have a list of the prime numbers up to the given limit. To get the 10001st number of this, we just need to change the last statement from **⍳⍵** to **(⍳⍵)[10001]**, and we are done!

Problem 8 – Largest product in a series

Problem 8 throws a 1000-digit number at us, asking for the greatest product of 13 adjacent digits.

> The four adjacent digits in the 1000-digit number that have the greatest product are $9 \times 9 \times 8 \times 9 = 5832$.
>
> 7316717653133062491922511967442657474235534919493496983520312774506326239578318016984801869478851843
> 8586156078911294949545950173795833195285320880551112540698747158523863050715693290963295227443043557
> 6689664895044524452316173185640309871112172238311362229893423380308135336276614282806444486645238749
> 3035890729629049156044077239071381051585930796086670172427121883998797908792274921901699720888093776
> 6572733300105336788122023542180975125454059475224352584907711670556013604839586446706324415722155397
> 5369781797784617406495514929086256932197846862248283972241375657056057490261407972968652414535100474
> 8216637048440319989000889524345065854122758866688116427171479924442928230863465674813919123162824586
> 1786645835912456652947654568284891288314260769004224219022671055626321111109370544217506941658960408
> 0719840385096245544436298123098787992724428490918884580156166097919133875499200524063689912560717606
> 0588611646710940507754100225698315520005593572972571636269561882670428252483600823257530420752963450
>
> Find the thirteen adjacent digits in the 1000-digit number that have the greatest product. What is the value of this product?

This is the perfect opportunity to introduce reading input from a file. My solution uses that and looks like this:

```
0⍎⌈/13×/⍎¨⊃⎕NGET'/path/to/p08.txt'1
```

To prepare for this, I first copy&pasted the number from the PE website as is to a file named `p08.txt` (the name doesn't matter, of course). If you leave it at this, hence keeping the line breaks, the solution should work as is.

Dyalog's standard function to read from text files is `⎕NGET`, and you need to specify the file name including the complete path enclosed in `''`. We also usually put a 1 behind this in order to let `⎕NGET` return the lines as boxed strings.

More importantly, we need to prepend *First* (⊃) in order to only get the file's content, because ⎕NGET also returns some additional info (see below).

As already discussed, *First* will return the first element of its input. In case of a list of numbers, this will just be the first number. If the input is partitioned, like the output of ⎕NGET, it will return the first partition:

```
    ⎕NGET'/path/to/testfile.txt'1
┌──────────────┬──────────┬──┐
│┌────────────┐│UTF-8-NOBOM│10│
│|Some Content|||          │  │
│└────────────┘│          │  │
└──────────────┴──────────┴──┘

    ⊃⎕NGET'/path/to/testfile.txt'1
┌────────────┐
|Some Content|
└────────────┘
```

After this, our number is now a list of boxed strings, one for every line in the file. Usually, that's fine, but in this case we just want an unsegmented list of all numbers, so we use *Enlist* to take care of this:

```
    ⊃⎕NGET'/path/to/Euler/p08.txt'1
┌──────────────────────────────────────────────────┬────┐
|7316717653133062491922511967442657474235534919493|9698 ...
└──────────────────────────────────────────────────┴────┘

    ∈⊃⎕NGET'/path/to/Euler/p08.txt'1
73167176531330624919225119674426574742355349194934 ...
```

But those are still characters, so we first have to convert them to numbers in order to do our calculations. For this, we can use **Execute** (⍎), which is basically the reverse of *Format*, and converts strings to numbers. It can also execute APL expressions stored as a string, hence the name. Don't forget to append *Each* to *Execute*, beacuse we want every number as a seperate item, not the 1000-digit number translated as a whole:

```
    ⍎¨∈⊃⎕NGET'/path/to/Euler/p08.txt'1
7 3 1 6 7 1 7 6 5 3 1 3 3 0 6 2 4 9 1 9 2 2 5 1 1 9 6 ...

    ⍎∈⊃⎕NGET'/path/to/Euler/p08.txt'1
7.316717653E999
```

To get all products of 13 adjacent digits, we can make use of a nice function called *N-Wise Reduce*. If you put a number N before the operator when using it with *Reduce*, it will be applied to all consecutive sublists of length N. As an example, imagine you want the sums or maxima of all adjacent number pairs in a list. This can be done with **2+/** or **2⌈/** like so:

```
    2+/1 4 8 1 6 5
5 12 9 7 11

    2⌈/1 4 8 1 6 5
4 8 8 6 6
```

In our case, we use **13×/** to get all products of 13 adjacent digits, which of course results in many of them being zero as there are a lot of zeros in the list:

```
    13×/⍎¨⊃⎕NGET'/path/to/p08.txt'1
5000940 0 0 0 0 0 0 0 0 0 0 0 0 0 4199040 4898880 9797760 ...
```

The last step is to use **⌈/** to get the maximum of all results. But since APL uses scientific notation for all numbers with more than 10 digits by default, we also need to prepend **0⍕**, which converts the result to a string with 0 decimal digits:

```
    2*50
1.125899907E15

    0⍕2*50
1125899906842624
```

There are ways to change this behaviour, but since it doesn't matter if we get the result as a number or a string, this is the easiest and quickest way.

Problem 9 – Special Pythagorean triplet

Pythagorean triplets are the topic of problem 9, and we have to find the one for which $a + b + c = 1000$.

> A Pythagorean triplet is a set of three natural numbers, $a < b < c$, for which, $a^2 + b^2 = c^2$
>
> For example, $3^2 + 4^2 = 9 + 16 = 25 = 5^2$.
>
> There exists exactly one Pythagorean triplet for which $a + b + c = 1000$. Find the product abc.

While I know that there are ways to solve this analytically, even with pen&paper in a few minutes if you know the formula, I decided to play around a bit and implement an algorithm which creates pythagorean triplets and then use that to find our candidate. Because it can never hurt to have a notebook of handy algorithms to use for later problems, right?

The algorithm which I use in `tri` is an implementation of Dickson's Method, and it is used in the worker function below it to find the answer:

```
tri←{⍵{⍺+⍵[1],⍵[2],+/⍵}¨{⍵,n÷⍵}¨{⍸0=⍵|⍨⍳⌊⍵*÷2}n←2÷⍨×⍨⍵×⍵}

{⍺←2◇+/n←⍵=+/¨tri ⍺:×/∈n/tri ⍺◇(⍺+2)∇⍵}1000
```

Please read the Wikipedia section on Dickson's Method if you want to know the details, I'll just explain my implementation of it. But first, let's see if it works as intended by giving it 6 as an input parameter to match the value of r in the article:

```
    tri 6
┌─────────┬─────────┬─────────┐
│7 24 25  │8 15 17  │9 12 15  │
└─────────┴─────────┴─────────┘
```

Apparently, it does! So let's dissect the function bit by bit. The input to the outer function is the value of r, which needs to be an even number. We also need the value of $\frac{r^2}{2}$, that's why the first step is to calculate it using `2÷⍨×⍨⍵×⍵`. I also store this result in `n`, because it's needed again later.

Then the rightmost inner function is used to calculate all "lower factors" of $\frac{r^2}{2}$ with `⍸0=⍵|⍨⍳⌊⍵*÷2`. I used Switch to reverse the arguments of the modulo operator to save a pair of parentheses, so the actual left argument is `⍳⌊⍵*÷2`, a list of all natural numbers up to the floor of the square root of $\frac{r^2}{2}$.

Just like in Problem 1, I now use *Where* to get the numbers for which the result is 0, thus being the "lower factors" of $\frac{r^2}{2}$. The result is as expected:

```
    {{⍸0=ω|⍨⍳⌊ω*÷2}n←2÷⍨ω×ω}6
1 2 3
```

Because we need the facor pairs (each "lower factor" paired with its corresponding "higher factor"), the next function takes this list and is used with *Each* to return a seperate result for each "lower factor". All it does is appending **n÷ω** to each "lower factor", and this is the outcome:

```
    {{ω,n÷ω}¨{⍸0=ω|⍨⍳⌊ω*÷2}n←2÷⍨ω×ω}6
┌─────┬─────┬─────┐
│1 18 │2 9  │3 6  │
└─────┴─────┴─────┘
```

The last function finally calculates the triplets. It uses this result as its right argument, and it's again applied to *Each* factor pair seperately. The left input is r. This doesn't need much explanation, there's nothing new there. It's just the calculation of what is $r+s$, $r+t$, $r+s+t$ in the Wikipedia article. r is our **⍺**, and we can just factor this out, resulting in **⍺+ω[1],ω[2],+/ω**.

Now to our worker function:

```
{⍺←2◇+/n←ω=+/¨tri ⍺:×/∈n/tri ⍺◇(⍺+2)∇ω}1000
```

It gets 1000 (the perimeter) as it's right input and sets **⍺←2** as the initial value for r. Then the usual if...then...else guard follows, which first evaluates **+/n←ω=+/¨tri ⍺**. This calculates all perimeters of the triplets returned by **tri ⍺**, again using *Each* to get seperate results for each triplet. The list of perimeters is then compared with **ω** (which is 1000), resulting in a boolean list with mostly 0s, until the recursion arrives at a value for r which produces the triplet we are looking for.

The list is stored in **n** because I need it in the *then* case as a filter. But because the *if* statement can't make use of a list of results (even if all of them are 0 or 1), I use **+/** to get the sum of all results. Because we know that there is only one triplet which has a perimeter of 1000, we also know that this sum will always be either 0 or 1, in which case we have found our result.

As long as this returns 0, the else case executes the next iteration with **α+2**, as Dickson's Method only works with even numbers. When the matching triplet is finally found, our *then* case comes into play. This executes **×/∈n/tri α**, which uses n as the filter for *Replicate* to extract the relevant triplet:

```
    {α←2⋄+/n←ω=+/¨tri α:n/tri α⋄(α+2)∇ω}1000
┌──────────┐
│200 375 425│
└──────────┘
```

Finally, *Enlist* converts the triplet to a list, and we apply **×/** to get the product abc.

Problem 10 – Summation of primes

Already at problem 10! And that's a nice one because we can reuse one of our previous solutions. We are asked to find the sum of all primes below two million.

> The sum of the primes below 10 is 2 + 3 + 5 + 7 = 17.
> Find the sum of all the primes below two million.

Remember problem 7? I already mentioned there that a simple Sieve of Eratosthenes is more than fast enough to solve this, and that's why we can use the exact same function, only modified to return the sum of the whole list of primes instead of just the 10001st one:

```
{n←⍵⋄3{n≥⍺×⍺:(⍺+2)∇(0@((⍺-1)↓⍺×⍳⌊n÷⍺))⍵⋄0⍕+/⍸⍵}0 1,(⍵-2)⍴1 0}2e6
```

You can refer to problem 7 for a detailed explanation of the function. I'll just mention that I used *Format* with `0⍕` again after the sum, to prevent the interpreter from returing it in scientific notation.

```
    ]runtime "{n←⍵⋄3{n≥⍺×⍺: ... ⋄0⍕+/⍸⍵}0 1,(⍵-2)⍴1 0}2e6"
* Benchmarking "{n←⍵⋄3{n≥⍺×⍺: ... ⋄0⍕+/⍸⍵}0 1,(⍵-2)⍴1 0}2e6"
            (ms)
CPU (avg):    63
Elapsed:      63
```

Maybe not as fast as the same algorithm would be in C, but still...

Problem 11 – Largest product in a grid

To solve problem 11 we need to find the largest product of four adjacent numbers in a 20x20 grid in different directions.

> In the 20×20 grid below, four numbers along a diagonal line have been marked in red.
>
> 08 02 22 97 38 15 00 40 00 75 04 05 07 78 52 12 50 77 91 08
> 49 49 99 40 17 81 18 57 60 87 17 40 98 43 69 48 04 56 62 00
> 81 49 31 73 55 79 14 29 93 71 40 67 53 88 30 03 49 13 36 65
> 52 70 95 23 04 60 11 42 69 24 68 56 01 32 56 71 37 02 36 91
> 22 31 16 71 51 67 63 89 41 92 36 54 22 40 40 28 66 33 13 80
> 24 47 32 60 99 03 45 02 44 75 33 53 78 36 84 20 35 17 12 50
> 32 98 81 28 64 23 67 10 **26** 38 40 67 59 54 70 66 18 38 64 70
> 67 26 20 68 02 62 12 20 95 **63** 94 39 63 08 40 91 66 49 94 21
> 24 55 58 05 66 73 99 26 97 17 **78** 78 96 83 14 88 34 89 63 72
> 21 36 23 09 75 00 76 44 20 45 35 **14** 00 61 33 97 34 31 33 95
> 78 17 53 28 22 75 31 67 15 94 03 80 04 62 16 14 09 53 56 92
> 16 39 05 42 96 35 31 47 55 58 88 24 00 17 54 24 36 29 85 57
> 86 56 00 48 35 71 89 07 05 44 44 37 44 60 21 58 51 54 17 58
> 19 80 81 68 05 94 47 69 28 73 92 13 86 52 17 77 04 89 55 40
> 04 52 08 83 97 35 99 16 07 97 57 32 16 26 26 79 33 27 98 66
> 88 36 68 87 57 62 20 72 03 46 33 67 46 55 12 32 63 93 53 69
> 04 42 16 73 38 25 39 11 24 94 72 18 08 46 29 32 40 62 76 36
> 20 69 36 41 72 30 23 88 34 62 99 69 82 67 59 85 74 04 36 16
> 20 73 35 29 78 31 90 01 74 31 49 71 48 86 81 16 23 57 05 54
> 01 70 54 71 83 51 54 69 16 92 33 48 61 43 52 01 89 19 67 48
>
> The product of these numbers is 26 × 63 × 78 × 14 = 1788696.
>
> What is the greatest product of four adjacent numbers in the same direction (up, down, left, right, or diagonally) in the 20×20 grid?

Similar to problem 8, I first copied the grid to a file and then used its contents for my solution as follows:

```
g←↑⍎¨⊃⎕NGET'/path/to/p11.txt'1
⌈/(∊4×/g),(∊4×⌿g),∊4×/¨{1 1⌽⍵}¨↑{(⍵↓g)(⍵↓⌽g)(⍵↓⍉g)(⍵↓⌽⍉g)}¨¯1+⍳17
```

The challenging part here was to find a way to get the products in the diagonals, as calculating them for the rows and columns is easy. But let's begin…

After pasting the grid in a file named `p11.txt`, I used `⊃⎕NGET` in the same fashion as in problem 8, again prepending *First* to only give me the contents. This, as usual, results in a segmented list with each segment containing one line of the file:

```
    ⍎⎕NGET'/path/to/p11.txt' 1
┌──────────────────────────────────────────────────────────────┬──────┐
│08 02 22 97 38 15 00 40 00 75 04 05 07 78 52 12 50 77 91 08│49 49 ...
└──────────────────────────────────────────────────────────────┴──────┘
```

Then I applied *Execute* to *Each* segment, resulting in a segmented list of numbers. To remove the segmentation and convert the list to a tabe, we can use **Mix**, the monadic form of *Take* (↑), which removes the segmentation and inserts each segment as a row in a matrix, filling up spaces with 0s as needed (which is not the case with our grid):

```
    (1 2)(3 4 5)(6 7 8 9)
┌─────┬─────┬───────┐
│1 2│3 4 5│6 7 8 9│
└─────┴─────┴───────┘

    ↑(1 2)(3 4 5)(6 7 8 9)
1 2 0 0
3 4 5 0
6 7 8 9

    ↑⍎¨⍎⎕NGET'/path/to/p11.txt' 1
 8  2 22 97 38 15  0 40  0 75
49 49 99 40 17 81 18 57 60 87
81 49 31 73 55 79 14 29 93 71 ...
52 70 95 23  4 60 11 42 69 24
22 31 16 71 51 67 63 89 41 92
...
```

Finally, I use **g←** to store the resulting grid in a variable. Now, to get a list of the products of 4 adjacent numbers in the rows or columns, we can just use **∊4×/g** or **∊4×⌿g** respectively. See problem 1 for the difference between **/** and **⌿**, but it's suffice to say that **/** operates on the rows, and **⌿** on the columns.

So the first part of the final line in the solution does just that, catenating both lists and applying **⌈/** to get the maximum. But apparently, the product we are looking for sits on one of the diagonals, so we need to dig deeper.

There is no direct way to apply functions or operators to all diagonals, but at least we can make use of Dyadic transpose with **1 1⍉**. Transpose in its monadic form will flip a matrix.

Its dyadic form is normally only relevant for multi-dimensional arrays, so I won't cover that in detail here. But the special case `1 1⍉` can be used to extract the main diagonal of a matrix:

```
    3 3⍴⍳9
1 2 3
4 5 6
7 8 9

    ⍉3 3⍴⍳9
1 4 7
2 5 8
3 6 9

    1 1⍉3 3⍴⍳9
1 5 9
```

That helps, but it can only return the main diagonal starting from the top left corner. In order to get all diagonals, I use a function which continously drops rows and columns from the matrix until only four are left, which shifts the origin for the main diagonal accordingly. By default *Drop* removes rows. To let it work on the columns, we can either use *Transpose*, which is what I did, or specify "rank 2" by appending `[2]` to *Drop*.

Let's see how that works with a little example:

```
    □←mat←4 4⍴⍳16
 1  2  3  4
 5  6  7  8
 9 10 11 12
13 14 15 16

    1↓mat
 5  6  7  8
 9 10 11 12
13 14 15 16

    1↓⍉mat
2 6 10 14
3 7 11 15
4 8 12 16
```

```
    1↓[2]mat
 2  3  4
 6  7  8
10 11 12
14 15 16
```

Instead of dropping a fixed number of rows or columns, I apply the fuction with *Each* to `¯1+⍳17`, so it drops 0 to 16 rows or comlums. That is sufficient, because dropping more would result in diagonals with 3 or less elements, and we don't need those.

In the function itself, I append four terms: `(⍵↓⍉)(⍵↓⌽⍉)(⍵↓⊖⍉)(⍵↓⌽⊖⍉)`. The first two are the NW-SE diagonals, being the ones below the main diagonal and those above, respectively. To get the NE-SW direction, we just need to do the same with the revered matrix which is `⊖⍉`. By chaining them without *Catenate* and enclosed in parentheses, the result will be segmented, which will help when we apply the next function with *Each* in a minute:

```
    (1 2 3)(4 5 6)(7 8 9)
┌─────┬─────┬─────┐
│1 2 3│4 5 6│7 8 9│
└─────┴─────┴─────┘
```

I won't show the result of this here, as it wouldn't fit on a page in any readable way, but believe me (or try it our for yourself), that after applying *Mix* (`↑`)to the result, we have a segmented "list" of matrices with the numbers of rows or colums reduced accordingly.

Now it's finally time to apply `{1 1⌽⍵}` to *Each* of those segments to get all the diagonals extracted, and `×⌿⍤¯1` takes care of calcultaing the product for *Each* one. Finally, it's just a matter of using *Enlist* on all seperate result lists to remove any segmentation and we are done.

Problem 12 – Highly divisible triangular number

Problem 12 is another one which allows us to make use of a previous solution.

> The sequence of triangle numbers is generated by adding the natural numbers. So the 7th triangle number would be 1 + 2 + 3 + 4 + 5 + 6 + 7 = 28. The first ten terms would be:
>
> 1, 3, 6, 10, 15, 21, 28, 36, 45, 55, ...
>
> Let us list the factors of the first seven triangle numbers:
>
> 1: 1
> 3: 1,3
> 6: 1,2,3,6
> 10: 1,2,5,10
> 15: 1,3,5,15
> 21: 1,3,7,21
> 28: 1,2,4,7,14,28
>
> We can see that 28 is the first triangle number to have over five divisors.
> What is the value of the first triangle number to have over five hundred divisors?

Remember the search for the largest prime factor in problem 3? If we adjust the solution for that to not only return the largest, but all prime factors, we have an easy and fast way to calculate the number of divisors using the Divisor function. There is an easy way to find the number of divisors for small numbers by using `{+/0=(⍳⍵)|⍵}`, but that will fail due to filling up the memory when we need to investigate many large numbers like in this particular problem. Hence, my solution makes use of the adapted prime factor function and looks like this:

```
pf←{α←3⋄0=2|⊃⍵:α∇(÷∘2@1)⍵,2⋄0=α|⊃⍵:α∇(÷∘α@1)⍵,α⋄(α×α)<⊃⍵:(α+2)∇⍵⋄⍵~1}
ndivs←{{×/1++⌿⍵∘.∈∪⍵}pf ⍵}

{500>ndivs +/⍳⍵:∇⍵+1⋄+/⍳⍵}1
```

Please refer to problem 3 for the explanation of `pf`. It's basically unchanged except for the last statement, which now returns `⍵~1`. This uses **Without** (`~`) to return the complete list of prime factors without 1. It's kind of a "dirty" workaround because my function would need yet another guard to check if either the last prime factor is 2 or if we encounter the square of the last prime factor in the final iteration. In those cases the function will again divide the last factor by either 2 or `α` and `⊃⍵` becomes 1.

Two examples would be 1024 (or any power of 2 for that matter) and 121. In the first case, we can keep on dividing 1024 by 2 until we arrive at 2. At this point, the algorithm could stop, but the first guard still sees that `⊃⍵` is divisible by 2 and does another run.

In the case of 121, **⍺** will be raised by 2 until it arrives at 11, which divides 121. The result is again 11, so it will divide by **⍺** another time.

This is just a minor inconvenience, the resulting list *Without* 1 is still the correct prime factorization.

But let's quickly see how *Without* works:

```
    3 4 2 5 7 9 8~1 2 3
4 5 7 9 8
```

As expected, the result is the left argument without all matching members of the right argument. Now to get from the prime factors to the number of divisors, we can make use of the Divisor function which states that the number of divisors is equal to the product of all prime factor exponents increased by 1.

But how do we calculate the exponents? We just need to count how often a prime factor is occuring in the list delivered by **`pf`**. For example, the number 10080 has 72 divisors. Let's see its prime factorization:

```
    pf 10080
7 2 2 2 2 2 3 3 5
```

This shows that $10080 = 2^5 \cdot 3^2 \cdot 5^1 \cdot 7^1$ with the exponents 5, 2, 1 and 1. The number of divisors should be $(5+1) \cdot (2+1) \cdot (1+1) \cdot (1+1) = 6 \cdot 3 \cdot 2 \cdot 2$, and that is indeed 72.

Here is where the **`ndivs`** function comes into play. It takes the list of prime factors and uses the Divisor function to calculate the number of divisors. To get the exponents, being the count of every factor, we can again make use of *Outer Product* and also learn about the dyadic use of **`∊`**, which is **Membership**. Given a list to its left, it will return a boolean result with 1s on all positions where an element of the right argument is a member of the list:

```
    1 2 3 4 5 6 7 8 9∊3 4 5
0 0 1 1 1 0 0 0 0
```

When we combine this with *Outer Product*, we get a seperate result for every member of the right argument as a column (we might as well swap the arguments and the result will be in rows instead):

```
    1 2 3 4 5 6 7 8 9∘.∈3 4 5
0 0 0
0 0 0
1 0 0
0 1 0
0 0 1
0 0 0
0 0 0
0 0 0
0 0 0

    3 4 5∘.∈1 2 3 4 5 6 7 8 9
0 0 1 0 0 0 0 0 0
0 0 0 1 0 0 0 0 0
0 0 0 0 1 0 0 0 0
```

To apply this concept to our solution, we can take the prime factorization and use the *Unique* elements of it as the right argument:

```
    pf 10080
7 2 2 2 2 2 3 3 5

    ∪pf 10080
7 2 3 5

    {ω∘.∈∪ω}pf 10080
1 0 0 0
0 1 0 0
0 1 0 0
0 1 0 0
0 1 0 0
0 1 0 0
0 0 1 0
0 0 1 0
0 0 0 1
```

The first column shows the occurences of 7, the second column those of 2 etc. And we are nearly there! We just need to add the 1s in all columns to get the count of every prime factor.

This is easily done with **+⌿**, using *Reduce First* to work on the columns (see problem 1). The final steps are adding 1 to this and then applying **×/** to get the product:

```
    {+⌿ω∘.∈∪ω}pf 10080
1 5 2 1

    {×/1++⌿ω∘.∈∪ω}pf 10080
72
```

We now have all we need to solve this, we just need a worker function. Mine looks like this:

```
{500>ndivs +/⍳ω:∇ω+1⋄+/⍳ω}1
```

This uses the convenient fact that you can easily get the nth triangular number in APL with **+/⍳n**, because it's just the sum of the first **n** natural numbers. Or you can use *Scan* instead of *Reduce* to get the whole sequence up to the nth triangular number:

```
    +/⍳10
55

    +\⍳10
1 3 6 10 15 21 28 36 45 55
```

The function now uses this and checks with **ndivs** if the number of divisors is less than 500. As long as this evaluates to true, **ω** is raised by 1 and the function calls itself with that new input. When the number is finally found, it is returned with **+/⍳ω**.

Problem 13 – Large sum

To solve problem 13, we can again make use of file input.

> Work out the first ten digits of the sum of the following one-hundred 50-digit numbers.
>
> 37107287533902102798797998220837590246510135740250
> 46376937677490009712648124896970078050417018260538
> 74324986199524741059474233309513058123726617309629
> 91942213363574161572522430563301811072406154908250
> 23067588207539346171171980310421047513778063246676
> 89261670696623633820136378418383684178734361726757
> ...

This is easy to solve because we just need the first 10 digits and can just add the numbers as they are, despite Dyalog APL not supporting that many digits without scientific notation out of the box.

```
10↑1↓0⍕+/⍎¨⊃⎕NGET'/path/to/p13.txt'1
```

As usual, I directy pasted the list of numbers from the PE website into a text file which I named **`p13.txt`**. Then, after using **`⎕NGET`** in the usual manner, we have a segmented list of the numbers, still represented as strings. To sum them, we first need to convert *Each* to a number using *Execute*. After this, we can use **`+/`** to get the sum:

```
    ⊃⎕NGET'/path/to/p13.txt'1
┌──────────────────────────────────────────────────┬────┐
│37107287533902102798797998220837590246510135740250│463 ...
└──────────────────────────────────────────────────┴────┘

    +/⍎¨⊃⎕NGET'/path/to/p13.txt' 1
5.53737623E51
```

And to get rid of the scientific notation, we can apply *Format* with 0 as explained previously:

```
    0⍕+/⍎¨⊃⎕NGET'/path/to/p13.txt' 1
 5537376230390877_____________________________________
```

As we can see, the digits that go beyond the currently set precision are just represented by underscores, but that's not an issue. You can either copy the first 10 digits of that and call it a day, or you can use **`10↑1↓`** to extract them. We need to drop the first character, because when using **`0⍕`**, a space is inserted at the beginning.

Problem 14 – Longest Collatz sequence

Problem 14 deals with the Collatz conjecture:

> The following iterative sequence is defined for the set of positive integers:
>
> n → n/2 (n is even)
> n → 3n + 1 (n is odd)
>
> Using the rule above and starting with 13, we generate the following sequence:
>
> 13 → 40 → 20 → 10 → 5 → 16 → 8 → 4 → 2 → 1
>
> It can be seen that this sequence (starting at 13 and finishing at 1) contains 10 terms. Although it has not been proved yet (Collatz Problem), it is thought that all starting numbers finish at 1.
>
> Which starting number, under one million, produces the longest chain?
>
> NOTE: Once the chain starts the terms are allowed to go above one million.

My solution to that is nothing special and uses a cache of previously calculated lengths to speed th process up a bit:

```
l←⍬
{⍵=⌈/⍵}{n←⊃⌽⍵⋄n=1:l,←≢⍵⋄n<⊃⍵:l,←¯1+(≢⍵)+l[n]⋄0=2|n:∇⍵,n÷2⋄∇⍵,1+3×n}¨⍳1e6
```

However, I must admit that even with caching, calculating all chain lengths takes about 9 seconds on my laptop. But that's ok, we are well under the time limit of 1 minute that PE suggests for considering a solution being valid.

The first line initializes the cache as an empty list and stores that in `l` with `l←⍬`. **Zilde** (`⍬`) is just that, an empty numeric vector. Then comes the actual worker function which calculates all chain lengths for *Each* number in `⍳1e6`. I should have used 999999 instead, but then the function wouldn't fit in one line. Priorities.

In there, I first store the current last element of the sequence – which I get with *First* of the *Reverse* – in `n`. Then follows a guard which checks if `n=1`. If that is the case, the current sequence couldn't make use of the cache but is finished, and its length gets appended to `l`. To get the length of a list, we use **Tally** (`≢`) like so:

```
    ≢13 40 20 10 5 16 8 4 2 1
10

    {⍵,+/¯2↑⍵}⍣{10=≢⍺}1 2
1 2 3 5 8 13 21 34 55 89
```

You can see that I used `l,←≢⍵` to append and store the result at the same time. This is just the short version of `l←l,≢⍵` and similar to using `+=` in C:

```
      a←3
      a+←2
      a
5

      a,←6
      a
5 6
```

The next guard uses the cache list. `n<⊃⍵` checks if we arrived at a number smaller than the start value (e.g. 10 in the sequence starting with 13). In this case, we know that we already calculated the chain length starting with that number, so we can just add this knwon value (which is stored in `l[n]`) to the length of the current sequence. But we need to subtract 1, or else we would account for the current `n` twice.

If both guards evaluate to false, the sequence needs to be continued, and this happens with `0=2|n:∇⍵,n÷2⋄∇⍵,1+3×n`, thus appending `n÷2` or `1+3×n` to the current sequence – depending on the result of `2|n` – and calling the function again with that input.

After all chain lengths have been calculated, it's just a matter of using `{⍸⍵=⌈/⍵}` to find the index of the longest chain, being the starting number which produced it and also the solution for this problem.

Problem 15 – Lattice paths

Time for a breather, and problem 15 delivers!

> Starting in the top left corner of a 2×2 grid, and only being able to move to the right and down, there are exactly 6 routes to the bottom right corner.
>
> How many such routes are there through a 20×20 grid?

Fortunately, this boild down to a simple ”n choose k” problem, because given an NxN grid, there are ”2N choose N” possible paths when only being able to move down or to the right. And because Dyalog APL has a built in function for that, the solution is just…

```
0⍕20!40
```

…again using *Format* to get rid of the scientific notation.

The **Factorial/Binomial** operator (**!**) does just what its name implies: Used monadically, it returns the factorial of its input. Dyadically, it calculates ”n choose k” with **k ! n**:

```
    !9
362880

    2!10
45
```

And that already concludes this chapter.

Problem 16 – Power digit sum

Problem 16 demands a bit of creativity, because we will need to get way beyond APLs integer range:

> $2^{15} = 32768$ and the sum of its digits is $3 + 2 + 7 + 6 + 8 = 26$.
>
> What is the sum of the digits of the number 2^{1000}?

I didn't check if there is a big integer library for Dyalog APL, because I like to work with what is there by default, which will also help when you use other dialects of the language. My solution uses a list to store the digits and looks like this:

```
{+/{{0<+/n←⌊⍵÷10:(10|⍵)+1⌽n⍪⍵}⍵×2}⍣⍵⊢(301⍴0),1}1000
```

Phew, three nested functions and the *Power Operator* mixed together to form a cryptic one liner? Welcome to APL! Just kidding... it's *my* solution (not the best or only one, for sure), it works and it finishes in 15 milliseconds. That's where I stopped trying to optimize it any further. So let's take it apart step by step from the oudside.

The input to the outermost function is 1000 (or in general, the power of 2 which we want to calculate). The next function takes this as the parameter for the *Power operator*, so we know that it loops a 1000 times. The input to the function is `(301⍴0),1`, so 301 zeros and a 1 appended to that. This is the initial list, representing $2^0 = 1$, which will store our digits. We know the needed length of the list, because by using our favourite calculator or `2*1000` in APL, we can see that 2^{1000} is approximately $1 \cdot 10^{301}$, which has 302 digits. I prepended `+/` to the left of this function in order to return the sum of the digits.

Going one step further inside, we see that the innermost function gets `⍵×2` as its argument, which is the current list of digits representing 2^n multiplied by 2, being 2^{n+1}. But here we face the problem that any digit greater than 4 will result in a 2-digit number at that position in the list:

```
    0 5 1 2×2
0 10 2 4
```

To convert the list to single digits, we need to take one more step, and that's the task of the innermost function. It first checks if the list contains numbers greater than 9 with `0<+/n←⌊⍵÷10`. So we divide the list by 10 and take the floor of that. For any number up to 9, this will result in 0, so using `+/` to sum up all results and checking if that is

greater than 0 is the indicator we need. I also store the result of the floored division in **n** because I need it again in the next expression.

If we don't have any digis greater than 9, the list is just returned unchanged by the last statement **⍵**. But if we do, **∇(10|⍵)+1⌽n⋄⍵** gets executed, which uses the sum of **10|⍵** and **1⌽n** to shift the "greater than 10"-part one digit to the left and keeing only the remainder of "divided by 10" at the current position.

This uses **Rotate**, the dyadic form of *Reverse* to shift the digits. Given a positive number N to its left, *Rotate* will – as the name suggests – rotate the contents of its input by N steps to the left. A negative N will rotate to the right:

```
    1⌽1 2 3 4 5
2 3 4 5 1

    ¯1⌽1 2 3 4 5
5 1 2 3 4
```

As you can see, it is really a rotation where the elements that "fall over" get appended to the other end. This isn't a problem in our case as there are only zeros which get rotated to the other end until the function has finished working, and zeros have no influence on the sum which is used to build the "corrected" list. Now let's see if this works as intended using our little example from above:

```
    {0<+/n←⌊⍵÷10:(10|⍵)+1⌽n⋄⍵}0 5 1 2
0 5 1 2

    {0<+/n←⌊⍵÷10:(10|⍵)+1⌽n⋄⍵}0 5 1 2×2
1 0 2 4
```

It does! And that's all there is to it. We just repeat that a thousand times to get to 2^{1000} and there is our result (short of summing the digits):

```
    {{{0<+/n←⌊⍵÷10:(10|⍵)+1⌽n⋄⍵}⍵×2}⍣⍵⊢(301⍴0),1}1000
1 0 7 1 5 0 8 6 0 7 1 8 6 2 6 7 3 2 0 9 4 8 4 2 5 0 4 9 0 6 0 0 0 1 8 1
0 5 6 1 4 0 4 8 1 1 7 0 5 5 3 3 6 0 7 4 4 3 7 5 0 3 8 8 3 7 0 3 5 1 0 5
1 1 2 4 9 3 6 1 2 2 4 9 3 1 9 8 3 7 8 8 1 5 6 9 5 8 5 8 1 2 7 5 9 4 6 7
2 9 1 7 5 5 3 1 4 6 8 2 5 1 8 7 1 4 5 2 8 5 6 9 2 3 1 4 0 4 3 5 9 8 4 5
7 7 5 7 4 6 9 8 5 7 4 8 0 3 9 3 4 5 6 7 7 7 4 8 2 4 2 3 0 9 8 5 4 2 1 0
7 4 6 0 5 0 6 2 3 7 1 1 4 1 8 7 7 9 5 4 1 8 2 1 5 3 0 4 6 4 7 4 9 8 3 5
8 1 9 4 1 2 6 7 3 9 8 7 6 7 5 5 9 1 6 5 5 4 3 9 4 6 0 7 7 0 6 2 9 1 4 5
7 1 1 9 6 4 7 7 6 8 6 5 4 2 1 6 7 6 6 0 4 2 9 8 3 1 6 5 2 6 2 4 3 8 6 8
3 7 2 0 5 6 6 8 0 6 9 3 7 6
```

Bonus: If you want to make this a bit more universal, you can work with a dynamic list length, calculated with `⌊10⊗2*⍵`. This uses the floor of the base-10 logarithm of 2^n to get the number of digits minus 1 (because we still need to append 1 to this). Adjusting the function accordingly works fine:

```
    {{{0<+/n←⌊⍵÷10:(10|⍵)+1⌽n⋄⍵}⍵×2}⍣⍵⊢1,⍤(⌊10⊗2*⍵)⍴0}10
1 0 2 4

    {{{0<+/n←⌊⍵÷10:(10|⍵)+1⌽n⋄⍵}⍵×2}⍣⍵⊢1,⍤(⌊10⊗2*⍵)⍴0}20
1 0 4 8 5 7 6

    {{{0<+/n←⌊⍵÷10:(10|⍵)+1⌽n⋄⍵}⍵×2}⍣⍵⊢1,⍤(⌊10⊗2*⍵)⍴0}50
1 1 2 5 8 9 9 9 0 6 8 4 2 6 2 4
```

And with a few further adjustments, we can use it not just for powers of 2 but for any base number and exponent:

```
tothepowerof←{⍺{{0<+/n←⌊⍵÷10:∇(10|⍵)+1⌽n⋄⍵}⍵×⍺}⍣⍵⊢1,⍤(⌊10⊗⍺*⍵)⍴0}

    25 tothepowerof 10
9 5 3 6 7 4 3 1 6 4 0 6 2 5
```

Problem 17 – Number letter counts

Problem 17 is a different beast. It asks us to calculate the number of letters needed to write out all numbers from 1 to 1000 in words:

> If the numbers 1 to 5 are written out in words: one, two, three, four, five, then there are 3 + 3 + 5 + 4 + 4 = 19 letters used in total.
>
> If all the numbers from 1 to 1000 (one thousand) inclusive were written out in words, how many letters would be used?
>
> NOTE: Do not count spaces or hyphens. For example, 342 (three hundred and forty-two) contains 23 letters and 115 (one hundred and fifteen) contains 20 letters. The use of "and" when writing out numbers is in compliance with British usage.

I use a solution similar to what others have done and it looks like this:

```
ones←3 3 5 4 4 3 5 5 4
teens←3 6 6 8 8 7 7 9 8 8
tens←6 6 5 5 5 7 6 6
+/11,(ones+7),(∈(ones+10)∘.,n),n←ones,teens,tens,∈tens∘.+ones
```

In the first step, three lists are created containing the letter counts of the
`ones` ("one", "two", "three"... "nine")
`teens` ("ten", "eleven", "twelve"... "nineteen") and
`tens` ("twenty", "thirty", "forty" ... "ninety").

Then it's just a matter of combining them in the right amounts. For all numbers below 100, we need one of each, as well as all combinations of **`tens`** and **`ones`** which we can easily get with *Outer Product* using **`∈tens∘.+ones`**. This is also needed for all "hundreds", so I store that in **`n`**, and combine it with *Outer Product* to **`ones+10`**, which takes care of all "X hundred and".

For the even hundreds ("one hundred", "two hundred"...) we also need **`ones+7`**, and finally we have to account for "one thusand" by adding 11. All that's left to do is **`+/`**ing it all together.

Problem 18 – Maximum path sum I

Problem 18is a classic, and although it can be solved using brute force, there is a well known algorithm that speeds it up considerably and is absolutely needed for the similar (but much larger) problem 67.

> By starting at the top of the triangle below and moving to adjacent numbers on the row below, the maximum total from top to bottom is 23.
>
> ```
>    3
>   7 4
>  2 4 6
> 8 5 9 3
> ```
>
> That is, 3 + 7 + 4 + 9 = 23.
>
> Find the maximum total from top to bottom of the triangle below:
>
> ```
>                              75
>                            95  64
>                          17  47  82
>                        18  35  87  10
>                      20  04  82  47  65
>                    19  01  23  75  03  34
>                  88  02  77  73  07  63  67
>                99  65  04  28  06  16  70  92
>              41  41  26  56  83  40  80  70  33
>            41  48  72  33  47  32  37  16  94  29
>          53  71  44  65  25  43  91  52  97  51  14
>        70  11  33  28  77  73  17  78  39  68  17  57
>      91  71  52  38  17  14  91  43  58  50  27  29  48
>    63  66  04  68  89  53  67  30  73  16  69  87  40  31
> 04  62  98  27  23  09  70  98  73  93  38  53  60  04  23
> ```
>
> NOTE: As there are only 16384 routes, it is possible to solve this problem by trying every route. However, Problem 67, is the same challenge with a triangle containing one-hundred rows; it cannot be solved by brute force, and requires a clever method! ;o)

This problem is the perfect one to introduce the option of applying functions with **`Reduce/Fold`**. We already did that a thousand times with **`+/`**, but you can also use it perfectly fine with your own functions. My following solution makes use of that:

```
tri←⍎¨⊃⎕NGET'/path/to/p18.txt'1

{⍺+2⌈/⍵}/tri
```

After having pasted the triangle in **`p18.txt`**, I use **`⎕NGET`** to get a segmented list of

strings, and then *Execute* with *Each* to convert those to numbers, which results in this:

```
    tri
┌──┬─────┬────────┬───────────┬────────────┬───────────────┬───────────
│75│95 64│17 47 82│18 35 87 10│20 4 82 47 65│19 1 23 75 3 34│88 2 77 ...
└──┴─────┴────────┴───────────┴────────────┴───────────────┴───────────
```

And now *Reduce* makes it possible to use a tiny function to solve this problem (and problem 67 as well), but how does it work? Let's take the smaller triangle from the example, which makes it easier to show the process:

```
    tri←(3)(7 4)(2 4 6)(8 5 9 3)
    tri
┌─┬───┬─────┬───────┐
│3│7 4│2 4 6│8 5 9 3│
└─┴───┴─────┴───────┘
```

As you might know, the key to solve this efficiently is to go from the bottom up, calculating the pairwise maxima of the last row and adding this result to the second-to-last row. This then becomes the new last row and the process repeats until we arrive at the top, where the final sum is also the maximum path sum.

I already showed in an eralier chapter that we can use **2⌈/** to get the pairwise maxima of a list:

```
    2⌈/8 5 9 3
8 9 9
```

And the "secret" here is that by using *Reduce*, which inserts the function between all segments, **ω** will be the bottom row and **α** the one before it. There is a nice way to illustrate that by just applying the function **{αω}** with *Reduce*:

```
      {αω}/tri
┌──────────────────────────┐
│┌─┬──────────────────────┐│
││3│┌───┬───────────────┐││
││ ││7 4│┌─────┬───────┐│││
││ ││   ││2 4 6│8 5 9 3││││
││ ││   │└─────┴───────┘│││
││ │└───┴───────────────┘││
│└─┴──────────────────────┘│
└──────────────────────────┘
```

This shows through boxing that in the first step, the function uses `2 4 6` as `⍺` and `8 5 9 3` as `⍵`. In the next iteration, the restult of this is the new `⍵`, while `⍺` is now the next row before the former, being `7 4`.

It also shows another important aspect: Dyalog APL does a ”right fold”, meaning that the iteration begins at the end of the list. So if you plan to apply a function using *Reduce*, make sure to have your list ordered in a way that the first elements to work on are the last ones in the list.

And that's just what we needed. An algorithm that works it's way from bottom to top (or in this case from right to left) through the rows of the triangle:

```
    {⍺+2⌈/⍵}/tri
┌──┐
│23│
└──┘
```

If the leftover boxing is an issue, just prepend *Enlist* to the function, which will take care of that.

Problem 19 – Counting Sundays

To solve problem 19, we need to find out how may Sundays fell on the first of the month between 01/01/1901 and 12/31/2000.

> You are given the following information, but you may prefer to do some research for yourself.
>
> - 1 Jan 1900 was a Monday.
> - Thirty days has September,
>   April, June and November.
>   All the rest have thirty-one,
>   Saving February alone,
>   Which has twenty-eight, rain or shine.
>   And on leap years, twenty-nine.
> - A leap year occurs on any year evenly divisible by 4, but not on a century unless it is divisible by 400.
>
> How many Sundays fell on the first of the month during the twentieth century (1 Jan 1901 to 31 Dec 2000)?

The last clue doesn't apply for this time range, as the only century is 2000 which *is* divisible by 400, so we can just use the single rule that every year divisible by 4 is a leap year. My solution uses that, and the additional information that 12/31/1900 was a Monday:

```
m←31 28 31 30 31 30 31 31 30 31 30 31

{+/7=⊃¨(⍵/⍳≠⍵)⊆(+/⍵)⍴1⌽⍳7}∊{0=4|⍵:(29@2)m⋄m}¨1900+⍳100
```

The solution builds a calendar for the whole century and only at the end counts the number of Sundays which fell on the first of the month. It's actually easier to do it this way than to write a function which only calculates the days on the first of the month.

But the first step is to make a list of the lentghs of the months in a normal year and store that in a variable. The rightmost function then gets applied to *Each* year and returns either the list unchanged or uses *At* to replace the length of Febuary with `29` if it is a leap year (which we check with `0=4|⍵`). After this is done for the whole century, we can remove the segmentation with *Enlist*.

The second function does the actual work of creating the calendar. Starting from the right, I first create a continous list of the day numbers (using 1 for Monday, 2 for Tuesday etc.) for the whole century. Because 01/01/1901 was a Tuesday, the list needs to start with 2 and we can achieve that with *Rotate* using `1⌽⍳7`. This represents the first week

of 1901, and to extend that over the whole century, we can just use *Shape* to fill the list continously. The length of the list (being the number of days in the whole century) is **+/⍵**. To check if everything worked correctly, we can examine the last element, representing 31/12/2000, which was a Sunday:

```
    {(+/⍵)⍴1⍳7}∊{0=4|⍵:(29@2)m⋄m}¨1900+⍳100
2 3 4 5 6 7 1 2 3 4 5 6 7 1 2 3 4 5 6 7 1 2 3 4 5 6 7 ...

    ¯1↑{(+/⍵)⍴1⍳7}∊{0=4|⍵:(29@2)m⋄m}¨1900+⍳100
7
```

That seems to be working! But now we need to partition this, in order to identify the seperate months. And **Partition** (⊆) is indeed the tool we can make use of. You pass the list to be partitioned as the right argument, and a ”partitioning rule” as the left argument. The lenghths of both must match. You can then use ascending numbers to specify the partitions, using as many terms of the same number as you want the corresponding partition to have members. Clear as mud, isn't it? Let's see some examples:

```
    1 1 1 2 2 2⊆1 2 3 4 5 6
┌─────┬─────┐
│1 2 3│4 5 6│
└─────┴─────┘

    1 1 2 2 3 3⊆1 2 3 4 5 6
┌───┬───┬───┐
│1 2│3 4│5 6│
└───┴───┴───┘

    1 2 2 3 3 3⊆1 2 3 4 5 6
┌─┬───┬─────┐
│1│2 3│4 5 6│
└─┴───┴─────┘
```

You don't need to use 1, 2, 3 etc. or even adjacent numbers, as long as they are ascending, but doing so makes it a bit easier to grasp. Here is an example using arbitrary numbers.

```
    24 24 78 95 95 1235⊆1 2 3 4 5 6
┌───┬─┬─┬───┬─┐
│1 2│3│4 5│6│
└───┴─┴─┴───┴─┘
```

You can also use zeros to leave out the corresponding elements:

```
     1 1 0 0 2 2⊆1 2 3 4 5 6
┌─────┬─────┐
│1 2 │5 6 │
└─────┴─────┘
```

Now back to our case. I use **(⍵/⍳≢⍵)** as the "partitioning rule". This uses *Replicate* to create **⍵** copies of **⍳≢⍵**. The latter being a list of natural numbers from 1 to 1200, because we have 1200 months in the century and **⍵** is the list of all month lengths. *Replicate* then uses **⍵** to make as much copies of each number as there are days in each month. So we start with 31 ones, then 28 twos, 31 threes and so on:

```
     {⍵/⍳≢⍵}∊{0=4|⍵:(29@2)m⋄m}¨1900+⍳100
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
3 3 3 3 3 3 3 3 3 4 ...
```

And this is just what we need to split the continous list of days back into partitions of month lengths:

```
     {(⍵/⍳≢⍵)⊆(+/⍵)ρ1⌽⍳7}∊{0=4|⍵:(29@2)m⋄m}¨1900+⍳100
┌──────────────────────────────────────────────────────────────────────────────┬─────┐
│2 3 4 5 6 7 1 2 3 4 5 6 7 1 2 3 4 5 6 7 1 2 3 4 5 6 7 1 2 3 4 5 6 7 1 2 3 4│5 6 7 ...
└──────────────────────────────────────────────────────────────────────────────┴─────┘
```

The last step is then to apply **+/7=⊃** with *Each* to this, checking if the first element of each month is a 7, and finally summing everything up.

```
     {7=⊃¨(⍵/⍳≢⍵)⊆(+/⍵)ρ1⌽⍳7}∊{0=4|⍵:(29@2)m⋄m}¨1900+⍳100
0 0 0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 ...
```

I know that *Partition* isn't the easiest operator to understand, but you'll quickly get the hang of it, for sure.

Problem 20 – Factorial digit sum

Similar to problem 16, I used the list representation of big numbers in problem 20, which deals with a large factorial.

> n! means n × (n − 1) × ... × 3 × 2 × 1
>
> For example, 10! = 10 × 9 × ... × 3 × 2 × 1 = 3628800, and the sum of the digits in the number 10! is 3 + 6 + 2 + 8 + 8 + 0 + 0 = 27.
>
> Find the sum of the digits in the number 100!

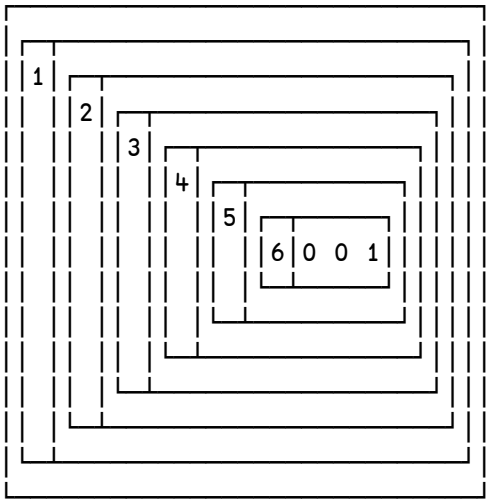
As 100! has 158 digits, it's far too big to be handled without scientific notation in Dyalog APL out of the box. But we did most of the work already for problem 16. Also, I use *Reduce* to create the factorial loop similar to problem 18. And here is my solution, combining both approaches:

```
{+/∊{{0<+/n←⌊ω÷10:∇(10|ω)+1⌽n◇ω}ω×α}/(⍳ω),⊂((⌊10⍟!ω)⍴0),1}100
```

The argument for the outermost function is the factorial which we want to calculate, so 100 in this case. Then, similar to problem 16, a list of the needed length is created which consists of zeros and a single 1 at the last position. The needed length for shape is calculated with **⌊10⍟!ω**, using the floor of the base-10 logarithm of **!ω** to get the number of digits of minus 1.

This list is then boxed using **Enclose** (**⊂**), and will be the first instance of **ω** for *Reduce* (please refer to problem 18 for the details on how this works). Our list of arguments for **α** is just **⍳ω**. And a function applied to this with *Reduce* will work in this fashion:

```
{{α ω}/(⍳ω),⊂((⌊10⍟!ω)⍴0),1}6
```



That's just what we need: Multiplying the list with all numbers decreasing from the upper limit to 1. The rest of the function works exactly as explained in problem 16, so please refer to that chapter for details. The only difference here is that I need to apply *Enlist* to the result in order to get rid of the boxing. I'll just show the result before we apply **+/** for the sum:

```
    {∈{{0<+/n←⌊ω÷10:∇(10|ω)+1⌽n⋄ω}ω×α}/(⍳ω),⊂((⌊10⍟!ω)⍴0),1}100
9 3 3 2 6 2 1 5 4 4 3 9 4 4 1 5 2 6 8 1 6 9 9 2 3 8 8 5 6 2 6 6 7
0 0 4 9 0 7 1 5 9 6 8 2 6 4 3 8 1 6 2 1 4 6 8 5 9 2 9 6 3 8 9 5 2
1 7 5 9 9 9 9 3 2 2 9 9 1 5 6 0 8 9 4 1 4 6 3 9 7 6 1 5 6 5 1 8 2
8 6 2 5 3 6 9 7 9 2 0 8 2 7 2 2 3 7 5 8 2 5 1 1 8 5 2 1 0 9 1 6 8
6 4 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

And to show that the "dynamic" calculation of the list length works fine, here is the result of the same function calculating 10!

```
    {∈{{0<+/n←⌊ω÷10:∇(10|ω)+1⌽n⋄ω}ω×α}/(⍳ω),⊂((⌊10⍟!ω)⍴0),1}10
3 6 2 8 8 0 0
```

So while it may be a bit more work initially, it always pays off to design a solution as universal as possible. You never know when you'll need it again for a similar task.

Problem 21 – Amicable numbers

Problem 21 wants us to find the sum of all amicable numbers below 10000:

> Let d(n) be defined as the sum of proper divisors of n (numbers less than n which divide evenly into n).
>
> If d(a) = b and d(b) = a, where a ≠ b, then a and b are an amicable pair and each of a and b are called amicable numbers.
>
> For example, the proper divisors of 220 are 1, 2, 4, 5, 10, 11, 20, 22, 44, 55 and 110; therefore d(220) = 284. The proper divisors of 284 are 1, 2, 4, 71 and 142; so d(284) = 220.
>
> Evaluate the sum of all the amicable numbers under 10000.

My solution is pretty straightforward and looks like this:

```
divsum←{+/¯1↓⍳0=(⍳⍵)|⍵}

+/⍳{(⍵=divsum ds)∧⍵≠ds←divsum ⍵}¨⍳9999
```

Because we only need to deal with small numbers, I didn't use prime factorization to get the sum of the divisors. It's easier to do with **⍳0=(⍳⍵)|⍵**, but we also need to drop the last element of the result because we are only interested in the proper divisors. **divsum** then works as intended:

```
    {¯1↓⍳0=(⍳⍵)|⍵}220
1 2 4 5 10 11 20 22 44 55 110

    {+/¯1↓⍳0=(⍳⍵)|⍵}220
284

    {+/¯1↓⍳0=(⍳⍵)|⍵}284
220
```

The worker function then needs to identify all numbers, for which applying **divsum** twice cycles back to the number (e.g. the divsum of the divsum of 220 is 220). But we also need to filter out all perfect numbers, which are equal to the sum of their proper divisors. To combine both conditions, I use logical **And** (∧), and I also store the result of **divsum ⍵** in **ds** because we need that twice and it speeds the function up a bit (and saves some memory) if we only calculate it once. We can use **Not Equal** (≠) for the task of discarding perfect numbers.

And here is the result short of using **+/** to get the sum:

```
    ⍸{(⍵=divsum ds)∧⍵≠ds←divsum ⍵}¨⍳9999
220 284 1184 1210 2620 2924 5020 5564 6232 6368
```

Problem 22 – Names scores

We can use problem 22 to learn about two new system functions and how to sort lists.

> Using names.txt (right click and 'Save Link/Target As...'), a 46K text file containing over five-thousand first names, begin by sorting it into alphabetical order. Then working out the alphabetical value for each name, multiply this value by its alphabetical position in the list to obtain a name score.
>
> For example, when the list is sorted into alphabetical order, COLIN, which is worth 3 + 15 + 12 + 9 + 14 = 53, is the 938th name in the list. So, COLIN would obtain a score of 938 × 53 = 49714.
>
> What is the total of all the name scores in the file?

```
{+/⍵×⍳≠⍵}{+/¯64+⎕UCS ⍵}¨{⍵[⍋⍵]}1⊃⎕CSV'/path/to/names.txt'
```

As the names in the file are comma separated values (CSV), we can use the system function of the same name to read the contents. Compared to using **⎕NGET**, this has the advantage that the input is partitioned automatically. Also, the double quotes which enclose all names are removed by default. We would need to take care of both things manually if we use **⎕NGET**. Let's look at the output of **⎕CSV**:

```
    ⎕CSV'/path/to/names.txt'
┌─────┬───────┬─────┬───────┬─────────┬────────┬─────┬─────┬──
│MARY│PATRICIA│LINDA│BARBARA│ELIZABETH│JENNIFER│MARIA│SUSAN│MA ...
└─────┴───────┴─────┴───────┴─────────┴────────┴─────┴─────┴──
```

That's just what we want! The next step would be to sort the names. But we first need to take one extra step, and that is to extract the first row using **Index** (**⊃**).

Why that? While it looks just like an ordinary segmented list, it's actually an array with one row. You can see this when you apply *Shape* monadically:

```
    ⍴⎕CSV'/path/to/names.txt'
1 5163
```

This teels us that we have an array with one row and 5163 columns. Given a simple list, *Shape* would just return it's length:

```
    ⍴⍳10
10
```

But why is that important? It is, because applying functions to this can result in a *Rank Error*. Consider this example:

```
    a←1 2 3 4
    b←5 6 7 8
    c←1 4⍴5 6 7 8

    a
1 2 3 4

    b
5 6 7 8

    c
5 6 7 8

    a+b
6 8 10 12

    a+c
RANK ERROR: Mismatched left and right argument ranks
    a+c
     ^
```

While **b** and **c** look exactly the same, **b** is a list and **c** is and array. You can get around this by explicitly specifying the rank which you want to work on, but in our case it's much easier if we just extract the first (and only) row of our "array", and we can use *Index* for that.

Given a single number as its left argument, *Index* will return the corresponding row of an array. You can also give it list (row, col) to specify the index of a single element:

```
    □←mat←3 3⍴⍳9
1 2 3
4 5 6
7 8 9

    2⌷mat
4 5 6

    3 2⌷mat
8
```

In our case, we can use `1⌷` to get the first (and only) row, which now is a simple list looking exactly the same as before. But if we apply *Shape* again, we see the difference:

```
    ⍴1⌷⎕CSV'/path/to/names.txt'
5163
```

Now it's time to actually work on our list of names. The first step is to sort them into alphabetical order. Sorting can be done with **Grade Up** (`⍋`) and **Grade Down** (`⍒`). If you apply one of those to a list, it will return the indices which would sort the list into ascending or descending order:

```
    ⍋5 7 9 6 1 4 3 8 2
5 9 7 6 1 4 2 8 3

    ⍒5 7 9 6 1 4 3 8 2
3 8 2 4 1 6 7 9 5
```

So in the first case, an ascending order would be achieved by the element at index 5 (which is 1), then the element at index 9 (which is 2) and so on. To actually get our sorted list, we can make use of the `[⍵]` notation:

```
    {⍵[⍋⍵]}5 7 9 6 1 4 3 8 2
1 2 3 4 5 6 7 8 9

    {⍵[⍒⍵]}5 7 9 6 1 4 3 8 2
9 8 7 6 5 4 3 2 1
```

We can now apply the same function to our unsorted list of names, because *Grade Up/Down* work equally well with strings:

```
    {⍵[⍒⍵]}1⌷⎕CSV'/path/to/names.txt'
┌─────┬─────┬─────┬─────┬──────┬───┬────┬──────┬───────┬─────┐
│AARON│ABBEY│ABBIE│ABBY │ABDUL │ABE│ABEL│ABIGAIL│ABRAHAM│ABRAM│ ...
└─────┴─────┴─────┴─────┴──────┴───┴────┴──────┴───────┴─────┘
```

Now it's time to calculate what the problem calls the "alphabetical value" for each name. In order to do this, we need to translate 'A' to 1, 'B' to 2 and so on. Fortunately, Dyalog APL has the system function **Unicode Convert** (`⎕UCS`) which takes care of converting characters to integer numbers (and vice versa).

The result will be the decimal value of the character in the ASCII-Table. To get the actual position in the alphabet, we need to subtract 64 from that (Because 'A' has the decimal value of 65). Let's use that on the name "COLIN" from the problem's example:

```
    ⎕UCS'COLIN'
67 79 76 73 78

    ¯64+⎕UCS'COLIN'
3 15 12 9 14

    +/¯64+⎕UCS'COLIN'
53
```

And that already explains the second function. We just apply the last step of this example as a function to *Each* name, and the result is a list of the name values:

```
    {+/¯64+⎕UCS ⍵}¨{⍵[⍋⍵]}1⊃⎕CSV'/path/to/names.txt'
49 35 19 30 40 8 20 41 44 35 6 14 78 46 19 20 23 23 37 41 27 32 46 50 ...
```

Just one more thing to do! To get the name scores, we need to multiply each of the values with their position in the list. This is easily done by multiplying this result with a list of 1 to 5163, because there are 5163 names in the list. Remember that applying an operator to lists of the same length results in pairwise operation? That's just what we need! The first value will be mutliplied by 1, the second value by 2 and so on.

We don't need to explicitly give 5163 as a parameter to `⍳` because we can just use *Tally* to get the number of values, and use `⍵×⍳≠⍵` to calculate the name scores:

```
    {⍵×⍳≠⍵}{+/¯64+⎕UCS ⍵}¨{⍵[⍋⍵]}1⊃⎕CSV'/path/to/names.txt'
49 70 57 120 200 48 140 328 396 350 66 168 1014 644 285 320 ...
```

And finally, we just have to use `+/` again to get the total. That maybe wasn't the typical PE problem and had not much to to with number theory, but we learned a few new things nonetheless.

Problem 23 – Non-abundant sums

Problem 23 deals with abundant numbers:

> A perfect number is a number for which the sum of its proper divisors is exactly equal to the number. For example, the sum of the proper divisors of 28 would be 1 + 2 + 4 + 7 + 14 = 28, which means that 28 is a perfect number.
>
> A number n is called deficient if the sum of its proper divisors is less than n and it is called abundant if this sum exceeds n.
>
> As 12 is the smallest abundant number, 1 + 2 + 3 + 4 + 6 = 16, the smallest number that can be written as the sum of two abundant numbers is 24. By mathematical analysis, it can be shown that all integers greater than 28123 can be written as the sum of two abundant numbers. However, this upper limit cannot be reduced any further by analysis even though it is known that the greatest number that cannot be expressed as the sum of two abundant numbers is less than this limit.
>
> Find the sum of all the positive integers which cannot be written as the sum of two abundant numbers.

That looks like a case for *Outer Product* again, and indeed my solution uses that:

```
{+/⍳⍵∘.+⍨⍵}∈{⍵<+/¯1↓⍸0=(⍳⍵)|⍵:⍵◇⍬}¨⍳←⍳28123
```

I first made a list of all numbers up to 28123 and stored that in `⍳`. Then I apply this function to *Each* number:

```
{⍵<+/¯1↓⍸0=(⍳⍵)|⍵:⍵◇⍬}
```

If you remember problem 21, you'll see that I use the same function to get the sum of the proper divisors as I did there. I just added a guard which checks if that sum is greater than `⍵` (thus `⍵` being abundant). If that is the case, the function returns `⍵`, or else an empty vector. The result is a partitioned list of abundant numbers and empty vectors, which we can convert to a regular list using *Enlist*:

```
    {⍵<+/¯1↓⍸0=(⍳⍵)|⍵:⍵◇⍬}¨⍳←⍳28123
┌┬┬┬┬┬┬┬┬┬┬┬┬─┬┬┬┬┬┬─┬┬─┬┬┬┬─┬┬┬┬┬┬─┬┬┬┬┬┬─┬┬┬┬┬┬─┬┬┬┬─┐
│││││││││││12│││││││18│⋄│20│⋄⋄│24│⋄⋄⋄⋄⋄⋄│30│⋄⋄⋄⋄⋄⋄│36│⋄⋄⋄⋄│40│ ...
└┴┴┴┴┴┴┴┴┴┴┴┴─┴┴┴┴┴┴─┴┴─┴┴┴┴─┴┴┴┴┴┴─┴┴┴┴┴┴─┴┴┴┴┴┴─┴┴┴┴─┘

    ∈{⍵<+/¯1↓⍸0=(⍳⍵)|⍵:⍵◇⍬}¨⍳←⍳28123
12 18 20 24 30 36 40 42 48 54 56 60 66 70 72 78 80 84 88 ...
```

There we have our list of abundant numbers. Now my approach was to get the sums of all combinations of this using *Outer Product* and then apply this using *Without* to **⍳**, which will remove all numbers from **⍳** that can be written as the sum of two abundant numbers:

```
    {⍳∼∘.+⍨⍵}∈{⍵<+/¯1↓⍸0=(⍳⍵)|⍵:⍵⋄⍬}¨⍳←⍳28123
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 25 26 ...
```

And finally – what would we do without it? – **+/** get's our answer.

Problem 24 – Lexicographic permutations

To solve problem 24, we need to find the millionth lexicographic permutation of `0 1 2 3 4 5 6 7 8 9`.

> A permutation is an ordered arrangement of objects. For example, 3124 is one possible permutation of the digits 1, 2, 3 and 4. If all of the permutations are listed numerically or alphabetically, we call it lexicographic order. The lexicographic permutations of 0, 1 and 2 are:
>
> 012 021 102 120 201 210
>
> What is the millionth lexicographic permutation of the digits 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9?

Now if you want to take the easy way out, you could google your way to Dyalog's dfns page about permutations, and find a very fast permutation function, made by array language veteran Roger Hui, called `pmat2`. Using this, we can get the answer in a matter of milliseconds with `¯1+1e6⌷pmat2 10`.

But if you are like me and like to find a solution using only what you know so far, feel free to follow along. My solution is based on the method I found here:

> Step 1. Take the previously printed permutation and find the rightmost character in it, which is smaller than its next character. Let us call this character as 'first character'.
>
> Step 2. Now find the ceiling of the 'first character'. Ceiling is the smallest character on right of 'first character', which is greater than 'first character'. Let us call the ceil character as 'second character'.
>
> Step 3. Swap the two characters found in above 2 steps.
>
> Step 4. Sort the substring (in non-decreasing order) after the original index of 'first character'.

My solution below does just that, and it's awfully slow compared to pmat2, taking 10 seconds to finish on my laptop. But I wrote it myself, it works, and I understand what it does – which I can't say about pmat2...

```
iab←{a←⌈/⍳2</⍵⋄a,a+⌈/⍳⍵[a]<a↓⍵}
sort←{((⊃⍺)↑⍵),⍋(⊃⍺)↓⍵}

{i←iab ⍵⋄i sort(⍋⌷⍵)⍵}⍣999999⊢¯1+⍳10
```

But let's see what's going on here. The first function `iab` returns the indices of the numbers from step 1 and step 2 as a list of two elements:

```
    iab 0 1 2 3 4 5 6 9 8 7
7 10
```

a is found with **⌈/⍳2</⍵**. If we break that down from right to left, the first action is applying *Less Than* pairwise to the list, to identify all numbers which are smaller than their successor:

```
    2</ 0 1 2 3 4 5 6 9 8 7
1 1 1 1 1 1 1 0 0
```

We get a boolean list of the results, which shows that in this example, all numbers except 9 and 8 are greater than their successor. We are only interested in the first number from the right for which the result is true. We can get that by using Where to get the indices of the 1s and taking the maximum of that:

```
    ⍳2</ 0 1 2 3 4 5 6 9 8 7
1 2 3 4 5 6 7

    ⌈/⍳2</ 0 1 2 3 4 5 6 9 8 7
7
```

The result is stored in **a**. **b** is found with **a+⌈/⍳⍵[a]<a↓⍵**. Again coming in from the right, I first use **a↓⍵** to drop all elements up to (and including) **a** from the list. Then I use **⍳⍵[a]<** to get the indices of all numbers in this sublist which are greater than **⍵[a]** (all in this case):

```
   {a←⌈/⍳2</⍵⋄a↓⍵}0 1 2 3 4 5 6 9 8 7
9 8 7

    {a←⌈/⍳2</⍵⋄⍳⍵[a]<a↓⍵}0 1 2 3 4 5 6 9 8 7
1 2 3
```

We can again use **⌈/** to find the rightmost index. If we then add **a** to that, we have our **b**. And appending this to **a** finishes the function:

```
    {a←⌈/⍳2</⍵⋄⌈/⍳⍵[a]<a↓⍵}0 1 2 3 4 5 6 9 8 7
3

    {a←⌈/⍳2</⍵⋄a+⌈/⍳⍵[a]<a↓⍵}0 1 2 3 4 5 6 9 8 7
10

    {a←⌈/⍳2</⍵⋄a,a+⌈/⍳⍵[a]<a↓⍵}0 1 2 3 4 5 6 9 8 7
7 10
```

Next comes the **sort** function which is responsible for step 4. But as it happens (and if you follow this method) the sublist behind **a** and after applying step 3 is always in descending order, so it's sufficient to just rotate it. Let's have a look at the function again:

```
sort←{((⊃⍺)↑⍵),⌽(⊃⍺)↓⍵}
```

It uses the result of **iab** as it's left argument and the list of numbers as it's right input. Then it takes the first element of **iab** and uses *Take* to get the sublist up to **a**. To this it appends the reversed sulbist after **a** unsing the same index with Drop.

Now to the permutation function:

```
{i←iab ⍵⌽i sort(⌽@i)⍵}⍣999999⊢¯1+⍳10
```

It uses the *Power Operator* to run the permutation 999999 times, because the first permutation is already the initial input. The first step is to use **iab** to get the current values for **a** and **b**, which get stored in **i**. This is then the left argument for sort as well as the right argument for *Reverese At* (⌽@), which just results in **⍵[a]** and **⍵[b]** swapping paces like so:

```
    (⌽@(3 5))8 7 6 1 2
8 7 2 1 6
```

And that is all that's needed to solve the problem. It's just the method described at the beginning translated to APL, but apparently somewhat inefficient. The same method in C returns the result almost instantly, so maybe it's just a case of not thinking APL-like enough yet…

Problem 25 – 1000-digit Fibonacci number

Problem 25 offers another opportunity to make us of a previous solution. We already worked on two problems, where list representations of big integers were used, and we can again adapt the principles used there to solve this problem.

> The Fibonacci sequence is defined by the recurrence relation:
>
> $F_n = F_{n-1} + F_{n-2}$, where $F_1 = 1$ and $F_2 = 1$.
>
> Hence the first 12 terms will be:
>
> $F_1 = 1$
> $F_2 = 1$
> $F_3 = 2$
> $F_4 = 3$
> $F_5 = 5$
> $F_6 = 8$
> $F_7 = 13$
> $F_8 = 21$
> $F_9 = 34$
> $F_{10} = 55$
> $F_{11} = 89$
> $F_{12} = 144$
>
> The 12th term, F_{12}, is the first term to contain three digits.
>
> What is the index of the first term in the Fibonacci sequence to contain 1000 digits?

```
sum←{{0<+/n←⌊ω÷10:∇(10|ω)+1⌽n⋄⊂ω}∈ω[1]+ω[2]}

{{α←1⋄0=⊃∈ω[1]:(α+1)∇ω[2],(sum ω)⋄α}⍣(2(ω-1)⍴0),1 1}1000
```

I know that there are ways to get the result using the golden ratio and a simle formula, but why not actually calculate the terms of the sequence, when we can? So let's do it.

The inner block of the `sum` function uses the exact same process to get rid of list elemens larger than 9 as in problems 16 and 20, so I won't cover that again. The input to this is `∈ω[1]+ω[2]`, and to understand that better, we first need to look at the input to the worker function, being `...⍣(2(ω-1)⍴0),1 1}1000`.

We need two lists of the length specified by the input argument (1000 in this case) to store numbers for the fibonacci sequence, because every new term is the sum of the previous two. In order to build the initial state, I set up a segmented list of two `ω`-element lists. They both consist of `ω-1` zeros and a 1 at the end, representing F_1 and F_2.

Let's look at the sequence of how that builds up step by step, using a smaller list size:

```
    (2 10⍴0)
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0

    (2 10⍴0),1 1
0 0 0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0 0 0 1

    ↓(2 10⍴0),1 1
┌─────────────────────┬─────────────────────┐
│0 0 0 0 0 0 0 0 0 0 1│0 0 0 0 0 0 0 0 0 0 1│
└─────────────────────┴─────────────────────┘
```

The **sum** function now uses those two lists and adds them. The result is then *Enlist*ed and brought down to single digits with the inner block. When this is done, we need to use *Enclose* (**⊂**) to box the result again, because the worker function (i.e. the way it is designed) expects a boxed list. Let's see the result with the shorter example-lists:

```
    {{0<+/n←⌊⍵÷10:∇(10|⍵)+1⌽n⋄⊂⍵}∈⍵[1]+⍵[2]}↓(2 10⍴0),1 1
┌─────────────────────┐
│0 0 0 0 0 0 0 0 0 0 2│
└─────────────────────┘
```

We now have the tool to sum the terms of the sequence, we just need a function that loops until the 1000th digit becomes non-zero. The worker function sets **⍺←1**, which will be our index of the nth term of the sequence. And because we are not actually interested in the Fibonacci number itself, but just its index, the final return statement is just **⍺**.

The guard in between first checks if the first digit of **⍵[1]** (being the current Fibonacci-number) is still zero using **0=⊃∈⍵[1]**. While that is the case, the function calls itself again with **⍺** raised by 1 and the sum of the current two terms appended to the second one, hence building the next iteration of the Fibonacci sequence.

And that's basically it. As you may have already guessed, I don't like to show the final results, because this book isn't supposed to be a copy&paste template for PE solutions. But those of you who like big numbers can see the resulting Fibonacci-term on the next page.

But to test if everything works as intended, let's check this with 3 as the input, which should result in 12, becasue the 12th term is the forst one to have 3 digits:

```
    {{⍺←1⋄0=⊃∈⍵[1]:(⍺+1)∇⍵[2],(sum ⍵)⋄⍺}⍬⍴(2(⍵-1)⍴0),1 1}3
12
```

And it does! And now, as promised, the first 1000-digit Fibonacci-number:

```
    {{⍺←1⋄0=⊃∈⍵[1]:(⍺+1)∇⍵[2],(sum ⍵)⋄∈⍕¨∈⍵[1]}⍬⍴(2(⍵-1)⍴0),1 1}1000
1070066266382758936764980584457396885083683896632151665013235203375314520604694040621889147582489792657804694888177591957484336466672569959512996030461262748092482186144069433051234774442750273781753087579391666192149259186759553966422837148943113074699503439547001985432609723067290192870526447243726117715821825548491120525013201478612965931381792235559657452039506137551467837543229119602129934048260706175397706847068202895486902666185435124521900369480641357447470911707619766945691070098024393439617474103736912503231365532164773697023167755051595173518460579954919410967778373229665796581646513903488154256310184224190259846088000110186255550245493937113651657039447629584714548523425950428582425306083544435428212611008992863795048006894330309773217834864543113205765659868456288616808718693835297350643986297640660000723562917905207051164077614812491885830945940566688339109350944456576357666151619317753792891661581327159616877487983821820492520348473874384736771934512787029218636250627816
```

Problem 26 – Reciprocal cycles

Problem 26 wants us to find the value of d<1000, for which 1/d has the longest recurring cycle in its decimal fraction part:

> A unit fraction contains 1 in the numerator. The decimal representation of the unit fractions with denominators 2 to 10 are given:
>
> 1/2 = 0.5 1/3 = 0.(3) 1/4 = 0.25 1/5 = 0.2 1/6 = 0.1(6) 1/7 = 0.(142857) 1/8 = 0.125 1/9 = 0.(1) 1/10 = 0.1
>
> Where 0.1(6) means 0.166666..., and has a 1-digit recurring cycle. It can be seen that 1/7 has a 6-digit recurring cycle.
>
> Find the value of d < 1000 for which 1/d contains the longest recurring cycle in its decimal fraction part.

My solution uses a pretty simple algorithm, which however has the disadvantage to only work for values of d which actually *have* a recurring cycle. So I need to filter out 1, all even numbers and all numbers which are divisible by 5, which fortunately is easily done in APL:

```
{⍵/⍨{⍵=⌈/⍵}{⍺←(⊃⍵)|10◇⍺≠1:((⊃⍵)|10×⍺)∇⍵,0◇≢⍵}¨⍵}1↓⍸∧⌿0≠2 5∘.|⍳999
```

The input list is created very similar to problem 1. We just remove all numbers which are divisible by 2 or 5 from the list of all numbers from 1 to 999 and drop the first element, which is 1.

The function then works by the principle that you start with *a=10 mod d*, and then continously set *a=(a*10) mod d* in each iteration until a gets 1, the number of iterations will be the length of the cycle. The right function is just a 1:1 translation of that in APL and shouldn't need further explanation if you followed the previous chapters. I just need to mention a little trick to keep track of the number of iterations: In each iteration, I append a 0 to the input number, and that's why I need to pass `⊃⍵` instead of just `⍵` to the modulo calculation.

At the end, `≢⍵` is returned, which is equal to the length of the cycle. We don't need to discard the first element (which is the input number), because the first iteration is already done before appending any 0 with `⍺←(1↑⍵)|10`. The resulting list of cycle lengths then gets passed to `{⍵=⌈/⍵}` which returns a boolean list with a 1 at the postion of the maximum length, and this is finally used to get the corresponding number with *Replicate*.

Problem 27 – Quadratic primes

Problem 27 deals with prime numbers, so we have the opportunity again to reuse some bits of previous solutions. In this case, it's the prime sieve that I used to solve problems 7 and 10.

> Euler discovered the remarkable quadratic formula:
>
> $n^2 + n + 41$
>
> It turns out that the formula will produce 40 primes for the consecutive integer values $0 \leq n \leq 39$. However, when $n = 40, 40^2 + 40 + 41 = 40(40 + 1) + 41$ is divisible by 41, and certainly when $n = 41, 41^2 + 41 + 41$ is clearly divisible by 41.
>
> The incredible formula $n^2 - 79n + 1601$ was discovered, which produces 80 primes for the consecutive values $0 \leq n \leq 79$. The product of the coefficients, −79 and 1601, is −126479.
>
> Considering quadratics of the form:
>
> $n^2 + an + b$, where $|a| < 1000$ and $|b| \leq 1000$
>
> where $|n|$ is the modulus/absolute value of n e.g. $|11| = 11$ and $|-4| = 4$
>
> Find the product of the coefficients a and b, for the quadratic expression that produces the maximum number of primes for consecutive values of n, starting with $n = 0$.

This is a small enough range to justify a brute force solution in my opinion. And because the equation needs to produce a prime for n=0, we know that b has to be a prime, thus limiting the range of b to all prime numbers from 2 to 997. This reduces the amount of possible a/b combinations from roughly 4,000,000 to 335,832!

My solution uses a primelist generated with the Sieve of Eratosthenes (see problem 7) to generate a list of primes to check against. I first set the limit very high, because one can't be sure beforehand what the highest prime number that the equation procuces will be.

It turns out that the highest prime for the a/b combination that solves the problem is quite a bit less than 2000, and the highest prime number for all combinations is less than 13,000. I set the limit to 2000, as the time-critical task in my solution is searching the list of primes for the current candidate.

```
p←{n←⍵⋄3{n≥⍺×⍺:(⍺+2)∇(0@((⍺-1)↓⍺×⍳⌊n÷⍺))⍵⋄⍸⍵}0 1,(⍵-2)⍴1 0}2000
ab←(¯1000+⍳1999)∘.,p[⍸p<1000]

{×/∊ab[⍸⍵=⌈/∊⍵]}{⍺←1⋄p∊⍨(⍺*2)+(⍵[1]×⍺)+⍵[2]:(⍺+1)∇⍵⋄⍺}¨ab
```

After having the prime list stored in p (please refer to problem 7 for an explanation of this function), I set up **ab** using *Outer Product* with *Catenate* (**∘.,**) to produce an array of all possible combinations of a (being **¯1000+⍳1999**) and b, which is **p** at all indices we get from **p<1000**.

Now it's time to apply the work funcion to *Each* of those. I use **⍺** for n and set that to 1 initially. We don't need to check if 0 produces a prime, because we already limited b to primes. Hence, the condition which evaluates if the function should continue is:

```
p∊⍨(⍺*2)+(⍵[1]×⍺)+⍵[2]
```

To the right, we first calculate the result of the quadratic equation. Then *Membership* (**∊**) is used to find out if the result is a member of **p** like so:

```
    7∊2 3 5 7 11
1

    8∊2 3 5 7 11
0
```

I used **⍨** to switch the order of the arguments, just to cut down on parentheses. While that evaluates to true, the function calls itself with **⍺+1**, else it will return **⍺**, which corresponds to the number of primes produced by the current a/b combo. The result is an array consisting of all cycle lengths.

I finally use **⍸⍵=⌈/∊⍵** in the prepended function to get the index of the maximum length and use this fo filter the corresponding a/b combination out of **ab**. Finally, **×/∊** calculates the product of both members.

Problem 28 – Number spiral diagonals

To solve problem 28, we need to find the sum of the numbers on the diagonals of a matrix, which is formed by a 1 in the middle and then, starting from the right of 1, wrapping the consecutive numbers in a clockwise direction around it.

> Starting with the number 1 and moving to the right in a clockwise direction a 5 by 5 spiral is formed as follows:
>
> ```
> 21 22 23 24 25
> 20  7  8  9 10
> 19  6  1  2 11
> 18  5  4  3 12
> 17 16 15 14 13
> ```
>
> It can be verified that the sum of the numbers on the diagonals is 101.
>
> What is the sum of the numbers on the diagonals in a 1001 by 1001 spiral formed in the same way?

If you play around a bit with this, you'll notice a pattern: The numbers on the corners of the first ring are all seperated by 2, the numbers on the second ring by 4, on the third ring by 6 etc.

Also, the difference between the last number on ring N-1 and the first number on ring N is again 2, 4, 6, 8 etc.

We can use this to build the following solution:

```
+/+\1,4/2×⍳500
```

The first thing to do is to create a list consisting of 1 and then 4 copies each of all even numbers up to the size of the matrix minus 1. Clear as mud, isn't it? But hang on, we'll get there.

In the case of the 5×5 matrix, this is 5-1=4 and we would need the following list:

```
    2×⍳2
2 4

    4/2×⍳2
2 2 2 2 4 4 4 4

    1,4/2×⍳2
1 2 2 2 2 4 4 4 4
```

This represents the numbers on the diagonals in the sense that we have a 1 in the middle, surrounded by 4 numbers seperated by 2, then four numbers seperated by 4. To get the actual numbers, all we need to do now is to use plus with *Scan* (**+**) to return the running sums which result from inserting plus between all numbers in the list. Then it's just a matter of **+/** to get the total:

```
    +\1,4/2×⍳2
1 3 5 7 9 13 17 21 25

    +/+\1,4/2×⍳2
101
```

And that's all there is to do to solve this! If you wanted, you could make it a little function that get's the width of the matrix as an input:

```
    {+/+\1,4/2×⍳2÷⍨⍵-1}5
101
```

Problem 29 – Distinct powers

Problem 29 asks us to find the number of distinct terms of a^b for all combinations of $2 \leq a \leq 100$ and $2 \leq b \leq 100$.

> Consider all integer combinations of ab for $2 \leq a \leq 100$ and $2 \leq b \leq 100$:
>
> $2^2 = 4, 2^3 = 8, 2^4 = 16, 2^5 = 32$
> $3^2 = 9, 3^3 = 27, 3^4 = 81, 3^5 = 243$
> $4^2 = 16, 4^3 = 64, 4^4 = 256, 4^5 = 1024$
> $5^2 = 25, 5^3 = 125, 5^4 = 625, 5^5 = 3125$
>
> If they are then placed in numerical order, with any repeats removed, we get the following sequence of 15 distinct terms:
>
> 4, 8, 9, 16, 25, 27, 32, 64, 81, 125, 243, 256, 625, 1024, 3125
>
> How many distinct terms are in the sequence generated by a^b for $2 \leq a \leq 100$ and $2 \leq b \leq 100$?

This is again pretty simple in APL (or any other array language for that matter), and my solution is:

```
≢∪∈∘.*⍨1+⍳99
```

This shouldn't need detailes explanations. It just a 1:1 translation of the problem, hence asking APL to calculate all a^b combinations, remove all duplicates from the result and print out the number of remaining items. We can see that *Outer Product* does again a good job of applying *Power* to all combinations, and because a and b share the same range, we can just use *Commute* (⍨) to mirror `1+⍳99` over to the left side.

You can also solve (or at least simplify) the problem by doing a bit of preparatory brain work, but this pure brute force solution takes just 3 milliseconds on my laptop, so in my opinion it's – buckle up – a no brainer.

Problem 30 – Digit fifth powers

Drum roll and we are already at problem 30! Let's hunt that one down!

> Surprisingly there are only three numbers that can be written as the sum of fourth powers of their digits:
>
> $1634 = 1^4 + 6^4 + 3^4 + 4^4$
> $8208 = 8^4 + 2^4 + 0^4 + 8^4$
> $9474 = 9^4 + 4^4 + 7^4 + 4^4$
>
> As $1 = 1^4$ is not a sum it is not included.
>
> The sum of these numbers is 1634 + 8208 + 9474 = 19316.
>
> Find the sum of all the numbers that can be written as the sum of fifth powers of their digits.

The problem implies that there is an upper limit, and if we fiddle around with a calculator a bit, then 354294 should be a reasonable choice. That is $6 \cdot 9^5$, or the fifth powers of the digits of 999999.

$7 \cdot 9^5$ is 413343, so still a 6-digit number. In fact, 194979 is the largest number which can be written as the sum of fifth powers of its digits, but we don't know that yet...

My solution looks like this:

```
{+/⍵/⍨{⍵=+/5*⍨⍵⊤⍨10⍴⍨1+⌊10⊗⍵}¨⍵}1+⍳354293
```

The input is a list from 2 to 354294. I then use **`10⍴⍨1+⌊10⊗⍵`** to create a list of as many 10s as the input number's digit count. One plus the floor of $log_{10}N$ will return the number of digits of N, and *Shape* with 10 as it's right argument will output as many 10s:

```
    {(1+⌊10⊗⍵)⍴10}12345
10 10 10 10 10
```

I just used *Switch* to switch the order of arguments for saving parentheses. The same then for **Encode** (`⊤`), which outputs an input number as a list of its digits, when the left argument has as many 10s as the number has digits.

It can also be used to convert decimals to binary, where the number of 2s you pass as the left argument specifies the number of bits the result should have:

```
    10 10 10⊤123
1 2 3

   {ω⊤⍨10⍴⍨1+⌊10⊗ω}12345
1 2 3 4 5

    (8⍴2)⊤123
0 1 1 1 1 0 1 1
```

Then again using *Switch* with *Power* and **5** to get the fifths powers, and finally **+/** for the sum:

```
    {5*⍨ω⊤⍨10⍴⍨1+⌊10⊗ω}12345
1 32 243 1024 3125

    {+/5*⍨ω⊤⍨10⍴⍨1+⌊10⊗ω}12345
4425
```

At the end, **ω=** makes sure that the function returns a 1 if the sum is euqal to the input number, or 0 if it's not. Then it's just a matter of using *Replicate* (with *Switch*, of course) to get all relevant numbers and **+/** for the total.

Note: There is an alternative (and much shorter) way to convert a number to a list of it's digits, and that is to combine *Execute Each* with *Format* like so:

```
    ⍎¨⍕12345
1 2 3 4 5
```

However, don't be misled by the terseness of this. the runtime will be *much* longer when you have to deal with a large number base. Using this, converting all numbers from 1 to 1,000,000 takes more than a minute on my laptop, while using *Encode* like in the above solution takes less than two seconds.

Problem 31 – Coin sums

Problem 31 is a classic one, asking us for the number of possible combinations to make £2 out of any number of coins (1p, 2p, 5p, 10p, 20p, 50p, £1 (100p), and £2 (200p)).

> In the United Kingdom the currency is made up of pound (£) and pence (p). There are eight coins in general circulation:
>
> 1p, 2p, 5p, 10p, 20p, 50p, £1 (100p), and £2 (200p).
>
> It is possible to make £2 in the following way:
>
> 1×£1 + 1×50p + 2×20p + 1×5p + 1×2p + 3×1p
>
> How many different ways can £2 be made using any number of coins?

There are typically two ways of solving this: Through recursion or dynamic programming, I chose the latter. The inner function works with recursion, but that's only used to do the loops, you could also implement it with the power operator or explicit for loops, depending on the APL dialect you are using.

And this is what came out of it:

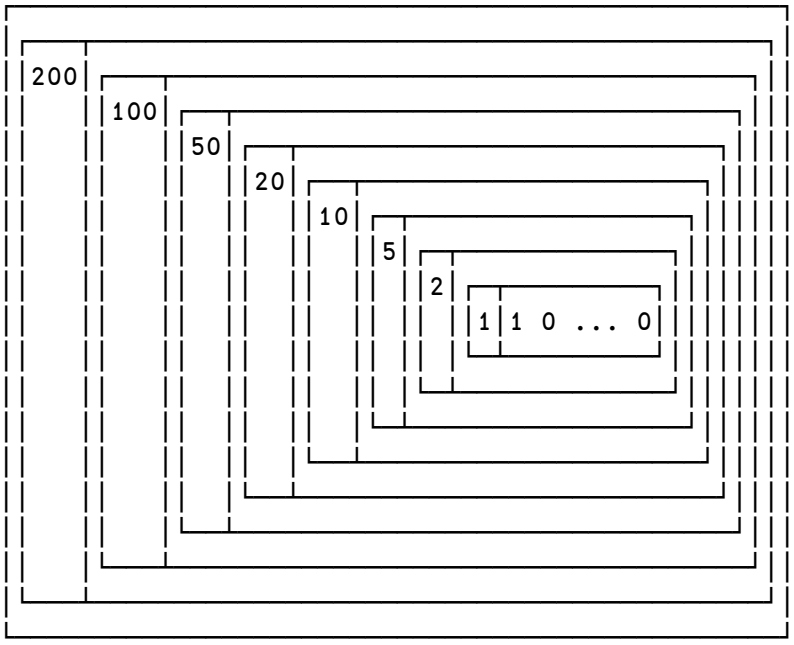
```
c←200 100 50 20 10 5 2 1

{⊃⌽∊{i←⍺⋄(⍺+1){⍺≤≠⍵:(⍺+1)∇((⍵[⍺]+⍵[⍺-i])@⍺)⍵⋄⍵}⍵}/c,⊂1,⍵⍴0}200
```

The algorithm is basically the same that's explained under the link above or similar sites. We have an outer loop that iterates over every coin value stored in `c`, and an inner loop which adds the current value in the list to the one thats `c[i]` positions in front.

The list itself is initially a 1 followed by as many 0s as the value we want to get the combinations vor. In our case it's a 1 followed by 200 zeros, which is initialized by `1,⍵⍴0` in the outermost function.

Similar to problems 18 and 20, I apply a function using *Reduce/Fold* `/`. And again, it can be shown what's actually happening when we just apply the function `{⍺⍵}` with *Fold* to this list:

```
{αω}/c,⊂1,200⍴0
```



It shows that in the first iteration, **α** will be the rightmost element of **c**, which is 1, and **ω** will be the initial list. In the second iteration, **α** will be 2 and **ω** will be the result of the first iteration. It is important to use Enclose **⊂** for the list again, because it needs to be a segment of the input, not just the numbers appended as single items.

The function itself is this:

```
{i←α⋄(α+1){α≤≠ω:(α+1)∇((ω[α]+ω[α-i])@α)ω⋄ω}ω}
```

The outer loop, if you will, first stores the current coin value in **i**, because I need that value unchanged in the inner function. Then **α+1** get's passed as the right argument to the inner function. I need **α+1** because I have my index origin set to 1 (which is default), and the first element of the list which needs to be changed is the second one.

Then the inner function iterates from **α** to **≠ω** and replaces **ω[α]** with **ω[α]+ω[α-i]**, using *At*. The outer loop makes sure that this process is done for all coin vaslues, and the answer to the problem is then the last element of the resulting list, which I extract with *First* of the *Reverse* of the *Enlist*ed (because it's still enclosed) list.

The benefit of this solution is that you not only get the answer for the value in question, but also vor any value up to this, which can be seen if we return the whole list short of the first element:

```
    {1↓∈{i←⍺◇(⍺+1){⍺≤⍵≠⍵:(⍺+1)∇((⍵[⍺]+⍵[⍺-i])@⍺)⍵◇⍵}⍵}/⍳,⊂1,⍵⍴0}200
1 2 2 3 4 5 6 7 8 11 12 15 16 19 22 25 28 31 34 41 44 51 54 61 68 75 ...
```

So there is 1 way of making 1p, 2 ways of making 2p (2x1p, 1x2p), two ways of making 3p (3x1p, 1p+2p) and so on.

Problem 32 – Pandigital products

To solve problem 32, we need to find the sum of all products, whose combination of multiplicand, multiplier and product is 1 through 9 pandigital:

> We shall say that an n-digit number is pandigital if it makes use of all the digits 1 to n exactly once; for example, the 5-digit number, 15234, is 1 through 5 pandigital.
>
> The product 7254 is unusual, as the identity, 39 × 186 = 7254, containing multiplicand, multiplier, and product is 1 through 9 pandigital.
>
> Find the sum of all products whose multiplicand/multiplier/product identity can be written as a 1 through 9 pandigital. HINT: Some products can be obtained in more than one way so be sure to only include it once in your sum.

We can limit the range of products to 4-digit numbers, because we know that we need to use all digits from 1 to 9 exactly once. If the product had 3 digits or less, we would need to use 6 digits or more for multiplicand and multiplier, which could never result in a product that small. Vice versa, if the product had 5 digits or more, we couldn't make that using only 4 digits or less for multiplicand and multiplier.

Also, because we can't use any digit more than once and also zeros are not allowed, we can limit the list of products to start at 1234 and end at 9876. That range still includes invalid numbers, of course, but that's good enough. We could further limit the input by excluding all primes etc., but that wouldn't significantly speed up the following solution:

```
pdp←{0<+/{10=≠∪∊(⍳⍴10)∘.⍵}¨{d←1↓⍸0=(⍳⌊⍵*÷2)|⍵⋄d,¨(⍵÷d),¨⍵}⍵}

+/{pdp ⍵:⍵⋄0}¨1233+⍳8643
```

I first created a function **pdp** to find out if a number produces a 1 throuhg 9 pandigital multiplicand/multiplier/product identity. This takes the number to examine as its input, and passes that through to the follwoing block:

```
{d←1↓⍸0=(⍳⌊⍵*÷2)|⍵⋄d,¨(⍵÷d),¨⍵}
```

This function first calculates all divisors up to the square root of **⍵** (see problem 21 if the method is unclear), and stored those in **d**. This will be the the list of multiplicands. Then it appends *Each* matching multiplier by calculating **⍵÷d**, and finally **⍵** itself, which produces multiplicand/multiplier/product-tuples. Let's see how that operates in sequence:

```
    {d←1↓⍸0=(⍳⌊⍵*÷2)|⍵∘.d}7254
2 3 6 9 13 18 26 31 39 62 78

    {d←1↓⍸0=(⍳⌊⍵*÷2)|⍵∘.d,⍤(⍵÷d)}7254
┌─────────┬─────────┐ ┌─────────┬─────────┬─────────┐
│2 3627   │3 2418   │...│39 186   │62 117   │78 93    │
└─────────┴─────────┘ └─────────┴─────────┴─────────┘

    {d←1↓⍸0=(⍳⌊⍵*÷2)|⍵∘.d,⍤(⍵÷d),⍤⍵}7254
┌──────────────┬──────────────┐ ┌──────────────┬──────────────┬──────────────┐
│2 3627 7254   │3 2418 7254   │...│39 186 7254   │62 117 7254   │78 93 7254    │
└──────────────┴──────────────┘ └──────────────┴──────────────┴──────────────┘
```

That's most of the work done! We now just need to seperate each tuple into it's digits and check if that list contains all numbers from 1 to 9. This is done by the next function block, which is applied to *Each* of the tuples:

```
{10=≢∪∈(4⍴10)⊤⍵}
```

I first use *Encode* with a right argument of **4⍴10** to split each element each of the tuple to a 4-digit list, resulting in 0 at empty decimal places for numbers with less than 4 digits. Then the resulting table is *Enlist*ed and the *Unique* elements extracted:

```
    {(4⍴10)⊤⍵}39 186 7254
0 0 7
0 1 2
3 8 5
9 6 4

    {∈(4⍴10)⊤⍵}39 186 7254
0 0 7 0 1 2 3 8 5 9 6 4

    {∪∈(4⍴10)⊤⍵}39 186 7254
0 7 1 2 3 8 5 9 6 4
```

The remaining 0 doesn't hurt, because it will be there for every possible tuple and we can just account for that by checking with **10=≢** if the resulting list has 10 elements. If that is true, we know that this multiplicand/multiplier/product tuple is 1 through 9 pandigital. The result of this will be a boolean list with 1 for every tuple that satisfies this condition.

We just need to sum the elements of this list to get a singular result which we can use to check if a number produces such a tuple:

```
    {{10=≢∪∈(4ρ10)⊤⍵}¨{d←1↓⍸0=(⍳⌊⍵*÷2)|⍵⋄d,¨(⍵÷d),¨⍵}⍵}7254
0 0 0 0 0 0 0 0 1 0 0

    {0<+/{10=≢∪∈(4ρ10)⊤⍵}¨{d←1↓⍸0=(⍳⌊⍵*÷2)|⍵⋄d,¨(⍵÷d),¨⍵}⍵}7254
1
```

I also prepended **0<** because the problem already states that some products can be obtained in more than one way, and I just need a 1 or 0 for the worker function, which looks like this:

```
+/{pdp ⍵:⍵⋄0}¨1233+⍳8643
```

It just checks for every possible product if it produces a 1-9 pandigital tuple and returns **⍵** if that's the case, else 0. Finally, **+/** gets our answer, being the sum of all relevant products.

Problem 33 – Digit cancelling fractions

Problem 33 is an interesting one:

> The fraction 49/98 is a curious fraction, as an inexperienced mathematician in attempting to simplify it may incorrectly believe that 49/98 = 4/8, which is correct, is obtained by cancelling the 9s.
>
> We shall consider fractions like, 30/50 = 3/5, to be trivial examples.
>
> There are exactly four non-trivial examples of this type of fraction, less than one in value, and containing two digits in the numerator and denominator.
>
> If the product of these four fractions is given in its lowest common terms, find the value of the denominator.

This can be solved by pure brute force, of course. But we can simplify the solution by significantly reducing the amount of valid fractions $\frac{n}{d}$.

The first thing we can do is omitting all values which are multiples of 10. For those, the only option is to cancel out the 0s (otherwise, we would either divide by 0 or the result would be 0), and that is what the problem calls a "trivial example".

It can further be shown that valid solutions can only be achieved by canceling the second digit of the numerator and the first digit of the denominator (see for example here).

This lead me to the following solution:

```
f←↑,/{⍵,¨⍵[⍸(⍵>⍵)∧(10|⍵)=⌊⍵÷10]}¨⍵←{⍵/⍨0≠10|⍵}10+⍳89

{⍵[2]÷∨/⍵}{∊×/⍵/f}{(÷/⍵)=(⌊⍵[1]÷10)÷10|⍵[2]}¨f
```

In the first line, I set up all possible n/d pairs and store them as a segmented list in `f`. Starting from the right, as usual, I first create a list of all 2-digit numbers greater than 10 with `10+⍳89`. This gets passed to `{⍵/⍨0≠10|⍵}`, which removes all multiples of 10. This reduced list is now stored in `⍵`.

The next function block is applied to *Each* of those numbers, takes the input number and appends every number of `⍵` which is larger than `⍵` (because $\frac{n}{d}$ needs to be smaller that 1) *And* also shared its first digit with the last digit of `⍵` using `(10|⍵)=⌊⍵÷10`.

The result is now a segmented list of segmented n/d pairs:

```
    {⍵,¨⍳[⍸(⍳>⍵)∧(10|⍵)=⌊⍳÷10]}¨⍳←{⍵/⍨0≠10|⍵}10+⍳89
┌──────────────────────────────────────────────────────┬──────────┐
│┌─────┬─────┬─────┬─────┬─────┬─────┬─────┬─────┐     │┌─────┬─────┐
│|11 12|11 13|11 14|11 15|11 16|11 17|11 18|11 19|     ||12 21|12 22|...
│└─────┴─────┴─────┴─────┴─────┴─────┴─────┴─────┘     │└─────┴─────┘
└──────────────────────────────────────────────────────┴──────────┘
```

Because we don't need the results to be seperated by values of n, we can remove that level of segmentation. First, we can use *Ravel* with *Reduce* (`,/`) to laminate all n/d tuples to a sinle list, which however is still boxed. This unneeded box can be removed with *Mix*:

```
    ,/{⍵,¨⍳[⍸(⍳>⍵)∧(10|⍵)=⌊⍳÷10]}¨⍳←{⍵/⍨0≠10|⍵}10+⍳89
┌──────────────────────────────────────────────────────┐
│┌─────┬─────┬─────┬─────┬─────┬─────┬─────┬─────┬─────┬─────┐
│|11 12|11 13|11 14|11 15|11 16|11 17|11 18|11 19|12 21|12 22|...
│└─────┴─────┴─────┴─────┴─────┴─────┴─────┴─────┴─────┴─────┘
└──────────────────────────────────────────────────────┘
```

```
    ↑,/{⍵,¨⍳[⍸(⍳>⍵)∧(10|⍵)=⌊⍳÷10]}¨⍳←{⍵/⍨0≠10|⍵}10+⍳89
┌─────┬─────┬─────┬─────┬─────┬─────┬─────┬─────┬─────┬─────┐
|11 12|11 13|11 14|11 15|11 16|11 17|11 18|11 19|12 21|12 22|...
└─────┴─────┴─────┴─────┴─────┴─────┴─────┴─────┴─────┴─────┘
```

I store this in `f`, and now it's time to apply the worker function to this:

```
{⍵[2]÷∨/⍵}{∊×/⍵/f}{(÷/⍵)=(⌊⍵[1]÷10)÷10|⍵[2]}¨f
```

The first function block checks for *Each* n/d tuple, if $\frac{n}{d} = \frac{\lfloor n/10 \rfloor}{d \bmod 10}$, thus returning a 1 for all valid fractions that satisfy the problem. This result is then used in the next block to filter out the corresponding n/d tuples using `⍵/f`. And to get the product of those as a simple list, we just need to prepend `∊×/`:

```
    {⍵/f}{(÷/⍵)=(⌊⍵[1]÷10)÷10|⍵[2]}¨f
┌─────┬─────┬─────┬─────┐
|16 64|19 95|26 65|49 98|
└─────┴─────┴─────┴─────┘
```

```
    {∊×/⍵/f}{(÷/⍵)=(⌊⍵[1]÷10)÷10|⍵[2]}¨f
387296 38729600
```

To get the answer to the problem (the denominator of this fraction when it has been reduced to its lowest common terms), we can apply *GCD* (the monadic form of `∨`) with *Reduce* to this list and divide the second element (which is the denominator) by the result of this.

I must admit that while this solution works, it's a bit cumbersome and probably could have been simplified further. But we all have those days, don't we?

Problem 34 – Digit factorials

To solve problem 34, we need to find all numbers which are equal to the sum of the factorial of their digits:

> 145 is a curious number, as 1! + 4! + 5! = 1 + 24 + 120 = 145.
>
> Find the sum of all numbers which are equal to the sum of the factorial of their digits.
>
> Note: As 1! = 1 and 2! = 2 are not sums they are not included.

As the problem implies, there is an upper limit, and we can define that as $7 \cdot 9! = 2540160$, because $8 \cdot 9! = 2903040$, which is still a 7-digit number. This is still way too high, as you will see in a minute, but when you can't be sure and don't have the time (or the patience) to think about a way to further decrease this, it's better to start with an upper bound that is too high, than to miss a relevant number.

I set the lower bound to 3, because 1 and 2 should be skipped according to the problem.

My solution, using these bounds, looks like this:

```
{+/⍵/⍨{⍵=+/!⍵⍳⍨10⍴⍨1+⌊10⊗⍵}¨⍵}2+⍳2540160
```

And this is basically the same solution as the one I used to solve problem 30, except for the number range and replacing the fifth power with the factorial. So please see there for an explanation of the function.

Problem 35 – Circular primes

Primes again, finally! Problem 35 deals with so called circular primes:

> The number, 197, is called a circular prime because all rotations of the digits: 197, 971, and 719, are themselves prime.
>
> There are thirteen such primes below 100: 2, 3, 5, 7, 11, 13, 17, 31, 37, 71, 73, 79, and 97.
>
> How many circular primes are there below one million?

Time to get ye olde prime sieve out of the attic, which I used to build the following solution:

```
p←{n←⍵⋄3{n≥⍺×⍺:(⍺+2)∇(0@((⍺-1)↓⍺×⍳⌊n÷⍺))⍵⋄⍵}0 1,(⍵-2)⍴1 0}1e6
rot←{10⊥⍣{n≠r←⊂1⌽⍥⌽⍵:∇⍵,r⋄⍵}n←{⊂⍵⊤⍨10⍴⍨1+⌊10⍟⍵}⍵}

+/{(≠rn)=+/p[rn←rot ⍵]}⍤⍸⊢p
```

The prime sieve that I use to build **p** is the same as in problems 7 and 10, except for one minor but important detail: I don't return the primes themselves using **⍸⍵** but just the boolean list **⍵**. I do this because checking if a number **n** is prime or not is much faster with **p[n]** than using *Membership* with **p∊n**. It is 30 times faster for the complete solution in this case!

After the prime sieve for all numbers below one million is done, I set up the **rot** function, which – given a number – returns a list of all its rotations:

```
    rot 12345
12345 23451 34512 45123 51234
```

The first (rightmost) block in **rot** takes the input number and splits it into its digits using the same technique as in problems 30 and 34. The result is then *Enclose*d and stored in **n**. Enclosing it is necessary because I first build all rotations as lists of digits before converting them back to numbers. Hence, I need a partitioned list of rotated lists. It'll become clearer in a minute, but let's see the result up to this point:

```
    {{⊂⍵⊤⍨10⍴⍨1+⌊10⍟⍵}⍵}12345
┌─────────┐
│1 2 3 4 5│
└─────────┘
```

The next block takes this as it's input, and uses a guard whose *if* condition checks `n≠r←⊂1ϕ⊃ϕω`. This compares `n` with the 1-step-to-the-left-rotation (`1ϕ`) of the last element of `ω`, which we get in the usual way using *First* of the *Reverse* (`⊃ϕω`). The result of this is also stored in `r`, because I need it again in the following loop.

I compare `n` to `⊂1ϕ⊃ϕω`, because in the loop I append exatly this to `ω` and let the funciton call itself with this as the new input. So in essence, the function will keep appending rotations until it arrives at the starting point, which is `n`. The result is the aforementioned paritioned list of rotated lists:

```
    {{n≠r←⊂1ϕ⊃ϕω:∇ ω,r ⋄ ω}n←{⊂ωT⍤10ρ⍤1+⌊10⊗ω}ω}12345
┌─────────┬─────────┬─────────┬─────────┬─────────┐
│1 2 3 4 5│2 3 4 5 1│3 4 5 1 2│4 5 1 2 3│5 1 2 3 4│
└─────────┴─────────┴─────────┴─────────┴─────────┘
```

All that's left to do now is to use *Decode* with base 10 and *Each* to convert this back to a list of rotated numbers:

```
    {10⊥⍤{n≠r←⊂1ϕ⊃ϕω:∇ ω,r ⋄ ω}n←{⊂ωT⍤10ρ⍤1+⌊10⊗ω}ω}12345
12345 23451 34512 45123 51234
```

Now it's finally time to start the worker function:

```
+/{(≠rn)=+/p[rn←rot ω]}⍤⍳p
```

This uses `⍳p` as its input, which is just the list of prime numbers under one million. Then, for *Each* of those, I build the list of rotations and store that in `rn`. Then `p[rn←rot ω]` is used to return a list of 0s and 1s, depending on if the numbers in the list of rotations are prime or not.

To check if all numbers in the list are prime, I sum the result of this and compare it to `≠rn`. Because if we have as many 1s as the number of elements in `rn`, that's a bingo. This will return a list of 1s (for circular primes) and 0s, so we just need to apply `+/` to this to get the final answer.

Problem 36 – Double-base palindromes

Problem 36 deals with palindromic numbers, but this time in two bases.

> The decimal number, 585 = 1001001001_2 (binary), is palindromic in both bases.
>
> Find the sum of all numbers, less than one million, which are palindromic in base 10 and base 2.
>
> (Please note that the palindromic number, in either base, may not include leading zeros.)

The last line already limits the problem to odd numbers, because all even numbers have a 0 at the last bit in base 2, which would result in a leading zero for the reverse. My following solution accounts for that:

```
split←{ω⊤⍨⍵⍴⍨1+⌊⍺⊛⍵}

+/{⍵/n}{{ω≡⌽⍵}{2 10∘.split ⍵}⍵}⍨n←¯1+2×⍳5e5
```

The list of odd numbers under one million is built with `¯1+2×⍳5e5` and stored in `n`. Then I use the same method as in the previous problem (also in problems 30 and 34) to convert *Each* number to a list of it's digits, but this time as a function which takes the base as its left argument:

```
    10 split 585
5 8 5

    2 split 585
1 0 0 1 0 0 1 0 0 1
```

In the worker function, I use *Outer Product* with 2 and 10 as the left arguments for `split` to get a partintioned result:

```
    2 10∘.split 585
┌─────────────────┬─────┐
│1 0 0 1 0 0 1 0 0 1│5 8 5│
└─────────────────┴─────┘
```

This result get's passed to the function `{ω≡⌽⍵}`, which checks if the reverse of both bases matches the input. The resulting boolean list, which contains a 1 for each number that satisfies the condition, is then used to get the corresponding numbers using `⍵/n`. Finally, as always, `+/` gets our final answer.

Problem 37 – Truncatable primes

Primes again in problem 37!

> The number 3797 has an interesting property. Being prime itself, it is possible to continuously remove digits from left to right, and remain prime at each stage: 3797, 797, 97, and 7. Similarly we can work from right to left: 3797, 379, 37, and 3.
>
> Find the sum of the only eleven primes that are both truncatable from left to right and right to left.
>
> NOTE: 2, 3, 5, and 7 are not considered to be truncatable primes.

I wanted to reuse the sieve that I used to solve e.g. problems 7 and 10, but I needed to define an arbitrary upper limit, because I couldn't think of an easy method to calculate one. So i set it to one million and luckily, that was sufficient:

```
p←{n←⍵⋄3{n≥⍺×⍺:(⍺+2)∇(0@((⍺-1)↓⍺×⍳⌊n÷⍺))⍵⋄⍵}0 1,(⍵-2)⍴1 0}1e6
trunc←{({10<n←⊃⍵:∇(⌊n÷10),⍵⋄⍵}⍵),{10<n←⊃⌽⍵:∇⍵,n|⍨10*⌊10⊗n⋄1↓⍵}⍵}

+/4↓{(≢t)=+/p[t←trunc ⍵]:⍵⋄0}¨⍳p
```

The `trunc` function is responsible for truncating in both directions and returning a list of all results:

```
    trunc 3797
3 37 379 3797 797 97 7
```

I designed it to return kind of a "mirrored" image, just for fun. But let's look at both subfunctions. The left one is responsible for truncating from right to left by continously dividing the number by 10 (and flooring the result to get an integer), building a list of these results on the way. The *if* condition `10<n←⊃⍵` checks if the remaining number is still greater than 10.

The right function does the left-to-right truncation. This uses `10*⌊10⊗n` to get the power of 10 that matches the current digit count and modulo to truncate one digit from the left. At the end, `1↓⍵` is returned in order to not include the initial number twice.

The worker function is then similar to problem 35:

```
+/4↓{(≢t)=+/p[t←trunc ⍵]:⍵⋄0}¨⍳p
```

This checks for *Each* prime number in **p** if all elements resulting of **trunc ⍵** are prime. If that is the case, **⍵** gets returned, else 0. Then we need to drop the first four numbers of the resulting list, because the problem states to not include 2, 3, 5 and 7. Finally, **+/** gets the answer again.

Problem 38 – Pandigital multiples

Problem 38 deals with pandigital numbers. Smells like splitting into digits again.

> Take the number 192 and multiply it by each of 1, 2, and 3:
>
> 192 × 1 = 192
> 192 × 2 = 384
> 192 × 3 = 576
>
> By concatenating each product we get the 1 to 9 pandigital, 192384576. We will call 192384576 the concatenated product of 192 and (1,2,3)
>
> The same can be achieved by starting with 9 and multiplying by 1, 2, 3, 4, and 5, giving the pandigital, 918273645, which is the concatenated product of 9 and (1,2,3,4,5).
>
> What is the largest 1 to 9 pandigital 9-digit number that can be formed as the concatenated product of an integer with (1,2, ... , n) where n > 1?

For this problem, finding a upper and lower bound for the input numbers to check is easy. We know that the input number needs to be lower than 10000, because the concatenated product of a 5-digit number and (1,2) will result in at least 10 digits. Similarly, if the number has 2 or 3 digits, we won't be able to get a 9-digit concatenated product.

So we are left with 4-digit numbers, and we can further limit the range a bit to all numbers from 9123 to 9876, because the input number can't have multiple digits itself. This also means that we only need to check for n=(1,2), because the concatenated product would contain too much digits if we go further.

And my solution using this range looks like this:

```
⌈/{{(∈⍤¨⍳9)≡⍵[⍋⍵]:±⍵◇0}{∈⍤¨1 2×⍵}⍵}¨9122+⍳754
```

After setting up the input list accordingly, I apply a block of two functions to *Each* number. The first one builds a list of both products with `1 2×⍵` and then uses *Format Each* and finally *Enlist* to get the concatenated product as a string:

```
    {1 2×⍵}9273
9273 18546

    {⍤¨1 2×⍵}9273
┌────┬─────┐
│9273│18546│
└────┴─────┘

    {∈⍤¨1 2×⍵}9273
927318546
```

The next function block checks if the sorted string matches `∈⍕⍤¨⍳9`, which is using the same technique as before to get the string "123456789". If that is the case, the function returns the string as a number using *Execute*, else 0:

```
    {{(∈⍕⍤¨⍳9)≡⍵[⍋⍵]:±⍵◇0}{∈⍕⍤¨1 2×⍵}⍵}9273
927318546
```

After this we just need to apply `⌈/` to get the maximum of all results, which is the answer to the problem.

Problem 39 – Integer right triangles

Problem 39 is about right angle triangles, and we can reuse the function to create those from problem 9.

> If p is the perimeter of a right angle triangle with integral length sides, {a,b,c}, there are exactly three solutions for p = 120.
>
> {20,48,52}, {24,45,51}, {30,40,50}
>
> For which value of p ≤ 1000, is the number of solutions maximised?

My solution uses Dickson's Method again, so please refer to problem 9 for an explanation of this part. It is also important to note that Dickson's method will return triangles with a perimeter ≤ 1000 up to an input number of 168, so we can limit the range to this value:

```
tri←{ω{α+ω[1],ω[2],+/ω}¨{ω,n÷ω}¨{⍸0=ω|⍨⍳⌊ω*÷2}n←2÷⍨ω×ω}

{c←+⌿ω∘.∈∪ω◇(∪ω)[⍸c=⌈/c]}{ω[⍸ω≤1000]}∈{+/¨tri ω}¨2×⍳84
```

The perimeters of all triplets found with *Each* input number are calculated using **`+/¨tri ω`**, and the result is then *Enlist*ed to get a simple list of the results:

```
    {+/¨tri ω}¨2×⍳84
┌──┬─────┬────────┬──────────┬──────────┬──────────────────────┐
│12│30 24│56 40 36│90 60 48  │132 84 60 │182 112 90 80 72 70 │...
└──┴─────┴────────┴──────────┴──────────┴──────────────────────┘

    ∈{+/¨tri ω}¨2×⍳84
12 30 24 56 40 36 90 60 48 132 84 60 182 112 90 80 72 70 ...
```

The next function block filters for all perimeters which are ≤ 1000, because Dickson's Method will produce some triplets that have a greater perimeter than this.

The last function then uses the same technique to get the counts of the individual perimeters as I used in problem 12 to get the count of the prime factors, so please refer to that chapter if you need clarifiaction on the **`+⌿ω∘.∈∪ω`** bit. The list of counts is stored in **`c`** and this is then used to filter out the perimeter with the highes count using **`(∪ω)[⍸c=⌈/c]`**.

Problem 40 – Champernowne's constant

Already at problem 40, only 10 more to go after this!

> An irrational decimal fraction is created by concatenating the positive integers:
>
> 0.12345678910111213141516171819202 1...
>
> It can be seen that the 12th digit of the fractional part is 1.
>
> If d_n represents the nth digit of the fractional part, find the value of the following expression.
>
> $d_1 \times d_{10} \times d_{100} \times d_{1000} \times d_{10000} \times d_{100000} \times d_{1000000}$
>
> It can be shown (or evaluated with APL), that concatenating the numbers up to 185185 is just sufficient to result in one million digits. My solution uses that as its limit:

```
{×/⍵[10*¯1+⍳7]}∊{⍵⊤⍨10⍴⍨1+⌊10⍟⍵}¨⍳185185
```

I use the same split-to-digits method as in many problems before, so I won't explain the first function block, which get's applied to *Each* input number and returns a partitioned list of each number's digits, which we can convert to an unsegmented list using *Enlist*:

```
      {⍵⊤⍨10⍴⍨1+⌊10⍟⍵}¨⍳20
┌─┬─┬─┬─┬─┬─┬─┬─┬─┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┬──┐
│1│2│3│4│5│6│7│8│9│1 0│1 1│1 2│1 3│1 4│1 5│1 6│1 7│1 8│1 9│2 0│
└─┴─┴─┴─┴─┴─┴─┴─┴─┴──┴──┴──┴──┴──┴──┴──┴──┴──┴──┴──┘

      ∊{⍵⊤⍨10⍴⍨1+⌊10⍟⍵}¨⍳20
1 2 3 4 5 6 7 8 9 1 0 1 1 1 2 1 3 1 4 1 5 1 6 1 7 1 8 1 9 2 0
```

After that, it's just a matter of using `⍵[10*¯1+⍳7]` to get the digits at indices 1, 10, 100, 1000, 10000, 100000 and 1000000, and `×/` takes care of the product.

Problem 41 – Pandigital prime

Pandigital numbers again in problem 41, this time: primes.

> We shall say that an n-digit number is pandigital if it makes use of all the digits 1 to n exactly once. For example, 2143 is a 4-digit pandigital and is also prime.
>
> What is the largest n-digit pandigital prime that exists?

This now puts my prime sieve to the test, because we need all primes up to 987654321. Or do we? It can be shwon that a number is divisible by 3, if the sum of its digits is divisible by 3. For a 1 through 9 pandigital number, the digit sum is 45. For a 1 to 8 pandigital number, it evaluates to 36. Both are divisible by 3, hence no 1 through 8 or 9 pandigital number can be prime, and we can lower the limit to a much more managable 7654321 (and in fact, the number of interest is very close to that):

```
p←{n←⍵⋄3{n≥⍺×⍺:(⍺+2)∇(0@((⍺-1)↓⍺×⍳⌊n÷⍺))⍵⋄⍺⍵}0 1,(⍵-2)⍴1 0}7654321

⌈/∊{⍵[⍋⍵]≡⍳≠⍵:10⊥⍵⋄⍬}¨{⍵⍪¨10⍴¨1+⌊10⊗⍵}¨p
```

The first step is again to split *Each* prime to a list of its digits (see problems 30, 34, 35). Then I apply `{⍵[⍋⍵]≡⍳≠⍵:10⊥⍵⋄⍬}` to *Each* of these results, which compares the sorted list of digits with a list from 1 up to the digit count. If that finds a match, it returns `10⊥⍵`, being the list of digits put back together to the number using *Decode*. Else, an empty vector `⍬` is returned.

The partitioned list of results is then *Enlist*ed and finally `⌈/` finds the maximum.

Problem 42 – Coded triangle numbers

Problem 42 gives us the opportunity to reuse most of what we needed to solve problem 22:

> The nth term of the sequence of triangle numbers is given by, tn = ½n(n+1); so the first ten triangle numbers are:
>
> 1, 3, 6, 10, 15, 21, 28, 36, 45, 55, ...
>
> By converting each letter in a word to a number corresponding to its alphabetical position and adding these values we form a word value. For example, the word value for SKY is 19 + 11 + 25 = 55 = t_{10}. If the word value is a triangle number then we shall call the word a triangle word.
>
> Using words.txt (right click and 'Save Link/Target As...'), a 16K text file containing nearly two-thousand common English words, how many are triangle words?

And here is my solution:

```
{+/ωϵ+\ι20}{+/¯64+⎕UCS ⍵}¨1⎕⎕CSV'/path/to/words.txt'
```

As I said, the first part up to the last function block is pretty much identical to my solution for problem 22, so please refer to this for detailed explanations. After we got our word values with `+/¯64+⎕UCS ⍵`, we just need to identify those which are triangle numbers. I already showed in problem 12, that the list of the first `n` triangle numbers is easily built in APL using `+\ι n`:

```
    +\ι20
1 3 6 10 15 21 28 36 45 55 66 78 91 105 120 136 153 171 190 210
```

And in fact, this is just the range we need, because the highest word value is just over 190. The last function now uses this list with *Membership*, to check which values are triangle numbers, and `+/` gets the answer one more time.

Problem 43 – Sub-string divisibility

Problem 43 has pandigital numbers on the menu again, but this time with a twist:

> The number, 1406357289, is a 0 to 9 pandigital number because it is made up of each of the digits 0 to 9 in some order, but it also has a rather interesting sub-string divisibility property.
>
> Let d_1 be the 1st digit, d_2 be the 2nd digit, and so on. In this way, we note the following:
>
> $d_2d_3d_4 = 406$ is divisible by 2
> $d_3d_4d_5 = 063$ is divisible by 3
> $d_4d_5d_6 = 635$ is divisible by 5
> $d_5d_6d_7 = 357$ is divisible by 7
> $d_6d_7d_8 = 572$ is divisible by 11
> $d_7d_8d_9 = 728$ is divisible by 13
> $d_8d_9d_{10} = 289$ is divisible by 17
>
> Find the sum of all 0 to 9 pandigital numbers with this property.

This is one of those problems which are actually faster to solve with pen&paper than by designing an algorithm for it. But let's try that nonetheless:

```
d←{⍎⍵/⍨(3⍴10)⊤n}¨⍋0=2 3 5 7 11 13 17∘.|n←{⍵/⍨{⍵=⍎∪⍕⍵}¨⍵}11+⍳976
comb←{⍵/⍨9=≢¨∪¨⍵}↑{{(1↑⍵),3↓⍵}¨{⍵/⍨{(2↑1↓⍵)≡2↑3↓⍵}¨⍵},⍺∘.,⍵}/d

0⍎+/{10⊥((⍳9)~⍵),⍵}¨comb
```

What the...? I must admit it looks terrible, but despite being brute force, it solves the problem in 35 milliseconds and that's not bad at all! Let's examine this bit by bit, and you'll see that it actually makes sense. Kind of...

The first line sets up $d_2d_3d_4$, $d_3d_4d_5$ etc. I first created a list of all numbers from 12 to 987, because that's the range in which all digit triples must sit, in order to satisfy the problem. The prepended function `{⍵/⍨⍵≡⍎∪⍕⍵¨⍵}` takes *Each* number, converts it to a string with *Format*, then takes the *Unique* "numbers" and converts that to a number again with *Execute*. The result of this is then compared to the original number. I do this to identify all numbers which have multiple digits, because those can't be any of the digit triples:

```
    {⍵/⍨{⍵=⍎∪⍕⍵}¨⍵}11+⍳976
12 13 14 15 16 17 18 19 20 21 23 24 25 26 27 28 29 30 31 32 34 ...
```

After the list has been reduced to numbers without mutliple digits, it gets stored in **n** for later re-use. Then I calculate the results of 0=n mod 2 3 5 7 11 13 17 with *Outer Product* (see problem 1) and convert the table of results to a partitioned list using *Split*:

```
    ↓0=2 3 5 7 11 13 17∘.|n←{⍵/⍨{⍵=⍵∪⍵}⍣≡⍵}11+⍳976
┌──────────────────────────────────────────────────────────────────────────────┐
│1 0 1 0 1 0 1 0 1 0 0 1 0 1 0 1 0 1 0 1 1 0 1 0 1 0 1 0 1 ...                 │
└──────────────────────────────────────────────────────────────────────────────┘
```

Each partition holds the results of one of the 0=x mod n comparisons, and the list will be used as a filter for **n** converted to 3-digit lists with *Encode* like so:

```
    (3⍴10)⊤n
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 3 3 3 ...
2 3 4 5 6 7 8 9 0 1 3 4 5 6 7 8 9 0 1 2
```

After using *Each* of the filter partitions on this result, we get the numbers that are divisible by 2, 3, 5 etc. as seperate lists:

```
    {⍵/(3⍴10)⊤n}¨↓0=2 3 5 7 11 13 17∘.|n←{⍵/⍨{⍵=⍵∪⍵}⍣≡⍵}11+⍳976
┌─────────┐ ┌─────────┐
│0 0 0 0 0│ │0 0 0 0 0│
│1 1 1 1 2│ │1 1 1 2 2│
│2 4 6 8 0│ │2 5 8 1 4│
└─────────┘ └─────────┘
```

And because *Encode* returned the lists of digits as columns, I used *Transpose* to convert that to rows and then **Split** to get a partitioned list of partitioned digit-lists like so:

```
{⊖⍵/(3⍴10)⊤n}¨↓0=2 3 5 7 11 13 17∘.|n←{⍵/⍨{⍵=⍵∪⍵}⍣≡⍵}11+⍳976
┌─────┬─────┬─────┬─────┬─────┬─────┬─────┐
│0 1 2│0 1 2│0 1 5│0 1 4│1 3 2│0 1 3│0 1 7│
│0 1 4│0 1 5│0 2 0│0 2 1│1 4 3│0 2 6│0 3 4│
│0 1 6│0 1 8│0 2 5│0 2 8│1 5 4│0 3 9│0 5 1│
│0 1 8│0 2 1│0 3 0│0 3 5│1 6 5│0 5 2│0 6 8│
│0 2 0│0 2 4│0 3 5│0 4 2│1 7 6│0 6 5│0 8 5│
                  ...
```

```
    {↓⍉ω/(3⍴10)⊤n}¨↓0=2 3 5 7 11 13 17∘.|n←{ω/⍨{ω≡⍋∪⍒ω}¨ω}11+⍳976
┌───────────────────────────────────────────┬────────────────────────────────
│┌─────┬─────┬─────┬─────┐     ┌─────┬─────┐│┌─────┬─────┬─────┐
││0 1 2│0 1 4│0 1 6│0 1 8│ ... │9 8 4│9 8 6│││0 1 2│0 1 5│0 1 8│ ...
│└─────┴─────┴─────┴─────┘     └─────┴─────┘│└─────┴─────┴─────┘
└───────────────────────────────────────────┴────────────────────────────────
```

Phew, that was a lot to digest, and there's probably an easier way to get to this result, but at least it worked. So what do we have now: A partitioned list with each partition containing the possible digit-triples for $d_2d_3d_4$, $d_3d_4d_5$ etc. as a segmented list. This now gets stored in **d**.

Still awake? good! So let's continue with the second line, which is used to build the valid combinations of all triples:

```
comb←{ω/⍨9=≢¨∪¨ω}↑{{(1↑ω),3↓ω}¨{ω/⍨{(2↑1↓ω)≡2↑3↓ω}¨ω},α∘.,ω}/d
```

The rightmost function gets applied to **d** with *Fold* (please refer to problem 18 for a refresher on this topic). In the first iteration, this first creates all possible combinations of $d_7d_8d_9$ and $d_8d_9d_{10}$ using *Catenate* with *Outer Product* (**α∘.,ω**). The resulting table is then converted back to a segmented list with **Ravel** (**,**). Let's see the result of this if we simulate the first iteration by only applying it to **↑d[6]** and **↑d[7]**:

```
    (↑d[6]){,α∘.,ω}↑d[7]
┌───────────┬───────────┬───────────┬───────────┬───────────┐
│0 1 3 0 1 7│0 1 3 0 3 4│0 1 3 0 5 1│0 1 3 0 6 8│0 1 3 0 8 5│...
└───────────┴───────────┴───────────┴───────────┴───────────┘
```

But we are not interested in all combinations, but only those for which the last two digits of $d_7d_8d_9$ and the first two digits of $d_8d_9d_{10}$ (etc.) are the same. So I filter this result using **{ω/⍨{(2↑1↓ω)≡2↑3↓ω}¨ω}**. This checks exactly that for *Each* partition. **2↑1↓ω** gets digits 2 and 3, **2↑3↓ω** gets digits 4 and 5. The result of this now only contains all combinations where those digits overlap:

```
    (↑d[6]){{ω/⍨{(2↑1↓ω)≡2↑3↓ω}¨ω},α∘.,ω}↑d[7]
┌───────────┬───────────┬───────────┬───────────┬───────────┐
│0 1 3 1 3 6│0 3 9 3 9 1│0 5 2 5 2 7│0 7 8 7 8 2│0 9 1 9 1 8│...
└───────────┴───────────┴───────────┴───────────┴───────────┘
```

To finally get the combination (being $d_7d_8d_9d_{10}$ in this case), I apply **{(1↑⍵),3↓⍵}** to *Each* result, which combines the first digit with three digits dropped, thus removing one instance of the overlapping pair:

```
    (↑d[6]){{(1↑⍵),3↓⍵}¨{⍵/⍨{(2↑1↓⍵)≡2↑3↓⍵}¨⍵},⍺∘.,⍵}↑d[7]
┌─────────┬─────────┬─────────┬─────────┬─────────┬─────────┬─────┐
│0 1 3 6  │0 3 9 1  │0 5 2 7  │0 7 8 2  │0 9 1 8  │1 3 0 6  │1 5 6 1  │...
└─────────┴─────────┴─────────┴─────────┴─────────┴─────────┴─────┘
```

And because I apply this with *Fold* to **d**, the function will continoulsy work it's way forward, until in the last step $d_2d_3d_4$ is combined with all valid (i.e. having overlapping digit pairs) $d_3d_4d_5d_6d_7d_8d_9d_{10}$ combinations:

```
    ↑{{(1↑⍵),3↓⍵}¨{⍵/⍨{(2↑1↓⍵)≡2↑3↓⍵}¨⍵},⍺∘.,⍵}/d
┌─────────────────┬─────────────────┬─────────────────┬─────┐
│0 1 2 3 5 7 2 8 9│0 1 2 9 5 2 8 6 7│0 1 8 3 5 7 2 8 9│...
└─────────────────┴─────────────────┴─────────────────┴─────┘
```

We now have a list of all combinations, but still need to get rid of all of those that have multiple digits. And we can do this in the usual manner by checking if the length of the *Unique* list is 9, thus applying **{⍵/⍨9=≢¨∪¨⍵}** to *Each* partition:

```
    {⍵/⍨9=≢¨∪¨⍵}↑{{(1↑⍵),3↓⍵}¨{⍵/⍨{(2↑1↓⍵)≡2↑3↓⍵}¨⍵},⍺∘.,⍵}/d
┌─────────────────┬─────────────────┬─────────────────┬─────┐
│1 0 6 3 5 7 2 8 9│1 3 0 9 5 2 8 6 7│1 6 0 3 5 7 2 8 9│...
└─────────────────┴─────────────────┴─────────────────┴─────┘
```

This leaves us with just six results, which are now all $d_2d_3d_4d_5d_6d_7d_8d_9d_{10}$ combinations which satisfy the problem, and I store this in **comb**. There is just one more thing to do, and that is to prepend d_1 to all results. We know that d_1 can only be the number from 1 to 9 which is not a member of the result. To find this, we can use *Whithout* in **(⍳9)~⍵**, which removes all members of **⍵** out of the list of numbers from 1 to 9:

```
    {(⍳9)~⍵}1 0 6 3 5 7 2 8 9
4
```

We can now just append **⍵** to this, and we finally have our finished lsit of digits, which we can convert to the corresponding number with *Decode* using **10⊥**.

Putting all of this together now gets our six resulting numbers:

```
    {10⊥((⍳9)~⍵),⍵}¨comb
4106357289 4130952867 4160357289 1406357289 1430952867 1460357289
```

And the last step, as usual, is to apply `+/` to this and we can use `0⍕` to get rid of the scientific notation.

I know this wasn't the most intuitive solution and a bit quick&dirty, but I'm just glad that I found something that worked at all. And because it's really fast, I don't feel the urge to optimize it any further. I just hope you could take something away from it!

Problem 44 – Pentagon numbers

Problem 44 brings us back to calmer waters.

> Pentagonal numbers are generated by the formula, $P_n = n(3n-1)/2$. The first ten pentagonal numbers are:
>
> 1, 5, 12, 22, 35, 51, 70, 92, 117, 145, ...
>
> It can be seen that $P_4 + P_7 = 22 + 70 = 92 = P_8$. However, their difference, $70-22 = 48$, is not pentagonal.
>
> Find the pair of pentagonal numbers, P_j and P_k, for which their sum and difference are pentagonal and $D = |P_k-P_j|$ is minimised; what is the value of D?

My solution works, but it works on the assumption that the first pair for which the sum and the difference are pentagonal is the one with the lowest difference. I'm not totally sure that if this is true, but it is the right solution nonetheless.

```
d←{{diff←∊∘.-⍨⍵◇0<+/i←(diff∊⍵)∧(∊∘.+⍨⍵)∊⍵:diff[⌊i]◇0}{2÷⍨⍵×¯1+3×⍵}¨⍳⍵}

{0<n←d ⍵:n◇∇⍵+1000}1000
```

The first line does the actual work this time. It gets a number **⍵** as its input, and then creates a list of the first **⍵** pentagonal numbers with **2÷⍨⍵×¯1+3×⍵¨⍳**. The next function then creates a list of the differences of all combinations of these numbers using *Outer Product* with **∊∘.-⍨⍵**, and stores that list in **diff**. Then it checks if there is a pair for which also the sum is pentagonal with:

```
0<+/i←(diff∊⍵)∧(∊∘.+⍨⍵)∊⍵
```

So I use *Membership* to get a boolean list that has a 1 for all differences which are in the list of pentagonal numbers. Then I do the same for the sums, and combine both results with *And*, which gets stored in **i**. If we have a result, the sum of all elements of this will be 1, hence I check if that sum is greater than 0. If it is, then *Where* is used to find the index of that 1 and the list of difference is filtered for that index. Else, the function will return 0.

The second function then starts with a limit of 1000 for **d** and if a result is found, it is returned. If **d** returns 0, i.e. no result was found, then the function calls itself with the limit raised by 1000.

I use this extra function because I didn't want to set an arbitrary limit for **d**. Of course you could just call **d** with 2500 and call it a day.

Problem 45 – Triangular, pentagonal, and hexagonal

Figurative numbers again in problem 45:

> Triangle, pentagonal, and hexagonal numbers are generated by the following formulae:
>
> | | | |
> |---|---|---|
> | Triangle | $T_n = n(n+1)/2$ | 1, 3, 6, 10, 15, ... |
> | Pentagonal | $P_n = n(3n-1)/2$ | 1, 5, 12, 22, 35, ... |
> | Hexagonal | $H_n = n(2n-1)$ | 1, 6, 15, 28, 45, ... |
>
> It can be verified that $T_{285} = P_{165} = H_{143} = 40755$.
>
> Find the next triangle number that is also pentagonal and hexagonal.

We can simplify this a bit, because all Hexagonal numbers are also Triangle numbers. Specifically, all T_n with an odd n are Hexagonal numbers. This means that we only need to create Hexagonal numbers and then check if they are also pentagonal, which I did in the following solution. I also used the test for pentagonal numbers which can be found in the corresponding Wikipedia article:

```
{h←+/⍳⍵◇{(⌊n)=n←6÷⍨¯1+(1+24×⍵)*÷2:1◇0}h:h◇∇⍵+2}287
```

The function takes 287 as its initial argument, because that will create the next higher hexagonal number. The number itself is then calculated with `+/⍳⍵` (see problem 12) and stored in `h`. The following guard uses the test for pentagonal numbers mentioned above to check if that produced a pentagonal number. If it did, the number is returned, else the function calls itself again with the next higher odd number. And that's it!

Problem 46 – Goldbach's other conjecture

Problem 46 deals with one of Christian Goldbach's conjectures:

> It was proposed by Christian Goldbach that every odd composite number can be written as the sum of a prime and twice a square.
>
> $9 = 7 + 2\times1^2$
> $15 = 7 + 2\times2^2$
> $21 = 3 + 2\times3^2$
> $25 = 7 + 2\times3^2$
> $27 = 19 + 2\times2^2$
> $33 = 31 + 2\times1^2$
>
> It turns out that the conjecture was false.
>
> What is the smallest odd composite that cannot be written as the sum of a prime and twice a square?

My solution re-uses the Sieve of Eratosthenes function from problems 7 and 10, so please refer to those for details:

```
soe←{n←⍵⋄3{n≥⍺×⍺:(⍺+2)∇(0@((⍺-1)↓⍺×⍳⌊n÷⍺))⍵⋄⍸⍵}0 1,(⍵-2)⍴1 0}
```

```
{⌊/∈{l←2÷⍨⍵-p[⍸p<⍵]⋄0<≢l/⍨{⍵=⌊⍵}l*÷2:⍬⋄⍵}¨(1+2×⍳2÷⍨⍵)~(p←soe ⍵)}10000
```

I set a fixed limit of 10000 this time, but you could easily adjust the function to continously raise the limit if no solution is found with a lower one. After the list of primes under 1000 is built and stored in `p`, I set up a list of odd numbers excluding all primes using *Without* in `(1+2×⍳2÷⍨⍵)~(p...`. The actual worker function is this:

```
{l←2÷⍨⍵-p[⍸p<⍵]⋄0<≢l/⍨{⍵=⌊⍵}l*÷2:⍬⋄⍵}
```

Applied to *Each* odd composite number in the list, it first creates a list by subtracting each prime lower than the current input from this number with `⍵-p[⍸p<⍵]`. The result is then divided by 2 (with switch to save parentheses), because the difference between the number and the prime should be twice a square. The resulting list is stored in `l`.

Then follows a guard which checks if `0<≢l/⍨{⍵=⌊⍵}l*÷2`. This takes the square root of each element in `l` and `⍵=⌊⍵` tests if the result is integer by comparing it to its floor. If this results in a number (or a list of numbers), an empty vector is returned, because then the conjecture was true for that number. If the rsult is empty, the function returns the number. Because there is more than one result under 10000, I prepend `⌊/∈` to get the minimum of the *Enlist*ed results.

Problem 47 – Distinct primes factors

Primes again in problem 47:

> The first two consecutive numbers to have two distinct prime factors are:
>
> $14 = 2 \times 7$
> $15 = 3 \times 5$
>
> The first three consecutive numbers to have three distinct prime factors are:
>
> $644 = 2^2 \times 7 \times 23$
> $645 = 3 \times 5 \times 43$
> $646 = 2 \times 17 \times 19$
>
> Find the first four consecutive integers to have four distinct prime factors each. What is the first of these numbers?

I went all the way back to problem 3/12 to make use of the prime factorization function again. And my solution looks like this:

```
upf←{α←3⋄0=2|⊃ω:∇(÷⍟2@1)ω,2⋄0=α|⊃ω:α∇(÷⍟α@1)ω,α⋄(α×α)<⊃ω:(α+2)∇ω⋄≢∪ω~1}

{α←0⋄4=upf ω:(α+1)∇ω+1⋄α=4:ω-4⋄0∇ω+1}210
```

It's a bit slow (7 seconds on my laptop), but we don't care, do we? Well, we probably should, but I need to work out a faster prime factorization function first. Hence, the `upf` function is exactly the same as in problem 12, except for the fact that it returns `≢∪ω~1` instead of just `ω~1`, to get the number of distinct factors.

The worker function starts with 210 (because this is 2x3x5x7, the smallest number with four distinct prime factors) and an initial value of 0 for `α`. Then it checks if the current `ω` has four distinct prime factors. If that evaluates to true, the function calls itself again with `α+1` and `ω+1`. So `α` acts as the counter, and when it is increased to 4 (i.e. four consecutive numbers with four distinct prime factors were found), then it returns `ω-4`, being the lowest one of those. Else, `α` is reset to zero and the function continues with `ω+1`.

Problem 48 – Self powers

Problem 48 deals with self powers up to 1000^{1000}, so we need to figure out a workaround for big integers again:

> The series, $1^1 + 2^2 + 3^3 + ... + 10^{10} = 10405071317$.
>
> Find the last ten digits of the series, $1^1 + 2^2 + 3^3 + ... + 1000^{1000}$.

That "last ten digits" bit comes to our rescue, because we can just truncate each step with modulo calculations while we build up the self powers:

```
¯10↑0⍕+/{⍵{⍺×1e10|⍵}⍣⍵⊢1}¨⍳1000
```

The input is a list of the numbers from 1 to 1000, as you would expect. The inner function then uses the *Power Operator* to calculate the nth power of n. The left argument is the base, the argument for the *Power Operator* is the exponent, and the initial input is 1, i.e. n^0. This example shows that for 2^{10}:

```
    2{⍺×1e10|⍵}⍣10⊢1
1024
```

The use of `1e10|⍵` instead of just `⍵` makes sure that we don't get more digits than we need (or APL can digest). We just need to pass the input number as base and exponent to this, and then sum the list of results. *Format* with `0` makes sure that we don't get the result in scientific notation, and `¯10↑` returns the last 10 digits.

Problem 49 – Prime permutations

Primes and permutations – could you ask for more? And problem 49 delivers.

> The arithmetic sequence, 1487, 4817, 8147, in which each of the terms increases by 3330, is unusual in two ways: (i) each of the three terms are prime, and, (ii) each of the 4-digit numbers are permutations of one another.
>
> There are no arithmetic sequences made up of three 1-, 2-, or 3-digit primes, exhibiting this property, but there is one other 4-digit increasing sequence.
>
> What 12-digit number do you form by concatenating the three terms in this sequence?

I use my trusty prime sieve (see problems 7 and 10) to create a list of primes between 1000 and 10000 for this solution:

```
p←{n←⍵⋄3{n≥⍺×⍺:(⍺+2)∇(0@((⍺-1)↓⍺×⍳⌊n÷⍺))⍵⋄(1e3<⍸⍵)/⍸⍵}0 1,(⍵-2)⍴1 0}1e4
can←{⍵/⍨{⍵[3]∊p}¨⍵}{⍵,⍵[2]--/⍵}¨↑,/{⍵,¨p[⍸p>⍵]}¨p~1487

{↑∊¨⍎¨⍕¨⍵/can}{2=+/2≡/{⍵[⍋⍵]}¨⍎¨⍕¨⍵}¨can
```

The first line sets up our prime list as usual, except for the fact that I limit the result to primes larger than 1000 to comply with the problem. This list is now the input to the next function, and I used *Without* in `p~1487` to exclude the example.

The function itself now takes *Each* prime from the list and appends every prime that is larger than itself with `⍵,¨p[⍸p>⍵]`. To remove a level of segmentation and the surrounding box from this (essetially making it a partitioned list of prime pairs), I used *Ravel* with *Reduce* and then *Mix* which results in this:

```
    ↑,/{⍵,¨p[⍸p>⍵]}¨p~1487
┌─────────┬─────────┬─────────┬─────────┬─────────┬─────────┐
│1009 1013│1009 1019│1009 1021│1009 1031│1009 1033│1009 1039│...
└─────────┴─────────┴─────────┴─────────┴─────────┴─────────┘
```

Now we need to add the third term to *Each* of those pairs, which is the second term plus the difference of the second and first term. A short way of getting this is `⍵[2]--/⍵`, which results in the following list of triples:

```
    {⍵,⍵[2]--/⍵}¨↑,/{⍵,¨p[⍸p>⍵]}¨p~1487
┌─────────────┬─────────────┬─────────────┬─────────────┬─────────────┐
│1009 1013 1017│1009 1019 1029│1009 1021 1033│1009 1031 1053│...
└─────────────┴─────────────┴─────────────┴─────────────┴─────────────┘
```

Now we need to remove all triples whose the third term isn't prime. Do do this, I use **ω/⍨⍵[3]∈p¨⍵** which checks with *Membership* if the third term of *Each* triple is an Element of the prime list and then uses the resulting list as a filter for the list of candidates:

```
    {ω/⍨{ω[3]∈p}¨ω}{ω,ω[2]--/ω}¨↑,/{ω,¨p[⍳p>ω]}¨p~1487
┌─────────────────┬─────────────────┬─────────────────┬─────────────────┐
│1009 1021 1033│1009 1039 1069│1009 1051 1093│1009 1063 1117│...
└─────────────────┴─────────────────┴─────────────────┴─────────────────┘
```

I decided to split the function at this point, because it would get rather long otherwise, so I stored the resulting candidates in **can**. We now need to filter out all triples whose numbers are permutations of each other. To do this, I apply **2=+/2≡/⍵[⍋⍵]¨⍎¨⍵** to *Each* triple. This first converts *Each* number of the triple to a string and then sorts *Each* string in ascending order:

```
    {{ω[⍋ω]}¨⍎¨ω}¨can
┌─────────────┬─────────────┬─────────────┬─────────────┐
│ ┌────┬────┬────┐ │ ┌────┬────┬────┐ │ ┌────┬────┬────┐ │ ┌────┬────┬────┐ │
│ │0019│0112│0133│ │ │0019│0139│0169│ │ │0019│0115│0139│ │ │0019│0136│1117│ │...
│ └────┴────┴────┘ │ └────┴────┴────┘ │ └────┴────┴────┘ │ └────┴────┴────┘ │
└─────────────┴─────────────┴─────────────┴─────────────┘
```

We can now insert *Match* pairwise, which results in **1 1** if all three strings are the same, like in this example:

```
    {2≡/{ω[⍋ω]}¨⍎¨ω}1487 4817 8147
1 1
```

And if we sum this result with **+/** and compare that to **2**, we know that we have found the triple we are looking for. The last block now uses the boolean list resulting from this as a filter for **can**, converts *Each* digit from **Each** number to a character and then *Enlist*s the result which is our final answer.

Problem 50 – Consecutive prime sum

And we are finally ready to tackle the last problem of this book: Problem 50. And a nice one it is:

> The prime 41, can be written as the sum of six consecutive primes: 41 = 2 + 3 + 5 + 7 + 11 + 13
>
> This is the longest sum of consecutive primes that adds to a prime below one-hundred.
>
> The longest sum of consecutive primes below one-thousand that adds to a prime, contains 21 terms, and is equal to 953.
>
> Which prime, below one-million, can be written as the sum of the most consecutive primes?

As usual with primes, I use the prime sieve explained in problems 7 and 10, up to alimit of one million for this solution:

```
p←{n←⍵⋄3{n≥⍺×⍺:(⍺+2)∇(0@((⍺-1)↓⍺×⍳⌊n÷⍺))⍵⋄⍳⍵}0 1,(⍵-2)⍴1 0}1e6
sums←{{⍺←1⋄1e6>s←+/⍺↑⍵:(⍺+1)∇⍵⋄⍺{p∊⍨⍵:⍵,⍺⋄⍺>1:(⍺-1)∇(⊃⍵)-⍙[⍺]}s}⍙←⍵}

{⊃∊⍵[⍸⊃¨⍵]}{sums ⍵↓p}¨⍳1000
```

The *sums* function is responsible of outputting the maximum sum and chain length found by starting the sum at each prime number. The first guard sets `⍺←1` for the initial run, and then continuously builds up the sum of all following primes until that would get larger that one million with `1e6>s←+/⍺↑⍵:(⍺+1)∇⍵`. As you can see, the sum is stored in `s`.

When the point is reached where the sum would get larger than one million, the *Else* case takes effect, which contains the following function:

```
⍺{p∊⍨⍵:⍵,⍺⋄⍺>1:(⍺-1)∇(⊃⍵)-⍙[⍺]}s
```

This gets the current value of `⍺` as its left argument, and the sum `s` as its right input. This checks if the current sum is already a prime with `p∊⍨⍵`. If that is the case, it returns the sum and appends `⍺` to it, which gives us a sum/chain length pair. If the current sum is not a prime number, the function continously subtracts the last added primes using `(⍺-1)∇(⊃⍵)-⍙[⍺]`, with `⍙` being a copy of the outer functions input (you'll see why in a minute). This goes on until the sum is reduced to a prime.

The worker function now calls `sums` with the prime list as its input, but in each iteration one more element is dropped from the beginning of the list, which essentialy works offsets the starting index of `sums` to the next higher prime number.

The end result of this is a partitioned list of sum/length paris, here with just the first five results:

```
    {sums ⍵↓p}¨⍳1000
┌──────────┬──────────┐   ┌─────────┐
│920291 525│978037 539│...│742757 89│
└──────────┴──────────┘   └─────────┘
```

I arbitrarily set the limit to 1000, because I assumed that the longest chains would result from the lowest primes, and indeed you could set that much, much lower and still get the correct answer.

Now we just need to find the pair with the longest chain and get the corresponding sum. This is done with `⊃∊⍵[⍒⊃¨⍧¨⍵]`, which takes all chain lengths with `⊃¨⍧¨⍵` and uses that with *Grade Down* to sort the list of results. The sum in question is then just the first element of the *Enlist*ed result.

Postface

So that's the end of my "book" (or collection of short stories). Kudos to you if you made it all the way through!

As I already mentioned in the preface, please don't take this as the definitve guide to APL. It's just a documentation of my progress as I went along, and using Project Euler is my favourite way of learning the basics of a new programming language. You definitely shouldn't hesitate to challenge my solutions as I'm sure there are better ones.

Have a good one!