Practical Numbers

# Introduction

To find out if a number is a practical number, first work out its **proper divisors**.Theproper divisors of a number are all of its factors except itself. The proper divisors of 6 are,

1, 2, 3

Then work out all of the possible ways that you could add these factors together. For the factors of 6, this would be,

**1**

**2**

**3**

1 + 2 = **3**

1 + 3 = **4**

2 + 3 = **5**

1 + 2 + 3 = **6**

If all of the numbers up to the number can be found by adding together combinations of factors, then the number is a practical number. 6 is a practical number because the digits, 1, 2, 3, 4 and 5 appear in this list.

Take the number 5. It is a prime number, so its only proper divisor is 1. 1 is the only digit that can be made from the factors and so 5 is not a practical number.

# Task 1 - Mindset

The algorithm for finding practical numbers is going to be more complex than you would need for most number sequences. You are going to have to develop and test lots of small sections of code to be able to solve this problem. Adding comments to your code as you go will be helpful.

The diagram shows the 3 main problems that you have to solve.

# Task 2 – All About The Factors

If it were just factors that we were talking about, this wouldn’t be too tricky. For most sequences, you either output, count or total the factors.None of those processes need you to store the factors. For this one, you will need to be able to store and use the factors that you find.

Your program won’t know in advance what number it is checking and how many factors there are. You can’t declare multiple variables to cover all bases and an array needs to have its size set at the start of the program. The best structure for this program is called a **List**. You can add as many items as you want to a list. Visual Studio also allows you to convert a list to an array with a single statement. Below is a Visual Basic declaration of a list of integers called **pd** (proper divisors). The second line adds 1 to the list, a divisor for any number we look at.

Dim pd As New List(Of Integer)()

pd.Add(1)

1. Start your program with the lines of code shown above. Set another variable called **possible\_factor** to 1. Use a while loop whilst this variable stores a value less than or equal to half of the number whose factors you are finding. Don’t include the number itself in the list of factors.

The following line will convert your list of factors into an array.

Dim divs As Integer() = pd.ToArray()

The page on one dimensional arrays has example code that will remind you how to output the elements of an array with a for loop.

1. Output your list of factors and check that your program is working correctly.

# Task 3 – Power Set

This task is the trickiest part of this problem. Once you have a list of proper divisors, you will need to be able to calculate all of the sums of all of the different combinations of them. Mathematically speaking, this is called the **power set**.

It’s relatively easy to use nested loops to list all combinations. We only need the sums of these combinations though. For us, 1 + 2 is the same as 2 + 1. Calculating all of the other combinations will slow our program down. If each number only had 2 factors, our program will calculate double the number of sums it needs to. This might not seem like a problem. Let’s think about it mathematically for a while.

Let’s say that a number has **n** proper divisors.

Those divisors have to be summed in all possible combinations of **1 to n** numbers taken from the list.

If n is 1, we have only one factor to sum.

If n is 2, we have 2 single factor sums, and one double factor sum.

If n is 3, we have 3 single factor sums, 3 double factor sums and one triple factor sum.

If n is 4, we have 4 single factor sums, 6 double factor sums, 4 triple factor sums and 1 quadruple factor sum.

The number of sums we’d do is 1, 3, 7, 15.

The sequence is **2n – 1**. The reason that we subtract 1 is that we are not using the one other combination that is possible, 0 divisors.

When things come in powers of 2, binary is always your BFF. Imagine that our number had exactly 3 factors. Using 3 bits, the possible binary values are,

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Denary** | **Binary** | **A** | **B** | **C** |
| 0 | 000 | 0 | 0 | 0 |
| 1 | 001 | 0 | 0 | 1 |
| 2 | 010 | 0 | 1 | 0 |
| 3 | 011 | 0 | 1 | 1 |
| 4 | 100 | 1 | 0 | 0 |
| 5 | 101 | 1 | 0 | 1 |
| 6 | 110 | 1 | 1 | 0 |
| 7 | 111 | 1 | 1 | 1 |

Ignoring the zero, look at the last 3 columns on each row. If we wrote the letter headings out only where there is a 1 in the column, we get,

**A**

**B**

**A + B**

**C**

**A + C**

**B + C**

**A + B + C**

Joyfully, you can use binary numbers to carry out this super-tricky stage of the program. It’s still not straightforward though.

A neat trick for working with binary numbers is to use the bitwise shift operator. A 1 in binary means a 1 under the units column. If it is left-shifted by the number of divisors, it will be a one under a higher place value.

This is the code you will need for your outer loop to list the binary values.

For i As Integer = 1 To (1 << divs.Length) – 1

Next

For each of these binary numbers you will need to look at each bit and, if it is a 1, output the divisor at that position in the array you made. You’ll need to nest another loop inside this one.

For j As Integer = 0 To divs.Length - 1

Next

The last tricky part involves using a logic operator to see if the bit is a 1 in each place value. You’re asking a question, so you need to use an IF statement. The condition is as follows,

(i And (1 << j)) <> 0

To understand this, remember that the integers i and j are stored by the computer in binary. If we use the And operator, we are doing a logical operation on the bits of the numbers. Remember your logic gates, you need both inputs to be a 1 for the output to be a 1. The left shift creates a binary integer with a 1 underneath the place value you are checking. All of the other place values are 0. If the logic operation gives an answer that isn’t 0, it means that the binary number had a 1 under this column.

If the condition is met, output divs(j) and a space. Use Console.Write(). Once the j loop is over, add Console.Writeline.

1. Add the code that you need to list all of the divisor combinations and give it a damn good testing.
2. Adapt the code to instead calculate the sum of each set of factors. Set a variable to 0 just before the j loop. Add the divisor instead of outputting. Test again.

# Task 4 – Keeping Track Of The Sums

So, now you can list all the unique factor combinations using some neat tricks with bitwise operators and binary numbers. The next step is to adapt the code to keep track of whether or not each of the numbers up to n is found among the factor sums that you were calculating.

We can do this by making an array before calculating the factor sums.

1. Declare an integer array of size n-1 called **found**. You need to do this before the i loop starts. Make the element in positions 0 and 1 both equal to 1. Use a loop to make all other elements in the array equal to 0. At the end of the j loop, when the total has been establised, set found(sum) to equal 1.
2. After the i loop is all done, use a for loop to add up the ones in the found array. If it is equal to the number you were checking, then the number is a practical number.
3. Test test test.

# Task 5 – Many Practical Numbers

1. Get some loop action going to output all of the practical numbers up to 1000.