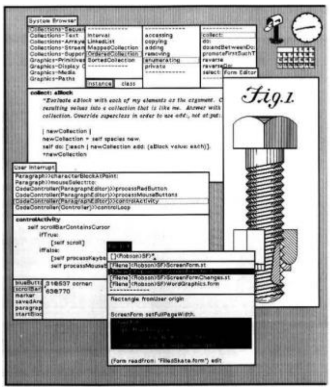
Code by Numbers



```
collect: aBlock
    "Evaluate aBlock with each of my elements as the argument. C
    resulting values into a collection that is like me. Answer with
    collection. Override superclass in order to use add:, not at:put:

    | newCollection |
    newCollection ← self species new.
    self do: [:each | newCollection add: (aBlock value: each)].
    ↑newCollection
```

```
Paragraph>>characterBlockAtPoint:
Paragraph>>mouseSelect:to:
CodeController(ParagraphEditor)>>processRedButton
CodeController(ParagraphEditor)>>processMouseButtons
CodeController(ParagraphEditor)>>controlActivity
CodeController(Controller)>>controlLoop
```

```
controlActivity
    self scrollBarContainsCursor
        ifTrue:
            [self scroll]
        ifFalse:
            [self processKeybo
            self processMouseB
```

```
Rectangle fromUser origin

ScreenForm setFullPageWidth.

(Form readFrom: 'FiledSkate.form') edit
```

Is your computer running slow? Time to upgrade? Think again; computer upgrades are not what they used to be. Improvements in computing speed traditionally came from CPU and memory upgrades; most improvements now come from improved algorithms and software developments. Graphical user interfaces demand more power and memory. To understand what is happening, we need to step back a few decades.

In 1983 Apple® introduced LISA, the first *commercially* available computer with a graphical user interface (GUI), the 5Mb hard drive was an optional extra. The inspiration for Lisa came from something Xerox® had been using for almost a decade. The photocopying behemoth had created a computer called: “Alto” to help make their offices run more efficiently. The GUI that Xerox developed (1974) was called Smalltalk. Users had a three-button mouse, desktop clock, calendar, email, spreadsheet and a word-processer. It is fair to say Xerox was creating the paperless office, not exactly something a photocopying company would want widely adopted. Ironically, they also developed the laser printer, something that would become more pervasive through the use of personal computers.

Smalltalk was the first object-oriented language allowing programmers to use these objects within their programs without having to know *how* the object worked. In 1985 Microsoft® introduced Windows®. Applications in Windows ran on a platform called MS.DOS (Microsoft’s Disk Operating System). Creating an application to work in Windows meant you could rely on MS.DOS to handle all the background computer management. Windows 1.0 required 256kB of memory and at least two floppy drives or 256kB hard-drive. The processor speed was approximately 5MHz and only 16bit. Fast-forward 35 years, Windows 11 requires a minimum 1 GHz twin core 64bit processor, 4GB RAM and 64GB storage. How did we get to the point? The simple answer is that processors and memory became cheaper allowing developers to add more and more layers to their code. From the 1980’s to early 2000’s, computer processing speeds increased almost exponentially. For the past 15 years, the same speed increases have not been possible.

This booklet aims to illustrate how mathematics and coding can be used to improve computational speeds without relying on hardware updates. The activities and investigations are also designed to improve understanding of number. The hierarchical nature of the mathematical content and structure of the coding require these activities be completed sequentially. Each activity includes coding instructions and references, visual and instructional support for mathematical content, reflective questions and a detailed investigation. The 10 minutes of coding references should be completed before commencing the corresponding activity. Coding instructions are included in each activity; however, it is also recommended that the code be modified to improve performance. The first activity “Factors that Count” involves writing a program to count the quantity of factors for any given number. The sample code provided is a ‘brute force’ approach; this code can be modified to achieve the same result much, much faster! The “Euler Totient” activity repeatedly requires information about the factors of a number; the factor program could be called upon for this purpose, however alternative algorithms can also be used that make the program run many, many times faster.

Why focus on number? Aside from the fact that the mathematical content is accessible, the importance of factors, or the lack of them (prime numbers) is critical to our world’s economy thanks to encryption. Many trillions of dollars are digitally transacted every day, the security of these transfers relies on prime numbers and the fact that current algorithms are not particularly efficient at *disassembling* numbers.

Table of Contents

Factors that Count

TI-Codes Lessons:

Unit 1 – Skill Builder 1

⇩

Unit 4 – Skill Builder 1

Commands:

- Request <input>
- For <counter> EndFor
- If <condition> Then <instruction>
- Disp <output>

Finding and Counting Factors

There are many ways to determine the quantity of factors for a specified number. The most common method is to test the divisibility for all applicable numbers. For example, suppose we want to determine the quantity of factors for 18. We can determine the quotient and remainder for all the numbers from 1 to 18.

Table 1A

Divisor	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Quotient	18	9	6	4	3	3	2	2	2
Remainder	0	0	0	2	3	0	4	2	0

Table 1B

Divisor	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**
Quotient	1	1	1	1	1	1	1	1	1
Remainder	8	7	6	5	4	3	2	1	0

Our conclusion is that 18 has six factors since there are six occasions whereby the remainder is equal to zero.

This divisibility check for all numbers is exhaustive. You may have ideas about how this process can be made more efficient, however, this method will provide a basis for an algorithm on which to write a simple program to count the quantity of factors for a given number. You can make the necessary improvements and checks once your initial program is complete and functioning.

Question: 1.

Write a description of the steps required to determine the quantity of factors for any whole number: n.

Instructions:

Start a new document and insert a calculator application.

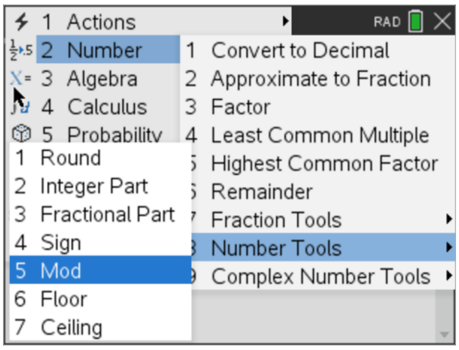
Locate the **mod** command using: **Number** > **Number Tools** > **Mod**

Determine the result of the following calculations:

Mod(18,6)

Mod(18,5)

Mod(18,12)



Question: 2.

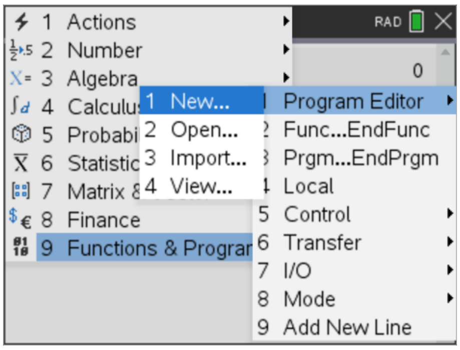
Based on your experimentation, what value does the MOD command return?

Question: 3.

If MOD(*a*, *b*) = 0, what does this say about the relationship between *a* and *b*?

Author: P.Fox

Writing a Program



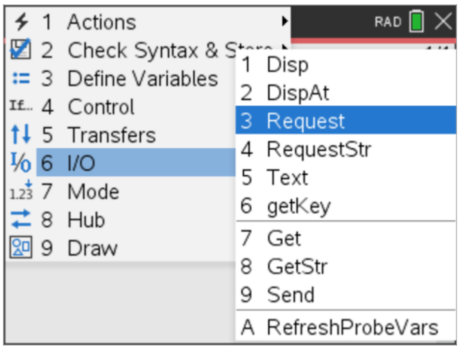
Create a new program by selecting:

Functions & Programs > **Program Editor** > **New**

Call the program: FactorCount

Note that 'FactorCount' is one word as program names cannot contain spaces.

> The Programming application is launched on the same page as the Calculator Application. The page-layout in the document menu can be used to give each application its own page.
> **Short-cut**: [**Ctrl**] + [**6**]. Page 1.1 = Calculator Application. Page 1.2 = Program Application.

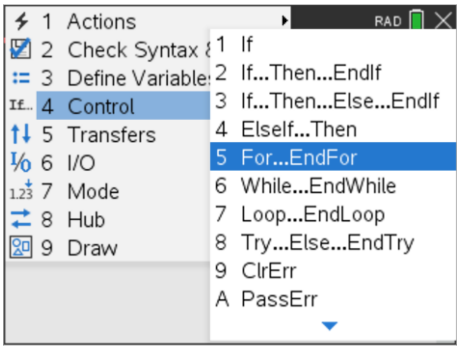


The first task is to request a number from the program user.

Use the I/O (input / output) menu to access the **Request** command. The request command can include a text prompt followed by a variable to store the number.

Request "Enter a number",n

> - Quotation marks: **" "** can be entered by pressing [**Ctrl**] + [**x**] (multiplication sign)
> - The comma is located in the bottom left corner of the keyboard.



A counter will be used to 'count' the quantity of factors. The counter must be set to zero before the counting process begins.

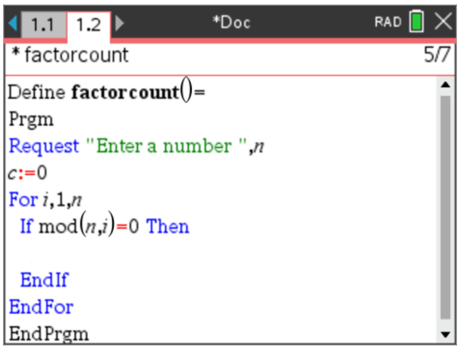
c : = 0

Start a **For** loop by selecting:

Control > **For … EndFor**

The loop will start at 1 and finish at n and use ***i*** to track the number of times the loop has been executed.

For *i* , 1, n



```
Define factorcount()=
Prgm
Request "Enter a number ",n
c:=0
For i,1,n
 If mod(n,i)=0 Then

 EndIf
EndFor
EndPrgm
```

An **IF** statement will be used to check if the user's number has a factor each time the program executes the loop.

The IF command can be selected by:

Control > **If** … **Then** … **EndIf**

Between the IF and THEN statement insert the command:

mod(n, *i*) = 0

Note that 'mod' can be typed directly from the keyboard or accessed through the catalogue.

Move the cursor into the empty line between **THEN** and **EndIf**. This line of code is only executed if the condition: **MOD**(n, i) = 0 is TRUE.

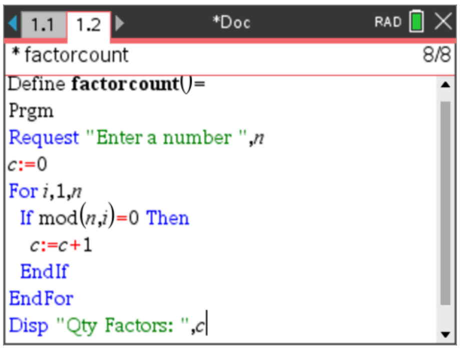
Insert the command:

c := c + 1

Create another line between **EndFor** and **EndPrgm**

From the I/O menu select **Disp** and type the command:

Disp "Qty Factors: ", c



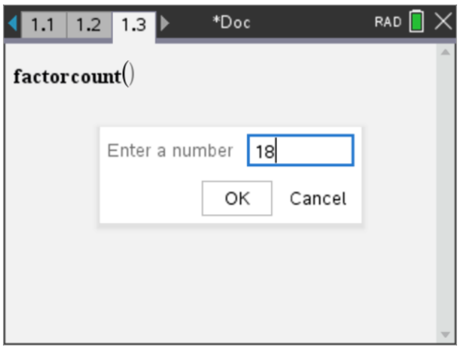
```
Define factorcount()=
Prgm
Request "Enter a number ",n
c:=0
For i,1,n
 If mod(n,i)=0 Then
  c:=c+1
 EndIf
EndFor
Disp "Qty Factors: ",c
```

Save the program and launch it by pressing [**Ctrl**] + [**R**]. A new Calculator application will be created and the program name automatically pasted.

Press [**Enter**] to launch the program.

Start by checking the factor count for 18.

The table at the start of this activity indicates the program should identify 6 factors.



factorcount()

Enter a number 18

Question: 4.

Determine the quantity of factors for each of the following numbers:

a. 24

b. 36

c. 37

d. 144

Check each of your answers by writing down all the factors.

Question: 5.

Determine the quantity of factors for each of the following numbers. Identify a specific characteristic about the quantity of factors and use this to classify the numbers into two groups, explain your classification.

29, 84, 104, 87, 220, 37, 101, 97, 45, 43, 133, 153, 173, 107.

Question: 6.

Determine the quantity of factors for each of the following numbers. Identify a specific characteristic about the quantity of factors and use this to classify the numbers into two groups, explain your classification.

28, 30, 90, 45, 50, 60, 120, 72, 25, 49, 81, 144, 441, 82, 24, 720

Question: 7.

The FactorCount program works, but it could be more efficient. Use a stop watch to time how long the program takes to count the number of factors for: 10,000; 20,000 and 30,000. Use these times to predict how long the program will take to count the factors for 40,000. Test your answer!

If you are satisfied with your prediction, how long would it take to find factors for the following number:

[250 digits!]

Note: This number is associated with RSA Encryption.

Investigation

Why are factors important? To answer this question, consider the opposite situation, the *absence* of factors. Billions of dollars are moved around electronically every day, to do this securely, the electronic transfers must be encrypted. The most common encryption method (RSA) is built around very large prime numbers, numbers with an *absence* of factors! All encryption methods are essentially built on numbers, so being able to 'assemble' and 'disassemble' numbers is extremely important.

Your task is to find a rule that determines the quantity of factors for any whole number, given the prime factorisation of that number.

A few clues are provided along the way to help you on your factor sleuth journey. Document your search findings and conclusions using the clues and your constructed program to help expedite your investigation.

Clue 1:

Determine the prime factorisation and corresponding quantity of factors for the following numbers: 36; 100; 441; 3025 & 48841.

Clue 2:

Determine the prime factorisation and corresponding quantity of factors for the following numbers: 24; 250; 1029; 6655 and 198911.

Clue 3:

Based on the first two clues, generate some numbers that you believe have exactly 8 factors. Test your numbers and comment on the results.

Clue 4:

Determine the prime factorisation and corresponding quantity of factors for the following numbers: 2000; 64827; 107811; 668168 and 1585615607. [Note: For this last number you will need a fast algorithm!]

Clue 5:

Create some numbers of the form: $m^2 \times n^5$ where m and n are both prime. Determine the quantity of factors for each of your numbers. [Note: You may want to choose relatively small prime numbers for m and n.]

Continue the exploration, tabulate your results and record your thoughts, hypotheses, tests and reflections as you go. Documenting findings is an important part of the investigative process. Detectives may have many suspects in their initial investigations, however as more clues surface they develop hypotheses. Detectives test each hypothesis, review what they already know or go in search of more clues. Some investigations end up as Cold Cases, however it is critical that detailed documentation of all aspects of their investigation are retained in the event the investigation is re-opened. Some crimes remain unsolved despite having significant suspects, in mathematics these are often called 'conjectures', a theory that seems to work but has never been proven.

Euclid's Algorithm

TI-Codes Lessons:

Unit 1 – Skill Builder 1

⇩

Unit 4 – Skill Builder 2

Commands:

- Request <input>
- While <condition> EndWhile
- If <condition> Then <instruction> Else <instruction>
- Disp <output>

Highest Common Factors

The Highest Common Factor (HCF) or Greatest Common Divisor (GCD) of two numbers is useful for many reasons. The process is valuable when working with fractions, solving packaging problems, developing traffic light sequences and encrypting content for digital communications. Developed more than 2000 years ago, Euclid's algorithm is still the most efficient process used to determine the Highest Common Factor of two numbers.

Euclid's Algorithm:

LINE #1: IF A = 0 THEN GCD(A,B) = B since GCD(0,B) = B

LINE #2: IF B = 0 THEN GCD(A,B) = A since GCD(A,0) = A

LINE #3: A = B x Q + R … where Q is the quotient and R is the remainder

LINE #4: GCD(B,R) = GCD(A,B), now find GCD(B,R)

This algorithm will make more sense when some numbers are used for A and B. Suppose we want to find the highest common factor of (A) 1260 and (B) 385. As neither A = 0 or B = 0 we progress to LINE #3.

1260 = 385 x 3 + 105 [We say that 105 is the remainder when 1260 is divided by 385]

According to LINE #4 of Euclid's algorithm: GCD(1260,385) = GCD(385,105)

We apply the algorithm again. Since 385 $\neq$ 0 and 105 $\neq$ 0 we proceed to LINE #3.

385 = 105 x 3 + 70 [We say that 70 is the remainder when 385 is divided by 105]

According to LINE #4 of Euclid's algorithm: GCD(1260,385) = GCD(385,105) = GCD(105,70).

We apply the algorithm again. Since 105 $\neq$ 0 and 70 $\neq$ 0, we proceed to LINE #3

105 = 70 x 1 + 35 [We can say that 35 is the remainder when 105 is divided by 70]

We are getting close! According to LINE #4 of Euclid's algorithm: GCD(1260,385) = … = GCD(70,35)

Applying the algorithm one more time, as 70 $\neq$ 0 and 35 $\neq$ 0, we proceed to LINE #3.

70 = 2 x 35 + 0. [This time the remainder is 0!]

Now we can apply LINE #1 or LINE #2 since we have GCD(35,0) = 35.

Our conclusion is that the Highest Common Factor or Greatest Common Divisor of 1260 and 385 is 35.

Question: 1.

Use Euclid's algorithm to identify the highest common factor of: 3850 and 3234.

Author: P.Fox

Writing a Program

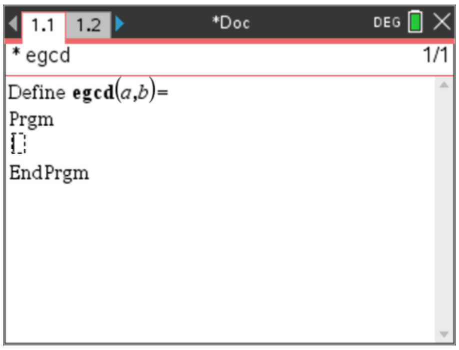
Instructions:

Start a new document; insert a new program.

Add Program Editor > **New**

Call the program: EGCD

Edit the program definition to include “a” and “b”. (See opposite)



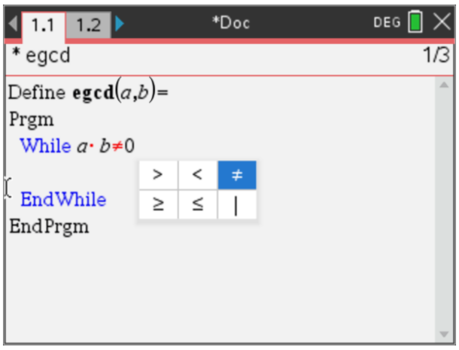
```
* egcd                1/1
Define egcd(a,b)=
Prgm

EndPrgm
```

Euclid’s algorithm ceases when either $a = 0$ or $b = 0$, an easy way to check this is: $a \times b = 0$. The “null factor law” states that if the product of two numbers is zero, then one or both of the numbers must be zero.

The algorithm should continue to run while $a \times b \neq 0$.

Menu > **Control** > **While … EndWhile**

The “not equals” sign can be accessed from the inequality flyout menu.



```
* egcd                1/3
Define egcd(a,b)=
Prgm
 While a·b≠0

 EndWhile
EndPrgm
```

Modular arithmetic returns the remainder when $a \div b$ (where $a > b$) so an If … Then … Else … statement can be used to process Line #3 of Euclid’s algorithm.

Menu > **Control** > **If...Then...Else... EndIf**

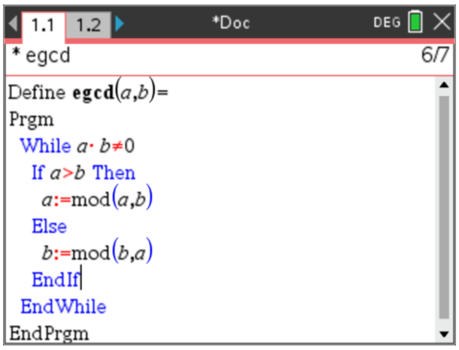
The mod() command can be typed directly or accessed from the catalogue. Enter the corresponding modular arithmetic calculations, note carefully the respective orders for *a* and *b*.

That’s the entire algorithm! The only thing remaining is to display the results. You can use a display command such as:

Disp *a*,*b*

This command can be placed in the loop, between EndIf and EndWhile to monitor the progress of the calculations.

Another display command should be used to display the highest common factor at the end of the program.



```
* egcd                6/7
Define egcd(a,b)=
Prgm
 While a·b≠0
  If a>b Then
   a:=mod(a,b)
  Else
   b:=mod(b,a)
  EndIf
 EndWhile
EndPrgm
```

Question: 2.

Determine the highest common factor of: 1914 and 7293 (by hand) using Euclid’s algorithm and use your results to check the program.

Question: 3.

Test your program on some smaller numbers where you know the highest common factor. Record your test results.

Author: P.Fox

Question: 4.

The **Number** menu in the Calculator Application contains a command to determine the highest common factor of **two** numbers. Edit your program to find the highest common factor of three numbers.
Example: EGCD(a,b,c)

Test and evaluate your program.

Question: 5.

Edit your program to test for the highest common factor of an entire list of numbers.
Note: The program can be defined as egcd(data) where data is a list of numbers: {#, #, # ...}. The **dim** command can be used to determine the dimensions (quantity of numbers) entered into the list.

Investigation

The prime factorisation of a number can be used to efficiently find the highest common factor of any two or more numbers. Use your program to find the highest common factor for each list of numbers (below). Write the original numbers and the highest common factor in terms of their prime factorisation. Try some of your own lists, then write a description of how you can use the prime factorisation to determine the highest common factor of any two or more numbers.

List 1: 1260, 1410, 2040, 4290 & 9570

List 2: 220, 1400, 1700, 30940 & 154700

List 3: 2964, 3588, 8892, 10764 & 409032

List 4: 399, 441, 1911, 3381, 5733 & 835107

Author: P.Fox

Euler Totient Function

TI-Codes Lessons:

Unit 1 – Skill Builder 1

⇩

Unit 4 – Skill Builder 1

Commands:

- Request <input>
- For <counter> Endfor
- If <condition> Then <instruction> Else <instruction>
- Disp <output>

Introduction

The Euler Totient Function for a whole number '*n*' counts the quantity of numbers that are co-prime up to the number *n*. To help understand this definition, consider the number 12.

We need to check which numbers: {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12} have a factor in common with 12, these numbers are discarded leaving us with the numbers that are co-prime. This is summarised in the table below.

Whole Numbers < n	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
Highest Common Factor	1	2	3	4	1	6	1	4	3	2	1	12

There are 4 numbers where the highest common factor is 1, these numbers are co-prime with 12: {1, 5, 7, 11}. The Euler Totient function for 12 is therefore equal to 4, this can be written as: $\varphi(12) = 4$.

Here is another example for the number 9.

Whole Numbers < n	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Highest Common Factor	1	1	3	1	1	3	1	1	9

The Euler Totient function for 9 is therefore equal to 6, this can be written as: $\varphi(9) = 6$.

Question: 1.

Create some pseudo-code for the Euler Totient function.

Writing a Program

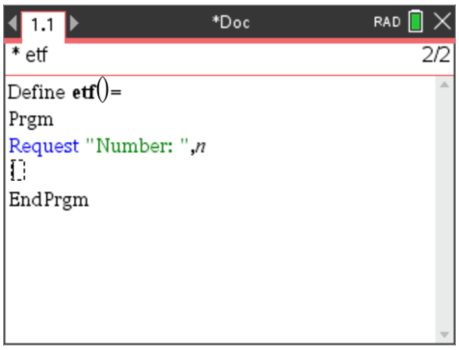
Instructions:

Start a new document; insert a new program.

Add Program Editor > **New**

Call the program: ETF

Use the Request command (I/O) to insert value.



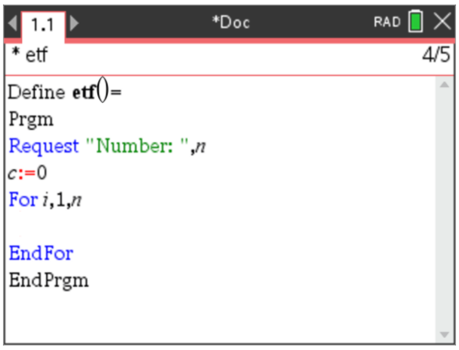
```
Define etf()=
Prgm
Request "Number: ",n

EndPrgm
```

Author: P.Fox

A counter is required to track the quantity of numbers that are co-prime with n.

Initialise a counter: c := 0; then insert a **For** loop.

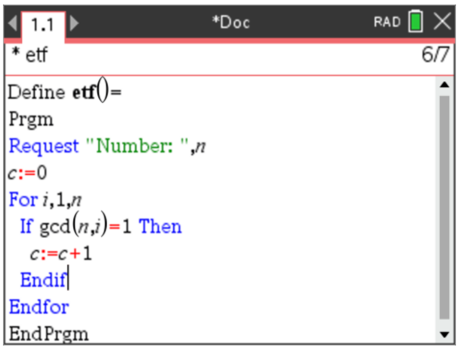
Just like the examples, all the numbers from 1 to n need to be checked.



```
Define etf()=
Prgm
Request "Number: ",n
c:=0
For i,1,n

EndFor
EndPrgm
```

If the highest common factor (greatest common divisor) between n and the loop counter is equal to 1, **then** increase the counter value.

Once the loop has finished, add a ‘display’ command to show the value of the Euler Totient function.



```
Define etf()=
Prgm
Request "Number: ",n
c:=0
For i,1,n
 If gcd(n,i)=1 Then
  c:=c+1
 Endif
Endfor
EndPrgm
```

Question: 2.

Check that your program produces the same results for the two worked examples, then try several others (by hand) and compare results.

Question: 3.

Explore the Euler Totient function for prime numbers, what do you notice?

Question: 4.

Determine the fraction: $\frac{n}{\varphi(n)}$ for the following values of n: 30, 60 and 90, comment on the results.

Question: 5.

The number 100 can be expressed as: $2^2 \times 5^2$. Compare the Euler Totient value for 100 with the following calculation:

$$100\times\left(1-\frac{1}{2}\right)\left(1-\frac{1}{5}\right)$$

Question: 6.

The number 1125 can be expressed as: $3^2 \times 5^3$. Compare the Euler Totient value for 1125 with the following calculation:

$$1125\times\left(1-\frac{1}{3}\right)\left(1-\frac{1}{5}\right)$$

Question: 7.

Use the previous to questions to explore the prime factorisation approach to the Euler Totient function with the Euler Totient value determined by your program.

Question: 8.

How does the prime factorisation approach to calculating the Euler Totient function explain your results to Question 4?

Question: 9.

Why does the 'short cut' approach to the Euler Totient function work?

Investigation

Re-write the Euler Totient function program to determine the Euler Totient function for a range of numbers, graph the results and explore any patterns.

Highly Composite Numbers

TI-Codes Lessons:

Unit 1 – Skill Builder 1

⇩

Unit 4 – Skill Builder 2

Commands:

- Request <input>
- While <condition> EndWhile
- If <condition> Then <instruction> Else <instruction>
- Disp <output>

Introduction

A highly composite number has more factors than any of its predecessors. Think of it as competition along the number line. The difficulty in locating highly composite numbers is that you must already know the previous highly composite number in order to identify how many factors the next number must have in order to qualify. Any search for highly composite number therefore generally starts at 1.

Whilst 1 only has one factor, there are no predecessors, so by default, 1 is the first highly composite number. Naturally 2 is the next highly composite number having two factors. The next is 4 with three factors then 6 with four factors. With one, two, three and four factors already checked, it would be easy to assume that the next highly composite number would have five factors, however 12 is the next highly composite number with six factors.

Question: 1.

Write a description of a program that will determine the Highly Composite number up to some value n.
Note: The quantity of factors for any number can be references as 'factor_count'.

Writing a Program

Instructions:

Start a new document and insert a new program.

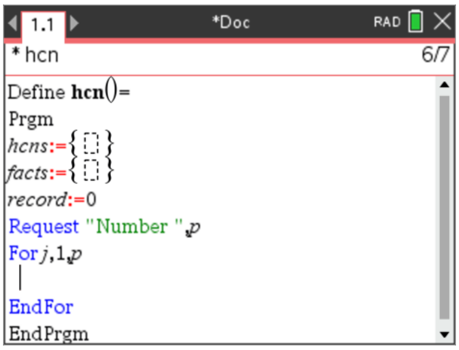
Add Program Editor > **New**

Call the program: HCN

There are a few variables to be set up for this program. It seems appropriate to record the highly composite numbers and as an added check, record the quantity of factors for each.

hcns:={ }	[List of Highly Composite Numbers]
facts:={ }	[List for the quantity of factors]
record:=0	Track most recent HCN.

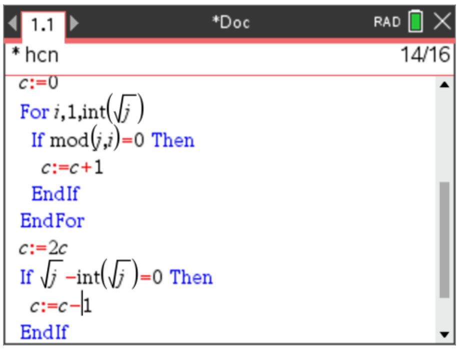
Request the highest number to be searched and start a For loop.



```
*hcn                                6/7
Define hcn()=
Prgm
hcns:={ }
facts:={ }
record:=0
Request "Number ",p
For j,1,p
 |
EndFor
EndPrgm
```

If the factor counting program is in the same document or available from the public library, it can be called up on here to count factors. It is very important however that variables are not duplicated. The factor counting program will also need to be very efficient as it is called upon many times!

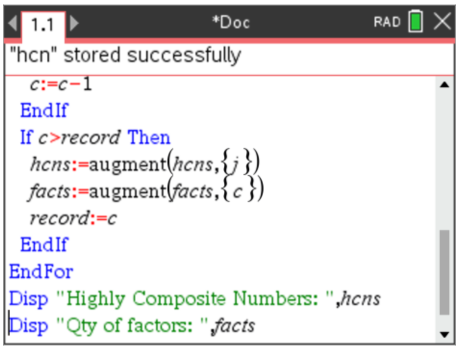
The sample code shown opposite is a relatively fast factor counting program that can be placed inside the previous loop.



```
*hcn                                14/16
 c:=0
 For i,1,int(√(j))
  If mod(j,i)=0 Then
   c:=c+1
  EndIf
 EndFor
 c:=2c
 If √(j) -int(√(j))=0 Then
  c:=c-1
 EndIf
```

Author: P.Fox

If the most recent factor count is higher than the previous record then the list of highly composite numbers (hcns) needs to be updated, so too the record for the quantity of factors (facts) and finally, the record itself needs to be updated.



```
"hcn" stored successfully
   c:=c-1
  EndIf
  If c>record Then
   hcns:=augment(hcns,{j})
   facts:=augment(facts,{c})
   record:=c
  EndIf
EndFor
Disp "Highly Composite Numbers: ",hcns
Disp "Qty of factors: ",facts
```

Question: 2.

Run your program and check that the first five highly composite numbers are: 1, 2, 4, 6, 12; then determine all the highly composite number from 1 to 100.

Question: 3.

Determine all the highly composite numbers from 1 to 1000 and their corresponding quantity of factors.

Question: 4.

Express each of the Highly Composite Number in the previous question as a product of its prime factors.

Question: 5.

Study the prime factorisations closely. Suggest a possible prime factorisation for the next highly composite number, the corresponding number and quantity of factors.
Note: You may have more than one educated guess.

Investigation

To continue exploring Highly Composite Numbers, a more efficient program (or new program) is required, one that no longer starts at 1, rather one that starts at some previously identified Highly Composite Number and uses information gleaned from the first sixteen highly composite numbers.

- Re-write your HCN program so that it can start at any HCN.
- Continue recording HCNs and the corresponding prime factorisations. When and what will be the next prime factor to be included in the prime factorisation?
- Identify any patterns you can find in the prime factorisation that would help in locating subsequent prime factorisations.
- What prior learning are you using to identify the quantity of factors, make predictions and search?