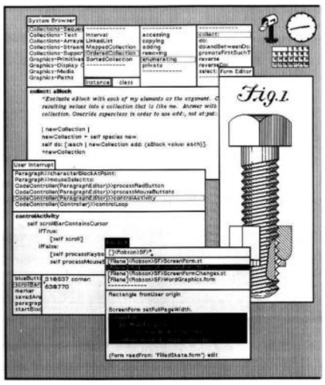
Code by Numbers



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.

Author: P. Fox

Table of Contents

Author: P.Fox

Factors that Count

Teacher Notes:

A PowerPoint slide show is provided with this activity as an introductory presentation for students to watch. The slides work through the algorithmic process for the determination of the factors for the number: 36. Slides reference mathematical terminology such as: divisor, quotient and remainder, the animations are designed to help students understand these terms.

The presentation does not cover all possible divisors, instead, it leaves students pondering the most efficient algorithm by stopping at a divisor of 9 and posing the comment: “The factors are now starting to repeat themselves”.

A perfect square has been used on purpose in the slide set, it serves as a subtle hint that it may only be necessary to search up to the square-root of the corresponding number. If this efficiency is incorporated, students would need to incorporate a check routine for perfect squares to ensure a double count of the square-root is not erroneously included in the factor count.

Instructions for the simplest program (not the quickest) are provided here so that students may also be assessed on their ability to independently arrive at a more efficient factor searching routine and conditions.

More advanced students may check if the input is odd and therefore start the loop counter with an odd number and use a step size two, therefore skipping divisibility for all subsequent even quantities.

TI-Codes Lessons:

Unit 1 – Skill Builder 1

⇩

Unit 4 – Skill Builder 1

Commands:

- input
- for (range)
- if
- print
- int (number types)
- [ ] (create a list)
- Append (add elements to a list)
- len (length of a list)
- % (module

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 – Finding the factors of 18

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 – Finding the factors of 18

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.

Author: P.Fox

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.

Answer: Student answers will vary, the pseudo-code provides the basis on which the program will be written.

Sample:

- Input number: n
- Set factor count to 0
- Loop from 1 to n
- If n ÷ (loop counter) has no remainder Then increase (factor count)
- End Loop
- Display (factor count)

Instructions:

Start a new document and insert a calculator application.

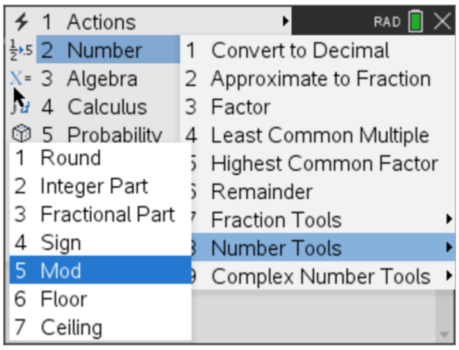
Locate the **mod** command using: **Number** > **Number Tools** > **Mod**

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?

The mod command returns the remainder upon division.

Question: 3.

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

If the 'remainder' of *a* ÷ *b* = 0 then *b* must be a factor of *a*.

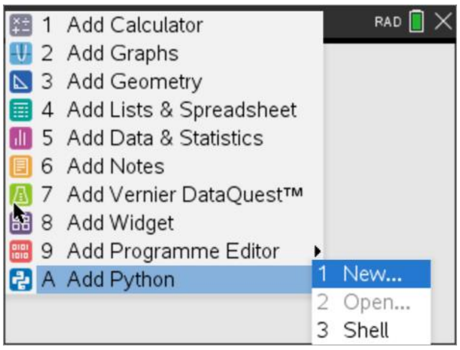
Writing a Program

Create a new Python program by pressing:

[ctrl] + [doc▾] , **Add Python** > **New…**

Call the program: FactorCount

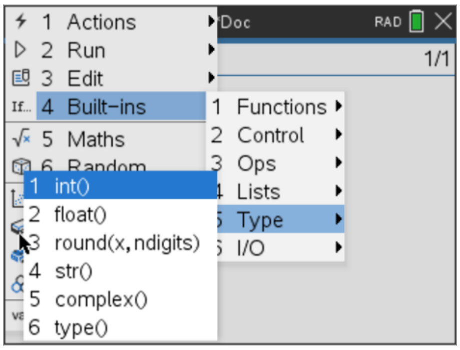
Note that 'FactorCount' is one word as program names cannot contain spaces.



If 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 and store it as a variable. Type in:

n =



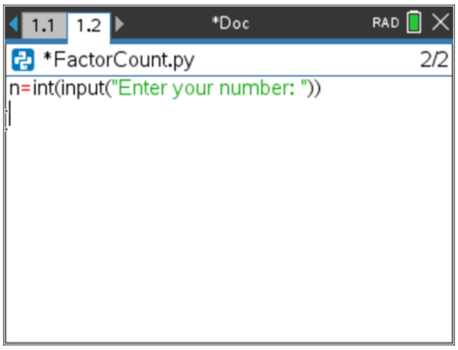
Python input defaults to text, so the next step is to restrict the input to a whole number (integer). The int() command is located in the "**type**" menu, alternatively it can be typed in directly from the keyboard:

Press:

menu > **Built-ins** > **Type** > **int()**

The next step is to enter the "input" command:

menu > **Built-ins** > **I/O** > **input()**



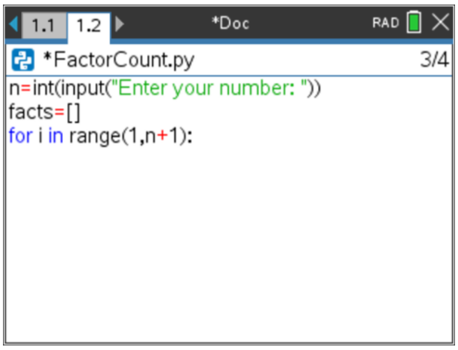
Finish the instruction by adding a text prompt.

When this command is executed, the user's numerical input will be stored in a variable "n"

- Quotation marks: **" "** can be entered by pressing [**Ctrl**] + [**x**] (multiplication sign)
- Colour is added automatically to help locate the various parts of the syntax.

We could just count factors, however, it is easy record and store them in a list which also helps with checking the results.

Factors = [ ]



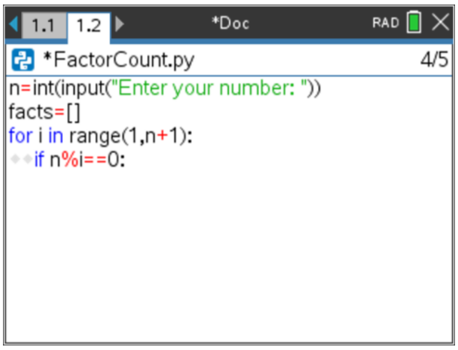
This creates an empty list called factors.

A "FOR" loop can be used to check for factors. We use a FOR loop because we can pre-determine the quantity of iterations the loop must perform.

menu > **Built-ins** > **Control** > **For index in Range(***size***)**

Python loop execution ceases when the counter reaches *size*, therefore the counter (*size*) needs to be one more than n.

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:

menu > **Built-ins** > **Control** > **If**

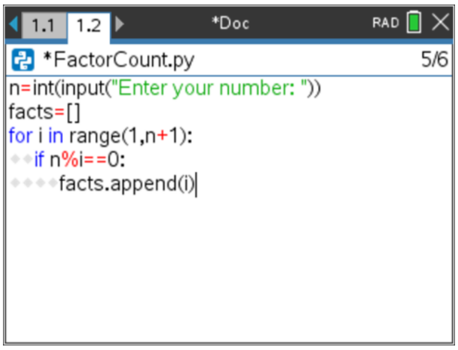
The % operation in Python is for modular arithmetic.

The percentage sign can be obtained from the punctuation fly-out menu.

Author: P.Fox

The ‘append’ command adds the latest factor to the current list of factors. This command can be typed in directly or accessed from the variable menu (facts) followed by:

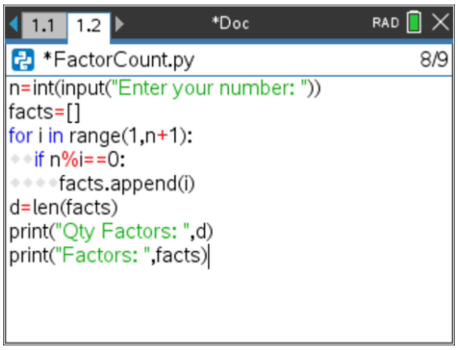
menu > **Built-ins** > **Lists** > **.append()**



The value to be added to the list, given that the division has not generated any remainder, is “*i*”, the loop counter.

That’s all for the algorithmic part of the program! The next step is to display the results. Start by deleting the indentions, the end of the “IF” condition and the For loop.

The list (“facts”) contains all the factors, len(facts) therefore returns the size of the list. This quantity can be stored in ‘d’.



Now the quantity of factors (d) and the actual factors can be printed to the screen.

Save the program and launch it by pressing [**Ctrl**] + [**R**]. A new Python shell will be created and the program name automatically pasted. Start by checking the factor count for 18.

The table at the start of this activity identifies 6 factors, compare this with the output from your program.

To check another number, press [**Ctrl**] + [**R**]

Question: 4.

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

a. 24 8 factors … {1, 2, 3, 4, 6, 8, 12, 24}

b. 36 9 factors … {1, 2, 3, 4, 6, 9, 12, 18, 36}

c. 37 2 factors (prime numbers have exactly 2 factors) … {1, 37}

d. 144 15 factors … {1, 2, 3, 4, 6, 8, 9, 12, 16, 18, 24, 36, 48, 72, 144}

Check each of your answers by writing down all the factors. (Factors listed above)

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 (2), 84 (12), 104 (8), 87 (4), 22 (4), 37 (2), 101 (2), 97 (2), 45 (6), 43 (2), 133 (4), 153 (6), 173 (2), 107 (2).
The numbers: 29, 37, 101, 97, 43 and 173 all have exactly two factors and form the group of ‘prime’ numbers.
Teacher Notes: Referring to prime numbers as having exactly two factors removes any potential ambiguity with regards to whether or not 1 is prime.

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 (6), 30 (8), 90 (12), 45 (6), 50 (6), 60 (12), 120 (16), 72 (12), 25 (3), 49 (3), 81 (5), 144 (15), 441 (9), 82 (4), 24 (8), 720 (30)
This classification is harder than the previous one … students need to pick the ‘odd’ ones out. If students can see that 25, 49, 81, 144 and 441 all have an odd number of factors they should also identify that these numbers are perfect squares. Perfect squares are the only numbers that have an odd number of factors since each contains a ‘repeated’ factor.

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: 100,000; 200,000 and 300,000. Use these times to predict how long the program will take to count the factors for 500,000. Test your answer!

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

2,140,324,650,240,744,961,264,423,072,839,333,563,008,614,715,144,755,017,797,754,920,881,418,023,447, 140,136,643,345,519,095,804,679,610,992,851,872,470,914,587,687,396,261,921,557,363,047,454,770,520,8 05,119,056,493,106,687,691,590,019,759,405,693,457,452,230,589,325,976,697,471,681,738,069,364,894,69 9,871,578,494,975,937,497,937 [250 digits!] **Note**: This number is associated with RSA Encryption.

Counting factors for 100,000 takes approximately 1 second. [TI-Nspire CX II series]
Counting factors for 200,000 takes approximately 1.5 seconds.
Counting factors for 300,000 takes approximately 2 seconds.
Students should see that each 100,000 numbers take approximately 0.5 seconds. Assuming that the larger numbers do not take any longer, 500,000 should take approximately 2.5 to 3.0 seconds.

Timed result: $\approx$ 3.1 seconds.

A simple improvement to the algorithm (working to square-root of n), however to include the square-root of a number the maths module needs to be imported first.

Even using the relatively simple improvement to the algorithm, the really big number containing 250 digits, would take 10^{238} years.
This time estimation doesn't allow for additional routines required to handle the quantity of digits in the number. This time factor is why super-computers are required to work on such large numbers and helps explain why these numbers are used in the public key encryption process.

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.

Teacher Notes

To help build relevance to this investigation, consider engaging students in a discussion about the Enigma Machine, a powerful encryption tool created and used by the Germans during World War II. The Imitation Game [movie] is a wonderful way to show students just how important mathematics and mathematicians are to the world. The movie looks at Alan Turing and a team of mathematicians as they build a computing device to help solve the enigma code. While the Enigma machine was not based on prime numbers, it helps illustrate a long history, pre-dating computers, pertaining to the importance of encryption.

In the 21st century, encryption requirements have become ubiquitous. RSA encryption, invented in 1977 by Ron **R**ivest, Adi **S**hamir and Leonard **A**dleman is based on prime numbers. The story reads like a Hollywood movie! Initially the encryption process was supposed to be reserved only for military communications, however Ron, Adi and Leonard were so confident their system could not be hacked, they released it to the world, even explaining how it works! Their encryption system is still in use today!

Imitation Game Move Trailer

Interview with Ron Rivest

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*.]

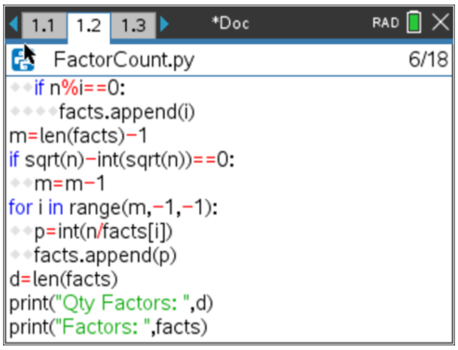
Continue the exploration, tabulate your results and record your thoughts, hypotheses, tests and reflections as you go. Documenting findings is an important part of the investigative process. Detectives may have many suspects in their initial investigations, however as more clues surface they develop 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.

Answers: Students should record their findings in a table. The clues prompt students to focus on the exponents rather than the bases. For example: $2^2 \times 3^2 = 36$ and $2^2 \times 5^2 = 100$ both have the same quantity of factors. They have the same indices but different bases, similarly with the other examples of 441, 3025 and 48841.

If students tabulate the indices and quantity of factors they should start to see the connection: {2, 2, 9}; {2, 3, 12}; {1, 2, 6} … adding one to each exponent and then calculating the product results in the quantity of factors.

Students should go beyond numbers with two prime factors and also check that the algorithm works for prime numbers.

Teacher Notes: A sample program loops up to the squareroot of the input number. Python does not have a square-root command so it must be imported first.



Euclid's Algorithm

Teacher Notes:

A PowerPoint slide show is provided with this activity as an introductory presentation for students to watch and help them understand how the algorithm works. The slides use 'lengths' to help explain why the algorithm works.

Following the 'lengths' examples, numbers are included to see how Euclid's algorithm translates to working with numbers

The calculator contains a command for the "Highest Common Factor" or "Greatest Common Divisor", it is however good for students to understand how the 'magic' happens.

The GCD command only compares 2 numbers, in the later stages of this activity, students extend their program to 3 numbers and then an entire list!

TI-Codes Lessons:

Unit 1 – Skill Builder 1

⇩

Unit 4 – Skill Builder 1

Commands:

- input
- while
- if
- else
- int (number types)
- print

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.

$70 = 2 \times 35 + 0$. [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.
Answer: 154

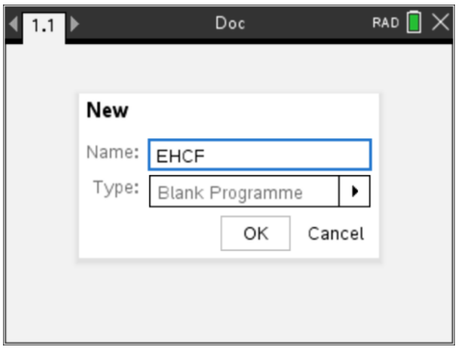
Writing a Program

Instructions:

Start a new document; insert a new Python program.

Add Python > **New**

Call the program: EHCF



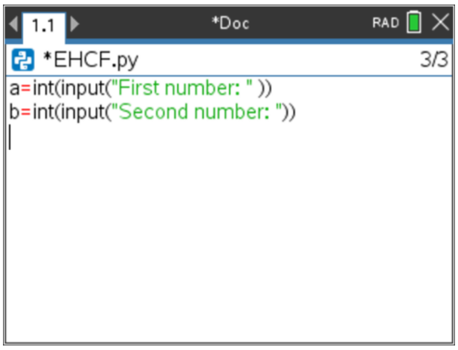
Insert two prompts for the integers 'a' and 'b'. The input type (integer) can be typed directly or entered via the menu.

Menu > **Built-ins** > **Type**

The input command can be typed directly or entered via the menu:

Menu > **Built-ins** > **I/O**

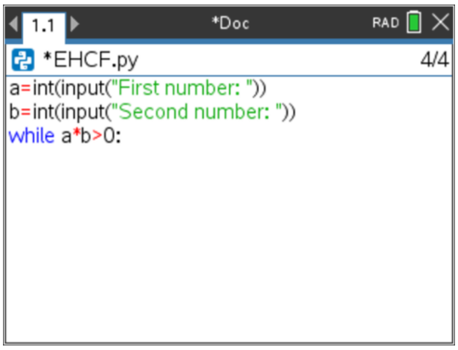
Include a text prompt for the user.



Euclid's algorithm ceases when either a = 0 or b = 0, an easy way to check this is: a x 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 x b $\neq$ 0.

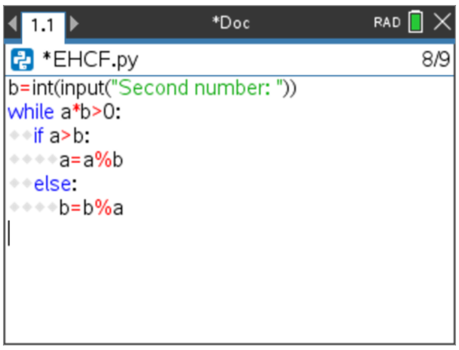
Menu > **Built-ins** > **Control > while..**



Modular arithmetic returns the remainder when a ÷ b (where a > b) so an if …else statement can be used to process Line #3 of Euclid's algorithm.

Menu > **Built-ins** > **Control > if … else**

Enter the condition, then press Tab to navigate to the instruction block and use the % sign for modular arithmetic, then Tab to the second block to complete the instruction block.

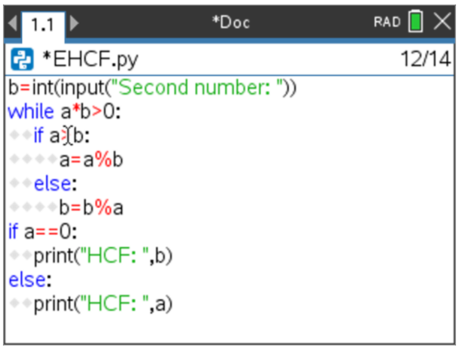


Author: P.Fox

That's the entire algorithm! The only thing remaining is to display the results. You could display as:

print(*a*,*b*)

Alternatively, use an if … else option to only display only the highest common factor rather than both a and b. (Shown opposite.)



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.

Answer: 33

Question: 3.

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

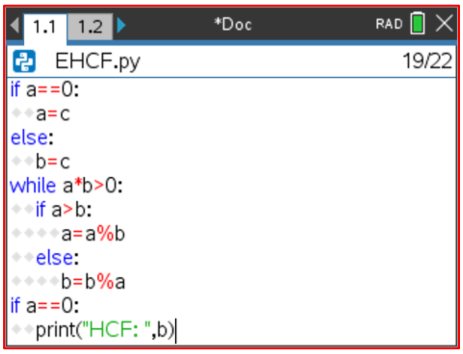
Answer: Answers will vary depending on what numbers student chose to explore and test.

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.

Answer: There are various ways students may edit their program to achieve the desired result. Students should also consider efficiency.



Teacher Notes: Sample Program

A simple addition to the original program is shown here. Following the original While loop, an IF statement is used test which variable, a or b, is equal to zero, the corresponding variable is then replaced with c.

If a = 0 **then** c is stored in a, otherwise c is stored in b.

The original While loop can then be pasted in its entirety.

Conceptually, copying and pasting the original loop leads students to an understanding that the original loop is repeated and therefore extending the program to a list of numbers involves an over-arching loop.

Students should provide a table of numbers that they have tested. Students should also think about how they generate the numbers to be tested. For example:

a = 29 x 35 = 1015 b = 41 x 35 = 1435 c = 97 x 35 = 3395

Generating the numbers as products with primes ensures the highest common factor of the three numbers is 35 in the above scenario.

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 **len** command can be used to determine the dimensions (quantity of numbers) entered into the list.

Answer:

Sample program opposite:

- While loop works for data entry with the "esc" key as the criteria to finish data entry
- The quantity of entries in the 'data' variable relates to the number of times the HCF loop needs to operate.
- Original while loop is also included with successive entries called from the 'data' variable.

For very large data sets it would be worthwhile sorting a list in descending order.

```
print("Enter your numbers, ESC to finish")
data=[]
a=None
while a != "esc":
◆◆a=input("Number: ")
◆◆if a!="esc":
◆◆◆◆a=int(a)
◆◆◆◆data.append(a)
d=int(len(data))
a=data[0]
print(a)
for n in range(1,d):
◆◆b=data[n]
◆◆print(a,b)
◆◆while a*b>0:
◆◆◆◆if a>b:
◆◆◆◆◆◆a=a%b
◆◆◆◆else:
◆◆◆◆◆◆b=b%a
◆◆if a==0:
◆◆◆◆a=b
if a==0:
◆◆print("HCF: ",b)
else:
◆◆print("HCF: ",a)
```

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

Teacher Notes:

A PowerPoint slide show is provided with this activity as an introductory presentation for students to watch and help them understand how the algorithm works. The slides work progressively through the number from 1 to n, capturing any numbers that are co-prime with the original number n.

There is no calculator command for the Euler Totient function, however there is a short cut approach using the prime factorisation of a number. Once students have completed their program, they use the prime factor approach and compare it to their program.

This coding activity also introduces the notion of a 'function' and 'sub-routine'. The Euler Totient function relies upon the 'highest common factor'. Students can use their HCF routine from the previous activity, or import the 'math' module and use the calculator's built-in HCF routine. The activity is written to utilise the previous activity to support the concept of 'chunking' (educational neuroscience)

TI-Codes Lessons:

Unit 1 – Skill Builder 1

⇩

Unit 4 – Skill Builder 1

Commands:

- input
- for (range)
- if
- print
- int (number types)
- def function
- while
- % (modular arithmetic)

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.

Answer: (Sample)

Request input
Reset Counter = 0
Loop from 1 to n
    If HCF(n,1) = 1 Then < increase counter >
End Loop

Author: P.Fox

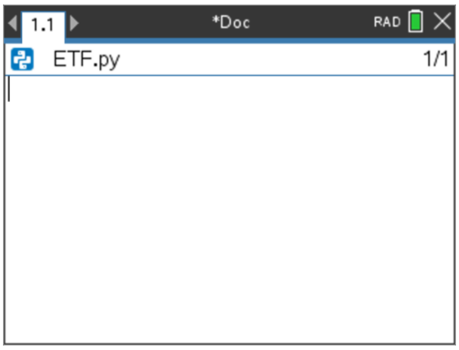
Writing a Program

Instructions:

Start a new document; insert a new Python program.

Add Python > **New**

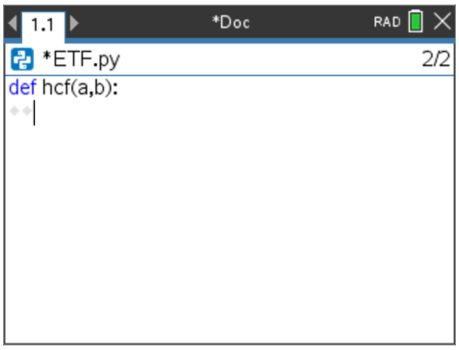
Call the program: ETF



The Euler Totient Function counts the quantity of numbers that are co-prime up to the specified number. When two numbers are co-prime their highest common factor is one, it therefore makes sense to use Euclid's algorithm to check the highest common factor. To do this efficiently, Euclid's algorithm can be defined as a function.

Built-ins > **Functions** > **def function()**

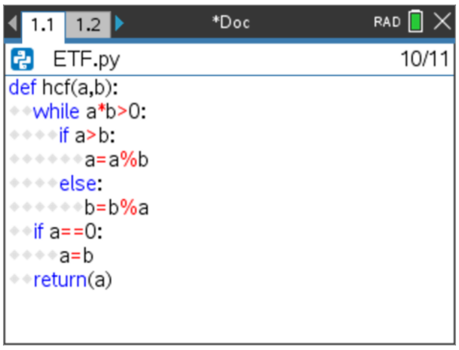
The function requires two parameters, the two numbers for which the highest common factor will be returned.



Euclid's algorithm for the Highest Common Factor can now be deployed through this function as per the activity on Euclid's algorithm. The only difference here is at the end of the function, 'return(a)', this is the value returned once the function has been called.

Note:

When the program runs, nothing happens with the function until it is called from the program.



The program needs to count the quantity of numbers that are co-prime with the selected (input) number. If two numbers are co-prime, their highest common factor is 1.

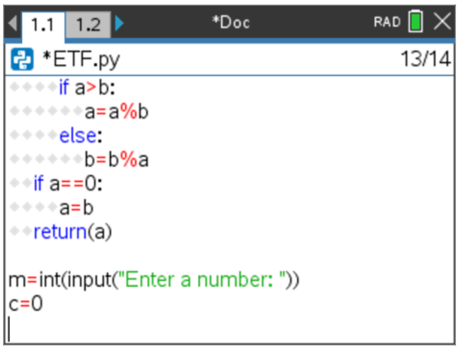
Start by requesting an input value and setting a counter equal to 0

m = int(input("Enter a number: ")

c = 0

Note:

Variables 'a' and 'b' are used in the function, so it is best to avoid using them anywhere else in the program. Longer, more meaningful variable names can be used but keep them brief to avoid long lines of code.



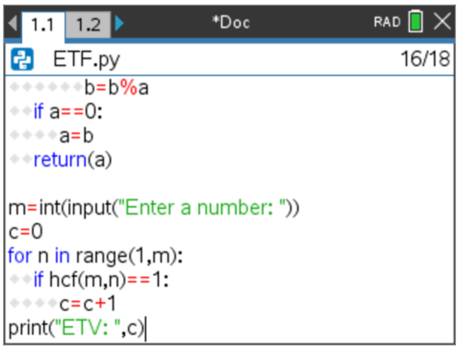
Author: P.Fox

The last part of the program is to scan the numbers from 1 to the designated value (m)* for numbers that are co-prime.

Each time one of these numbers is found, the counter increments by 1.

Note:

The loop will halt at 'm – 1', the hcf(m,m)≠1 so this last check would not change the counter value c.



Question: 2.

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

Answer: The program returns the correct values for all numbers.

Question: 3.

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

Answer: The Euler Totient function for a prime number 'n', returns the value n – 1.

Question: 4.

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

Answer: $\frac{30}{\varphi(30)}=\frac{30}{8}=3.75$, $\frac{60}{\varphi(60)}=\frac{60}{16}=3.75$ and $\frac{90}{\varphi(90)}=\frac{90}{24}=3.75$

Other values for n with the same fraction (ratio) include: 120, 150, 180, 240, 270, 300 but 210 and 330 have very different results.

Teacher Notes: Students may sample a selection of multiples of 30 (as above) and jump to a conclusion too quickly if they miss 210 and 330. The clue lies in the prime factorisation of the multiples of 30.
30 = 2 x 3 x 5; 60 = 2^2 x 3 x 5; 90 = 2 x 3^2 x 5; 120 = 2^3 x 3 x 5; 150 = 2 x 3 x 5^2; 180 = 2^2 x 3^2 x 5; however, 210 = 2 x 3 x 5 x 7 which results in prime factorisation involving 4 prime factors, specifically a departure from: 2^a x 3^b x 5^c. Students should establish that for the Euler Totient function, the bases that are important not the exponents.

Question: 5.

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

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

Answer: φ(100) = 40. 100 x ½ x 4/5 = 40. The results are the same.

Question: 6.

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

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

Answer: φ(1125) = 600. 1125 x 2/3 x 4/5 = 600. The results are the same!

Author: P.Fox

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.

Answer: Results will vary, depending on the values that students chose, however in each case the answers will be the same.

Question: 8.

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

Answer: The prime factorisation for 30, 60 and 90 are of the form: $2^a \times 3^b \times 5^c$. The prime factorisation approach for calculating the Euler Totient function can be considered as two parts, the first part being the original number, the second, a combination of the prime factors (bases only). By dividing out the original number, we are only left with a calculation involving the prime factors, ignoring duplicity.

Question: 9.

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

Answer: Each prime factor removes the corresponding fraction of the remaining numbers.
Example, if 2 is a prime factor of some number 'n', then ½ of the numbers up to (and including n) will have a factor in common (2). Similarly, if 3 is a prime factor of 'n', then 1/3 of the remaining numbers will also have a factor in common with 'n'. The co-primes will be the complement of these calculations.

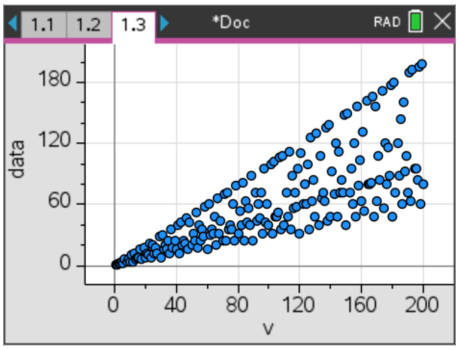
Investigation

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.

Note: Use the ti-system import module to share data from the program with the TI-Nspire document.

Answer: A sample of the Euler Totient program for a range of numbers is shown opposite. The program generates all the Euler Totient values from lower to upper and stores them in a list called: 'data'.

To generate a scatterplot, the whole numbers from 'lower' to 'upper' would need to be stored in a list, furthermore, the TI-System module would need to be installed so the variables from the Python shell can be shared to the TI-Nspire document variables.



ETFL.py

```
from ti_system import *
#Euclids Algorithm for HCF
def hcf(a,b):
  while a*b>0:
    if a>b:
      a=a%b
    else:
      b=b%a
  if a==0:
    a=b
  return(a)

lower=int(input("Enter lowest number: "))
upper=int(input("Enter highest number: "))
data=[]
ns=[]
for n in range(lower,upper+1):
  print(n)
  ns.append(n)
for n in range(lower,upper+1):
  c=0
  for j in range(1,n+1):
    if hcf(n,j)==1:
      c=c+1
  data.append(c)
print(data)
store_list("etvs",data)
store_list("number",ns)
```

There is a clear line of data above which there are no points. The points along this line are the prime numbers. $\varphi(p) = p - 1$, where p is prime.

There is also a 'line' of points assembled around $y = \frac{1}{2}(x-2)$. This collection of points corresponds to prime factorisations of the form: 2 x p, where p is prime.

With the exception of $\varphi(1) = 1$ and $\varphi(2) = 1$, $\varphi(n)$ is even. Why? This can be seen from the prime factorisation calculation method for the Euler Totient function. Consider *n* as even and then *n* as odd.

Another set of points of particular interest are those at the bottom of the graph:

1, 2, 3, 4, 6, 8, 10, 12, 14, 18, 20, 24, 30, 36, 42, 48, 60

The highly composite numbers are a subset of these numbers.

Number:	2	3	4	6	8	10	12	14
Euler Totient:	1	2	2	2	4	4	4	6
Prime Factorisation:	2	3	2^2	2×3	2^3	2×5	$2^2\times3$	2×7
Number:	18	20	24	30	36	42	48	60
Euler Totient:	6	8	8	8	12	12	16	16
Prime Factorisation:	2×3^2	$2^2\times5$	$2^3\times3$	$2\times3\times5$	$2^2\times3^2$	$2\times3\times7$	$2^4\times3$	$2^2\times3\times5$

Expressing each calculation using the prime factorisation method helps show why these numbers fall along the bottom of the graph.

Highly Composite Numbers

Teacher Notes:

A PowerPoint slide show is provided with this activity as an introductory presentation for students to watch and help them understand highly composite numbers.

Students may wish to call upon previous programs that count factors.

TI-Codes Lessons:

Unit 1 – Skill Builder 1

⇩

Unit 4 – Skill Builder 1

Commands:

- input
- for (range)
- if
- print
- int (number types)
- def function
- [ ] (create a list)
- Append (add elements to a list)
- % (modular arithmetic)
- Import module

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’.

Answer: Answers will vary, students must use a ‘record holder’ to track the current highly composite number.

Sample: Record:= 0

  Input <Number> n

  Loop start = 1, finish = n

    If factor_count(loop_counter) > record Then

      Increase record

      Store loop_counter

  End Loop

  Display Highly Composite numbers <stored_loop counters>

Teacher Notes:

Notice how referencing the “factor count” program simplifies the entire program. In programming languages this is often referred to as a sub-routine. In educational neuroscience this is referred to as ‘chunking’, putting procedures or a collection of procedures into bite size pieces making them easier to digest. In mathematics this might be referring to “solving simultaneously” as one step in a much larger problem. Simultaneous equation would have been taught as a topic unto itself, however, if students understand what ‘solving simultaneously’ means, they are able to refer to it as a single step in a much bigger problem.

Writing a Program

Instructions:

Start a new document; insert a new Python program.

Add Python > **New**

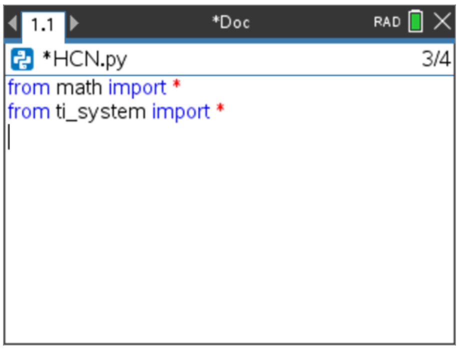
Call the program: HCN

To make the program efficient, it is desirable to have access to the 'square-root' function. Import the 'math' module.

Math > **from math import**

To access results outside the Python shell, import the TI-System module.

More Modules > **TI-System** > **from ti-system import**

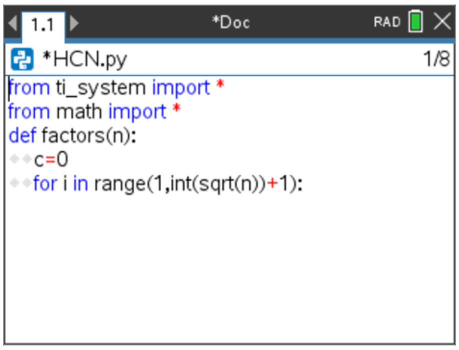


Creating a function to efficiently determine the quantity of factors will make the main program much easier.

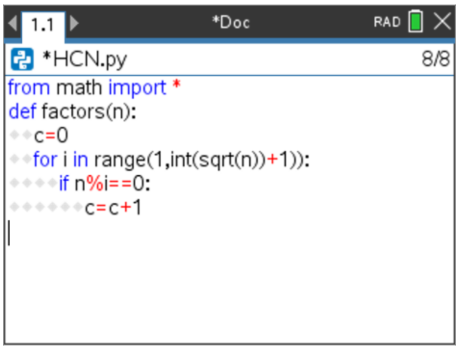
Define a function called "factors" with input 'n':

Built-ins > **functions** > **def function()**

A counter (c) will be used to count each factor and a loop to search for the factors. The loop only needs to go to the square-root of the chosen number, but a final check will be necessary in the event that the original number is a perfect square.



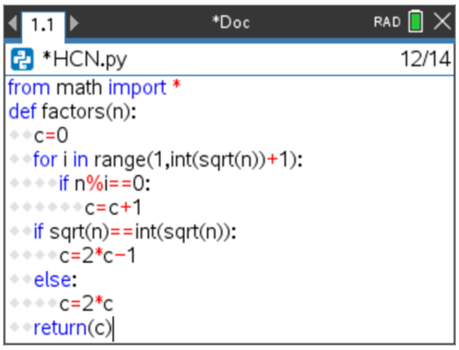
The loop checks if the current number (n) is divisible using modular arithmetic (%), if there is no remainder, then 'i' must be a factor of 'n', so the counter is increased by one.



Once the loop has finished, a check must be performed to see if the original number was a perfect square. If the original number was a perfect square, doubling the quantity of factors would count the square-root twice.

If the original number was not a perfect square, then the quantity of factors is doubled as all the factors counted to date have a 'partner'.

Finally, the quantity of factors (c) is returned to the program.



Author: P.Fox

Several variables need to be initialised at the start of the program.

- QTY = The quantity of factors for the highly composite number
- HCNS = Highly Composite Numbers
- Record = Quantity of factors for the current HCN.

The first highly composite number '1' is seeded into the variables as it is the only 'odd' highly composite number.

Note: "qty' and 'hcns' will hold a list of numbers that will be continually updated.

```
1.1    *Doc    RAD
*HCN.py    17/18
  if sqrt(n)==int(sqrt(n)):
    c=2*c-1
  else:
    c=2*c
  return(c)

qty=[1]
hcns=[1]
record=1
p=int(input("Number: "))
```

The loop can start at 2 since the first highly composite number (1) has already been stored. As all subsequent HCN's are even, the step counter can be set at 2.

The first instruction in the loop is to store the quantity of factors in variable 'n'; if this quantity is larger than the current record, the current record is updated and the 'qty' and 'hcns' lists are updated.

Note: The append command adds the specified value to the end of the specified list.

```
1.1    *Doc    RAD
*HCN.py    24/24
qty=[1]
hcns=[1]
record=1
p=int(input("Number: "))
for j in range(2,p+1,2):
  n=factors(j)
  if n>record:
    record=n
    qty.append(n)
    hcns.append(j)
```

Once the loop is finished, all the highly composite numbers have been stored and can therefore be displayed and transferred to variables that can be accessed by the current document.

More Modules > TI System > store_list("name",list)

"name" represents the name of the variable in the current document.

"list" refers to the list in the current program (Python shell).

The program is now complete and ready to run.

```
1.1    *Doc    RAD
*HCN.py    24/27
record=1
p=int(input("Number: "))
for j in range(2,p+1,2):
  n=factors(j)
  if n>record:
    record=n
    qty.append(n)
    hcns.append(j)
print(hcns)
store_list("hcns",hcns)
store_list("number",qty)
```

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.

Answer: Highly Composite Numbers: 1, 2, 4, 6, 12, 24, 36, 48 & 60.
Quantity of factors for each: 1, 2, 3, 4, 6, 8, 9, 10, 12.

Question: 3.

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

Answer:

HCNs	1	2	4	6	12	24	36	48	60
Qty Factors	1	2	3	4	6	8	9	10	12
HCNs	120	180	240	360	720	840			
Qty Factors	16	18	20	24	30	32			

Note: Students may be surprised that 144 is not a highly composite number given that $144 = 12^2$.

Author: P.Fox

Question: 4.

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

Answer:

HCNs	1	2	4	6	12	24	36	48
Prime Factorisation	1	2	2^2	2×3	$2^2\times3$	$2^3\times3$	$2^2\times3^2$	$2^4\times3$
HCNs	60	120	180	240	360	720	840	
Prime Factorisation	$2^2\times3\times5$	$2^3\times3\times5$	$2^2\times3^2\times5$	$2^4\times3\times5$	$2^3\times3^2\times5$	$2^4\times3^2\times5$	$2^3\times3\times5\times7$	

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.

Answer: Based on the previous prime factorisations... $2^3\times3$ went to $2^2\times3^2$, $2^3\times3\times5$ went to $2^2\times3^2\times5$, so it is likely that $2^3\times3\times5\times7$ will transition to: $2^2\times3^2\times5\times7$ (1260) which has 36 factors. The current calculator program validates this answer (prediction).

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?

Answer: There is a LOT to explore here, famous mathematicians such as Ramanujan explored HCNs, indeed, the back story makes for interesting reading. Prime factorisation can certainly act as a guide to predicting future HCN's.

Author: P.Fox

Current HCN:

$2^2 \times 3^2 \times 5 \times 7 = 1260$ (36 factors)

Based on previous HCN's there are a couple of options for the next HCN:

- $2^4 \times 3 \times 5 \times 7 = 1680$ (40 factors) [Increase exponent of 2, reduce exponent of 3]
- $2^3 \times 3^2 \times 5 \times 7 = 2520$ (48 factors) [Increase exponent of 2]
- $2^2 \times 3^2 \times 5^2 \times 7 = 6300$ (54 factors) [Increase exponent of 5]
- $2^2 \times 3^2 \times 5 \times 7 \times 11 = 13860$ (72 factors) [Introduce another prime factor]

Note: Increasing the exponent of 3 should not be a consideration. The result would produce the same quantity of factors as increasing the exponent of 2, but the numerical result would be greater.

Each option introduces more factors, however the numerical expense of repeating the 5 or introducing the next prime factor are too much (at this stage). The first option multiplies the previous HCN by 4/3. The second option multiplies the previous HCN by 2.

Student's should be confident of their HCN prediction which can be validated by the existing program structure. Further exploration using the existing program structure however will become problematic as the algorithm searches every number.

Current HCN:

$2^4 \times 3 \times 5 \times 7 = 1680$ (40 factors)

The next HCN is slightly less predictable. Using data collected so far, the prime factors 5 and 7 were introduced as similar junctions.

- $2 \times 3 \times 5 \times 7 \times 11 = 2310$ (32 factors) [Decrease all exponents, introduce another prime factor]
- $2^3 \times 3^2 \times 5 \times 7 = 2520$ (48 factors) [Decrease exponent of 2, increase exponent of 3]
- $2^2 \times 3 \times 5 \times 7 \times 11 = 4620$ (48 factors) [Decrease exponent of 3, introduce another prime factor]

Introducing the prime factor (11) is "too expensive" as a trade off with regards to the final calculation versus additional factors, indeed the first option produces less factors than the previous HCN.

Students should be reasonably confident that the next HCN is therefore 2520.

Students may also consider 'reverse engineering' a solution here by consideration of the quantity of factors. The missing options for the quantity of factors are: 41, 42, 43, 44, 45, 46 and 47. Using their understanding of how the quantity of factors can be calculated, HCNs with 41, 43 or 47 factors clearly don't work.

Consider: 42 = 6 x 7 or 2 x 3 x 7, the exponents could be: {5, 6} or {2, 3, 5}. The logical approach would be to place the largest exponents on the smallest bases:

- $2^6 \times 3^5 = 15552$ (42 factors)
- $2^5 \times 3^3 \times 5^2 = 21600$ (42 factors)

Neither of these results are satisfactory.

Consider: 44 = 11 x 4, a number with 44 factors could be produced using exponents of 10 and 3 only.

- $2^{10} \times 3^3 = 27648$.

Consider a number with 45 factors, it must be a perfect square since it has an odd number of factors!

Author: P.Fox

Since 45 = 9 x 5 = 3 x 3 x 5, the exponents could be either {8, 4} or {2, 2, 4}, which means the following numbers would be options:

- $2^8 \times 3^4 = 20736$ [$144^2 = 20736$]
- $2^4 \times 3^2 \times 5^2 = 3600$ [$60^2 = 3600$ and 60 is a previous HCN]

In the case of 3600, we note that 2520 has more factors. Why? The prime factorisation of 2520 involves the introduction of the prime factor 7.

Students should quickly realise that a number with 46 factors would require exponents of 22 and 1, the computed result would be much too large! This leads to the conclusion that then next HCN after 1680 must have 48 factors.

Current list of highly composite numbers:

1, 2, 4, 6, 12, 24, 36, 48, 60, 120, 180, 240, 360, 720, 840, 1260, 1680, 2520

Where $2520 = 2^3 \times 3^2 \times 5 \times 7$ (48 factors)

Now the highly composite numbers themselves provide a clue as to how many factors the next highly composite number might contain: 60 (factors).

$60 = 2^2 \times 3 \times 5$

This means the exponents could be:

- 1, 1, 2, 4
- 3, 2, 4

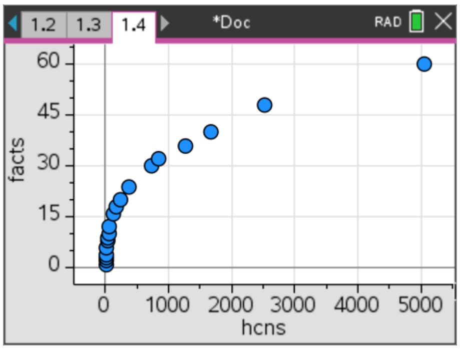
Applying these exponents in the appropriate order means the next HCN could be:

- $2^4 \times 3^2 \times 5 \times 7 = 5040$
- $2^4 \times 3^3 \times 5^2 = 10800$

At this point in time it is worth exploring a graph of the HCNs versus the quantity of factors.

The relationship looks almost logarithmic ... but it's not.

Programming



The existing HCN program can be modified by starting the search for the next series of HCNs at the last known value. The search loop should also use an increment of at least 30. For example, if the most recent HCN = 5040, it is not necessary to check 5041, we know from the prime factorisation, the next HCN will have factors of 2, 3 and 5. Once students are confident that 7 will be included in all subsequent HCN's, the step size can be 210 and eventually 210 x 11 = 2310.

Primorial Factorisation

Students may also be encouraged to explore primorial representation. Primorial (Harvey Dubner) is a mixture of prime numbers and factorial.

Example:

Factorial: 5! = 5 x 4 x 3 x 2 x 1 = 120

Primorial: 5# = 5 x 3 x 2 x 1 = 30 (Product of primes less than or equal to 5)

The use of primorial becomes 'obvious' when considering the prime factorisation of a number, particularly highly composite numbers.

Example:

$720720 = 2^4 \times 3^2 \times 5 \times 7 \times 11 \times 13 = 2^2 \times (3 \times 2) \times (13 \times 11 \times 7 \times 5 \times 3 \times 2) = 2^2 \times 3\# \times 13\#$ or $2^2 \times 6 \times 30030$