Highly Composite Numbers

Teachers Notes and Answers

7 8 **9** 10 11 12

TI-Nspire™ Activity Student 180 min

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.

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>
```

Author: P. Fox

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**

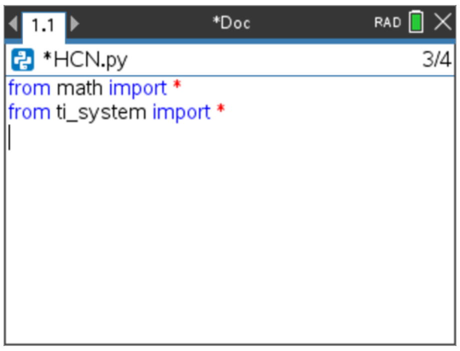
Call the program: HCN

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**



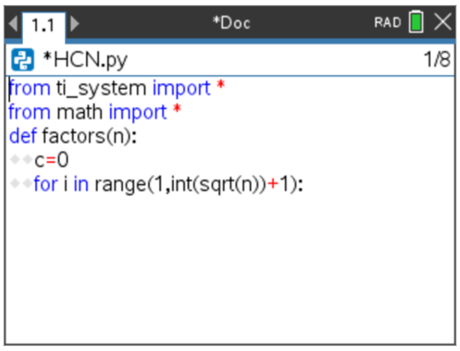
```
from math import *
from ti_system import *
```

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’:

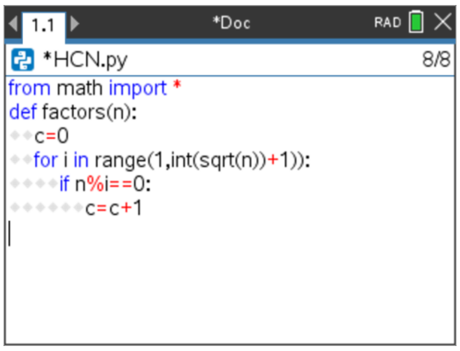
Built-ins > **functions** > **def function()**

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.



```
from ti_system import *
from math import *
def factors(n):
  c=0
  for i in range(1,int(sqrt(n))+1):
```

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.



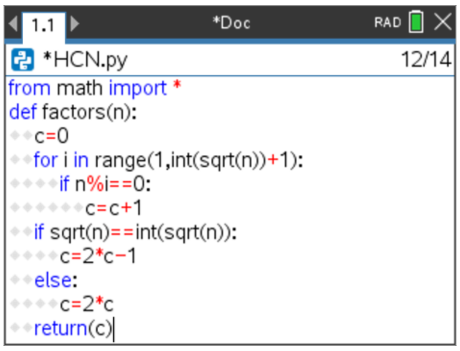
```
from math import *
def factors(n):
  c=0
  for i in range(1,int(sqrt(n))+1)):
    if n%i==0:
      c=c+1
```

Author: P. Fox

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.



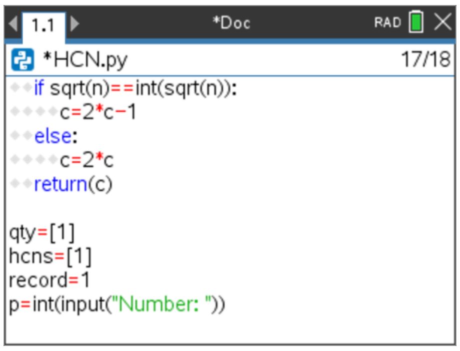
```
1.1  *Doc  RAD
*HCN.py  12/14
from math import *
def factors(n):
  c=0
  for i in range(1,int(sqrt(n))+1):
    if n%i==0:
      c=c+1
  if sqrt(n)==int(sqrt(n)):
    c=2*c-1
  else:
    c=2*c
  return(c)
```

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.



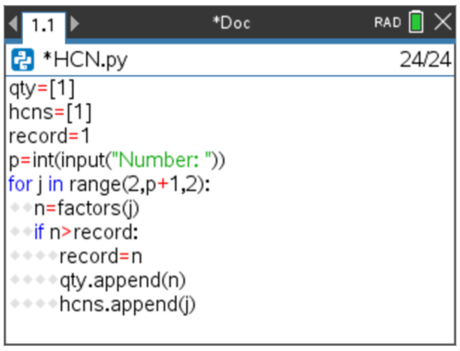
```
1.1  *Doc  RAD
*HCN.py  17/18
  if sqrt(n)==int(sqrt(n)):
    c=2*c-1
  else:
    c=2*c
  return(c)

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)
```

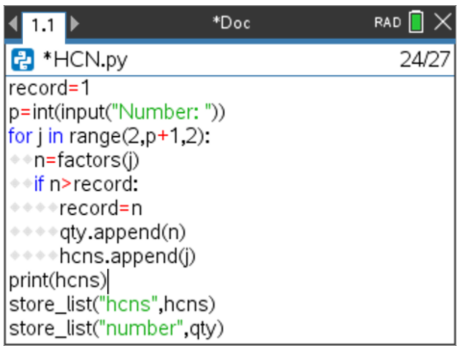
Once the loop is finished, all the highly composite numbers have been stored and can therefore be displayed and transferred to variables that can be accessed by the current document.

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.

Author: P. Fox

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.

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.

Current HCN:

$2^2\times3^2\times5\times7 = 1260$ (36 factors)

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

- $2^4\times3\times5\times7 = 1680$ (40 factors) [Increase exponent of 2, reduce exponent of 3]
- $2^3\times3^2\times5\times7 = 2520$ (48 factors) [Increase exponent of 2]
- $2^2\times3^2\times5^2\times7 = 6300$ (54 factors) [Increase exponent of 5]
- $2^2\times3^2\times5\times7\times11 = 13860$ (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:

$2^4\times3\times5\times7 = 1680$ (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.

- $2\times3\times5\times7\times11 = 2310$ (32 factors) [Decrease all exponents, introduce another prime factor]
- $2^3\times3^2\times5\times7 = 2520$ (48 factors) [Decrease exponent of 2, increase exponent of 3]
- $2^2\times3\times5\times7\times11 = 4620$ (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:

- $2^6\times3^5 = 15552$ (42 factors)
- $2^5\times3^3\times5^2 = 21600$ (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.

- $2^{10}\times3^3 = 27648$.

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

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:

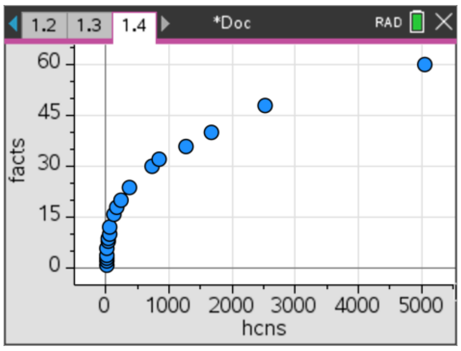
- 1, 1, 2, 4
- 3, 2, 4

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 $2^2 \times 6 \times 30030$