The CS 5 Times

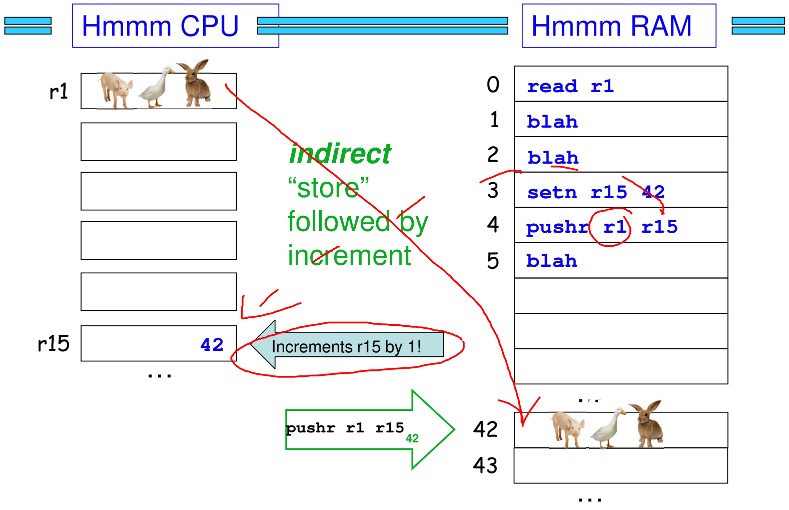
Penguin/Pig Gang Fight Brings Violence to Claremont

Victim of attack

Claremont (Farm News): Gang activity reached a new low when an angry group of penguins viciously beat a pig, a goose, and a duck in an apparently unprovoked attack. Witnesses said that the gang of birds waddled up to the victims, shouting something about an "invasion" and threatening that they would "make bacon bits" and "have a bit of foie gras."

At first, the farm animals attempted to defend themselves, but they found themselves outnumbered and were forced to retreat into a nearby business, the Claremont Village Grill. The owner of the business, Chef Boy Are We Hungry, welcomed them with open arms. The pig soon escaped through a back door, but the duck and goose have not been seen. Relatives now fear the worst.

pushr Goes TO Memory



Hmmm CPU

Hmmm RAM

r1

r15 42

. . .

indirect "store" followed by increment

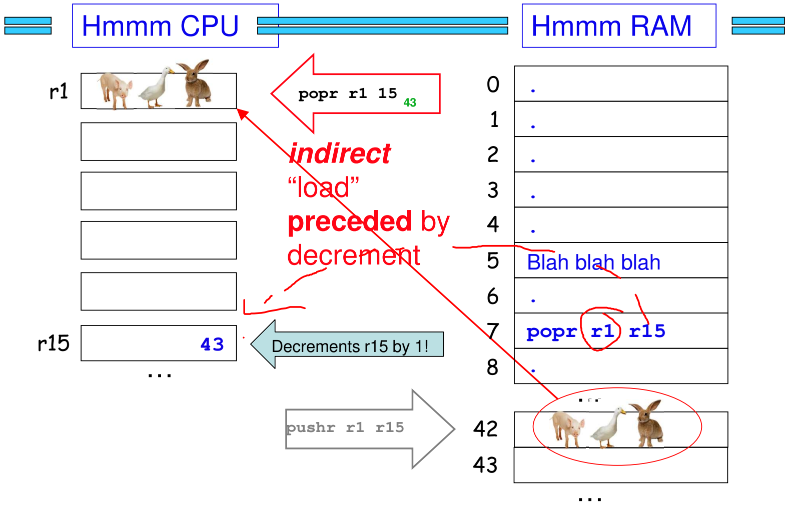
Increments r15 by 1!

pushr r1 r15 42

0	read r1
1	blah
2	blah
3	setn r15 42
4	pushr r1 r15
5	blah
42	
43	

. . .

popr Comes FROM Memory



Hmmm CPU

Hmmm RAM

r1

r15 43

. . .

popr r1 15 43

indirect "load" **preceded** by decrement

Decrements r15 by 1!

pushr r1 r15

0	.
1	.
2	.
3	.
4	.
5	Blah blah blah
6	.
7	popr r1 r15
8	.
42	
43	

. . .

calln = setn + jumpn!

A function call in python:

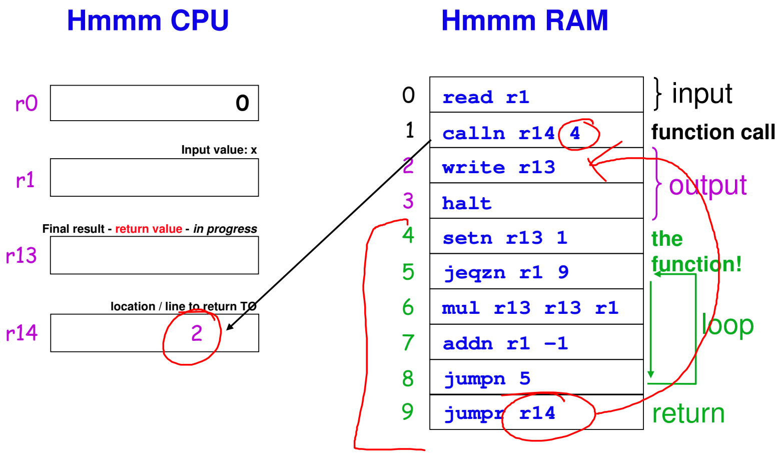
```
def main():
    r1 = input()
    result = factorial(r1)
    print(result)


def factorial(r1):
    # do work
    return result
```

puts NEXT line # into r14, then jumps to line 4

Hmmm's **call** operation:

```
0    read r1
1    calln r14 4
2    write r13
3    halt

4    do stuff and
5    answer in r13
6    jumpr r14
```

Factorial: *Function Call!*



Hmmm CPU

r0	0
r1	Input value: x
r13	Final result - return value - *in progress*
r14	location / line to return TO 2

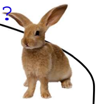
Hmmm RAM

```
0   read r1          } input
1   calln r14 4      function call
2   write r13        } output
3   halt
4   setn r13 1       the function!
5   jeqzn r1 9
6   mul r13 r13 r1   loop
7   addn r1 -1
8   jumpn 5
9   jumpr r14        return
```

Function Calls…

```
def main():
    r1 = input()<-----     r1=3
    r13 = emma(r1) <---   emma(3)
    r13 = r13 + r1
    print(r13)
    return

def emma(r1):  <-----     r1=3
    r1 = r1 + 1 <-----     r1=4
    r13 = sarah(r1)<---  sarah(4) r13=47
    r13 = r13 + r1 <---   r13=??
    return r13

def sarah(r1): <-----     r1=4
    r1 = r1 + 42 <-----    r1=46
    r13 = r1 + 1 <-----   r13=47
    return r13    <-----return(47)
```

Chew on this…



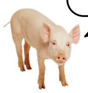
Function Calls…

```
def main():
    r1 = input()<-----     r1=3
    r13 = emma(r1) <---   emma(3)  r13=51
    r13 = r13 + r1 <---   r13=??
    print(r13)
    return

def emma(r1):  <-----     r1=3
    r1 = r1 + 1 <-----     r1=4
    r13 = sarah(r1)<---  sarah(4) r13=47
    r13 = r13 + r1 <---   r13=51
    return r13    <-----return(51)

def sarah(r1): <-----     r1=4
    r1 = r1 + 42 <-----    r1=46
    r13 = r1 + 1 <-----   r13=47
    return r13    <-----return(47)
```

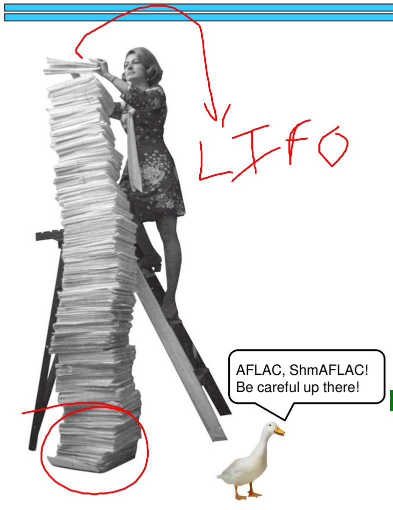
Function Calls…

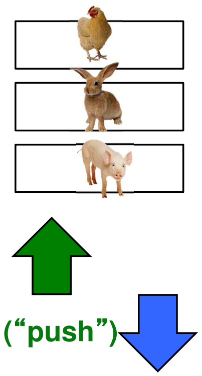
```
def main():
    r1 = input()<-----     r1=3
    r13 = emma(r1) <---   emma(3)   r13=51
    r13 = r13 + r1 <---   r13=54
    print(r13)
    return                 54

def emma(r1):  <-----     r1=3
    r1 = r1 + 1 <-----     r1=4
    r13 = sarah(r1)<---  sarah(4) r13=47
    r13 = r13 + r1 <---   r13=51
    return r13    <-----return(51)

def sarah(r1): <-----     r1=4
    r1 = r1 + 42 <-----    r1=46
    r13 = r1 + 1 <-----   r13=47
    return r13    <-----return(47)
```

Cool, but how does this work!?



The Stack!





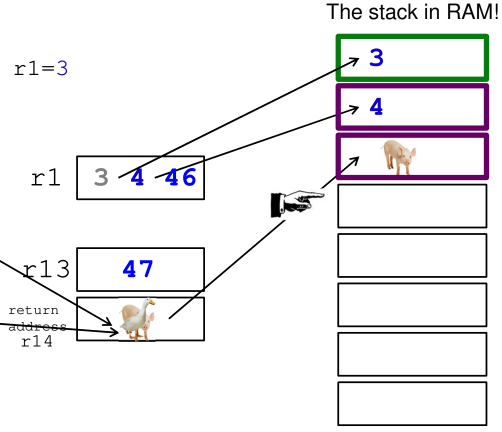
Insert (“push”)

Remove (“pop”)

Function Calls…

Hmmm code up here!

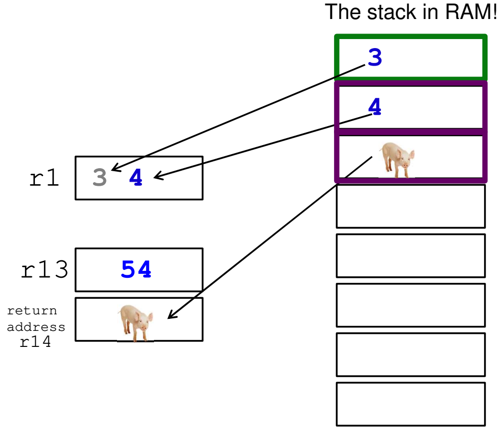
```
def main():
    r1 = int(input())  r1=3
    r13 = emma(r1)
    r13 = r13 + r1
    print(r13)
    return

def emma(r1):
    r1 = r1 + 1
    r13 = sarah(r1)
    r13 = r13 + r1
    return r13

def sarah(r1):
    r1 = r1 + 42
    r13 = r1 + 1
    return r13
```



Function Calls…

Hmmm code up here!

```
def main():
    r1 = int(input())
    r13 = emma(r1)
    r13 = r13 + r1
    print(r13)
    return

def emma(r1):
    r1 = r1 + 1
    r13 = sarah(r1)
    r13 = r13 + r1
    return r13

def sarah(r1):
    r1 = r1 + 42
    r13 = r1 + 1
    return r13
```



Implementing Functions

(1) Use **r15** as the ***stack pointer***. **setn r15 42**

or some other large-enough value

(2) Before the function call, **pushr r1 r15**
Store all “precious belongings” to the stack—and increment r15

store the return address **r14** and the inputs: **r1**, (**r2**), (**r3**)

(3) Get **r1**, (**r2**), (**r3**), … ready as function “arguments."

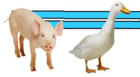
(4) Make the function call. **calln r14 #**
The result, if any, will be in r13.

line # of the function

(5) After the function call, **popr r1 r15**
Load “precious belongings” back from the stack (in reverse order)

for each item stored

Now Without Pigs and Geese!



It was better with pigs and geese!

```
def main():
    r1 = input()
    r13 = emma(r1)
    r13 = r13 + r1
    print(r13)
    return
```

```
00 setn r15 42     # set stack pointer to 42
01 read r1         # start of main
02 pushr r1 r15    # store r1 on the stack
03 calln r14 10    # call emma
04 popr r1 r15     # load r1 from the stack
05 add r13 r13 r1  # r13 = r13 + r1
06 write r13
07 halt
08 nop
09 nop
```

save

restore

do nothing

```
def emma(r1):
    r1 = r1 + 1
    r13 = sarah(r1)
    r13 = r13 + r1
    return r13
```

```
10 addn r1 1       # start of emma!
11 pushr r1 r15    # store r1 on the stack
12 pushr r14 r15   # save return addr on stack
13 calln r14 20    # call sarah
14 popr r14 r15    # load ret addr from stack
15 popr r1 r15     # load r1 from the stack
16 add r13 r13 r1  # r13 = r13 + r1
17 jumpr r14       # return!
18 nop
19 nop
```

save

restore

```
def sarah(r1):
    r1 = r1 + 42
    r13 = r1 + 1
    return r13
```

```
20 addn r1 42      # start of sarah!
21 setn r2 1       # put 1 in a register
22 add r13 r1 r2
23 jumpr r14       # return
```

Recursion?

```
def fac(N):

    if N <= 1:
        return 1

    else:
        return N * fac(N-1)
```

"The Stack"

Remembers all of the individual calls to fac

$$\underbrace{\texttt{fac(5)}}_{\texttt{5 * }\underbrace{\texttt{fac(4)}}_{\texttt{4 * }\underbrace{\texttt{fac(3)}}_{\texttt{3 * }\underbrace{\texttt{fac(2)}}_{\texttt{2 * }\underbrace{\texttt{fac(1)}}_{\texttt{1}}}}}}$$

Factorial via Recursion…

Python

```
n = int(input())
answer = fac(n)
print(n, answer)

def fac(n):
  """Recursive
     factorial!"""
  if n == 0:
    return 1
  else:
    res = fac(n-1)
    return n*res
```

r1 (N)	3 ~~2~~ ~~1~~ ~~0~~	
r13(Res)	~~1~~ ~~1~~ ~~2~~ 6	Return value
r14	~~P~~ ~~4~~ ~~6~~ ~~6~~	Return address

First let's try N=0 and then N=3

jumpr r14

jumpr r14

This is same as `return n*fac(n-1)` but done in 2 steps…

Python Hmmm

Python

match each piece of Python code with the Hmmm assembly code that implements it.

Hmmm

b → r1

```
n = int(input())
answer = fac(n)
print(answer)

def fac(n):
  """Recursive factorial!"""
  if n==0:
    return 1

  else:
    res = fac(n-1)
    return n*res
```

```
00 setn r15 42
01 read r1
02 calln r14 5
03 write r13
04 halt

05 jnezn r1 8
06 setn r13 1
07 jumpr r14

08 pushr r1 r15
09 pushr r14 r15
10 addn r1 -1
11 calln r14 5
12 popr r14 r15
13 popr r1 r15
14 mul r13 r13 r1
15 jumpr r14
```

Try to align the Python code to the Hmmm code…

as shown in the next slide…

Python Hmmm

Python

```
n = int(input())
answer = fac(n)
print(answer)

def fac(n):
  """Recursive factorial!"""
  if n==0:
    return 1

  else:
    res = fac(n-1)
    return n*res
```

r13 is the **answer**

r14 is the "return address"

r15 is the "stack pointer"

Hmmm

```
00 setn r15 42
01 read r1
02 calln r14 5
03 write r13
04 halt

05 jnezn r1 8
06 setn r13 1
07 jumpr r14

08 pushr r1 r15
09 pushr r14 r15
10 addn r1 -1
11 calln r14 5
12 popr r14 r15
13 popr r1 r15
14 mul r13 r13 r1
15 jumpr r14
```

Python Hmmm

Python

```
n = int(input())
answer = fac(n)
print(answer)

def fac(n):
  """Recursive factorial!"""
  if n==0:
    return 1

  else:
    res = fac(n-1)
    return n*res
```

r13 is the **answer**

r14 is the "return address"

r15 is the "stack pointer"

Hmmm

```
00 setn r15 42
01 read r1
02 calln r14 5
03 write r13
04 halt

05 jnezn r1 8
06 setn r13 1
07 jumpr r14

08 pushr r1 r15
09 pushr r14 r15
10 addn r1 -1
11 calln r14 5
12 popr r14 r15
13 popr r1 r15
14 mul r13 r13 r1
15 jumpr r14
```

Python Hmmm

Python

```
n = int(input())
answer = fac(n)
print(answer)

def fac(n):
  """Recursive factorial!"""
  if n==0:
    return 1

  else:
    res = fac(n-1)
    return n*res
```

r13 is the **answer**

r14 is the "return address"

r15 is the "stack pointer"

Hmmm

```
00 setn r15 42
01 read r1
02 calln r14 5
03 write r13
04 halt

05 jnezn r1 8
06 setn r13 1
07 jumpr r14

08 pushr r1 r15
09 pushr r14 r15
10 addn r1 -1
11 calln r14 5
12 popr r14 r15
13 popr r1 r15
14 mul r13 r13 r1
15 jumpr r14
```

Python Hmmm

Python

```
n = int(input())
answer = fac(n)
print(answer)

def fac(n):
  """Recursive factorial!"""
  if n==0:
    return 1

  else:
    res = fac(n-1)
    return n*res
```

r13 is the **answer**

r14 is the "return address"

r15 is the "stack pointer"

Hmmm

```
00 setn r15 42
01 read r1
02 calln r14 5
03 write r13
04 halt

05 jnezn r1 8
06 setn r13 1
07 jumpr r14

08 pushr r1 r15
09 pushr r14 r15
10 addn r1 -1
11 calln r14 5
12 popr r14 r15
13 popr r1 r15
14 mul r13 r13 r1
15 jumpr r14
```

Prepare for function call! All precious belongings must be saved on the stack!

Python Hmmm

Python

r13 is the answer

r14 is the "return address"

r15 is the "stack pointer"

Hmmm

```
n = int(input())
answer = fac(n)
print(answer)

def fac(n):
  """Recursive factorial!"""
  if n==0:
    return 1

  else:
    res = fac(n-1)
    return n*res
```

```
00 setn r15 42
01 read r1
02 calln r14 5
03 write r13
04 halt

05 jnezn r1 8
06 setn r13 1
07 jumpr r14

08 pushr r1 r15
09 pushr r14 r15
10 addn r1 -1
11 calln r14 5
12 popr r14 r15
13 popr r1 r15
14 mul r13 r13 r1
15 jumpr r14
```

Python Hmmm

Python

r13 is the answer

r14 is the "return address"

r15 is the "stack pointer"

Hmmm

```
n = int(input())
answer = fac(n)
print(answer)

def fac(n):
  """Recursive factorial!"""
  if n==0:
    return 1

  else:
    res = fac(n-1)
    return n*res
```

```
00 setn r15 42
01 read r1
02 calln r14 5
03 write r13
04 halt

05 jnezn r1 8
06 setn r13 1
07 jumpr r14

08 pushr r1 r15
09 pushr r14 r15
10 addn r1 -1
11 calln r14 5
12 popr r14 r15
13 popr r1 r15
14 mul r13 r13 r1
15 jumpr r14
```

Python Hmmm

Python

r13 is the answer

r14 is the "return address"

r15 is the "stack pointer"

Hmmm

```
n = int(input())
answer = fac(n)
print(answer)

def fac(n):
  """Recursive factorial!"""
  if n==0:
    return 1

  else:
    res = fac(n-1)
    return n*res
```

Function call over! All precious belongings back into their registers!

```
00 setn r15 42
01 read r1
02 calln r14 5
03 write r13
04 halt

05 jnezn r1 8
06 setn r13 1

07 jumpr r14
08 pushr r1 r15
09 pushr r14 r15
10 addn r1 -1
11 calln r14 5
12 popr r14 r15
13 popr r1 r15
14 mul r13 r13 r1
15 jumpr r14
```

Python Hmmm

Python

r13 is the answer

r14 is the "return address"

r15 is the "stack pointer"

Hmmm

```
n = int(input())
answer = fac(n)
print(answer)

def fac(n):
  """Recursive
     factorial!"""
  if n==0:
    return 1

  else:
    res = fac(n-1)
    return n*res
```

```
00 setn r15 42
01 read r1
02 calln r14 5
03 write r13
04 halt

05 jnezn r1 8
06 setn r13 1
07 jumpr r14

08 pushr r1 r15
09 pushr r14 r15
10 addn r1 -1
11 calln r14 5
12 popr r14 r15
13 popr r1 r15
14 mul r13 r13 r1
15 jumpr r14
```

Name: ______________________

Worksheet

Write down what happens in the registers and memory (the stack) as this program runs. Remember that `calln` sets r14 to the address of the *next* instruction!

Program (“low part of RAM”)

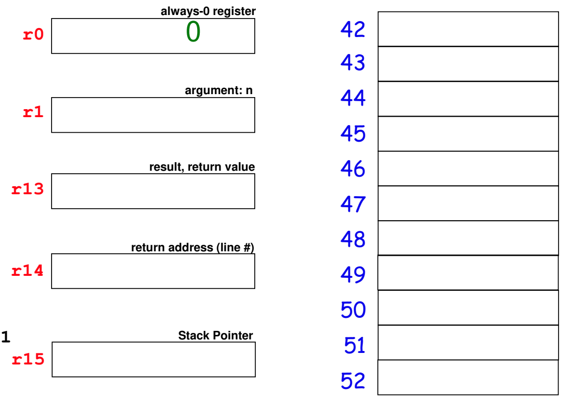
```
00 setn  r15 42
01 read  r1
02 calln r14 5
03 write r13
04 halt

05 jnezn r1 8
06 setn  r13 1
07 jumpr r14

08 pushr r1 r15
09 pushr r14 r15
10 addn  r1 -1
11 calln r14 5
12 popr  r14 r15
13 popr  r1 r15
14 mul   r13 r13 r1
15 jumpr r14
```

factorial function

The input is 3.

CPU Registers with labels

Memory (“high part of RAM”) **"the stack"**

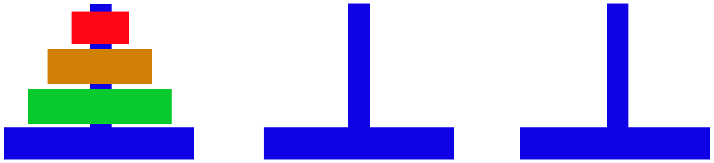


How low *could* we start the stack? How deep does the stack get? What are the possible values of r14?

Towers of Hanoi

This puzzle can get Hanoi'ing!

hanoi (Disks, From, To)
hanoi(3, 1, 3)

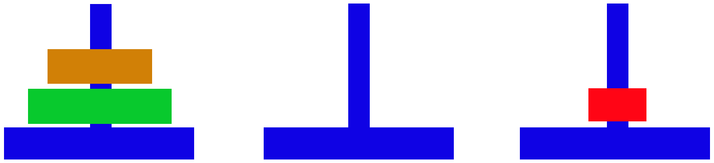


Peg 1 Peg 2 Peg 3

Towers of Hanoi

This puzzle can get Hanoi'ing!

hanoi (Disks, From, To)
hanoi(3, 1, 3)
1 to 3

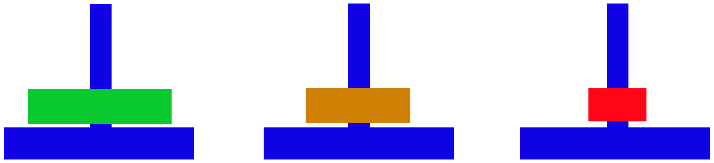


Peg 1 Peg 2 Peg 3

Towers of Hanoi

This puzzle can get Hanoi'ing!

hanoi (Disks, From, To)
hanoi(3, 1, 3)
1 to 3
1 to 2

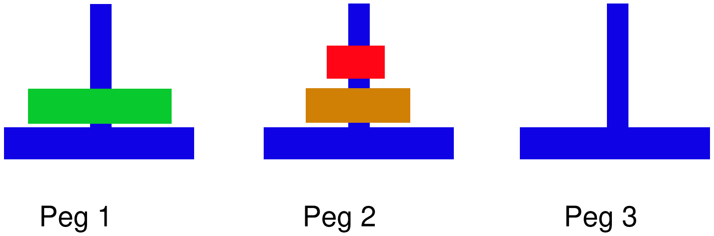


Peg 1 Peg 2 Peg 3

Towers of Hanoi

This puzzle can
get Hanoi'ing!

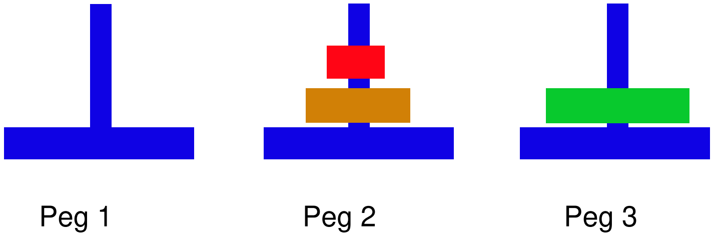
```
hanoi (Disks, From, To)
hanoi(3, 1, 3)
 1 to 3
 1 to 2
 3 to 2
```



Towers of Hanoi

This puzzle can
get Hanoi'ing!

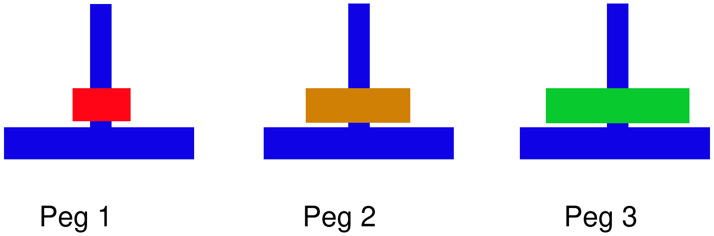
```
hanoi (Disks, From, To)
hanoi(3, 1, 3)
 1 to 3
 1 to 2
 3 to 2
 1 to 3
```



Towers of Hanoi

This puzzle can
get Hanoi'ing!

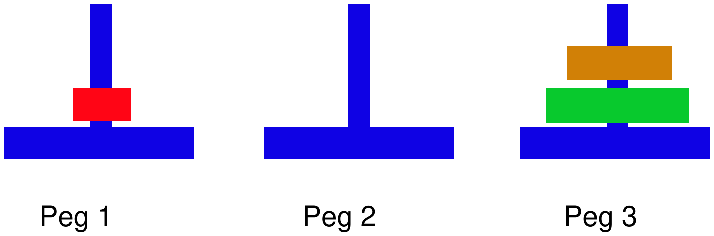
```
hanoi (Disks, From, To)
hanoi(3, 1, 3)
 1 to 3
 1 to 2
 3 to 2
 1 to 3
 2 to 1
```



Towers of Hanoi

This puzzle can
get Hanoi'ing!

```
hanoi (Disks, From, To)
hanoi(3, 1, 3)
 1 to 3
 1 to 2
 3 to 2
 1 to 3
 2 to 1
 2 to 3
```



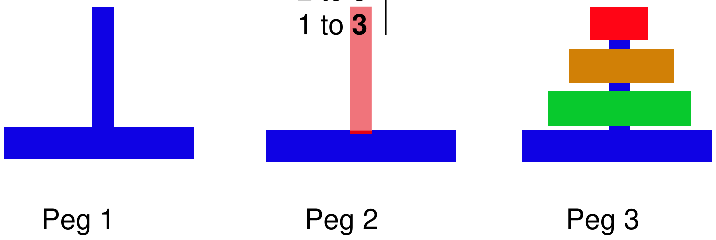
Towers of Hanoi

This puzzle can
get Hanoi'ing!

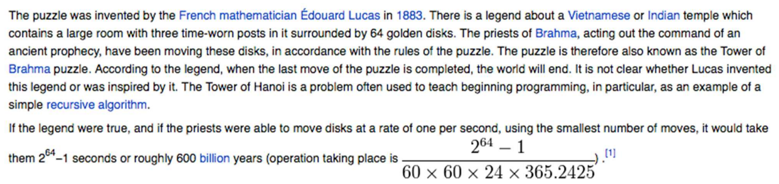
hanoi (Disks, From, To)
hanoi(3, 1, 3)

1 to 3
1 to 2
3 to 2
1 to 3
2 to 1
2 to 3
1 to **3**

7 = 2^3 - 1 moves



The Hanoi Legend

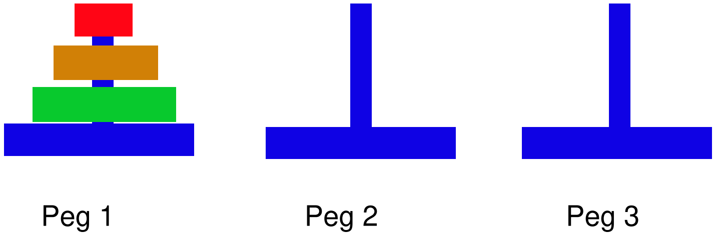


The puzzle was invented by the French mathematician Édouard Lucas in 1883. There is a legend about a Vietnamese or Indian temple which contains a large room with three time-worn posts in it surrounded by 64 golden disks. The priests of Brahma, acting out the command of an ancient prophecy, have been moving these disks, in accordance with the rules of the puzzle. The puzzle is therefore also known as the Tower of Brahma puzzle. According to the legend, when the last move of the puzzle is completed, the world will end. It is not clear whether Lucas invented this legend or was inspired by it. The Tower of Hanoi is a problem often used to teach beginning programming, in particular, as an example of a simple recursive algorithm.

If the legend were true, and if the priests were able to move disks at a rate of one per second, using the smallest number of moves, it would take them $2^{64}-1$ seconds or roughly 600 billion years (operation taking place is $\frac{2^{64}-1}{60 \times 60 \times 24 \times 365.2425}$).[1]

```
          3          1     3
hanoi  (Disks,  From,  To)
  if Disks == 1:
     print(str(From) + ”,” + str(To))
     return
  else:
     # COMPUTE “Other” peg
     hanoi(Disks-1, From, Other)
     hanoi(1, From, To)
     hanoi(Disks-1, Other, To)
     return
```

```
hanoi(2, 1, 2)
  hanoi(1, 1, 3)
  hanoi(1, 1, 2)
  hanoi(1, 3, 2)
hanoi(1, 1, 3)
hanoi(2, 2, 3)
```



What’s Next?

Cool application areas…
- Data compression
- Secret sharing
- AI and games

Object-oriented programs (OOPS!)

Limits of computation: Are there things computers cannot do?