The CS 5 Times

Penguin/Pig Gang Fight Brings Violence to Claremont

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.
def main():
    r1 = 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
def main():
    r1 = input()  # r1=3
    r13 = emma(r1)  # emma(3)
    r13 = r13 + r1
    print(r13)
    return

def emma(r1):
    r1 = r1 + 1  # r1=4
    r13 = sarah(r1)  # sarah(4)
    r13 = r13 + r1  # r13=??
    return r13

def sarah(r1):
    r1 = r1 + 42  # r1=4
    r13 = r1 + 1  # r13=46
    return r13  # return(47)
def main():
    r1 = input()  # r1=3
    r13 = emma(r1)  # emma(3)
    r13 = r13 + r1
    print(r13)
    return

def emma(r1):
    r1 = r1 + 1  # r1=4
    r13 = sarah(r1)  # sarah(4) r13=47
    r13 = r13 + r1  # r13=??
    return r13

def sarah(r1):
    r1 = r1 + 42  # r1=4
    r13 = r1 + 1  # r1=46
    return r13  # return(47)
def main():
    r1 = input() ← r1=3
    r13 = emma(r1) ← emma(3) ← r13=51
    r13 = r13 + r1 ← r13=??
    print(r13)
    return

def emma(r1):
    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 = r1 + 42 ← r1=4
    r13 = r1 + 1 ← r1=46 ← r13=47 ← return(47)

    return

r1 = 3
emma(3)
r1 = 3
r1 = 4
sarah(4)
r1 = 4
r1 = 46
r13 = 47
return(47)
def main():
    r1 = input()  # r1=3
    r13 = emma(r1)  # emma(3) r13=51
    r13 = r13 + r1  # r13=54
    print(r13)  # 54
    return

def emma(r1):
    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 = r1 + 42  # r1=4
    r13 = r1 + 1  # r13=46
    r13 = r13 + 1  # r13=47
    return r13  # return(47)

Cool, but how does this work!?
The Stack!

Insert ("push")

Remove ("pop")

AFLAC, ShmAFLAC! Be careful up there!
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
```python
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
```

The stack in RAM!

```
3 3 46
47
4 3
```
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
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
```python
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
```

The stack in RAM!

The stack in RAM!
Implementing Functions

(1) Use r15 as the stack pointer.

(2) Before the function call,
   Store all valuable data to the stack—and increment r15

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

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

(5) After the function call,
    Load valuable data back from the stack (in reverse order)

```
setn r15 42
pushr r1 r15
```

```
calln r14 #
```

```
popr r1 r15
```
Implementing Functions

(1) Use r15 as the stack pointer.

(2) Before the function call, store all valuable data to the stack—and increment r15.

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

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

(5) After the function call, load valuable data back from the stack (in reverse order).
def main():
    r1 = int(input())
    r1 = r1 + 1
    print(r1)
    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

We object to this!!!

We object to this!!!

RAM!

```
0 01001001
1 11001011
...
25 01001011
26
27
28
29
...
255
```
Now Without Pigs and Geese!

We object to this!!!

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
Now Without Pigs and Geese!

It was better with pigs and geese!

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

<table>
<thead>
<tr>
<th>Line</th>
<th>Assembler Code</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>setn r15 26</td>
<td>set stack pointer to 26</td>
</tr>
<tr>
<td>01</td>
<td>read r1</td>
<td>start of main</td>
</tr>
<tr>
<td>02</td>
<td>pushr r1 r15</td>
<td>store r1 on the stack</td>
</tr>
<tr>
<td>03</td>
<td>calln r14 08</td>
<td>call emma</td>
</tr>
<tr>
<td>04</td>
<td>popr r1 r15</td>
<td>load r1 from the stack</td>
</tr>
<tr>
<td>05</td>
<td>add r13 r13 r1</td>
<td>r13 = r13 + r1</td>
</tr>
<tr>
<td>06</td>
<td>write r13</td>
<td></td>
</tr>
<tr>
<td>07</td>
<td>halt</td>
<td></td>
</tr>
<tr>
<td>08</td>
<td>addn r1 1</td>
<td>start of emma</td>
</tr>
<tr>
<td>09</td>
<td>pushr r1 r15</td>
<td>store r1 on the stack</td>
</tr>
<tr>
<td>10</td>
<td>pushr r14 r15</td>
<td>save return addr on stack</td>
</tr>
<tr>
<td>11</td>
<td>calln r14 16</td>
<td>call sarah</td>
</tr>
<tr>
<td>12</td>
<td>popr r14 r15</td>
<td>load ret addr from stack</td>
</tr>
<tr>
<td>13</td>
<td>popr r1 r15</td>
<td>load r1 from the stack</td>
</tr>
<tr>
<td>14</td>
<td>add r13 r13 r1</td>
<td>r13 = r13 + r1</td>
</tr>
<tr>
<td>15</td>
<td>jmp r14</td>
<td>return</td>
</tr>
<tr>
<td>16</td>
<td>addn r1 42</td>
<td>start of sarah</td>
</tr>
<tr>
<td>17</td>
<td>setn r2 1</td>
<td>put 1 in a register</td>
</tr>
<tr>
<td>18</td>
<td>add r13 r1 r2</td>
<td>put result in r13</td>
</tr>
<tr>
<td>19</td>
<td>jmp r14</td>
<td>return</td>
</tr>
</tbody>
</table>
Now Without Pigs and Geese!

It was better with pigs and geese!

```python
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
```

```assembly
00 setn r15 26  # set stack pointer to 26
01 read r1       # start of main
02 pushr r1 r15  # store r1 on the stack
03 calln r14 08  # 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 addn r1 1    # start of emma!
09 pushr r1 r15  # store r1 on the stack
10 pushr r14 r15 # save return addr on stack
11 calln r14 16  # call sarah
12 popr r14 r15 # load ret addr from stack
13 popr r1 r15  # load r1 from the stack
14 add r13 r13 r1 # r13 = r13 + r1
15 jump r14      # return!

16 addn r1 42    # start of sarah!
17 setn r2 1     # put 1 in a register
18 add r13 r1 r2 # put result in r13
19 jump r14      # return
```
Memory-processing unit could bring memristors to the masses

Date: July 30, 2018
Source: University of Michigan
Summary: A new way of arranging advanced computer components called memristors on a chip could enable them to be used for general computing, which could cut energy consumption by a factor of 100.


I wonder when memristors will show up in Logisim?
Recursion?

```python
def fac(N):
    if N <= 1:
        return 1
    else:
        return N * fac(N-1)
```

"The Stack"

Remembers all of the individual calls to `fac`

```
fac(5)
  5 * fac(4)
    4 * fac(3)
      3 * fac(2)
        2 * fac(1)
          1
```
Factorial via Recursion...

Python

```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
```

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

First let’s try N=0 and then N=3

```
r1 (N) r13 (Res) r14
```

Return value

Return address

```
^ Jump r14
^ Jump r14
```

```python
return n*res
```
Python

```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
```

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  jumper 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  jumper r14
```

Try to align the Python code to the Hmmm code... as shown in the next slide...
```python
def fac(n):
    """ recursive factorial! ""
    if n==0:
        return 1
    else:
        res = fac(n-1)
        return n*res
```

Try to align the Python code to the Hmmm code...
```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
```

**Hmmm**

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

- **r13** is the answer
- **r14** is the "return address"
- **r15** is the "stack pointer"
```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"
```

```
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 jumper 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 jumper r14
```

r15 is the "stack pointer"
r14 is the "return address"
r13 is the answer
n = \texttt{int(input())} \\
\texttt{answer = fac(n)} \\
\texttt{print(answer)}

def fac(n):
    \"\"\" recursive factorial! \"\"\"
    \textbf{if} n==0:
        \texttt{return 1}
    \textbf{else}:
        \texttt{res = fac(n-1)}
        \texttt{return n*res}

Hmmm

- \texttt{r13} is the \texttt{answer}
- \texttt{r14} is the \texttt{\hspace{1mm}return address}\texttt{\hspace{1mm}}
- \texttt{r15} is the \texttt{\hspace{1mm}stack pointer}\texttt{\hspace{1mm}}

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

```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
```

Hmmm

- **r13** is the answer
- **r14** is the "return address"
- **r15** is the "stack pointer"

```
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 jumper 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 jumper r14
```
```python
def fac(n):
    """ recursive factorial! ""
    if n==0:
        return 1
    else:
        res = fac(n-1)
        return n*res
```

Python

<table>
<thead>
<tr>
<th>Python</th>
<th>Hmmm</th>
</tr>
</thead>
</table>
| ```n = int(input())
answer = fac(n)
print(answer)```
| ```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 jumper 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 jumper r14
```

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

r15 is the "stack pointer",
r14 is the "return address",
r13 is the answer.
```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
```
On your Worksheet…

Write down what happens in the registers and memory (the stack) as this program runs…

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 jump 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 jump r14
```

How low could we start the stack? How deep does the stack get? What are the possible values of r14?
Demo Factorial!
Towers of Hanoi

This puzzle can get Hanoi'ing!

hanoi (Disks, From, To)
hanoi(3, 1, 3)
Towers of Hanoi

This puzzle can get Hanoi'ing!

\[ \text{hanoi (Disks, From, To)} \]
\[ \text{hanoi(3, 1, 3)} \]
\[ \text{1 to 3} \]
Towers of Hanoi

This puzzle can get Hanoi'ing!

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

Peg 1

Peg 2

Peg 3
This puzzle can get Hanoi'ing!

Towers of Hanoi

hanoi (Disks, From, To)

hanoi(3, 1, 3)
1 to 3
1 to 2
3 to 2
1 to 3

Peg 1

Peg 2

Peg 3
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
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
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 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 $\frac{2^{64} - 1}{60 \times 60 \times 24 \times 365.2425}$).
Towers of Hanoi

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

Peg 1  Peg 2  Peg 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
What’s Next?

Cool application areas…
  • Data compression
  • “Big data” ethics
  • AI and games

Object-oriented programs (OOPS!)

Limits of computation: Are there things computers cannot do?
Program ("low part of RAM")

<table>
<thead>
<tr>
<th>Line</th>
<th>Assembly Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>setn r15 42</td>
<td>The input is 3.</td>
</tr>
<tr>
<td>01</td>
<td>read r1</td>
<td></td>
</tr>
<tr>
<td>02</td>
<td>calln r14 5</td>
<td></td>
</tr>
<tr>
<td>03</td>
<td>write r13</td>
<td></td>
</tr>
<tr>
<td>04</td>
<td>halt</td>
<td></td>
</tr>
<tr>
<td>05</td>
<td>jnezn r1 8</td>
<td></td>
</tr>
<tr>
<td>06</td>
<td>setn r13 1</td>
<td></td>
</tr>
<tr>
<td>07</td>
<td>jump r14</td>
<td></td>
</tr>
<tr>
<td>08</td>
<td>pushr r1 r15</td>
<td></td>
</tr>
<tr>
<td>09</td>
<td>pushr r14 r15</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>addn r1 -1</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>calln r14 5</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>popr r14 r15</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>popr r1 r15</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>mul r13 r13 r1</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>jump r14</td>
<td></td>
</tr>
</tbody>
</table>

CPU Registers with labels:
- r0: always-0 register
- r1: argument: n
- r13: result, return value
- r14: return address (line #)
- r15: Stack Pointer

Memory ("high part of RAM") "the stack"