It never ends!

Write a Hmmm program that:
1. never halts
2. is as short as possible
State any assumptions you make about registers / memory.

(Your response)
It never ends!

Write a Hmmm program that:
1. never halts
2. is as short as possible

State any assumptions you make about registers / memory.

0 jumpn 0
Hmmm conventions
Human programming practices that help us write correct and readable programs

\[ r_0 \text{ always contains the value } 0 \]

Write lots of comments / documentation!
Clearer is better than shorter (but shorter can be clearer).
Practice writing Hmmm (well)

Write a Hmmm program that reads two integers from input into registers r1 and r2. Then, the program should compute \( r1^{r2} \), store that value in register r13, and write that value out. 

You may assume that \( r2 \geq 0 \).

A suggested algorithm-algorithm

1. Write a few test cases
2. Write pseudocode (i.e., a sketch of the algorithm)
3. Write comments
4. Write Hmmm instructions for the math 
   leave space for the control flow (i.e., jumps)
5. Write Hmmm instructions for the control flow
6. Write the line numbers
Hmmm conventions

Human programming practices that help us right correct and readable programs

r0 always contains the value 0

r13 is for result values

Write lots of comments / documentation!
Clearer is better than shorter (but shorter can be clearer).
A program that computes $r_1^{r_2}$

Lots of comments have been omitted, so that the program fits on this slide!

```plaintext
# read the inputs from the user
0 read r1     # base
1 read r2     # exponent

# initialize the result (r13)
2 setn r13 1

# if the exponent (r2) is 0, the result is 1, so we can finish
3 jeqzn r2 7

4 mul   r13 r13 r1     # result *= base
5 addn  r2  -1         # decrement exponent
6 jgtzn r2  4          # do we need to keep iterating?

# we're done, print the output
7 write r13
8 halt
```
A program that computes \( r_1^{r_2} \)

Lots of comments have been omitted, so that the program fits on this slide!

```
0 read r1  # base
1 read r2  # exponent

# initialize the result (r13)
2 setn r13 1

# if the exponent (r2) is 0, the result is 1, so we can finish
3 jeqzn r2 0

r13 = r1^{r2}

4 mul  r13 r13 r1  # result *= base
5 addn r2 -1  # decrement exponent
6 jgtzn r2 4  # do we need to keep iterating?

7 write r13
8 halt
```
A function that computes $r_1^{r_2}$?

0 read r1       # base
1 read r2       # exponent

# initialize the result (r13)
2 setn r13 1

# if the exponent (r2) is 0, the result is 1, so we can finish
3 jeqzn r2 7
4 mul  r13 r13 r1  # result *= base
5 addn r2 -1      # decrement exponent
6 jgtzn r2 4      # do we need to keep iterating?

7 write r13
8 halt
A function that computes $r1^{r2}$?

0 read r1  # base
1 read r2  # exponent
2 r13 = r1^{r2}
3 write r13
4 halt

# initialize the result (r13)
6 setn r13 1

# if the exponent (r2) is 0, the result is 1, so we can finish
7 jeqzn r2 7
8 mul r13 r13 r1  # result *= base
9 addn r2 -1      # decrement exponent
10 jgtzn r2 4     # do we need to keep iterating?
A function that computes $r_1^{r_2}$?

Oops! We’ve re-ordered the code. How can we fix it?

0 read r1    # base
1 read r2    # exponent

2 r13 = r1\^{r_2}

3 write r13
4 halt

# initialize the result (r13)
5 setn r13 1

# if the exponent (r_2) is 0, the result is 1, so we can finish
6 jeqzn r2 7

7 mul r13 r13 r1    # result *= base
8 addn r2 -1        # decrement exponent
9 jgtzn r2 4        # do we need to keep iterating?
A function that computes $r1^{r2}$

That’s better.

```
0  read  r1    # base
1  read  r2    # exponent

2  r13 = r1^{r2}

3  write  r13
4  halt

# initialize the result (r13)
5  setn  r13  1

# if the exponent (r2) is 0, the result is 1, so we can finish
6  jeqzn  r2  10

7  mul  r13  r13  r1    # result *= base
8  addn  r2  -1    # decrement exponent
9  jgtzn  r2  7    # do we need to keep iterating?
10 jumpn  3    # return
```
Hmmm conventions
Human programming practices that help us right correct and readable programs

r0 always contains the value 0

r13 is for the return value

r14 is for the return location

Write lots of comments / documentation!
Clearer is better than shorter (but shorter can be clearer).
A function that computes $r1^{r2}$

Calls and returns in Hmmm: use `calln r14` # and `jumpn r14`

```
0  read  r1   # base
1  read  r2   # exponent
2  [white]calln r14 5
3  write  r13
4  halt

# initialize the result (r13)
5  setn r13 1

# if the exponent (r2) is 0, the result is 1, so we can finish
6  jeqzn r2 10

7  mul  r13 r13 r1   # result *= base
8  addn r2 -1         # decrement exponent
9  jgtzn r2 7         # do we need to keep iterating?
10 [red]jumpn r14     # return
```

# the same as:
```
  setn r14 3
  jumpn 5
```
A function that computes $r_1^{r_2}$

Calls and returns in Hmmm: use `calln r14` # and `jumpn r14`

```
0  read r1  # base
1  read r2  # exponent

2  calln r14 5  # r13 = r1^{r2}

3  write r13
4  halt

# initialize the result (r13)
5  setn r13 1

# if the exponent (r2) is 0, the result is 1, so we can finish
6  jeqzn r2 10

r13 = r1^{r2}

7  mul r13 r13 r1  # result *= base
8  addn r2 -1  # decrement exponent
9  jgtzn r2 7  # do we need to keep iterating?
10 jumpn r14  # return
```
A function that computes $r_1^{r_2}$

Uh, oh... What if we want to call the function lots of times, with different arguments?

0 read r1    # base
1 read r2    # exponent

2 calln r14 7 # r13 = r1^{r2}
3 write r13

4 calln r14 7 # r13 = r1^3 ← This doesn’t work!!!!
5 write r13

6 halt

$r_{13} = r_1^{r_2}$
A function that computes \( r_1^{r_2} \)

Uh, oh... What if we want to call the function lots of times, with different arguments?

0 | read \( r_1 \) # base
1 | read \( r_2 \) # exponent
2 | calln \( r_{14} \) 8 # \( r_{13} = r_1^{r_2} \)
3 | write \( r_{13} \)
4 | setn \( r_2 \) 3 # prepare the argument
5 | calln \( r_{14} \) 8 # \( r_{13} = r_1^{r_3} \)
6 | write \( r_{13} \)
7 | halt

\[ r_{13} = r_1^{r_2} \]
A function that computes $r_1^{r_2}$

Uh, oh. What if we need the register values after the call?

0 read r1    # base
1 read r2    # exponent

2 # $r_{13} = r_1^{r_2} + r_2$

3 write r13
4 halt

# initialize the result (r13)
5 setn r13 1

# if the exponent (r2) is 0, the result is 1, so we can finish
6 jeqzn r2 10

7 mul r13 r13 r1    # result *= base
8 addn r2 -1    # decrement exponent
9 jgtzn r2 7    # do we need to keep iterating?
10 jump r14    # return
Function calls in Hmmm

Caller (outside the function): assume the function writes to every register
Callee (inside the function): assume every register is yours

# save any register value that I’ll need later

# prepare the arguments

calln r14 N  # call the function

# restore all the register values that I saved

N # function start
   write to registers with gleeful abandon
   if the function should return a value, save it in r13
M jump r14  # return
The stack

A place in memory where we can save values for later

stack pointer
the “top” of the stack (where the next value will go)
Hmmm conventions
Human programming practices that help us right correct and readable programs

r0 always contains the value 0

r13 is for the return value

r14 is for the return location

r15 is for the stack pointer

Write lots of comments / documentation!
Clearer is better than shorter (but shorter can be clearer).
The stack

“Last-in, first out” (LIFO)

- `storer` `r1` `r15`
- `addn` `r15` 1
- `storer` `r2` `r15`
- `addn` `r15` 1

**Stack Pointer**
the “top” of the stack
(where the next value will go)

- `addn` `r15` -1
- `loadr` `r2` `r15`
- `addn` `r15` -1
- `loadr` `r1` `r15`
# Loads and stores

<table>
<thead>
<tr>
<th>Registers</th>
<th>RAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>r0</td>
<td></td>
</tr>
<tr>
<td>r1</td>
<td>100</td>
</tr>
<tr>
<td>r2</td>
<td>43</td>
</tr>
<tr>
<td>r3</td>
<td></td>
</tr>
<tr>
<td>r4</td>
<td></td>
</tr>
<tr>
<td>r5</td>
<td></td>
</tr>
<tr>
<td>r6</td>
<td></td>
</tr>
<tr>
<td>r7</td>
<td>-1</td>
</tr>
<tr>
<td>r8</td>
<td>51</td>
</tr>
<tr>
<td>r9</td>
<td></td>
</tr>
<tr>
<td>r10</td>
<td></td>
</tr>
<tr>
<td>r11</td>
<td></td>
</tr>
</tbody>
</table>

###.store

<table>
<thead>
<tr>
<th>&quot;local variable&quot;</th>
<th>RAM address</th>
</tr>
</thead>
<tbody>
<tr>
<td>storen</td>
<td>r1</td>
</tr>
<tr>
<td>storer</td>
<td>r1</td>
</tr>
</tbody>
</table>

###.load

<table>
<thead>
<tr>
<th>&quot;local variable&quot;</th>
<th>RAM address</th>
</tr>
</thead>
<tbody>
<tr>
<td>loadn</td>
<td>r7</td>
</tr>
<tr>
<td>loadr</td>
<td>r7</td>
</tr>
</tbody>
</table>
Function calls in Hmmm

Caller (outside the function): assume the function writes to every register

Callee (inside the function): assume every register is yours

```assembly
# initialize stack pointer
setn r15 S

# save any register value that I’ll need later
caller
storer rN r15
addn r15 1

# prepare the arguments
calln r14 N     # call the function

# restore all the register values that I saved (LIFO!)
callee
addn r15 -1
loadr rN r15

N   # function start
write to registers with gleeful abandon
if the function should return a value, save it in r13
M   jump r14     # return
```
Let’s practice!

Write a Hmmm program that reads a positive integer value \( n \), then writes the value \( n! + n \) to the screen.

**Bonus:** can you write a recursive factorial?

```
setn r13 1  
jeqzn r1 47  
mul r13 r13 r1  
addn r1 -1  
jumpn 43  
jumpr r14
```

\( r13 = r1! \)
Human programming practices that help us write correct and readable programs

- **r0** always contains the value 0
- **r13** is for the return value
- **r14** is for the return location
- **r15** is for the stack pointer

Write lots of comments / documentation! Clearer is better than shorter (but shorter can be clearer).
Function calls in Hmmm

Caller (outside the function): assume the function writes to every register

Callee (inside the function): assume every register is yours

```
# initialize stack pointer
setn  r15  S

# save any register value that I'll need later
storer r15  rN
addn  r15  1

# prepare the arguments

calln r14  N  # call the function

# restore all the register values that I saved (LIFO!)
addn  r15  -1
loadr  rN  r15

N  # function start
write to registers with gleeful abandon
if the function should return a value, save it in r13

M jumpn r14  # return
```