23. Example of Interrupt, Subroutines, Recursion
Chapter 10 November 19, 2018

- Review
  - Interrupt processing
  - Subroutines

- Recursion
  - Example: Towers of Hanoi
  - Euclid's algorithm for Greatest Common Divisor

Example of Interrupts

Program to generate a random number

- Main program increments a register
- Interrupted by keyboard input
  - Converts current number in register to ASCII and print is out

→ MAIN PROGRAM

→ INTERRUPT SERVICE ROUTINE

→ TABLE – GET STARTING ADDRESS OF ISR

```
        .ORIG x3000
        LEA     R6, #-1 ; initializing R6, to 3000 (empty stack)
        LD      R0, IEnable
        STI     R0, KBSR            SET UP INTERRUPT
; R1 keeps numbers up to 1024
        AND     R1, R1, #0
        LEA     R0, Welcome
        PUTS
        LD      R0, NL
        TRAP    x21
NEXTLN  ADD     R1, R1, #1
        LD      R0, MAX                                  (INFINITE)
        ADD     R0, R0, R1                               LOOP
        BRn     NEXTLN
        AND     R1, R1, #0      ; Reset number
        BRnzp   NEXTLN
        HALT
MAX     .FILL   #-999           ; negative of number
IEnable .FILL   x4000
KBSR    .FILL   xFE00
Welcome .STRINGZ    "Pick your lucky number: Press any key"
NL      .FILL   #13             ; new line
        .END
```

```
.ORIG x0100
.BLKW x80
.FILL x1000 ; starting address of the keyboard interrupt subroutine
.END
```

INTR. TABLE

ISR

```
        .ORIG x1000
; pushing on stack. Might not need that but I just put it in there as a reference
. R6 is initialized in the main code
        ADD     R6, R6, #-1
        STR     R0, R6, #0
        ADD     R6, R6, #-1
        STR     R1, R6, #0
        ADD     R6, R6, #-1
        STR     R7, R6, #0
;
        GETC                        ; Read input to clear keyboard
                                    ; Don't need to do anything with this char.
; rest of code goes here
        ADD     R0, R1, #0
        JSR     BIN2ASC
        LEA     R0, ASCIIBUF
        PUTS
        LD      R0, NL1
        TRAP    x21
        LDR     R7, R6, #0
        ADD     R6, R6, #1
        LDR     R1, R6, #0
        ADD     R6, R6, #1
        LDR     R0, R6, #0
        ADD     R6, R6, #1
        RTI
;
BIN2ASC         ST    R1, Save1
                ST    R2, Save2
                ST    R3, Save3
                LEA   R1, ASCIIBUF    ; R1 points to string being generated
                ADD   R0, R0,#0       ; R0 contains the binary value
                BRn   NegSign        ;
                LD    R2,ASCIIplus   ; First store the ASCII plus sign
                STR   R2,R1,#0
                BRnzp Begin100
NegSign         LD    R2,ASCIIminus ; First store ASCII minus sign
                STR   R2,R1,#0
                NOT   R0,R0          ; Convert the number to absolute
                ADD   R0,R0,#1       ; value; it is easier to work with.
;
Begin100        LD    R2,ASCIIoffset ; Prepare for "hundreds" digit
;
                LD    R3,Neg100      ; Determine the hundreds digit
Loop100         ADD   R0,R0,R3
                BRn   End100
                ADD   R2,R2,#1
                BRnzp Loop100
;
End100          STR    R2,R1,#1   ; Store ASCII code for hundreds digit
                LD     R3,Pos100
                ADD    R0,R0,R3   ; Correct R0 for one-too-many subtracts
;
                LD     R2,ASCIIoffset ; Prepare for "tens" digit
;
Begin10         LD     R3,Neg10   ; Determine the tens digit
Loop10          ADD    R0,R0,R3
                BRn    End10
                ADD    R2,R2,#1
                BRnzp  Loop10
;
End10           STR    R2,R1,#2   ; Store ASCII code for tens digit
                ADD    R0,R0,#10  ; Correct R0 for one-too-many subtracts
Begin1          LD     R2,ASCIIoffset ; Prepare for "ones" digit
```

? RESTORE REGS. FROM STACK

```
                ADD     R2,R2,R0
                STR     R2,R1,#3
                LD      R1, Save1
                LD      R2, Save2
                LD      R3, Save3
                RET
;
ASCIIBUF        .BLKW  #4
                .FILL  #13          ; Carriage return to end line
                .FILL  #0           ; Null for ascii string
ASCIIplus       .FILL  x002B
ASCIIminus      .FILL  x002D
ASCIIoffset     .FILL  x0030
Neg100          .FILL  xFF9C
Pos100          .FILL  x0064
Neg10           .FILL  xFFF6
Save1   .BLKW   #1
Save2   .BLKW   #1
Save3   .BLKW   #1
NL1     .FILL   #13
;
        .END
```

Subroutines

A subroutine is a program fragment that:

- lives in user space
- performs a well-defined task
- is invoked (called) by another user program
- returns control to the calling program when finished

Like a service routine, but not part of the OS

- not concerned with protecting hardware resources
- no special privilege required

Reasons for subroutines:

- reuse useful (and debugged!) code without having to keep typing it in
- divide task among multiple programmers
- use vendor-supplied *library* of useful routines

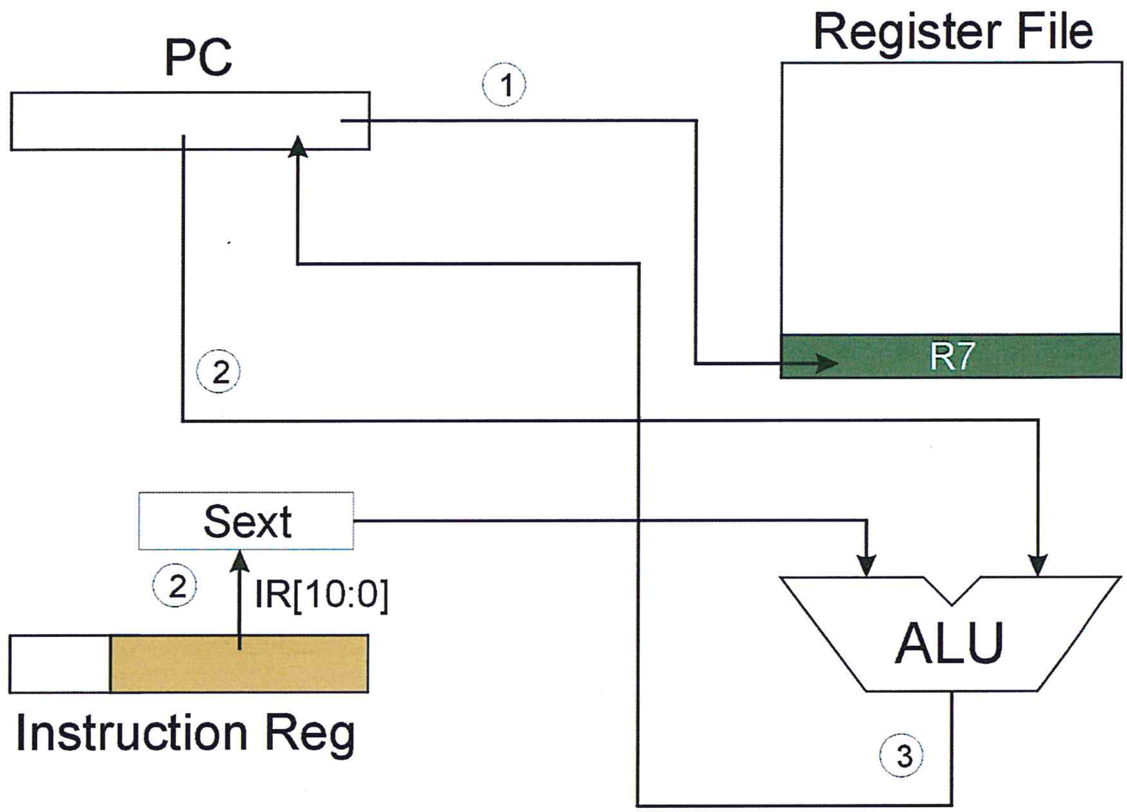
JSR Instruction

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
JSR	0	1	0	0	1	PCoffset11										

Jumps to a location (like a branch but unconditional), and saves current PC (addr of next instruction) in R7.

- **saving the return address is called "linking"**
- **target address is PC-relative** (PC + Sext(IR[10:0]))
- **bit 11 specifies addressing mode**
  - ➢ **if =1, PC-relative: target address = PC + Sext(IR[10:0])**
  - ➢ **if =0, register: target address = contents of register IR[8:6]**

JSR



NOTE: PC has already been incremented during instruction fetch stage.

JSRR Instruction

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
JSRR	0	1	0	0	0	0	0	Base			0	0	0	0	0	0

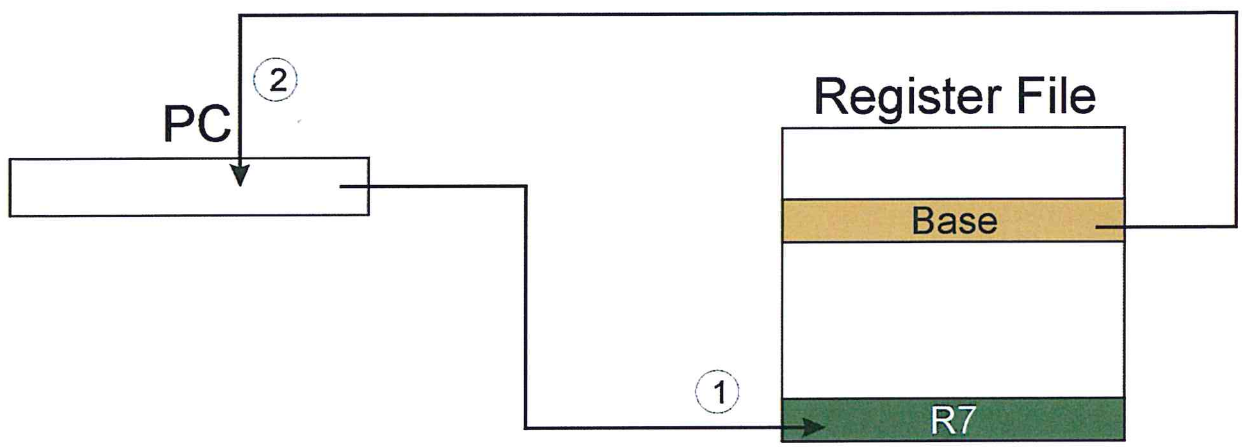
Just like JSR, except Register addressing mode.

- target address is Base Register
- bit 11 specifies addressing mode

What important feature does JSRR provide that JSR does not?

LIMITED RANGE

JSRR



NOTE: PC has already been incremented
during instruction fetch stage

Returning from a Subroutine

RET (JMP R7) gets us back to the calling routine.

- **just like TRAP**

Can a Subroutine Call Other Subroutines?

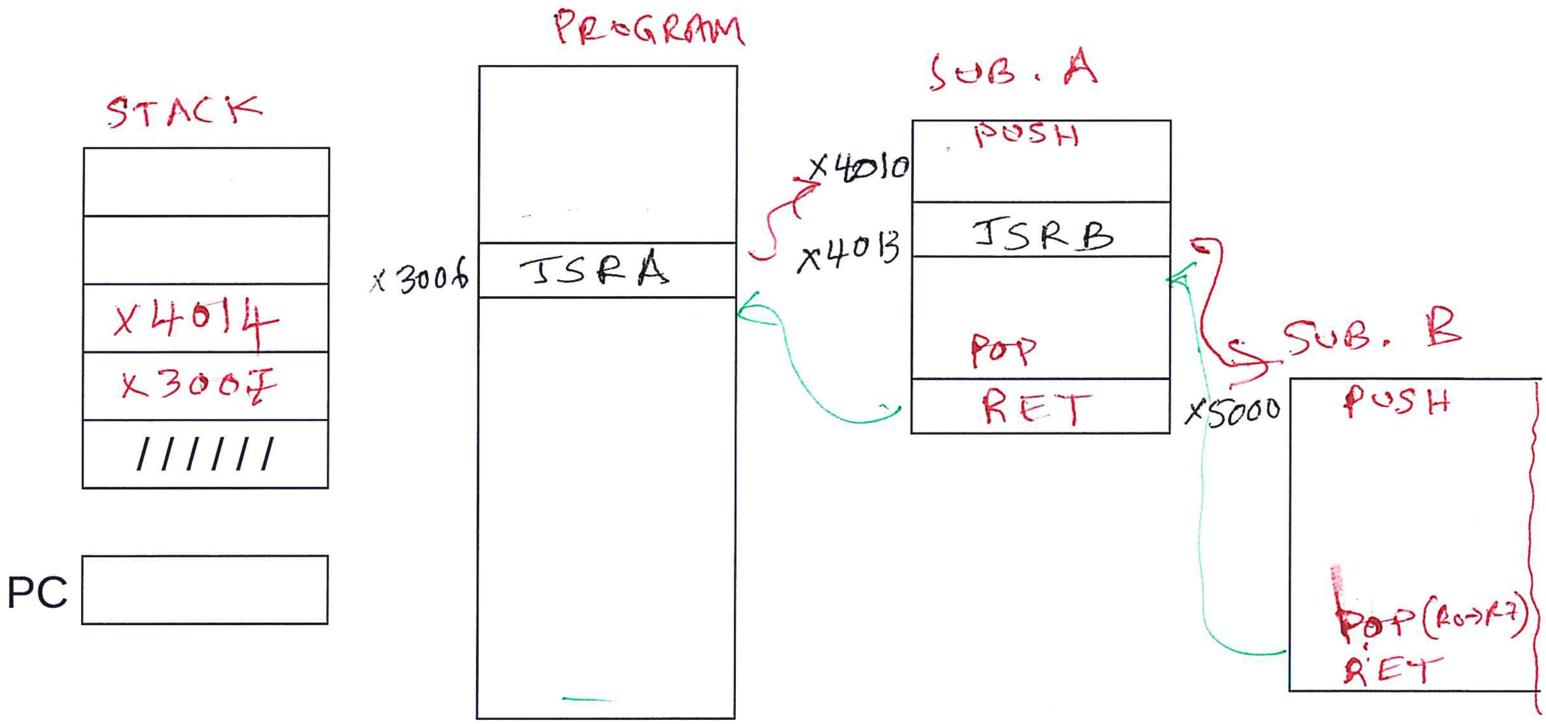
YES, BUT NEED TO SAVE R7.

WHAT ABOUT "NESTED" SUBROUTINES?

⇓

USE A STACK

Example



CAN A ROUTINE CALL ITSELF?

"RECURSION"

What is Recursion?

A **recursive function** is one that solves its task by **calling itself** on smaller pieces of data.

- Similar to recurrence function in mathematics.
- Like iteration -- can be used interchangeably; sometimes recursion results in a simpler solution.

Example: Running sum ($\sum_{1}^{n} i$)

Mathematical Definition:

RunningSum(1) = 1
RunningSum(n) =
    n + RunningSum(n-1)

Recursive Function:

```
int RunningSum(int n) {
  if (n == 1)
    return 1;
  else
    return n + RunningSum(n-1);
}
```

WHAT IS

$$\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \cdots$$

SAY

$$S = \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \cdots$$

$$2S = 1 + \underbrace{\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots}_{S}$$

$$2S = 1 + S$$

$$S = 1$$

TRIVIA

$$1 - \frac{1}{2} + \frac{1}{3} \cancel{+} \frac{1}{4} + \frac{1}{5} \cdots$$

GREATEST COMMON DIVISOR

EUCLID'S ALGORITHM

$GCD(N, M)$

IF $M = 0$, $GCD = N$

IF $N < M$, $= GCD(M, N)$

$GCD(N, M) = GCD(M, N - M)$

EXAMPLE

$GCD(64; 12)$

$= GCD(12, 52) = GCD(52, 12)$

$= GCD(12, 40) = GCD(40, 12)$

$= GCD(12, 28) = GCD(28, 12)$

$= GCD(12, 16) = GCD(16, 12)$

$= GCD(12, 4)$ ~~$= GCD(4, 12)$~~

$= GCD(4, 8) = GCD(8, 4)$

$= GCD(4, 4) = GCD(4, 0)$

$= 4$

GCD

SETUP
CALL GCD
END

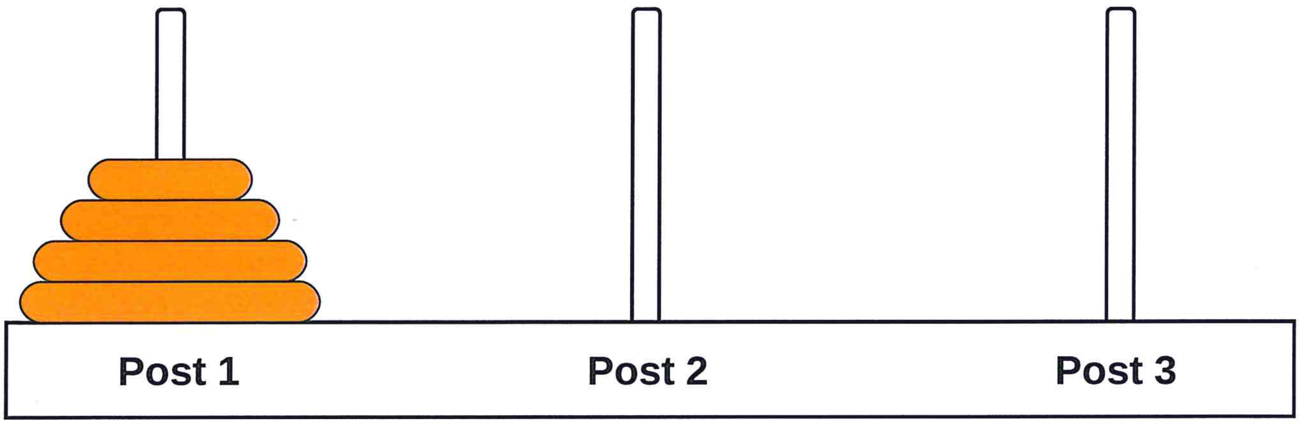
GCD : SAVE R7 (ON STACK)
<GCD ALGORITHM>
RESTORE R7
RETURN

```
; Program to compute GCD(n,m)
; Two numbers are stored in N and M
; Push return address on stack
;
; Call GCD
; Convert numbers to Ascii, print out
;
; GCD
; Put n,m in R0 and R1
; If n<m, swap
; if m = 0, then GCD = n, Return
; Call GCD (m, n-m)
; Return
; Stop
;
        .ORIG   x3000
;
        LEA     R6, StkBase      ;
        LDR     R6, R6, #0
        ADD     R6, R6, #-1      ; Initialize Stack Pointer
        LD      R0, N
        LD      R1, M            ; Get the two numbers
; Call GCD
        JSR     GCD              ; Call GCD (recursively)
        HALT
N       .FILL   #1
M       .FILL   #1
;
GCD     ADD     R2, R0, #0       ; Save R0
        ADD     R0, R7, #0       ; Return Address
        JSR     PUSH             ; Push Return Address on Stack
        ADD     R0, R2, #0       ; Restore R0
        NOT     R2, R0
        ADD     R2, R2, #1       ; Get -N in R2
        ADD     R2, R2, R1       ; M-N
        BRnz    NoSwap
        ADD     R2, R0, #0
        ADD     R0, R1, #0
        ADD     R1, R2, #0       ; Swapped R0 and R1
NoSwap  ADD     R1, R1, #0
        BRz     Done
        NOT     R3, R1
        ADD     R3, R3, #1       ; -M in R3
        ADD     R3, R3, R0       ; N-M in R3
        ADD     R0, R1, #0       ; M in R0
        ADD     R1, R3, #0       ; N-M in R1
        JSR     GCD
Done    ADD     R2, R0, #0       ; Save R0
        JSR     POP
        ADD     R7, R0, #0       ; Restore return address
        ADD     R0, R2, #0       ; Restore R0
        RET
;
;       PUSH, POP Routines
;
; Subroutines for carrying out the PUSH and POP functions.  This
; program works with a stack consisting of memory locations x3FFF
; (BASE) through x3FFB (MAX).  R6 is the stack pointer.
;
POP             ST      R2,Save2        ; are needed by POP.
                ST      R1,Save1
                LD      R1,BASE         ; BASE contains -x3FFF.
                ADD     R1,R1,#-1       ; R1 contains -x4000.
                ADD     R2,R6,R1        ; Compare stack pointer to x4000
```

```
                BRz     fail_exit       ; Branch if stack is empty.
                LDR     R0,R6,#0        ; The actual "pop."
                ADD     R6,R6,#1        ; Adjust stack pointer
                BRnzp   success_exit
PUSH            ST      R2,Save2        ; Save registers that
                ST      R1,Save1        ; are needed by PUSH.
                LD      R1,MAX          ; MAX contains -x3FFB
                ADD     R2,R6,R1        ; Compare stack pointer to -x3FFB
                BRz     fail_exit       ; Branch if stack is full.
                ADD     R6,R6,#-1       ; Adjust stack pointer
                STR     R0,R6,#0        ; The actual "push"
success_exit    LD      R1,Save1        ; Restore original
                LD      R2,Save2        ; register values.
                AND     R5,R5,#0        ; R5 <-- success.
                RET
fail_exit       LD      R1,Save1        ; Restore original
                LD      R2,Save2        ; register values.
                AND     R5,R5,#0
                ADD     R5,R5,#1        ; R5 <-- failure.
                RET
BASE            .FILL   xC001           ; BASE contains -x3FFF.
MAX             .FILL   xC005
Save1           .FILL   x0000
Save2           .FILL   x0000
;
StkBase          .FILL   x4000
        .END
```

High-Level Example: Towers of Hanoi

Task: Move all disks from current post to another post.



Rules:

(1) Can only move one disk at a time.

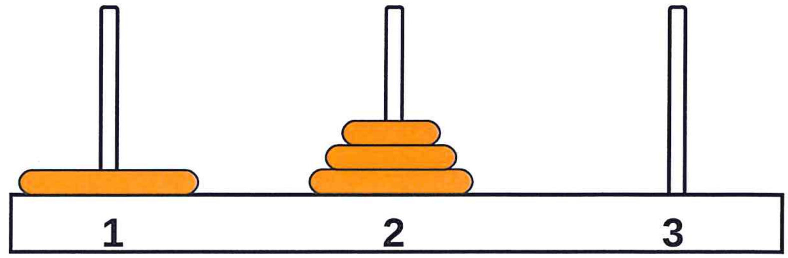
(2) A larger disk can never be placed on top of a smaller disk.

(3) May use third post for temporary storage.

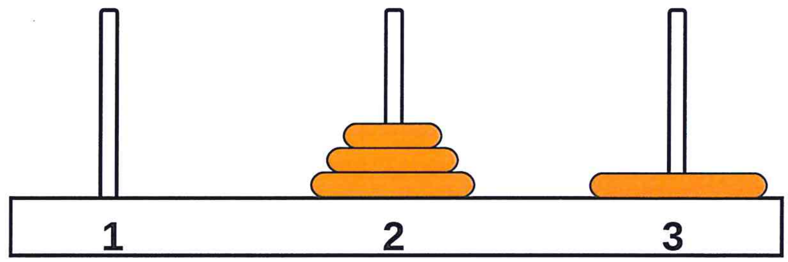
Task Decomposition

Suppose disks start on Post 1, and target is Post 3.

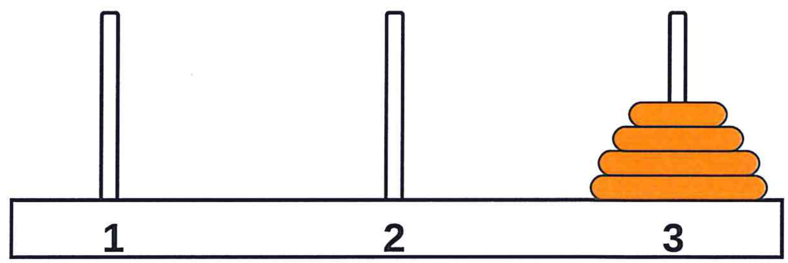
1. Move top n-1 disks to Post 2.



2. Move largest disk to Post 3.



3. Move n-1 disks from Post 2 to Post 3.



Task Decomposition (cont.)

Task 1 is really the same problem,
with fewer disks and a different target post.

- "Move n-1 disks from Post 1 to Post 2."

And Task 3 is also the same problem,
with fewer disks and different starting and target posts.

- "Move n-1 disks from Post 2 to Post 3."

So this is a recursive algorithm.

- The terminal case is moving the smallest disk -- can move directly without using third post.
- Number disks from 1 (smallest) to n (largest).