Recursion

Recursion captures the idea of something defined in terms of itself

↳ point a video camera @ a TV displaying what the video camera is recording

↳ point two mirrors at each other

↳ A function can call itself

- Solution to puzzles (Tower of Hanoi)
- Searching/Sorting (Binary/Merge)
- computations (Euclidean Algorithm)

↳ sequences defined by recurrence relations

⚠ we need to be very careful not to create circular definitions ⚠

i.e. All definitions must be in terms of simpler definitions

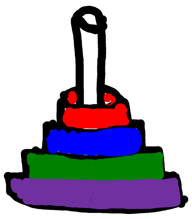
e.g. English

We cannot use the word we are attempting to define in it's own definition. Words must be defined in terms of simpler, already established words

Similarly, if a function were to call itself with the same input/arguments, this would result in an infinite loop & the program would not terminate

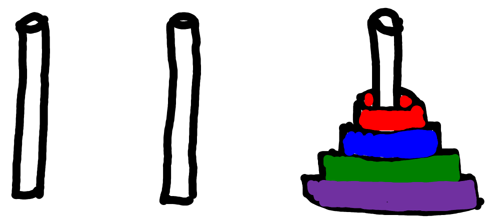
Towers of Hanoi

Starting Position



- n disks on the leftmost peg stacked according to size

Ending Position (Goal)

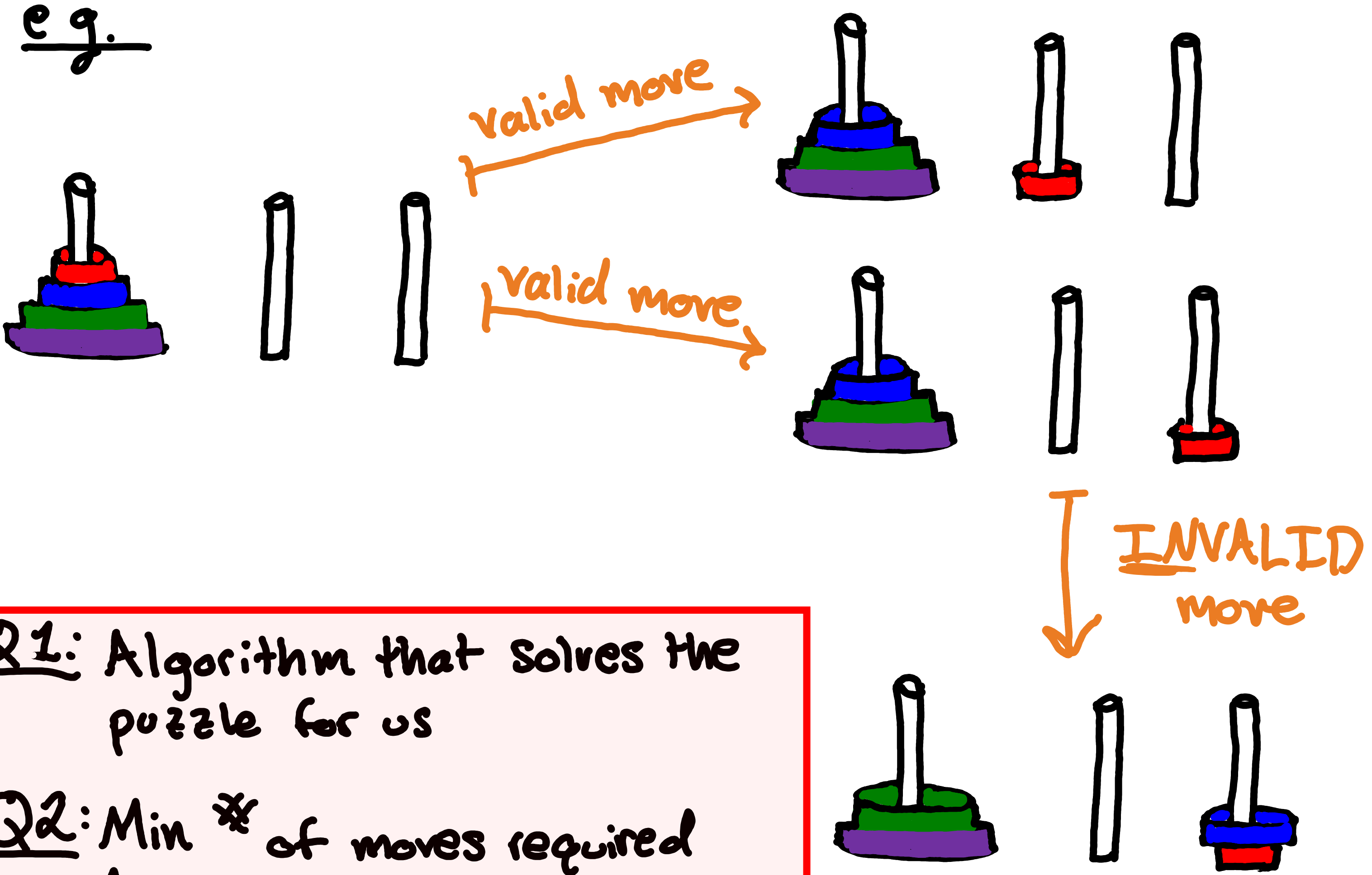


- n disks on the rightmost peg stacked according to size

Rules

you can move the top disk of any stack to another peg so long as you never place a larger disk on top of a smaller one.

e.g.



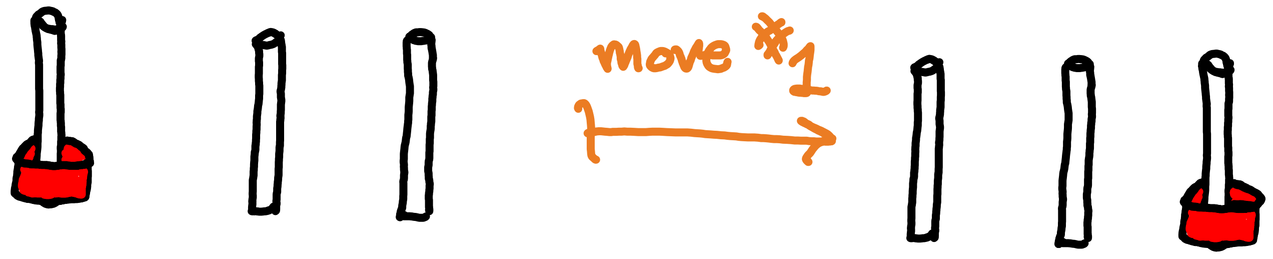
Q1: Algorithm that solves the puzzle for us

Q2: Min # of moves required to solve a puzzle w/ n disks

the blue disk is larger than the red disk so it cannot be placed on the red one.

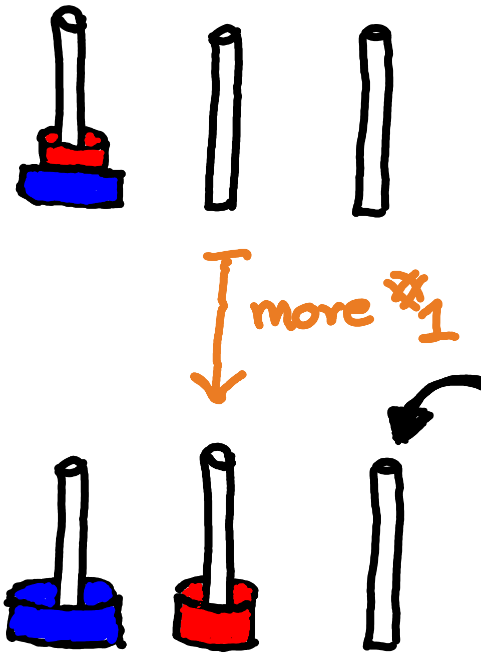
Let's Play!

n=1 (only a single disk)



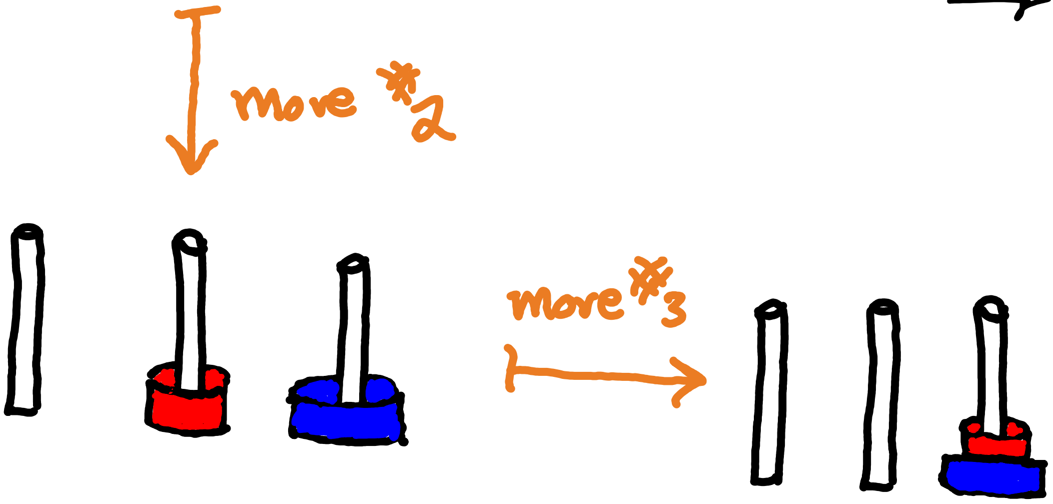
GAME OVER, WE WIN!

n=2



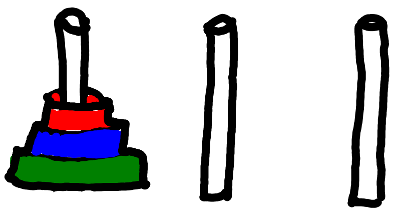
Let's think before we act.

- We know the red disk must be moved first (it is the only disk at the top of a stack)
- We want the blue disk to end @ the bottom of the rightmost peg. (this is half of the "goal" in this case)
- Therefore, let's keep that third peg open for blue to move to (red is the smallest disk, so it will <u>always</u> be able to move to <u>any</u> peg at <u>any</u> time.



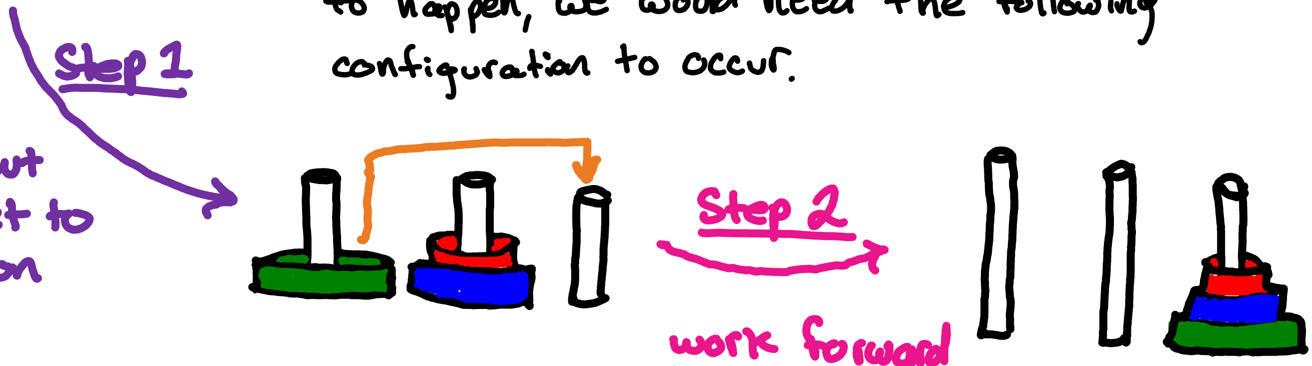
GAME OVER WE WIN

n=3   we need to find a strategy that works in general.



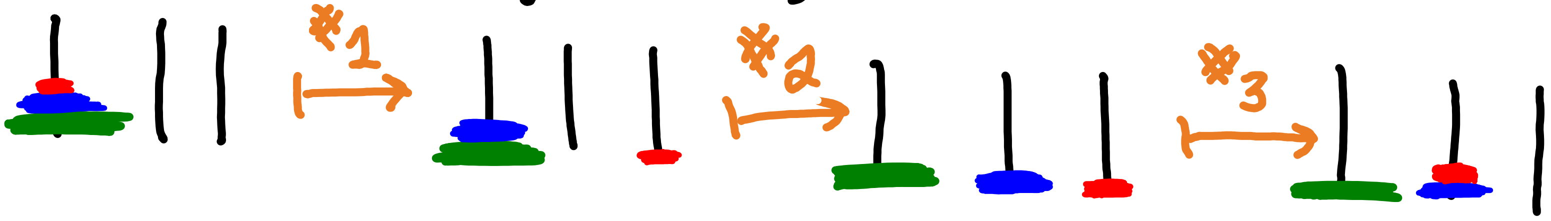
Idea: We would like to move green (i.e. the largest disk) directly to the rightmost peg the first (and only) time it is moved. For this to happen, we would need the following configuration to occur.

work backward to figure out how to get to this position



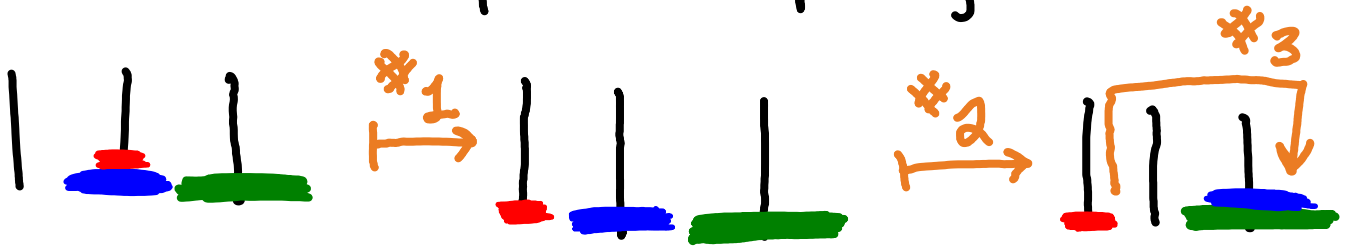
work forward toward goal from there

Step 1: We need the blue disk on the bottom of the middle peg so we begin by moving red to the rightmost peg (leaving the middle open for blue)



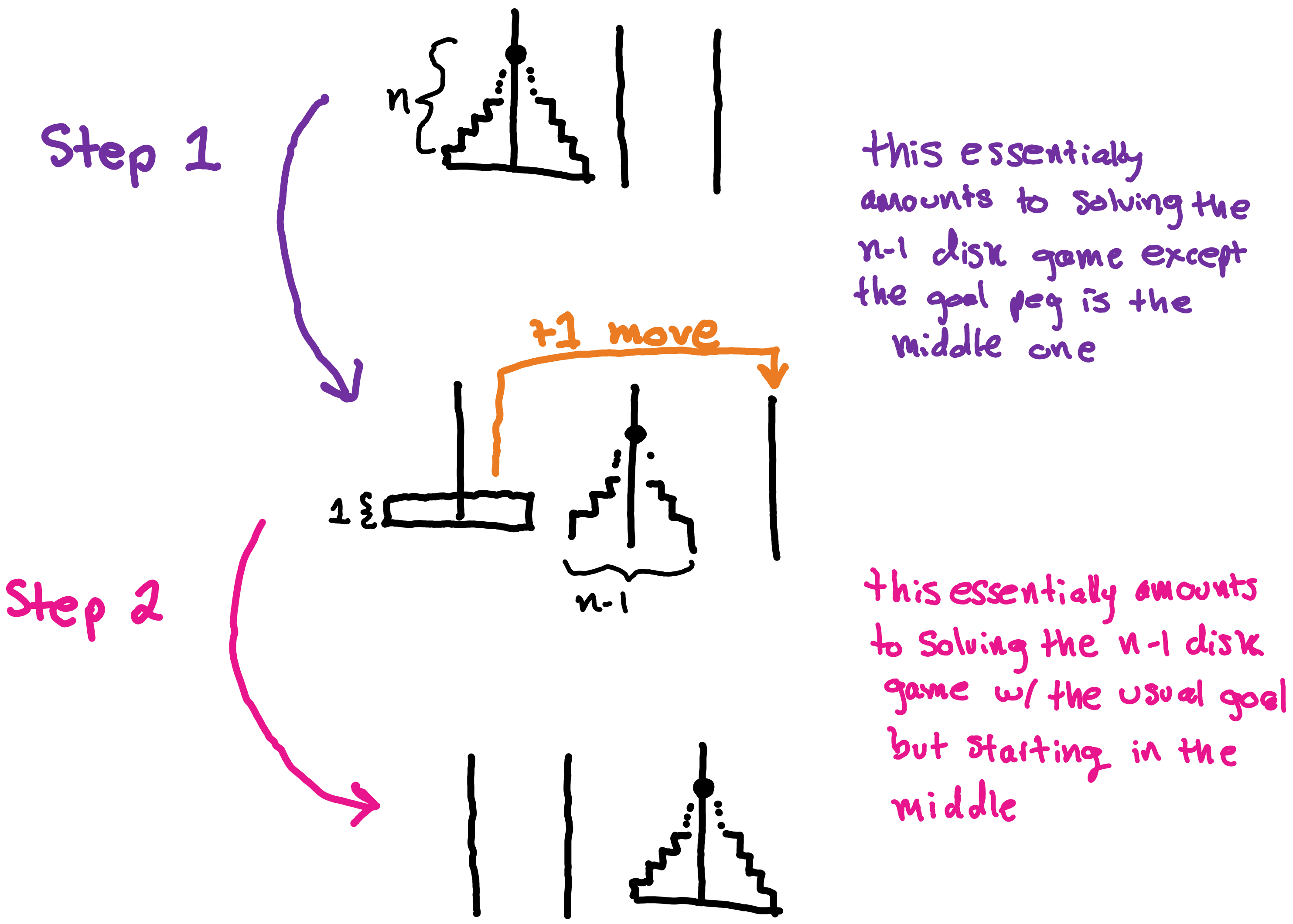
Use 3 moves = # moves required for 2 disk problem

Step 2: We just need to move red off blue so it can be placed on top of green



Use 3 moves = # moves required for 2 disk problem

The n-disk game



Step 1
n
this essentially amounts to solving the n-1 disk game except the goal peg is the middle one
+1 move
1
n-1
Step 2
this essentially amounts to solving the n-1 disk game w/ the usual goal but starting in the middle

The recursive relationship

$H_n :=$ min # of moves required to solve the n-disk game

$$H_n = H_{n-1} + 1 + H_{n-1}$$

$$= 2H_{n-1} + 1 \quad \text{(this looks familiar)}$$

$\Longrightarrow$ (lesson 20) $\boxed{H_n = 2^n - 1}$

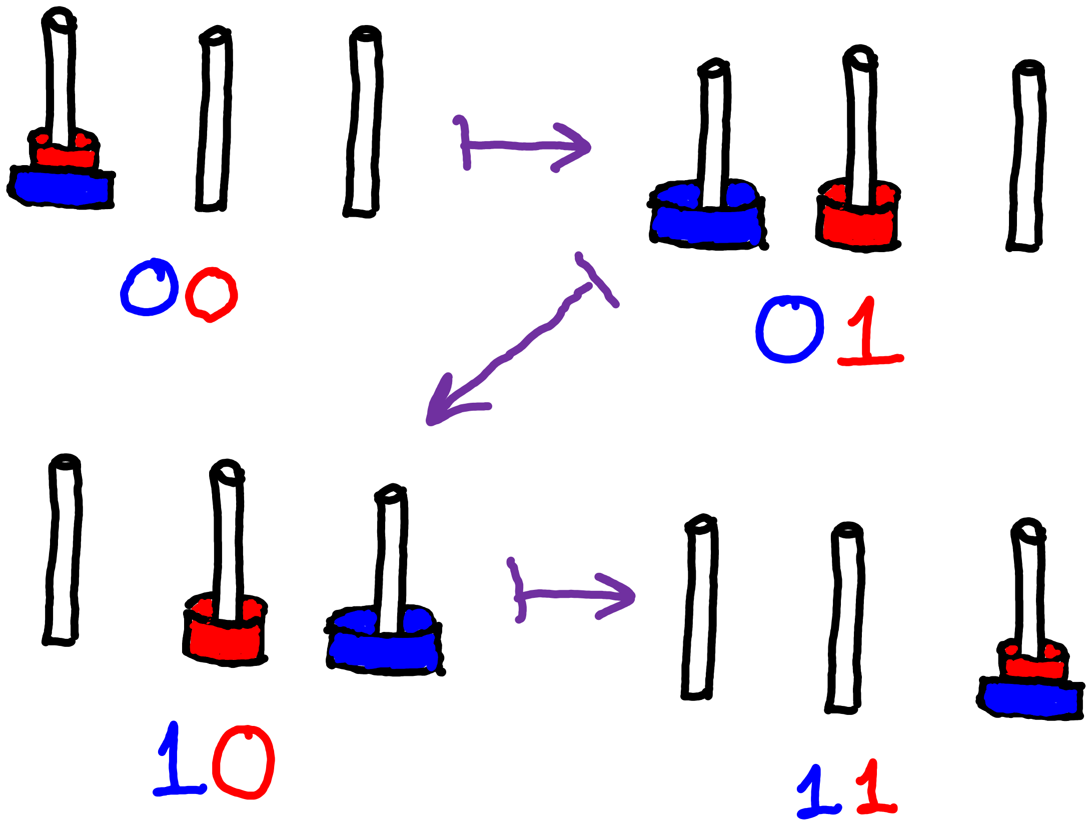
this answers both questions 1 & 2.

Another way to view the algorithm

The presence of 2^n suggests a connection to binary counting

n-disk game	counting from 0 to 2^n-1
k^{th} largest disk	k^{th} binary digit
move the k^{th} largest disk to a new peg	flip the k^{th} bit & all bits to the right of it

e.g. $n=2$



Note: This works for any n & even though binary counting won't help us know which peg to move the next disk to, there will only be 1 legal move any time the smallest disk is not being moved.

Q: How does the top disk move?

A: It depends on whether n is even or odd

<u>If n is odd</u>: first move takes smallest disk to far right peg

<u>If n is Even</u>: first move takes smallest disk to middle peg

& after the first move it should be clear how to move the smallest disk keeping the goal in mind.

<u>pseudocode</u>

<u>Goal</u>: print step by step instructions for solving the n disk towers of Hanoi game.

```
Towers(n, A, B, C)
    If n = 0 then return

    Else return Towers(n-1, A, C, B)
        print "Move top disk of (peg) A to (peg) C"
        Towers(n-1, B, A, C)
```

Each call of the Towers function w/ input n results in two recursive calls to the Towers function w/ input n-1, & 1 instruction is printed unless n=0 in which case the function returns.

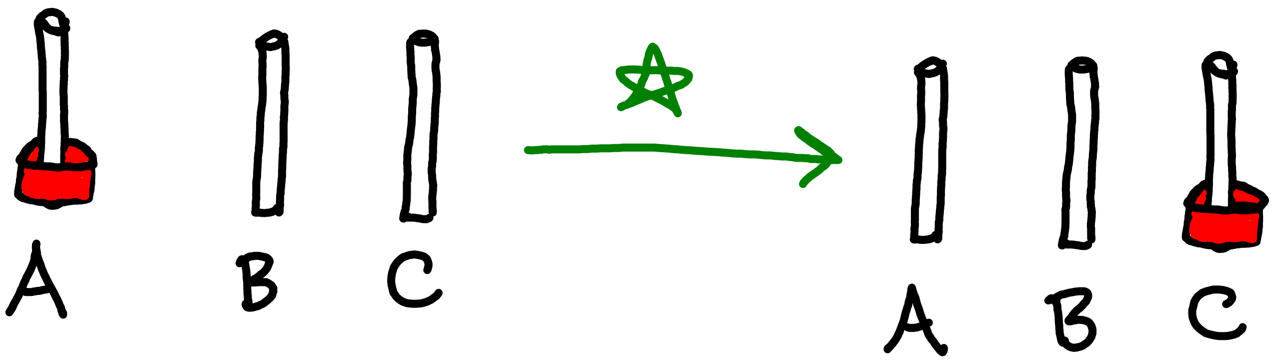
Tracing the code on small examples

Towers(1, A, B, C)

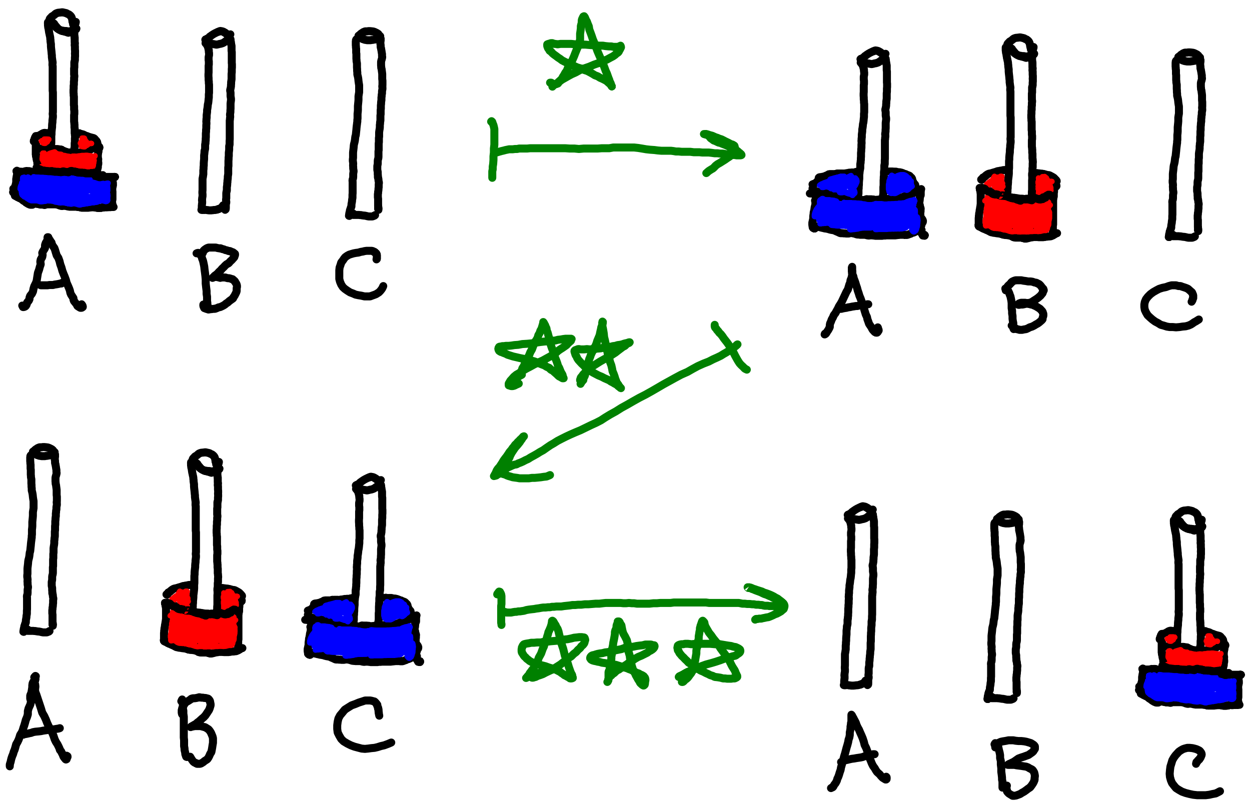
  Towers(0, A, B, C) ✓ Return

  Move top disk of A to C ☆

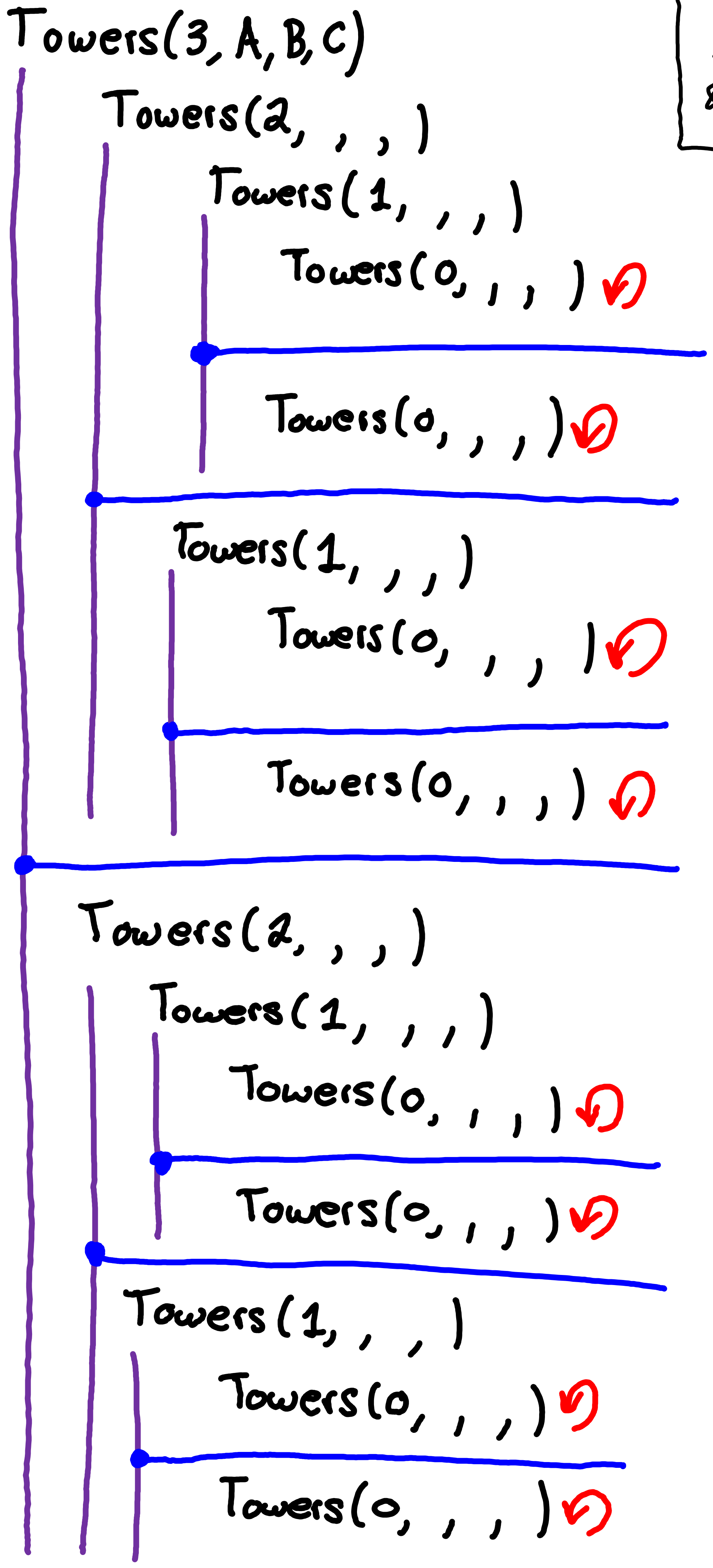
  Towers(0, B, A, C) ✓ Return



```
Towers(2, A, B, C)
    Towers(1, A, C, B)
        Towers(0, A, B, C) ↩ Return
        Move top disk of A to B
        Towers(0, C, A, B) ↩ Return
    Move top disk of A to C                          ☆☆
    Towers(1, B, A, C)
        Towers(0, B, C, A) ↩ Return
        Move top disk of B to C                      ☆☆☆
        Towers(0, A, B, C) ↩ Return
```

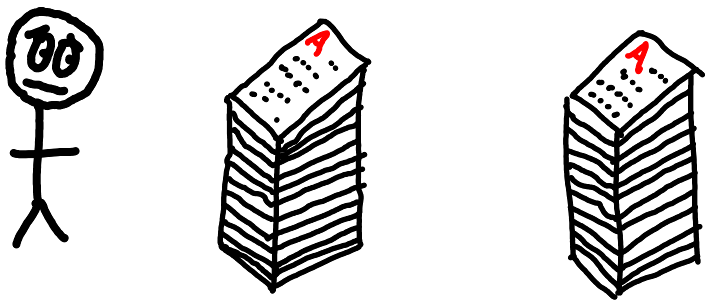


Exercise: fill in the missing arguments & print statements



Merge Sort

Merging: two branches of a company are moving into a new unified office space. Both branches keep their records in alphabetical order & now the combined collection of records must be alphabetized. As the unpaid intern, this task is delegated to you.



Solution: Compare the top two papers of each stack. Place the one that comes first (in the dictionary order) face down on the table & repeat.

L_1, L_2 (sorted lists) **pseudocode** L merged list w/ elements in increasing order

```
L := empty list
while L_1 & L_2 both nonempty,
remove smaller of first elements of L_1
& L_2 from its list; put it at the end of L
if this removal makes one list empty
then remove all elements from the other
list & append them to L
return L
```

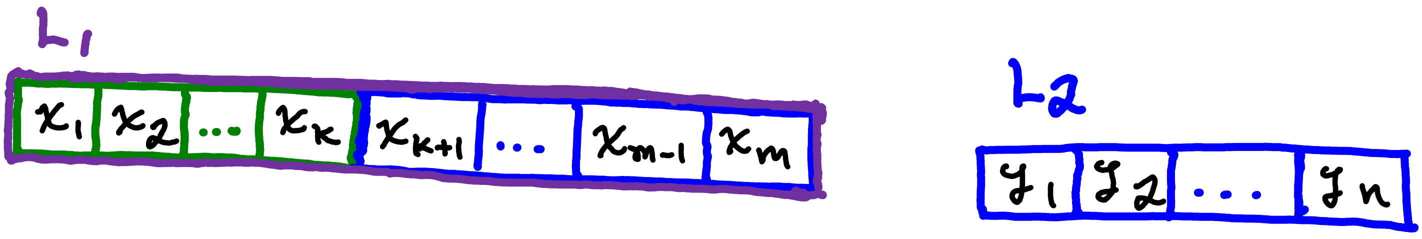
Elementary Operation

(i) comparisons of integers

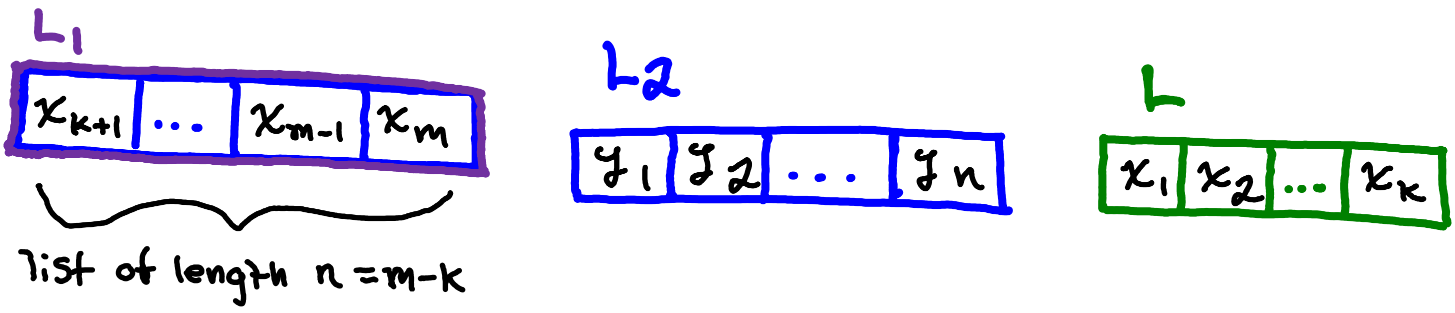
Suppose L_1 is a list of length m
& L_2 is a list of length n

$\Longrightarrow$ $m+n-1$ comparisons are required in the worst case scenario.

proof: Suppose (without loss of generality) that $m \geqslant n$ & moreover $n+k=m$ where $k \in \mathbb{N} = \{0,1,2,\dots\}$.



If $x_i < y_1 \ \forall i \leq k$ then after k comparisons we have



Now, if $x_{k+i} < y_i < x_{k+i+1} \quad \forall i \leq n$

(i.e. the merge algorithm alternates between appending an element from L_1, then one from L_2 until both are empty)

there will be $2n-1$ comparisons in total before the merge is complete.

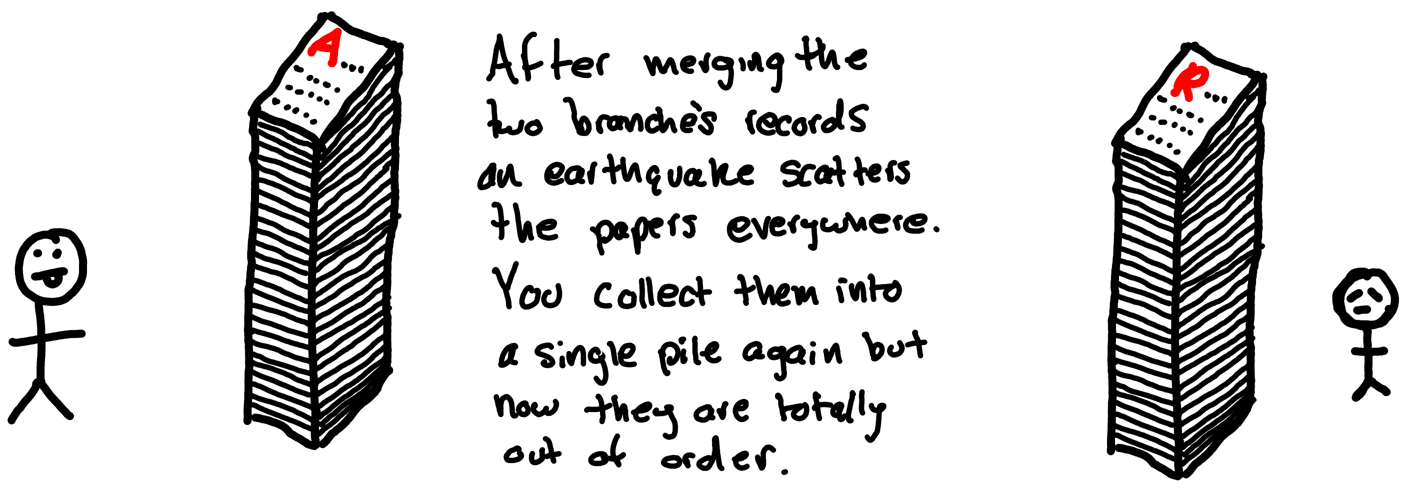
e.g. $\boxed{2}\boxed{5}\boxed{10}$ $\boxed{3}\boxed{7}\boxed{11}$ $(n=3)$

Comparisons			
1	$2<3$	$\Rightarrow$	$\boxed{2}$
2	$3<5$	$\Rightarrow$	$\boxed{2}\boxed{3}$
3	$5<7$	$\Rightarrow$	$\boxed{2}\boxed{3}\boxed{5}$
4	$7<10$	$\Rightarrow$	$\boxed{2}\boxed{3}\boxed{5}\boxed{7}$
5	$10<11$	$\Rightarrow$	$\boxed{2}\boxed{3}\boxed{5}\boxed{7}\boxed{10}$

At this point L_1 is empty & L_2 has 1 element

this proves max # comparisons used in the merge algorithm $\geq n+m-1$

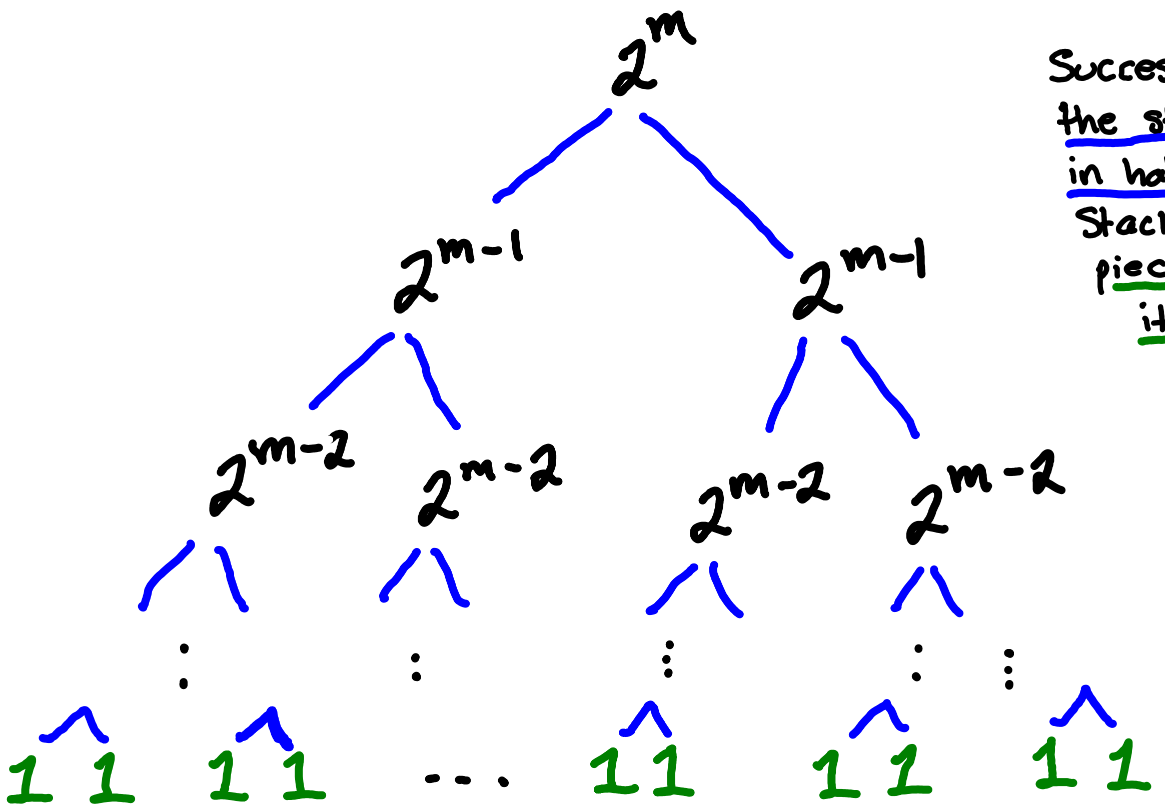
However, there can be at most this many comparisons used since each comparison results in one list decreasing in length by 1 & once one list is empty no elements from the remaining list (there is at least 1 such) needs to be compared w/ anything else.



After merging the two branches records an earthquake scatters the papers everywhere. You collect them into a single pile again but now they are totally out of order.

For Simplicity, we assume the # of papers is a power of 2, say 2^m The boss understands that sorting the papers this time will require much more work than simply merging two collections of already alphabetized stacks, so he assembles a team of people (including yourself) to fix this mess.

Strategy: Divide & Conquer

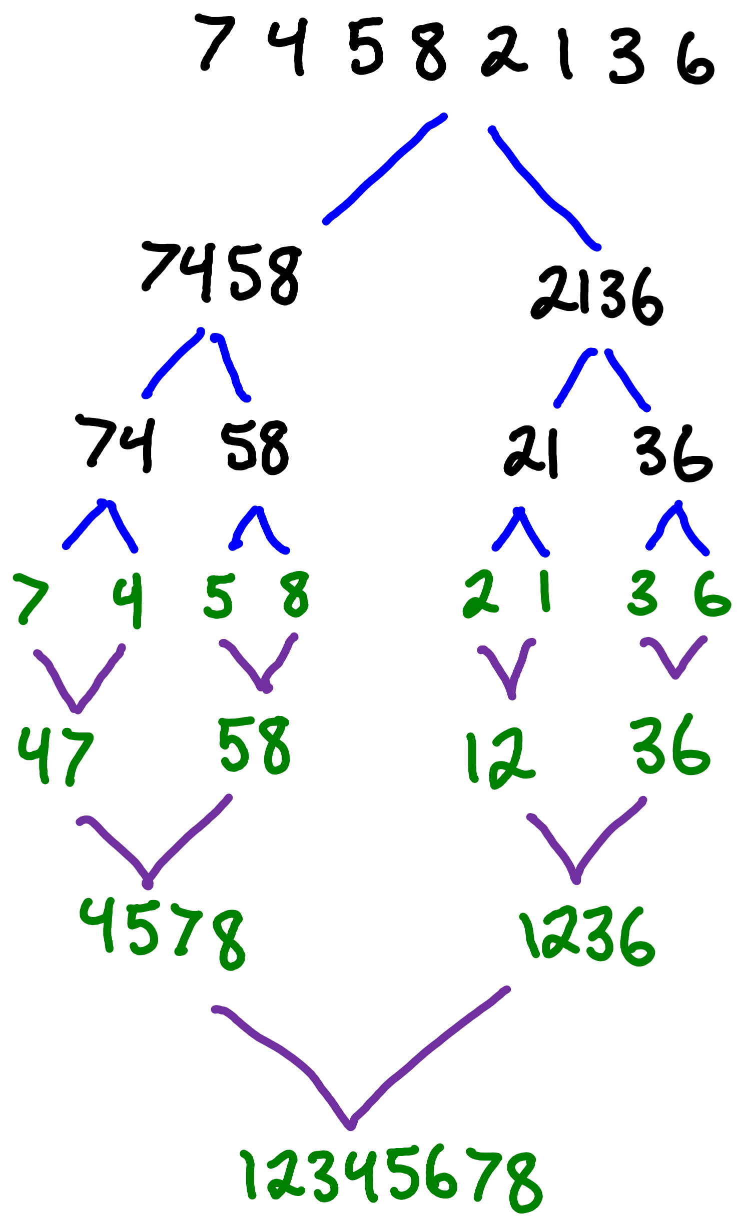


Successively divide the stack of papers in half until each Stack has only one piece of paper in it

Then, undo this separation process by successively merging the two halves of a stack

(this requires us to keep track of the heirarchy of stacks/divisions)

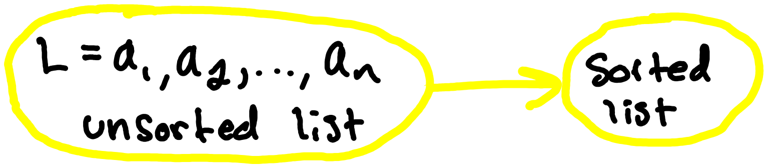
e.g.



🟦 = divide

🟪 = merge

🟩 = Sorted



Pseudocode

```
mergesort(L)
If n>1 Then
    $m := \lfloor n/2 \rfloor$
    $L_1 := a_1, \ldots, a_m$
    $L_2 := a_{m+1}, \ldots, a_n$
    $L :=$ merge(mergesort($L_1$), mergesort($L_2$))
Return L
```

Complexity of Mergesort

Elementary Operation

(i) comparisons of integers

1 Stack w/ 2^m papers in it

2 Stacks w/ 2^{m-1} papers in each

4 Stacks w/ 2^{m-2} papers in each

8 Stacks w/ 2^{m-3} papers in each

$\vdots$

2^k Stacks w/ 2^{m-k} papers in each

$\vdots$

2^m Stacks w/ 1 paper in each

Merging 2^k Stacks requires k uses of the merge function (each using at most $2 \cdot 2^{m-k} - 1$ comparisons)

Hence, $\sum_{k=1}^{m} 2^{k-1}(2^{m-k+1} - 1)$ is the # of comparisons needed

$$\sum_{k=1}^{m} 2^{k-1}(2^{m-k+1}-1) = \sum_{k=1}^{m} 2^m - 2^{k-1}$$

$$= \sum_{k=1}^{m} 2^m - \sum_{k=1}^{m} 2^{k-1}$$

$$= m2^m - (2^m - 1)$$

Setting $n = 2^m$ $(m = \log n)$ this becomes

$$n\log(n) - n + 1 \in O(n \log n)$$

Fact: Any $\underbrace{\text{comparison-based}}_{\text{the basic operation is comparisons of elements}}$ sorting algorithm

has at best $O(n \log n)$ time complexity.

Remark: if one happens to have special knowledge about the configuration of elements in the list to be sorted, then it may be possible to beat $n \log n$ but there is no such hope in general.