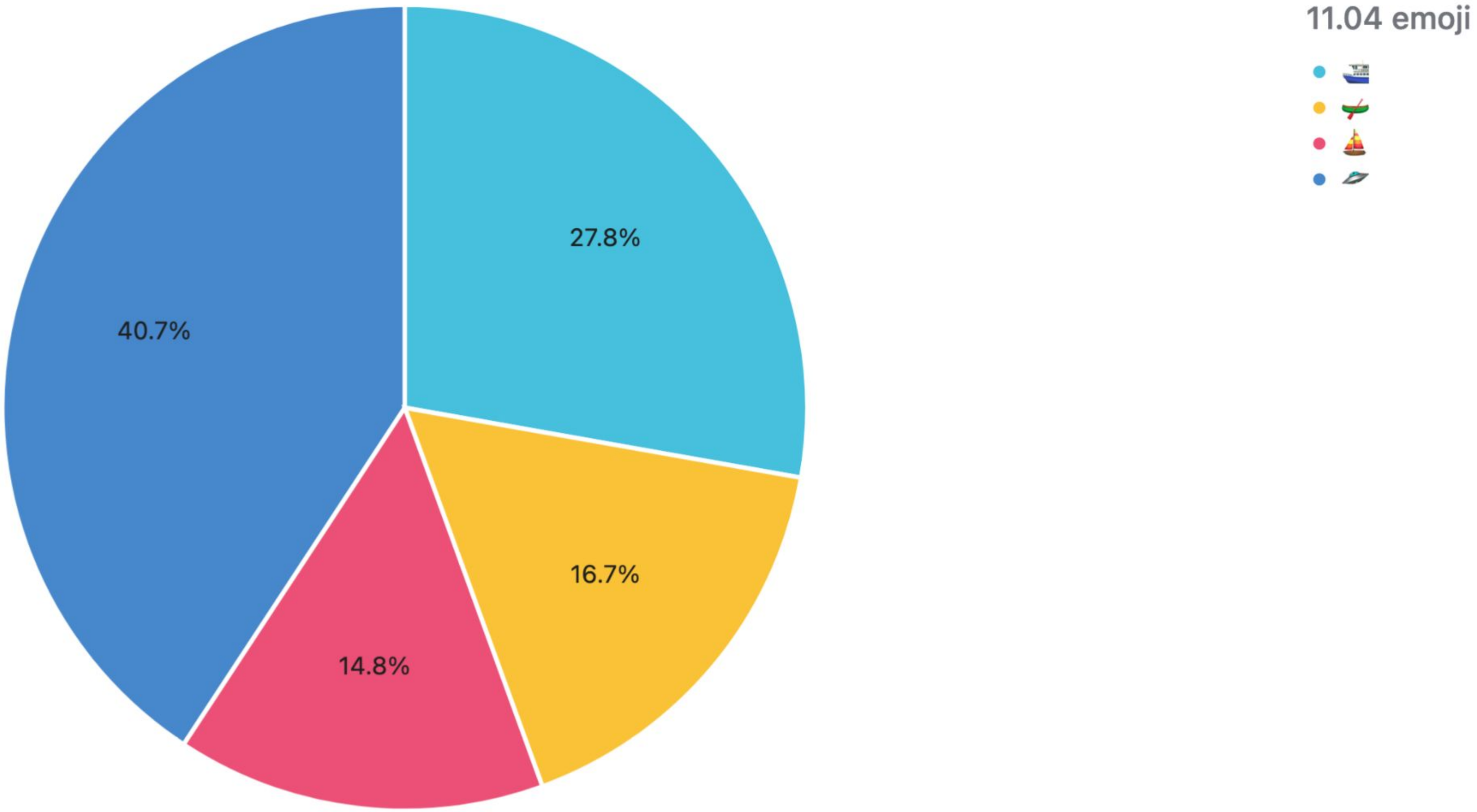
CS 103000
Prof. Madeline Blount

Week 11/12:
ALGORITHMS

https://proton-cs103-11.glitch.me/

Dall-E 2: cats learning C++ in the forest on '90's technology


11.04 emoji
● ⛴️
● 🛶
● ⛵
● 🛸
27.8%
16.7%
14.8%
40.7%

🧹 housekeeping

- extra credit, posted by Wed. - options, Dec.
- PROJECTS! also posted by Wed.
- No LAB/HOMEWORK for 2 weeks!
- Friday lab = project time
- pair programming: due end of day SUNDAY

There are layers of poesis around practice, cycles, and loops that I am trying to get under and articulate. Layers of salt, layers of soil, layers of sound, layers of soul that it builds up and breaks down only to rebuild again. It has something to do with the creative practice and simultaneously forgetting and remembering each other. It has everything to do with the lifecycle of the forest and teaching us about our ability to surrender long enough to let spirit show you the safety inside the regenerating life-force of faith. Is forgetting part of the practice? Forgetting in the face of fear only to remember in the face of another. Does the forgetting facilitate the rebirth? Must I always leave to return? What if the practice isn't staying in remembrance, what if the practice is creating a safe space for the cycle of forgetting — maintaining the garden of longing so we never stop looking? I'm learning to surrender control, let the saprophytes of my spirit do their job, and find safety in the cycle. To practice presence is a recursive move. A practice of a practice. See you in class tomorrow.

Recursion in JavaScript

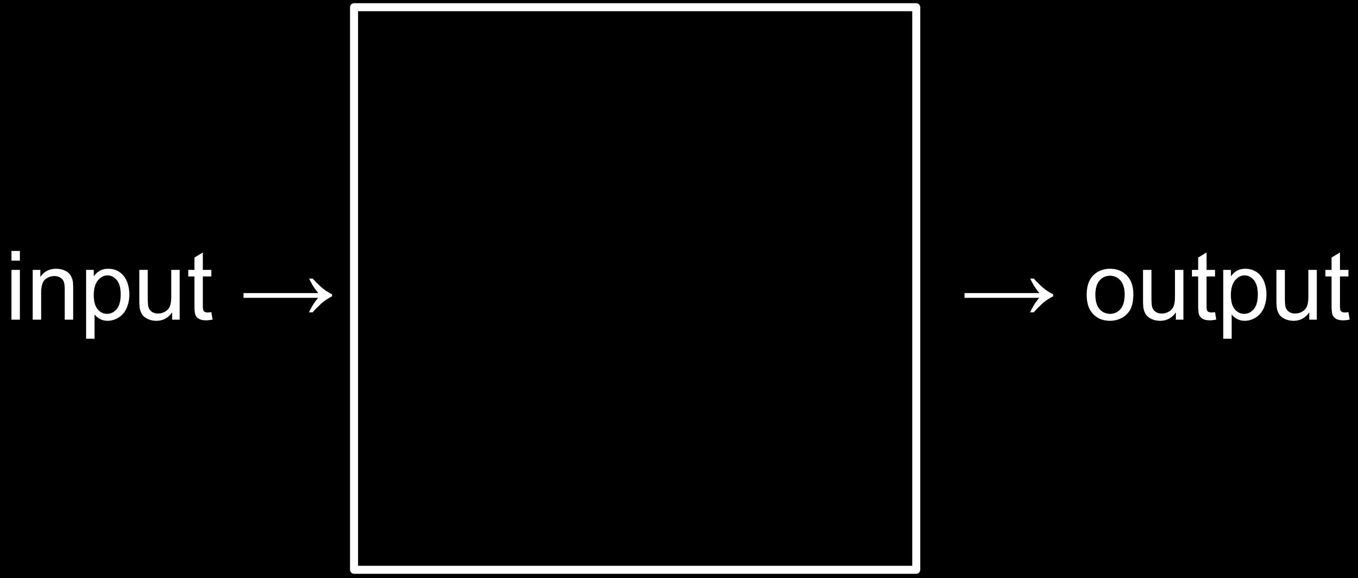
From inside the function of practicing presence my feelings of safety compound. Below is an example of recursion in JavaScript where my `safetyBaromenter` increments with each recursive loop.

```
const practicePresence = (safetyBarometer) => {
  meditate()
  write()
  walkWithTrees()
  selfPortrait()
  soakInWater()
  practicePresence(safetyBarometer++)
};

practicePresence(0)
```

To learn more about recursion and the increment operator in JavaScript check out the MDN docs linked above.

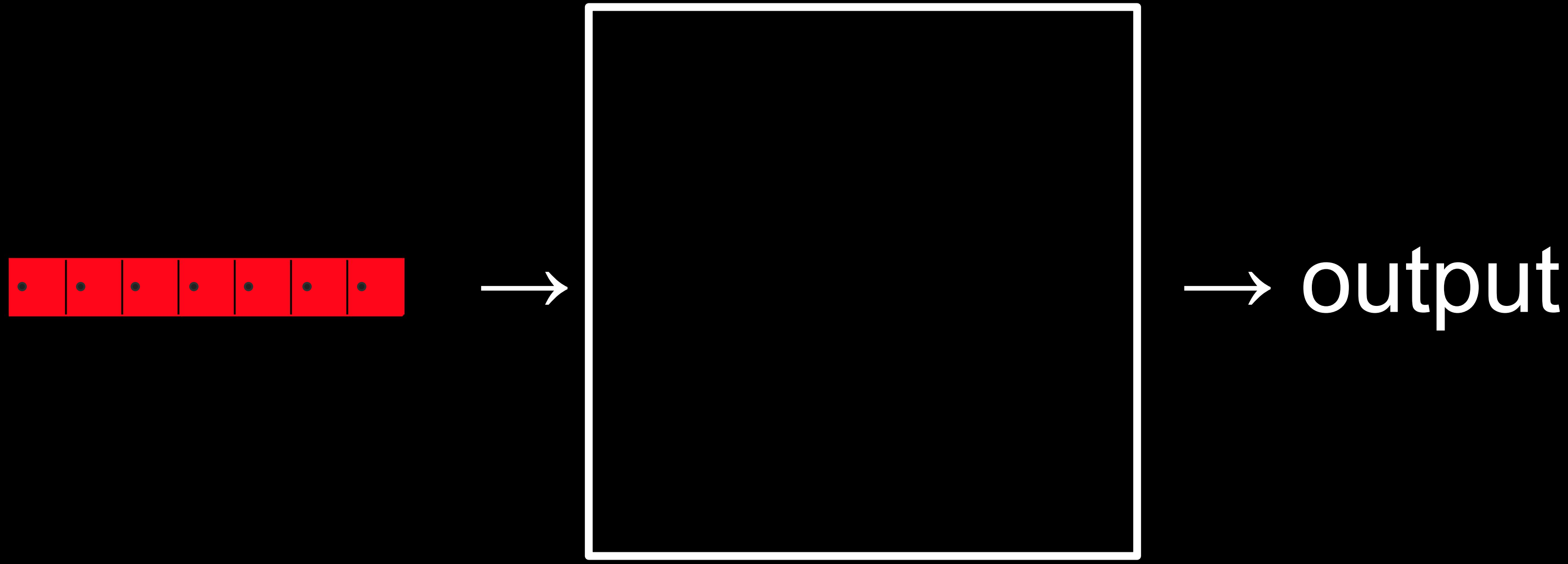
RECURSION + STACK:
spinning plates



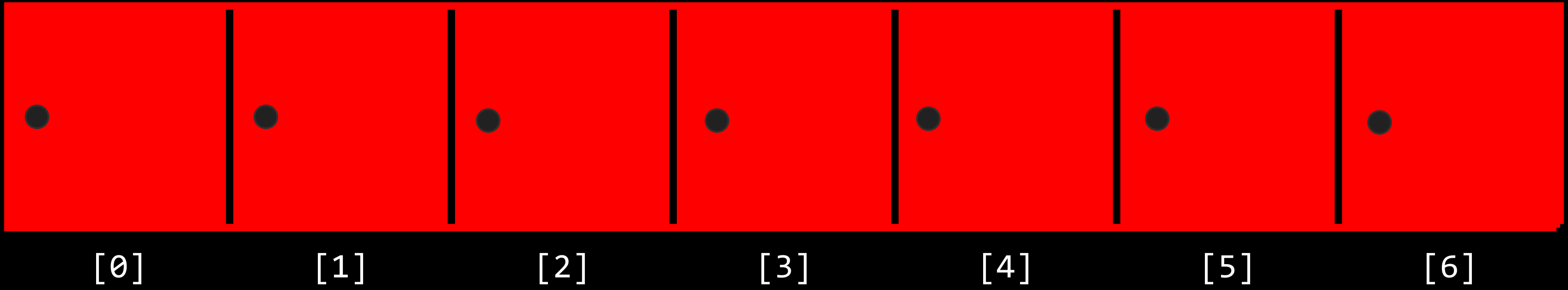
from Harvard CS50

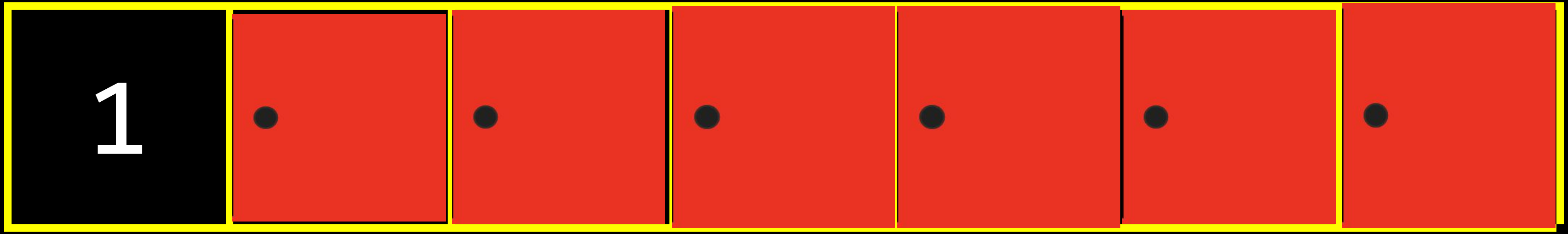
algorithm

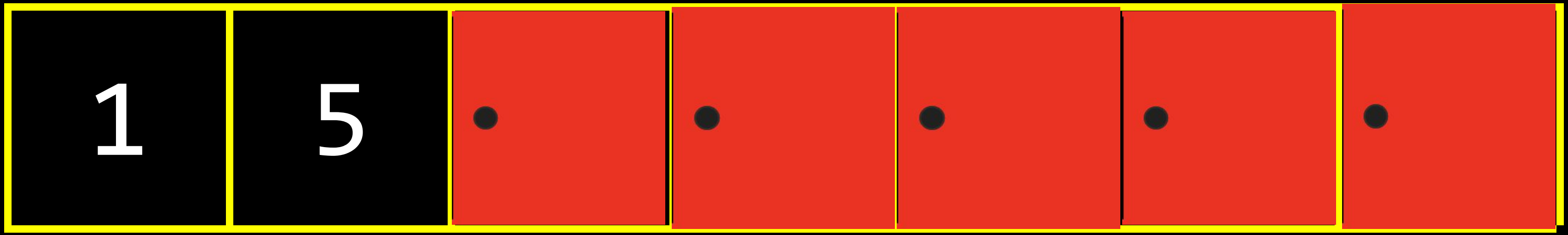
SEARCH

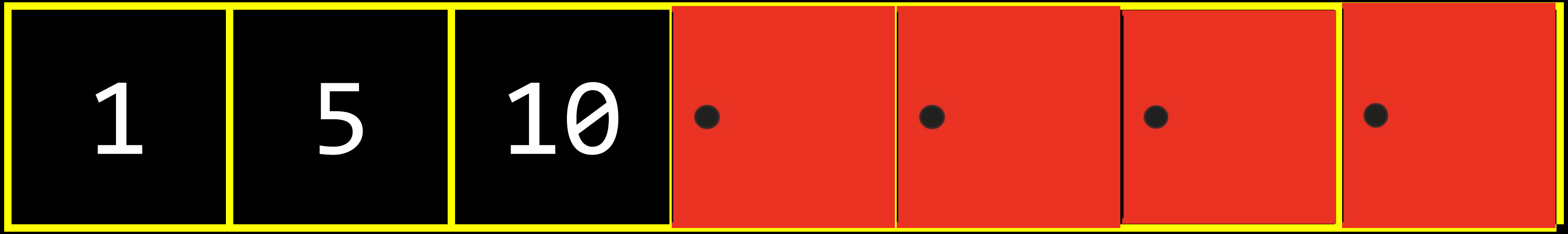


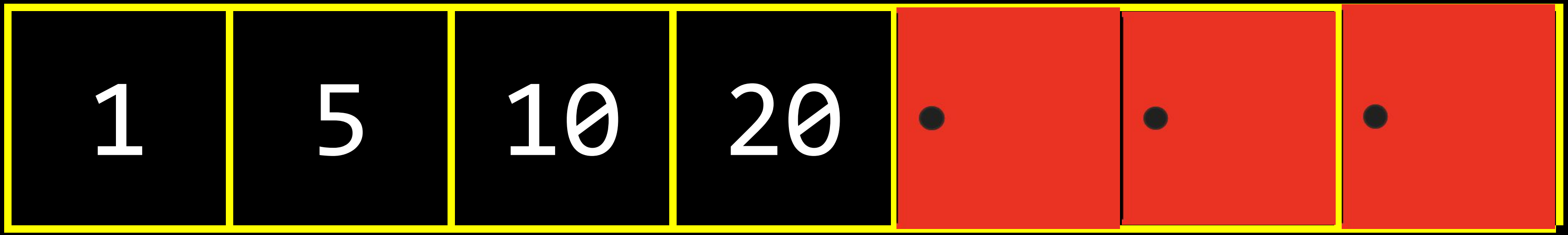
SEARCH

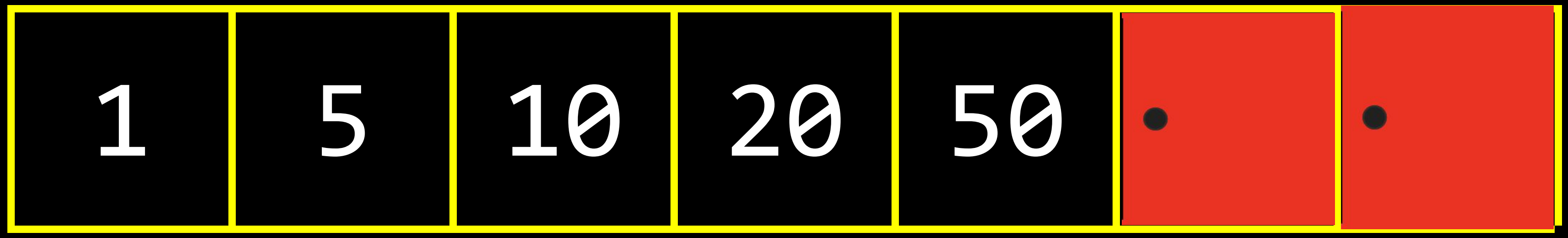


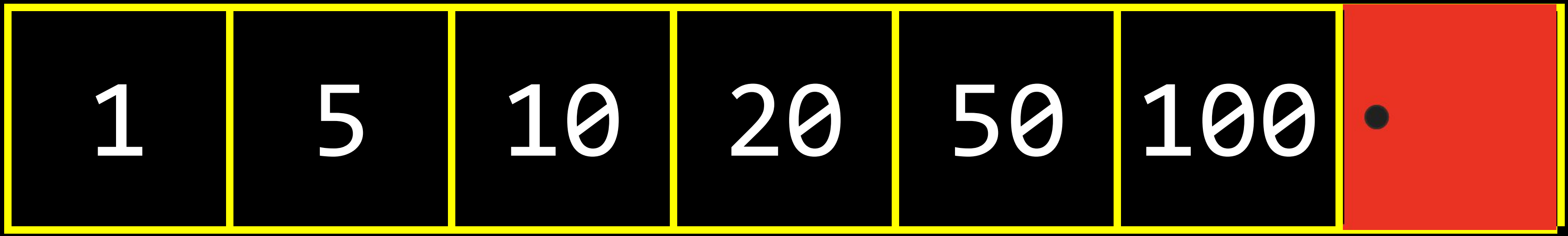

1


1
5

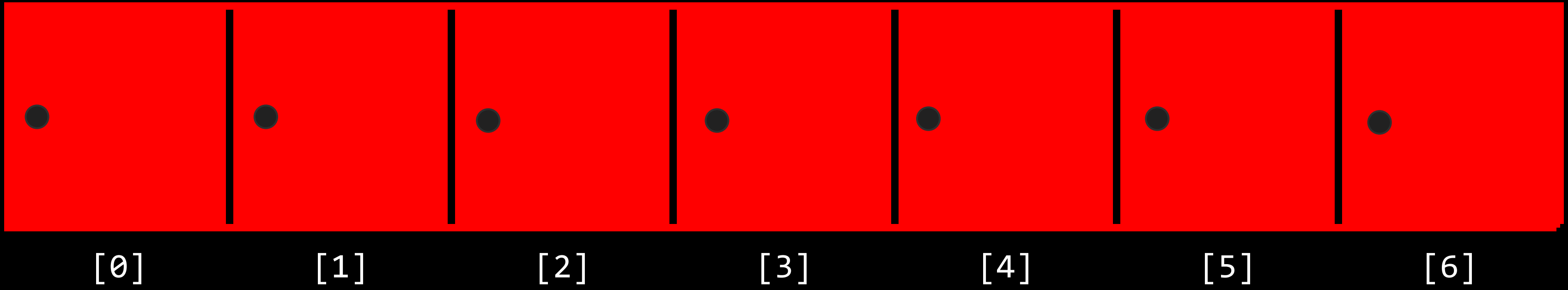

1
5
10


1
5
10
20

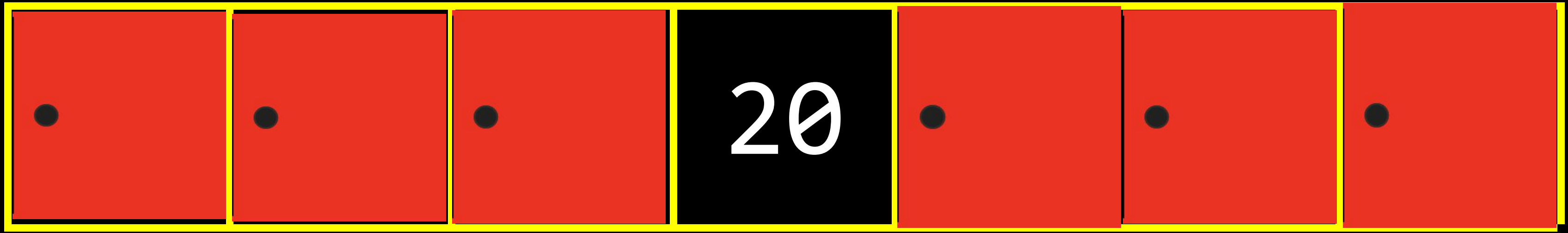

1
5
10
20
50

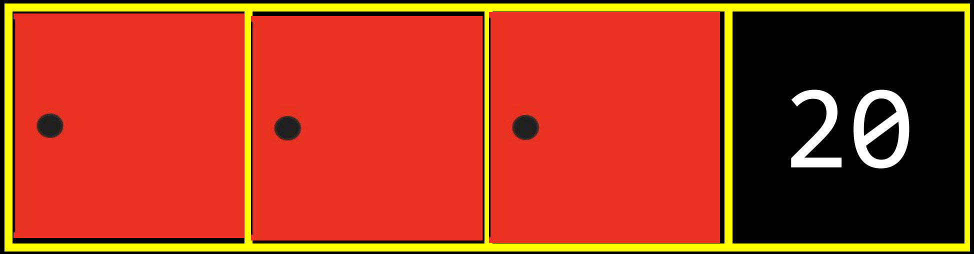

1
5
10
20
50
100

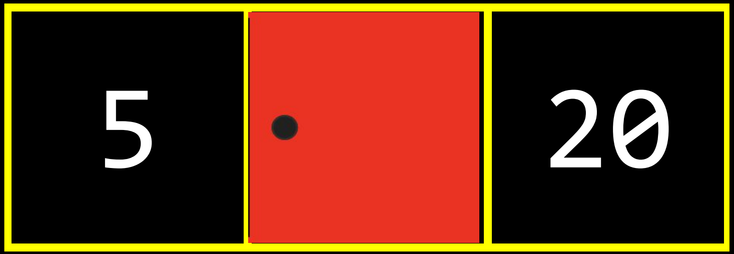
1	5	10	20	50	100	500

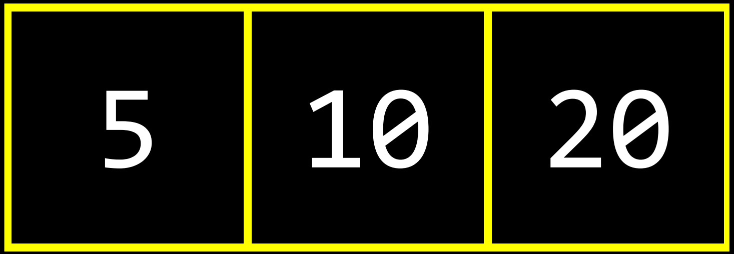


from Harvard CS50

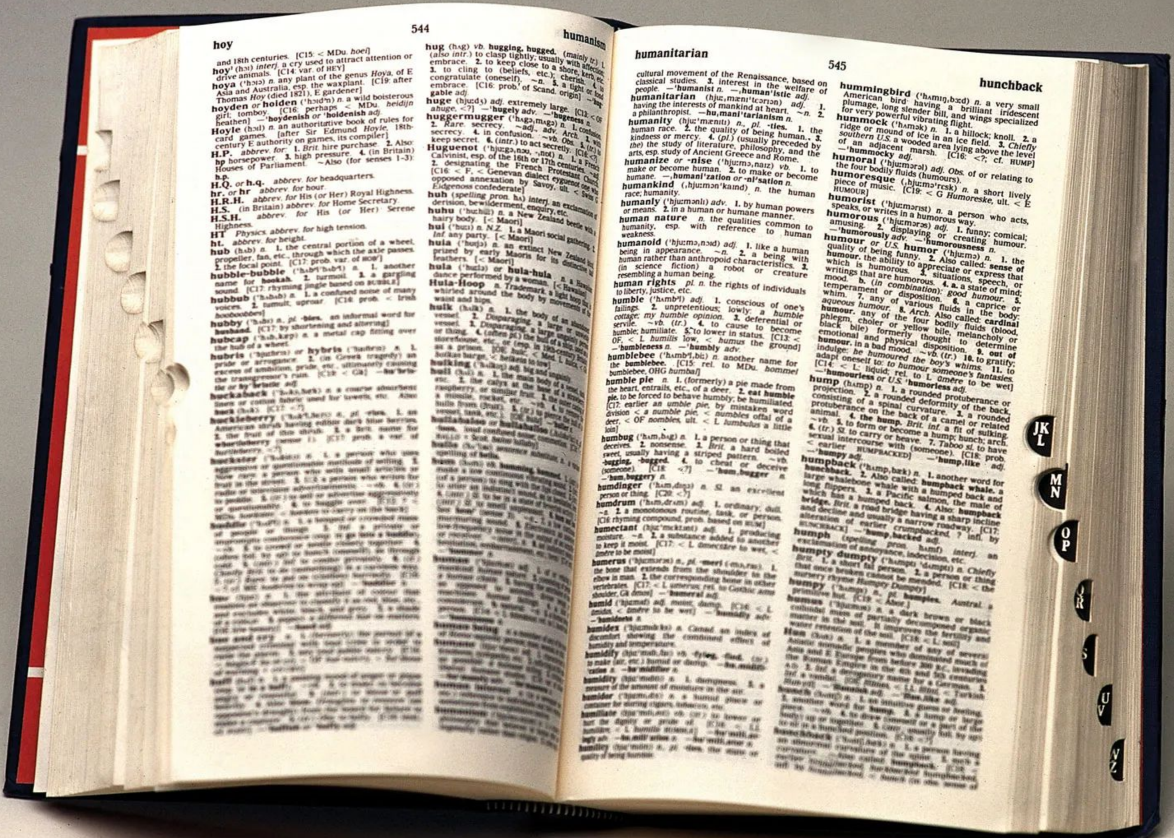

20


20


5
20


5
10
20

1	5	10	20	50	100	500


hoy
544
humanism
humanitarian
545
hunchback
JK L
MN
OP

🔎 BINARY SEARCH:

- sorted list

- check middle of list - if there, **return!**
- if search term > middle, **binary search** to the right
- if search term < middle, **binary search** to the left
- are there any doors left to check? if not, **return!**

LINEAR SEARCH: “brute force”

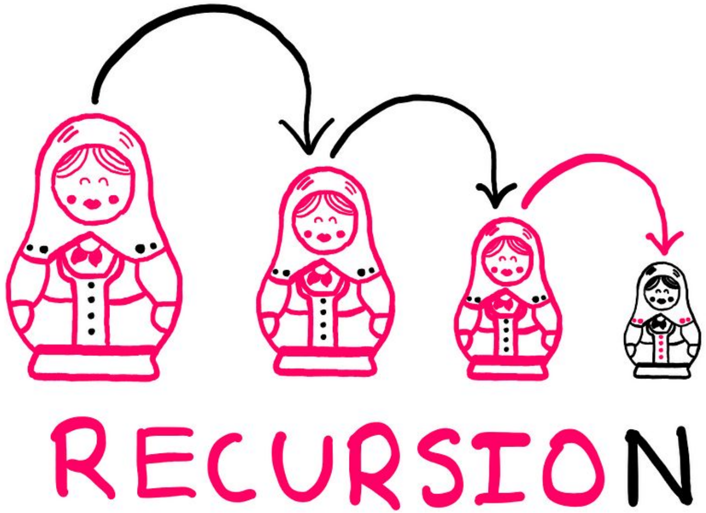
- check every element
- move onto next element

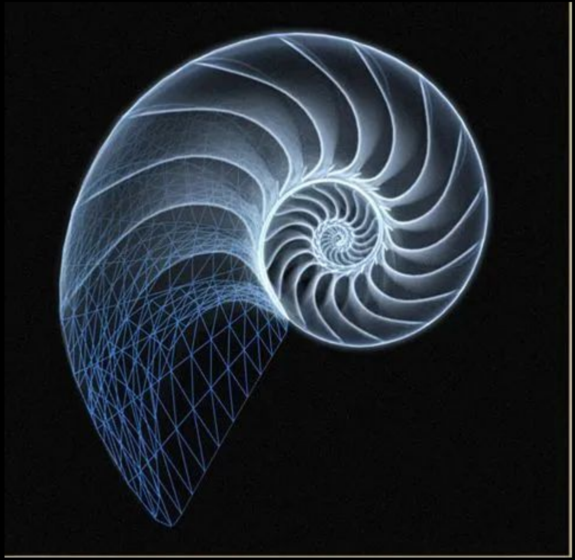
BINARY SEARCH: “divide and conquer”

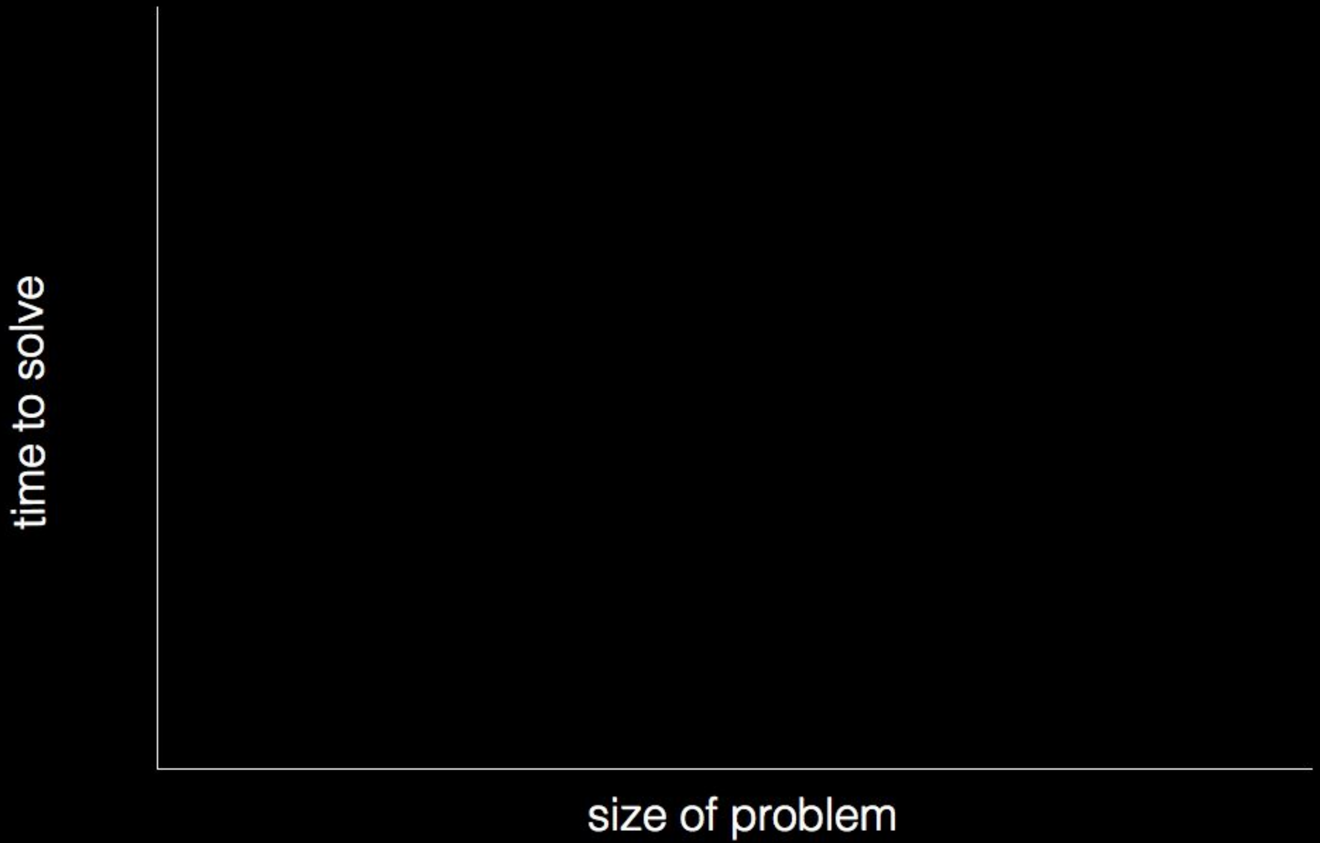
- sorted list
- check middle of list
- if search term > middle, **binary search** to the right
- if search term < middle, **binary search** to the left

recursion: when a process invokes itself

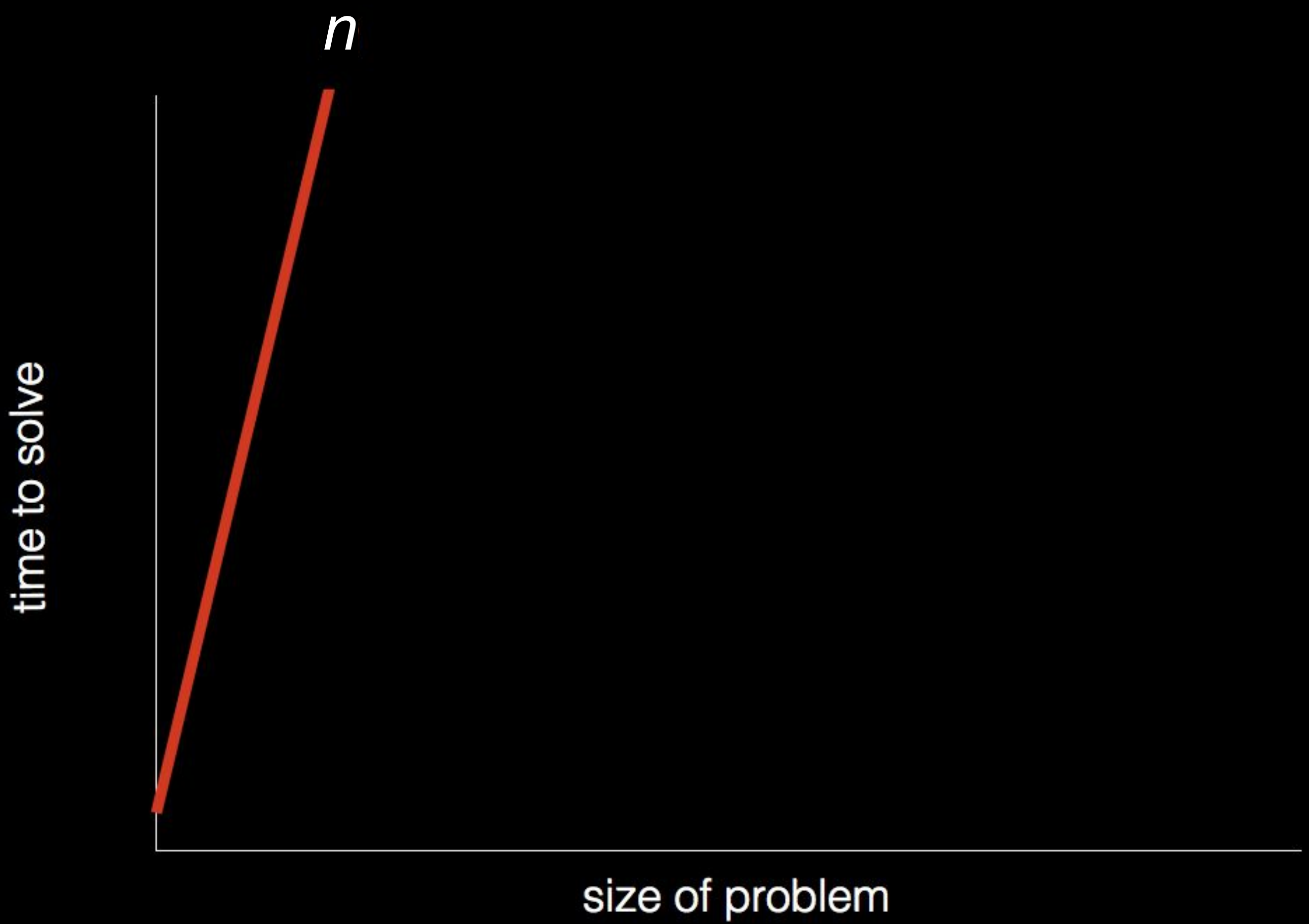
- **TO UNDERSTAND RECURSION, WE MUST FIRST UNDERSTAND RECURSION.**


RECURSION

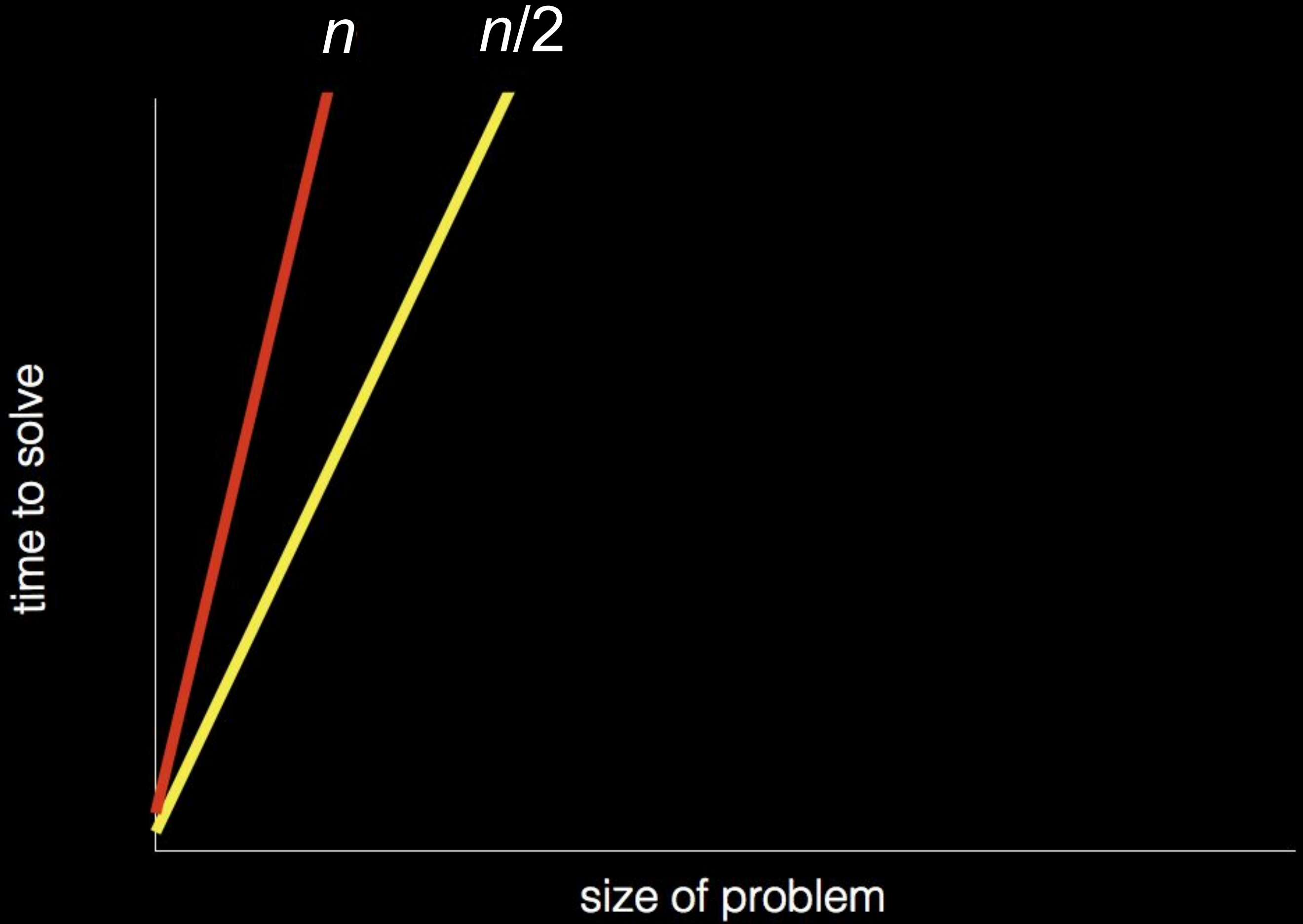




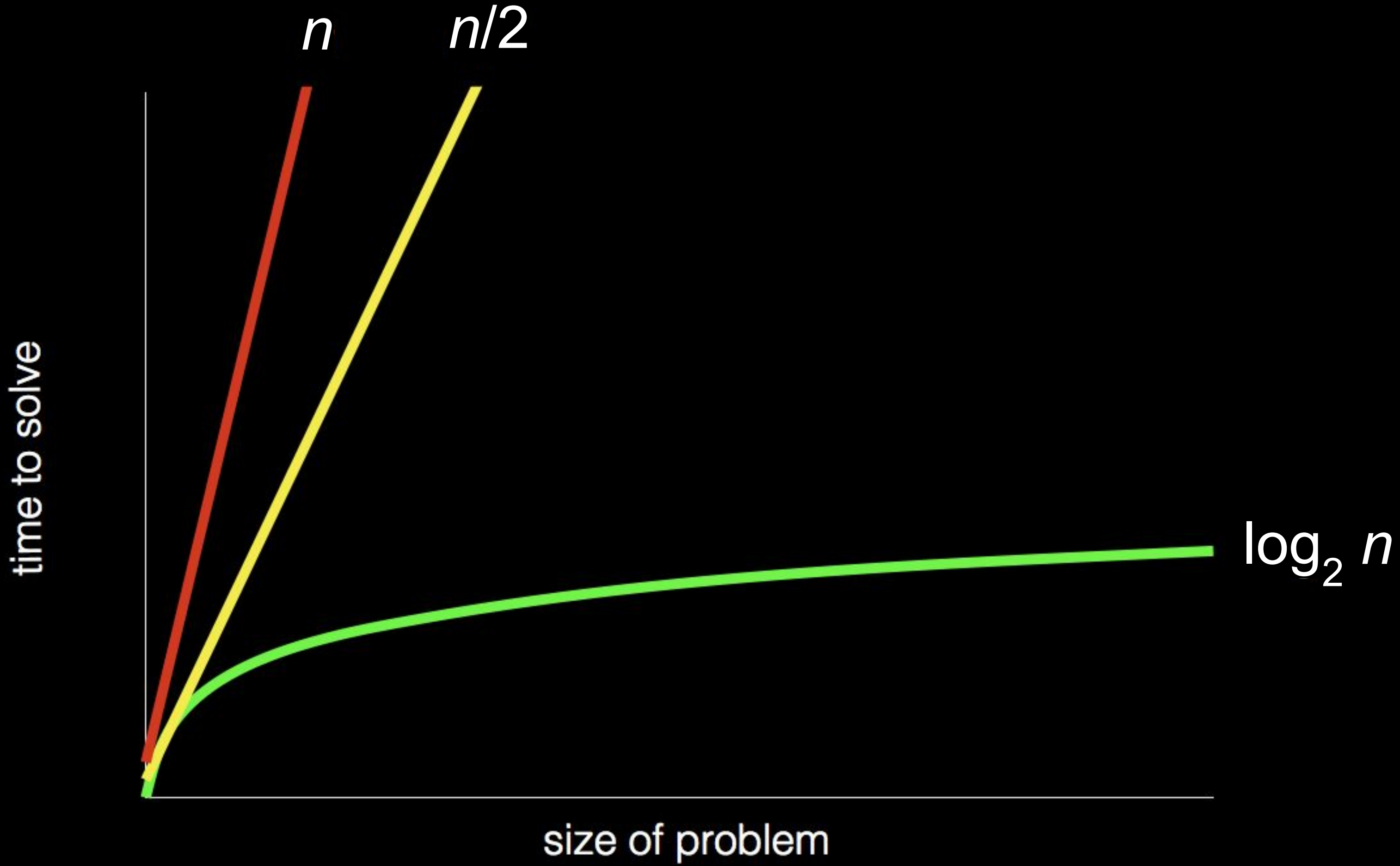
from Harvard CS50



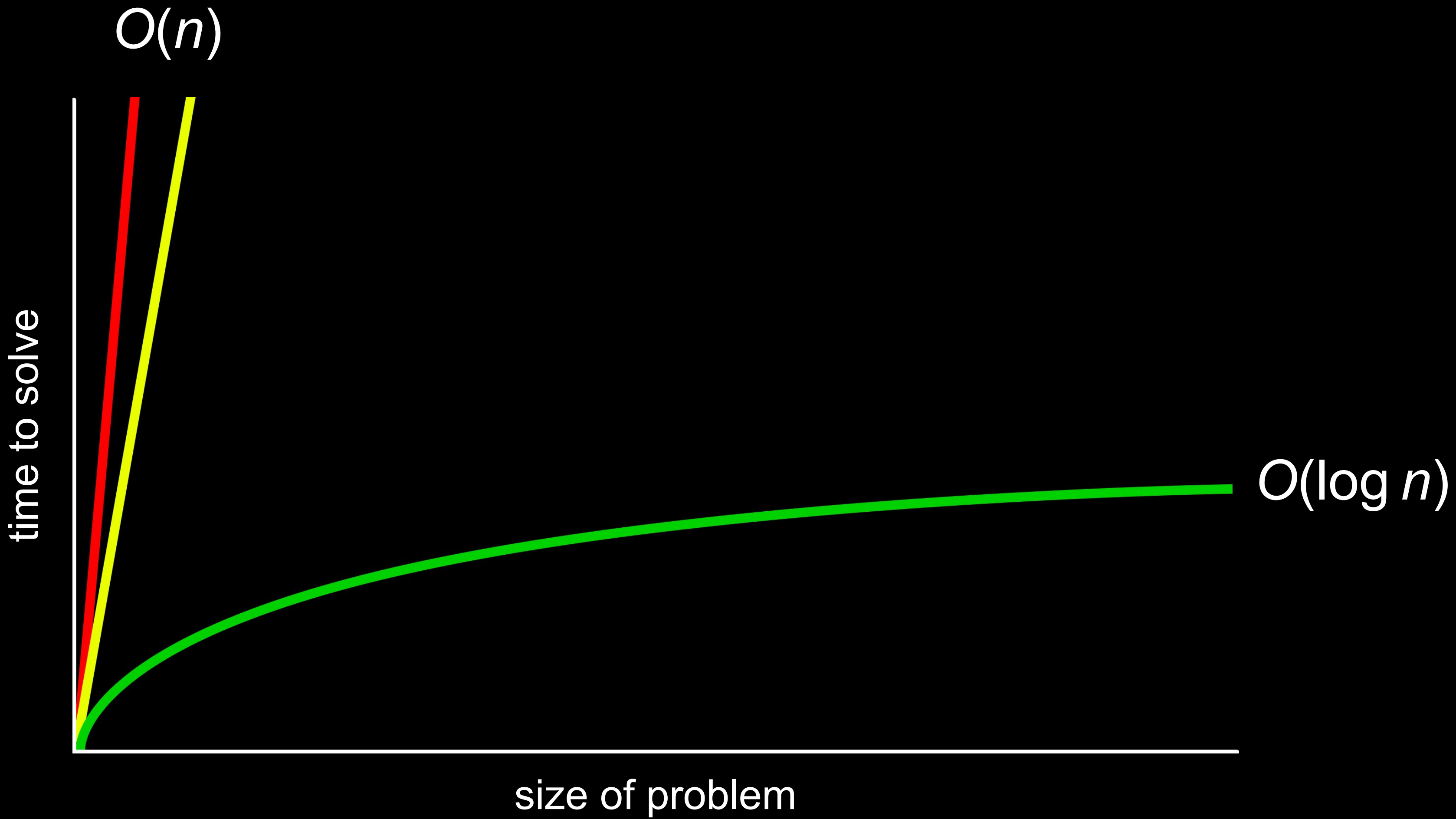
from Harvard CS50



from Harvard CS50



from Harvard CS50



from Harvard CS50

0

from Harvard CS50

$O(n^2)$

$O(n \log n)$

$O(n)$

$O(\log n)$

$O(1)$

$O(n^2)$	
$O(n \log n)$	
$O(n)$	linear search
$O(\log n)$	
$O(1)$	

$O(n^2)$	
$O(n \log n)$	
$O(n)$	linear search
$O(\log n)$	binary search
$O(1)$	

from Harvard CS50

Classes of Running Times

Constant - $O(1)$

Logarithmic - $O(\log N)$

Linear - $O(N)$

Linearithmic - $O(N \log N)$

Polynomial - $O(N^2), O(N^3), O(N^4)$, etc.

Exponential - $O(2^N), O(3^N), O(4^N)$, etc.

$$\Omega$$

from Harvard CS50

$\Omega(n^2)$

$\Omega(n \log n)$

$\Omega(n)$

$\Omega(\log n)$

$\Omega(1)$

$\Omega(n^2)$

$\Omega(n \log n)$

$\Omega(n)$

$\Omega(\log n)$

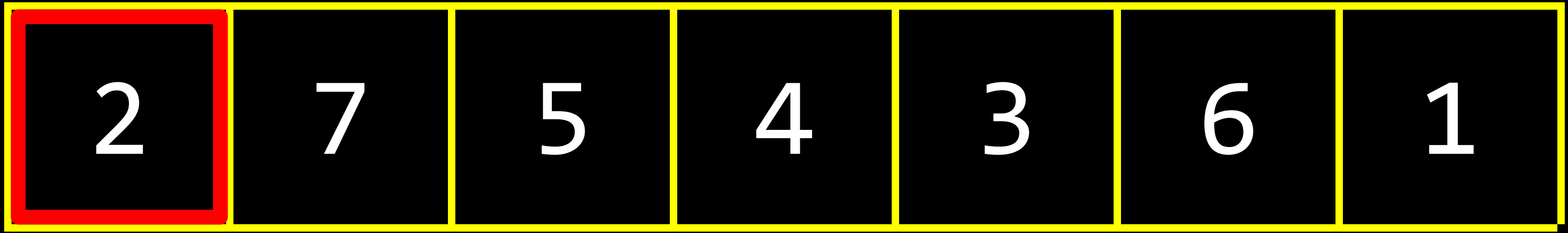
$\Omega(1)$ linear search

$\Omega(n^2)$	
$\Omega(n \log n)$	
$\Omega(n)$	
$\Omega(\log n)$	
$\Omega(1)$	linear search, binary search

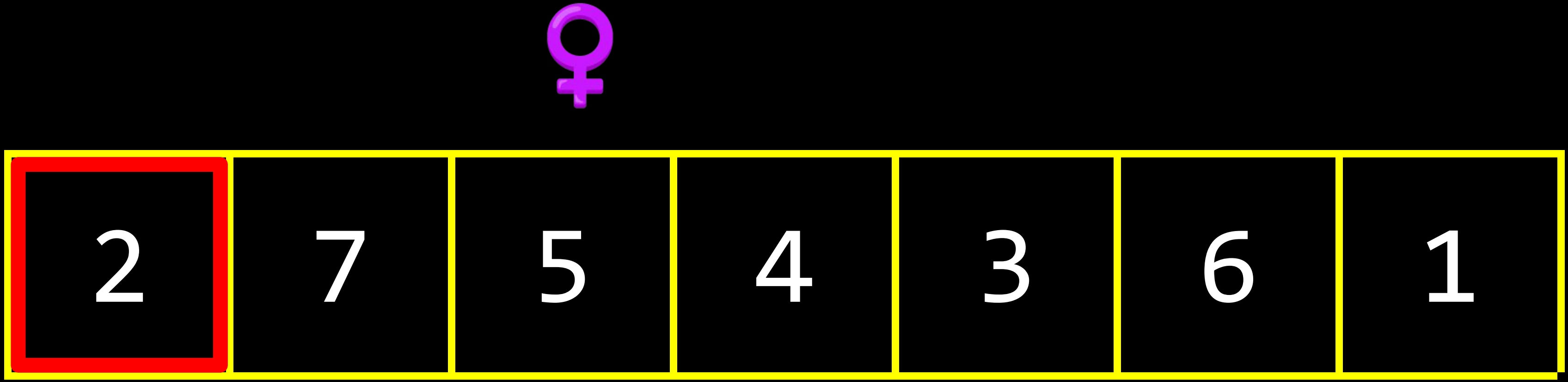
SELECTION SORT

2	7	5	4	3	6	1

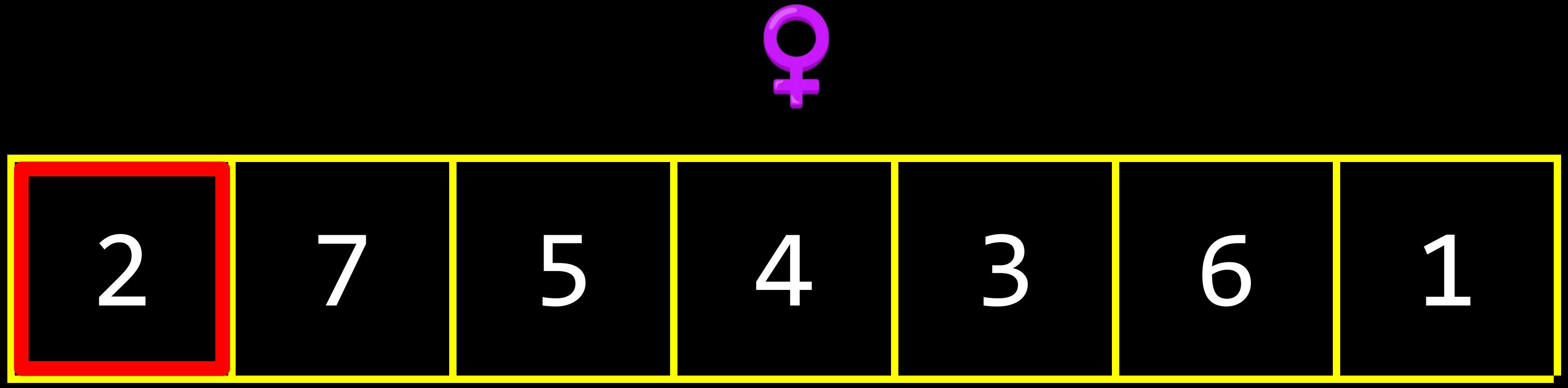
SELECTION SORT

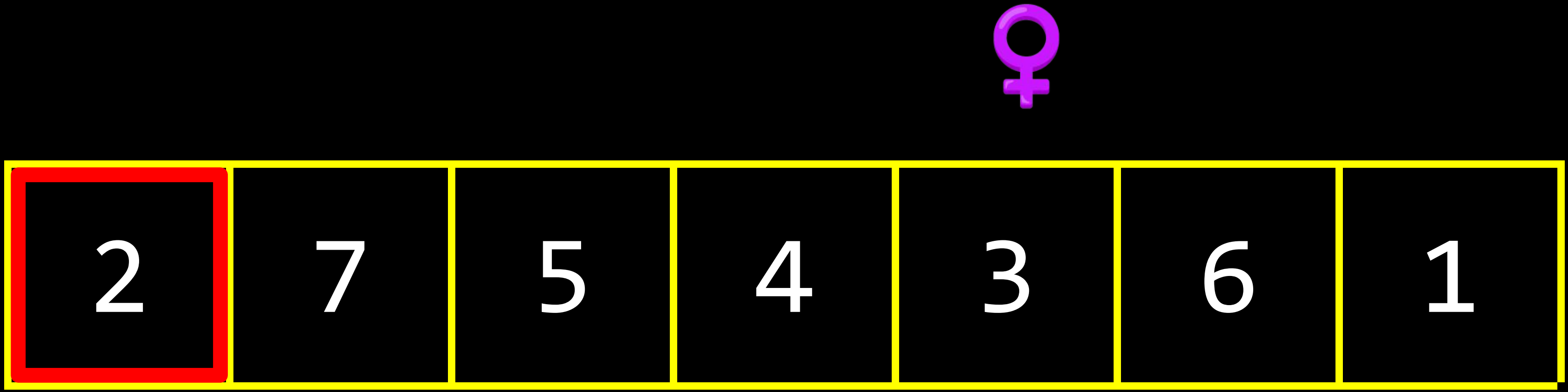


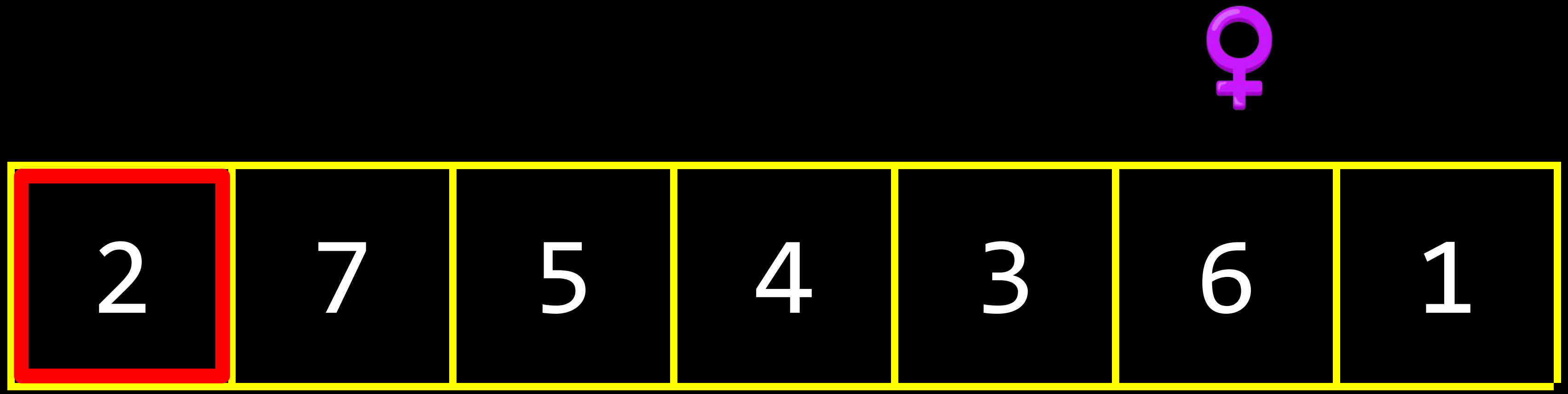
SELECTION SORT

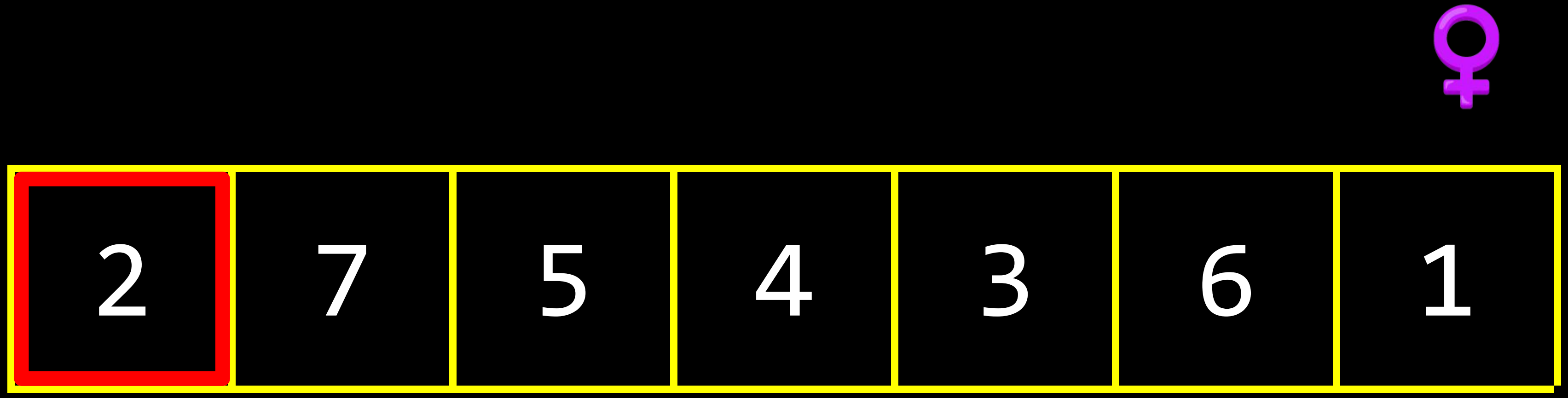


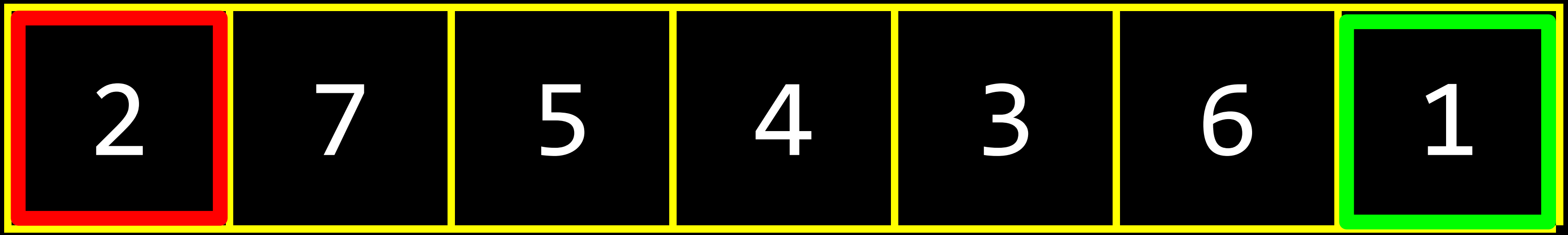
Check: smallest number in rest of the array …

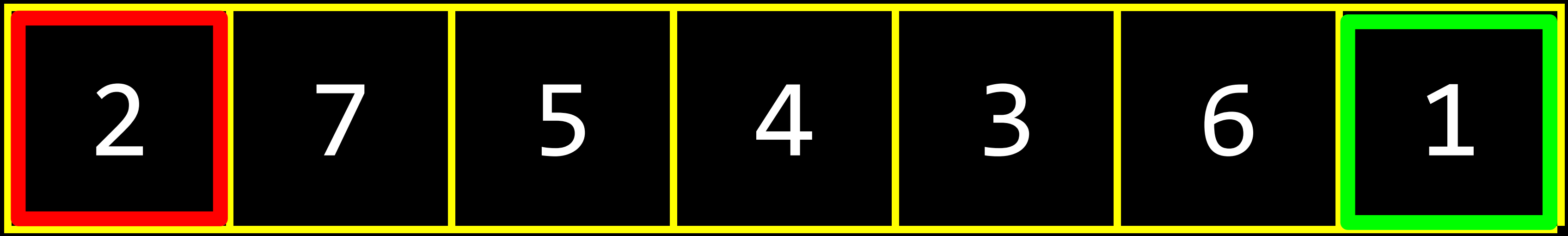

♀
2
7
5
4
3
6
1


♀
2
7
5
4
3
6
1

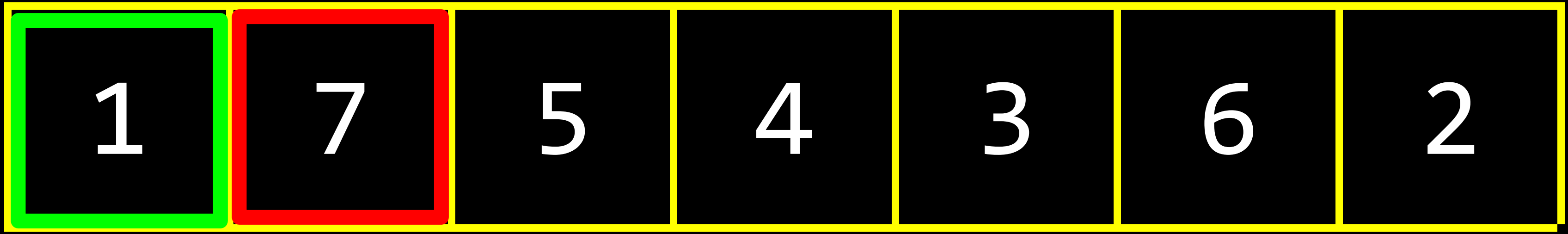

2
7
5
4
3
6
1


2
7
5
4
3
6
1

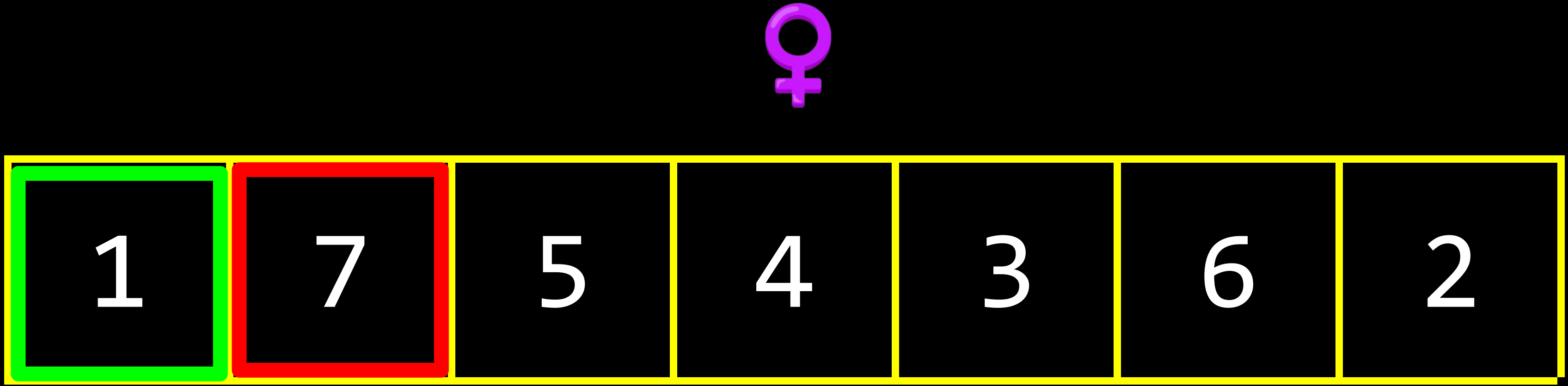

2
7
5
4
3
6
1



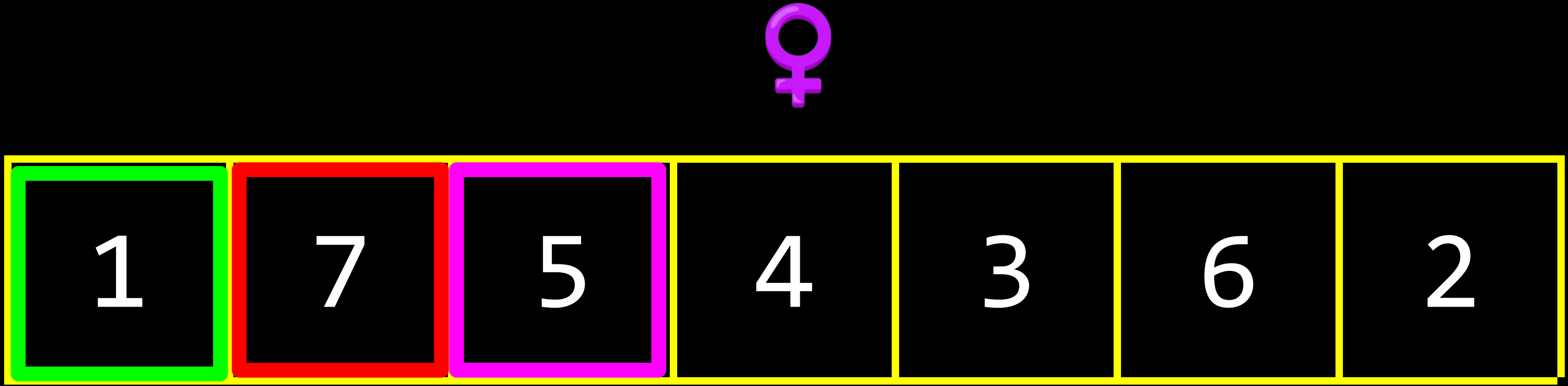
SWAP!



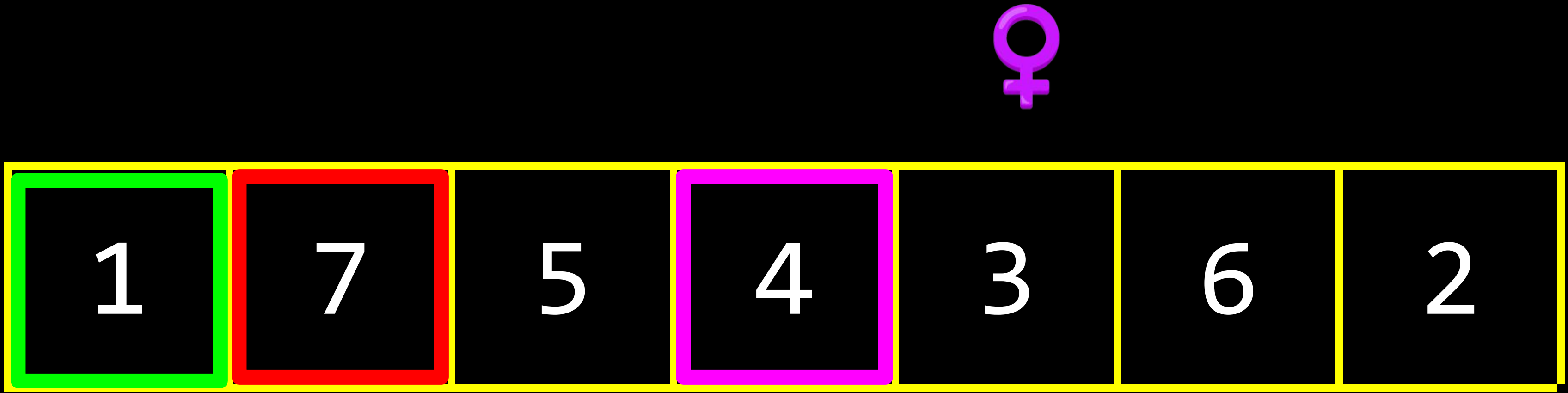
Start again…



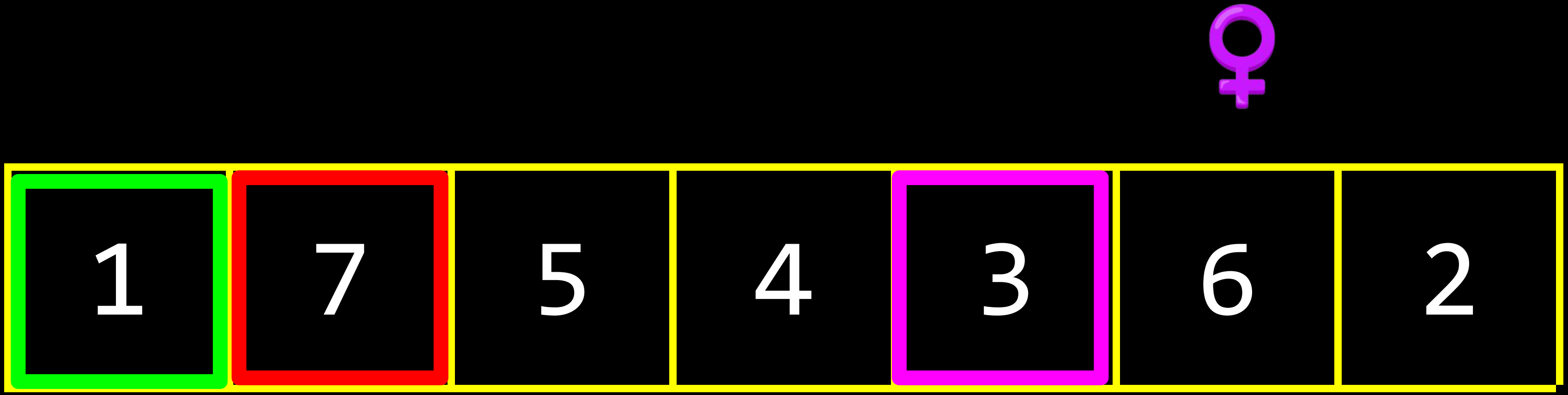
Check: smallest number in rest of the array …



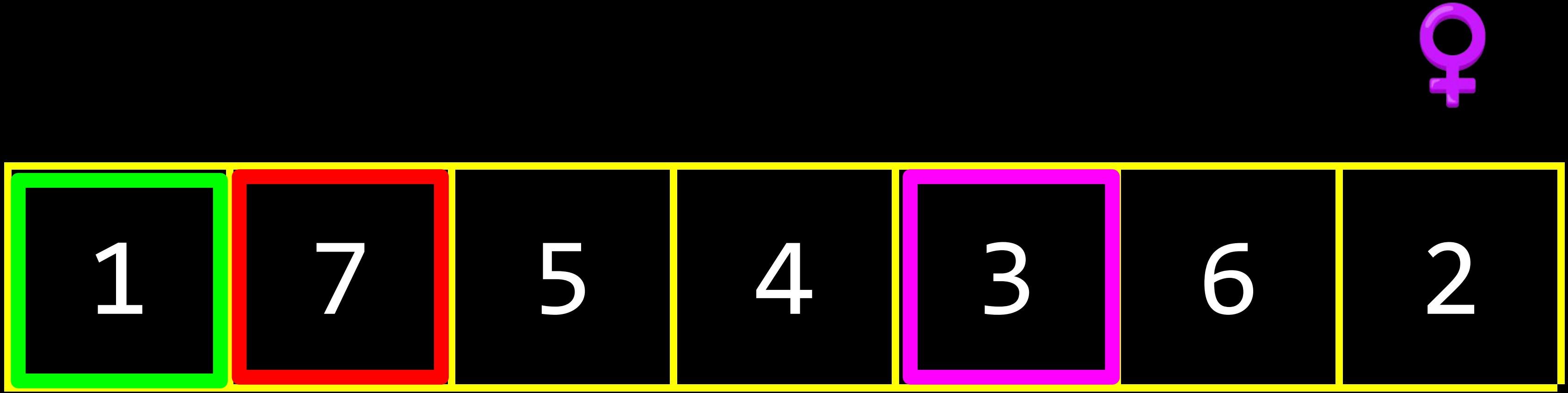
SO FAR: 5



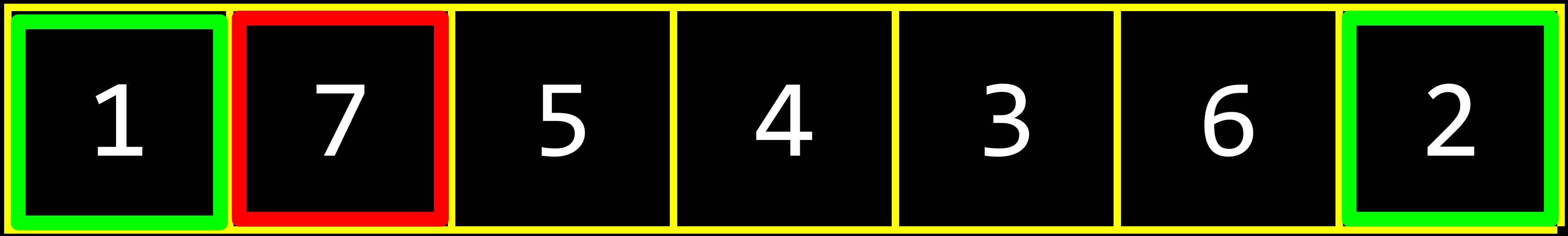
SO FAR: 4



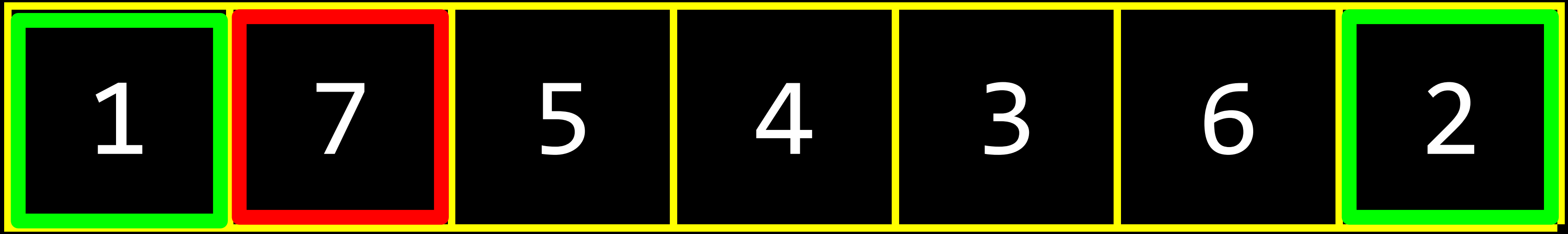
SO FAR: 3



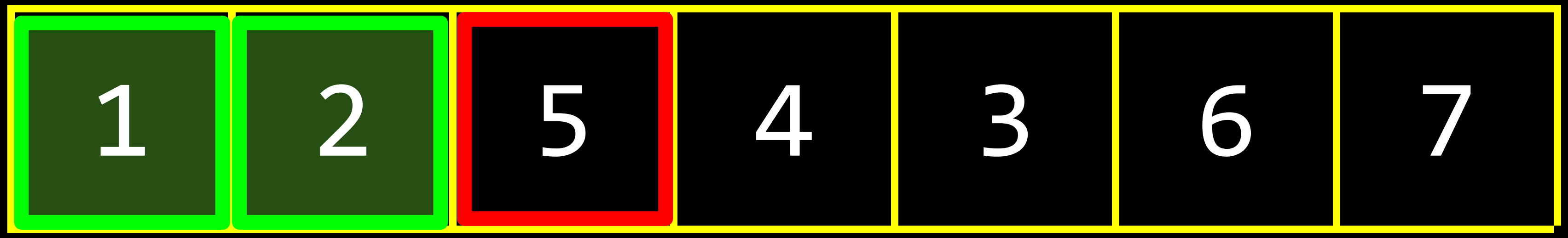
SO FAR: 3

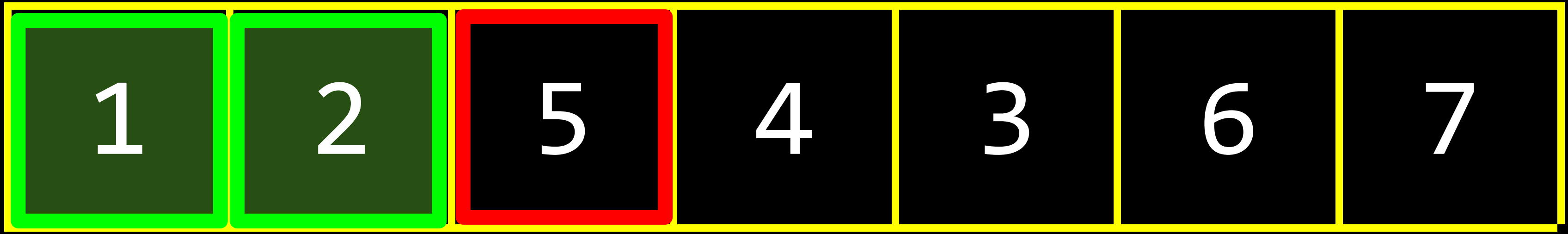


SMALLEST: 2

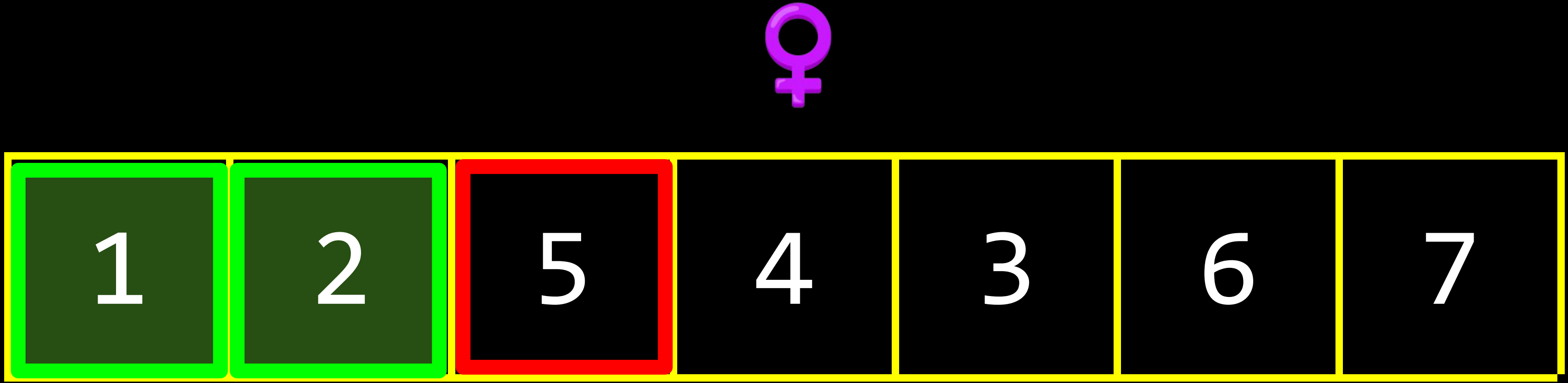


SWAP!

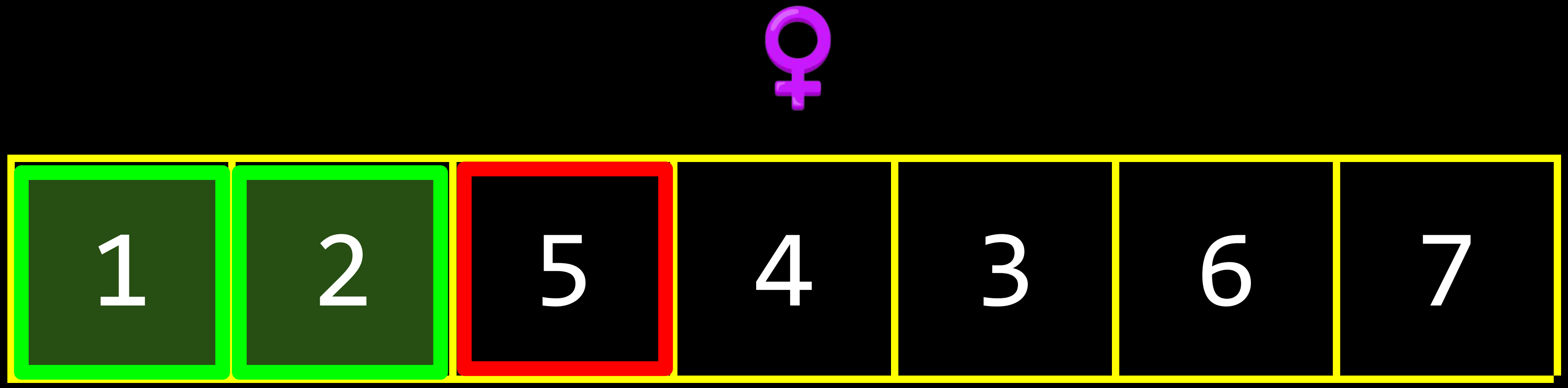

1
2
5
4
3
6
7



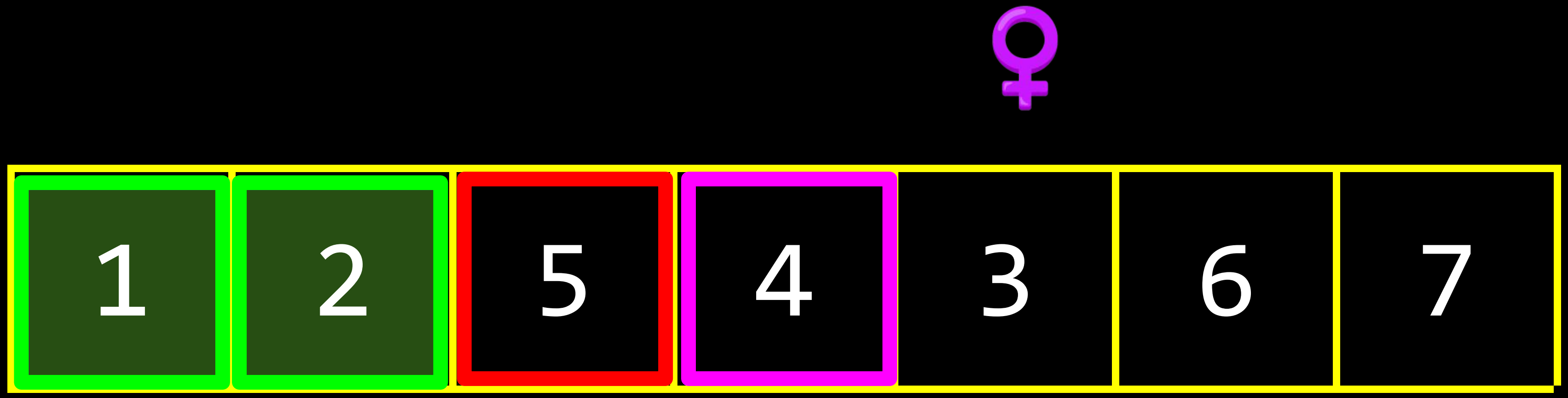
Start again…



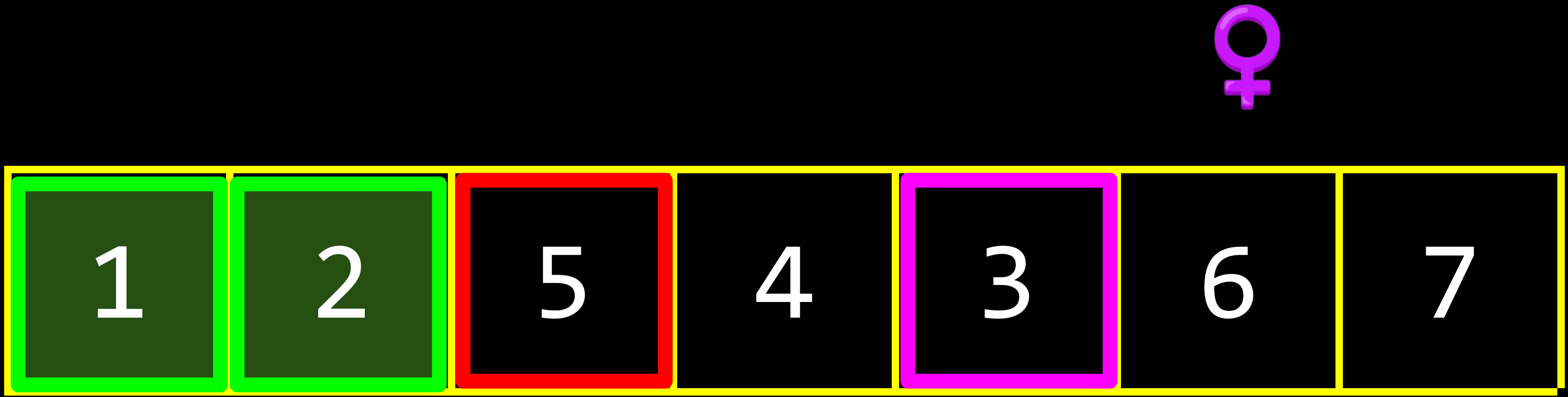
Check: smallest number in
rest of the array …



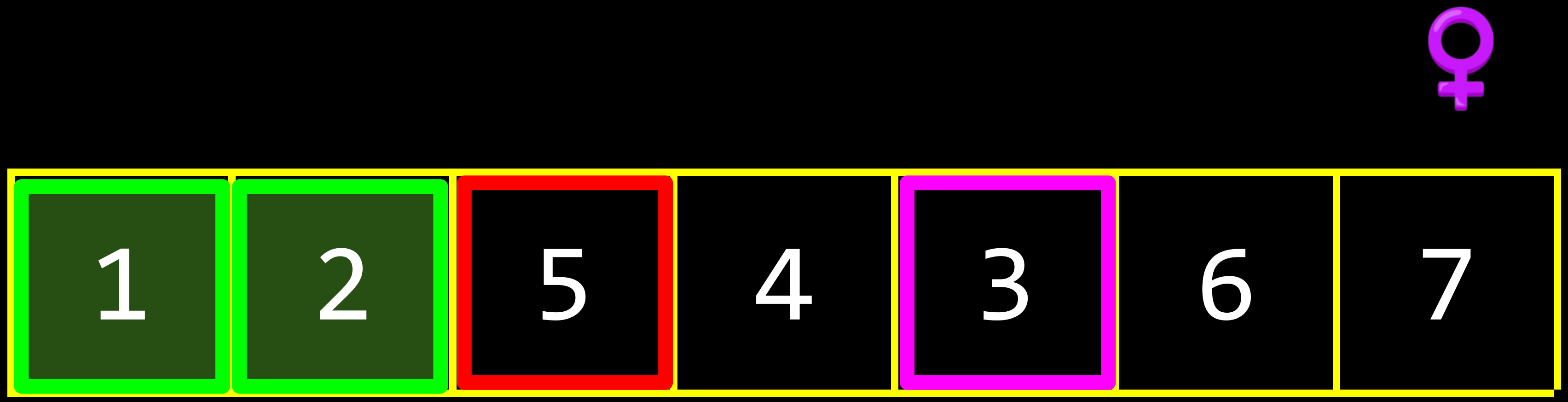
SO FAR: 5



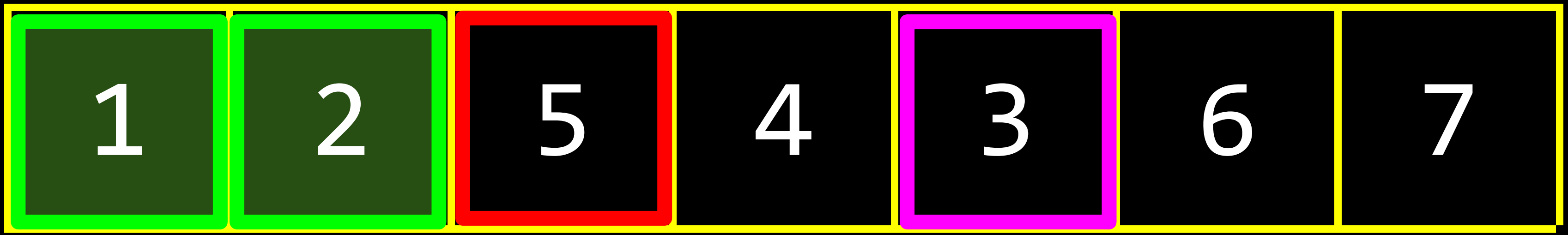
SO FAR: 4



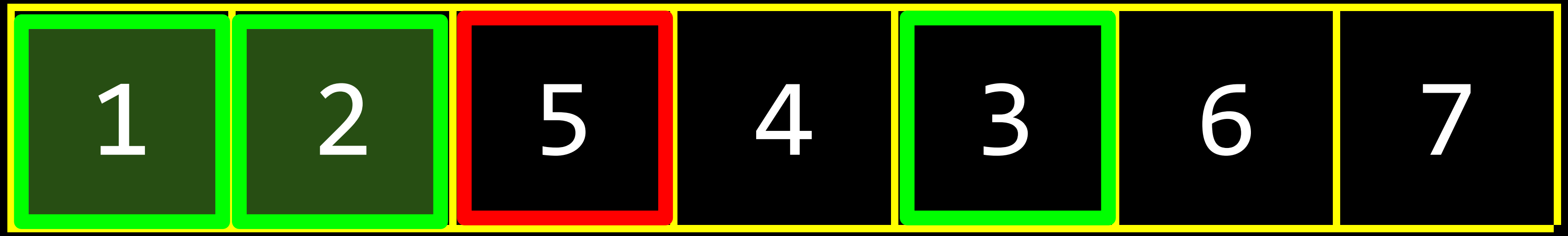
SO FAR: 3



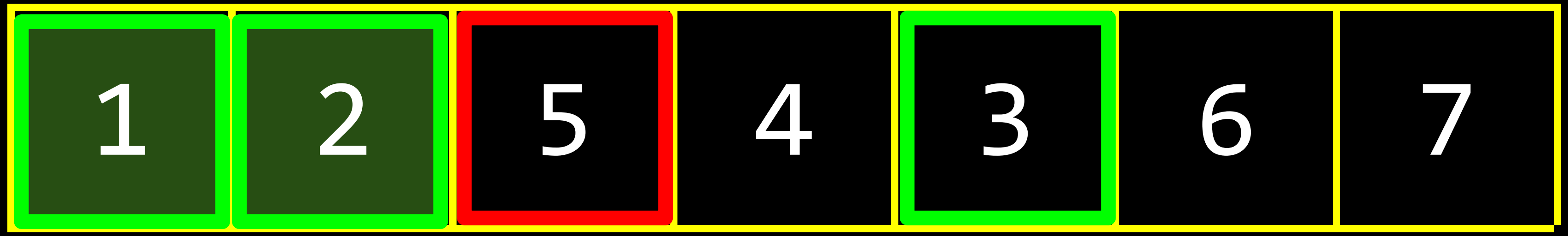
SO FAR: 3



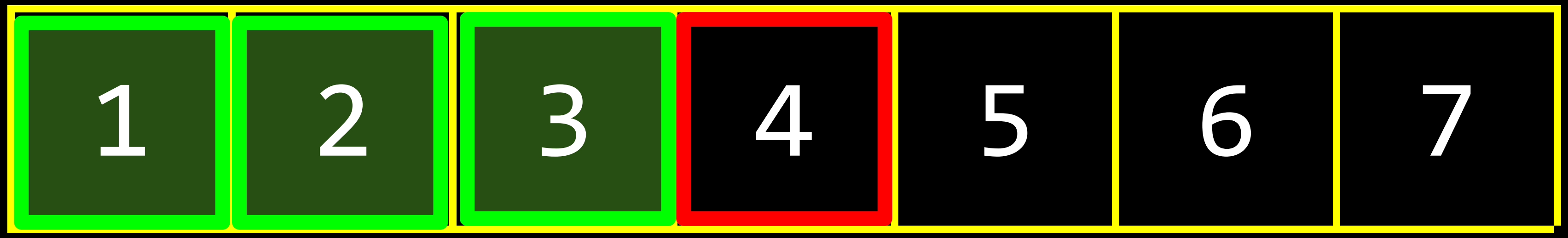
SO FAR: 3

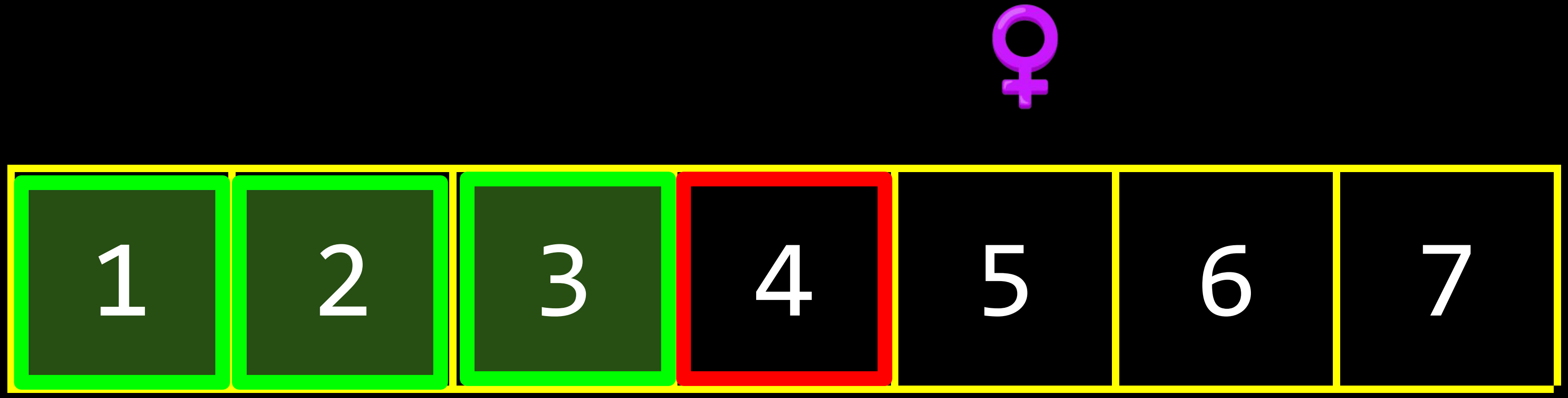


SMALLEST: 3

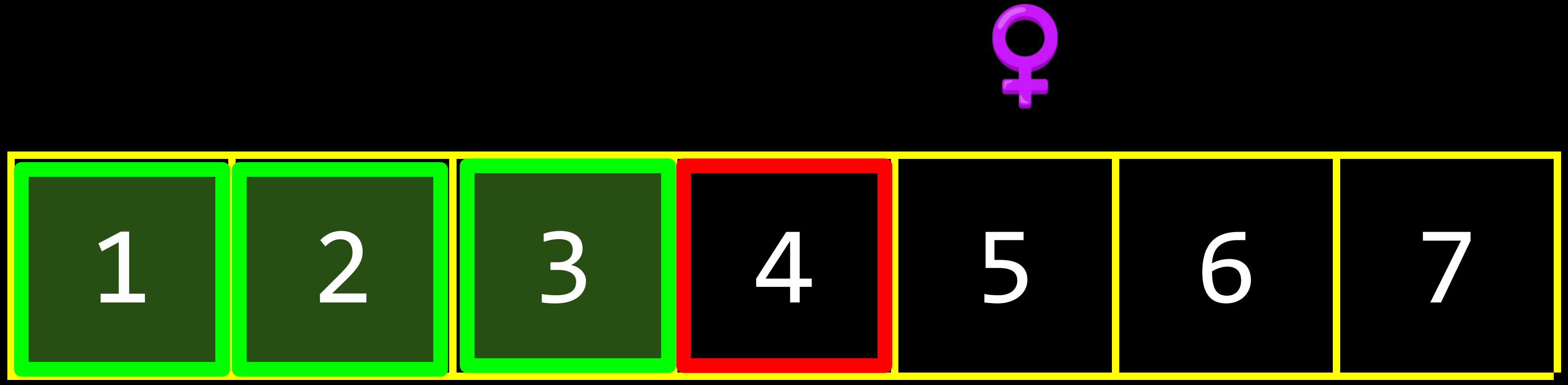


SWAP!

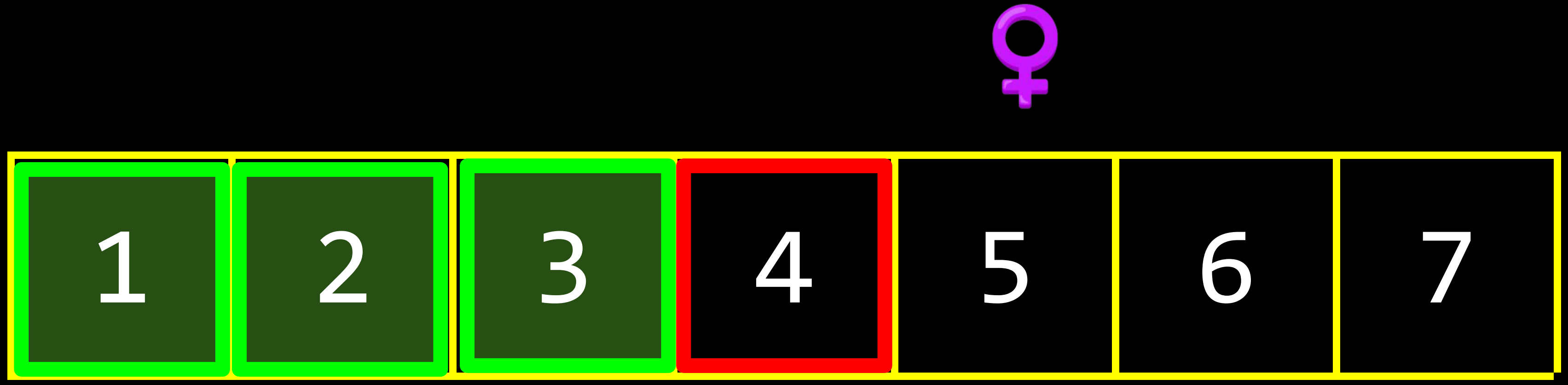

1
2
3
4
5
6
7



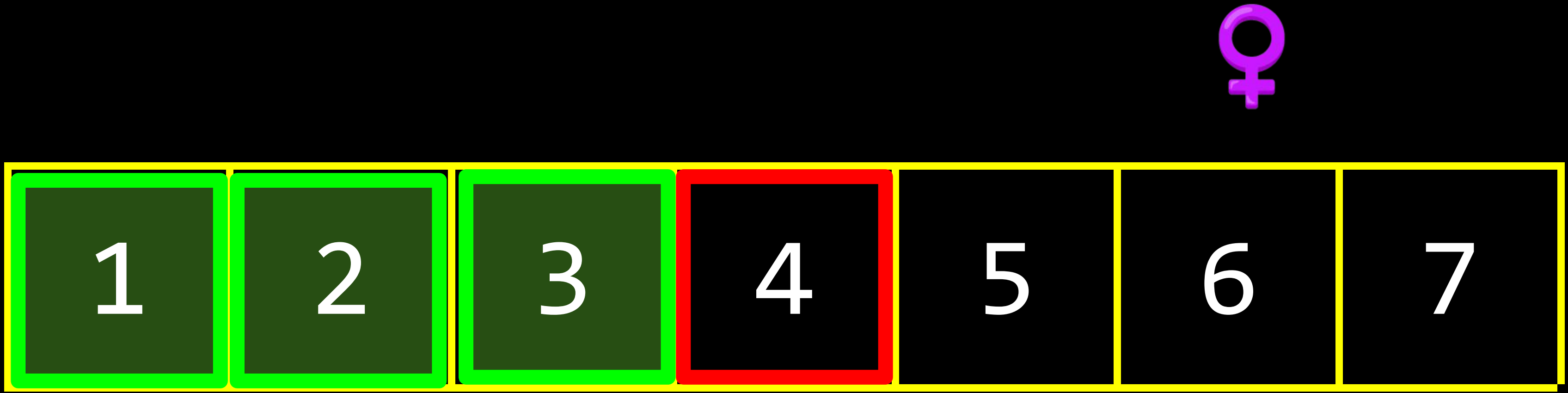
Start again…



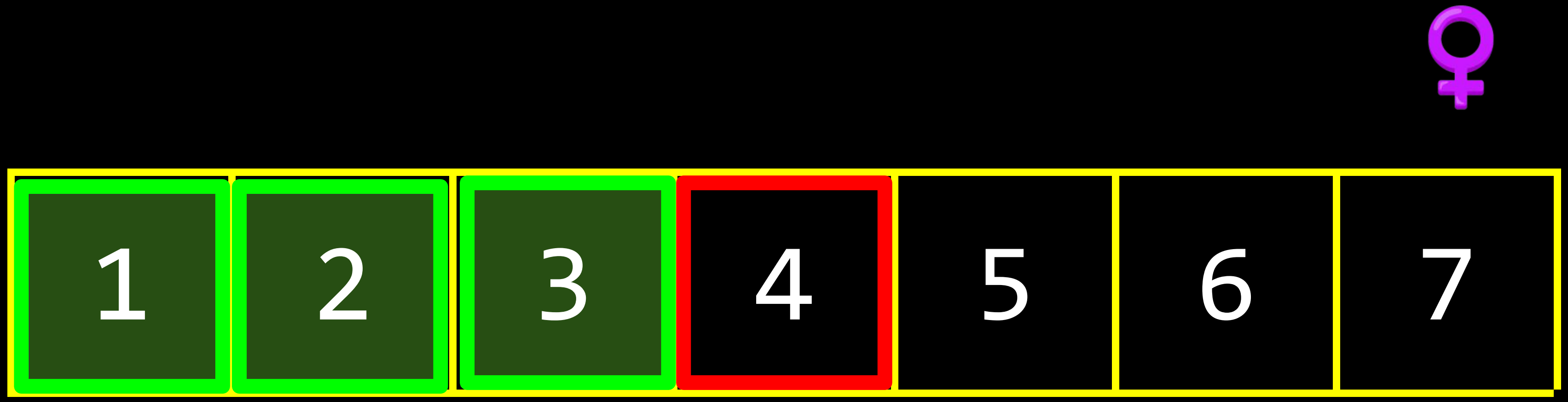
Check: smallest number in rest of the array …



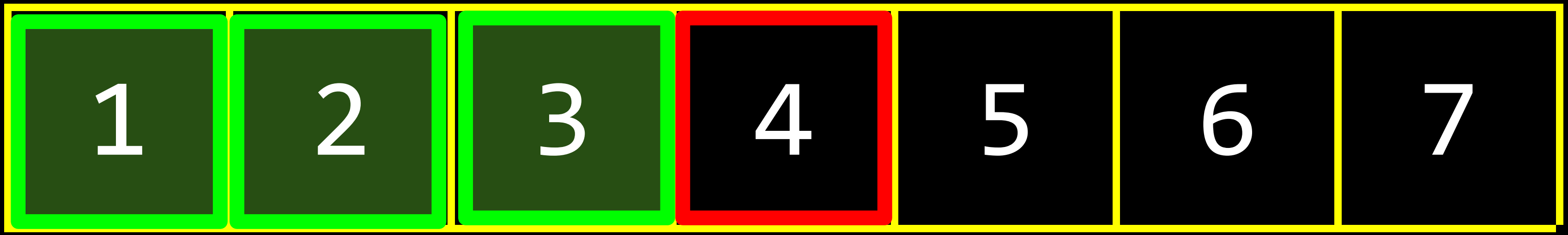
SO FAR: 4



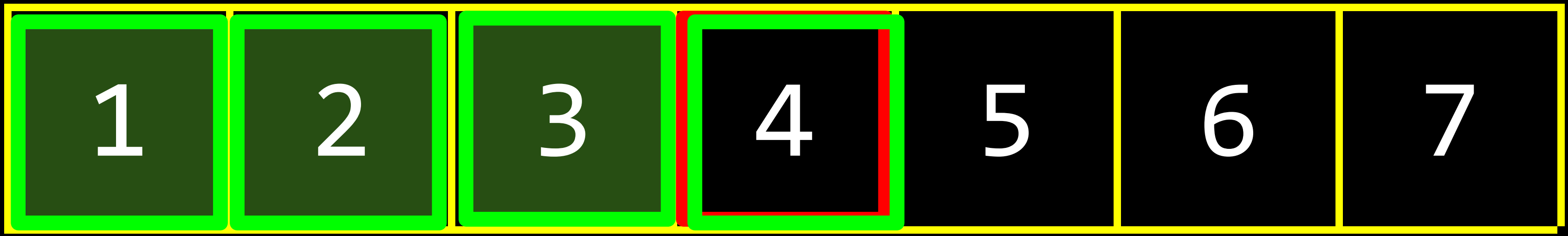
SO FAR: 4



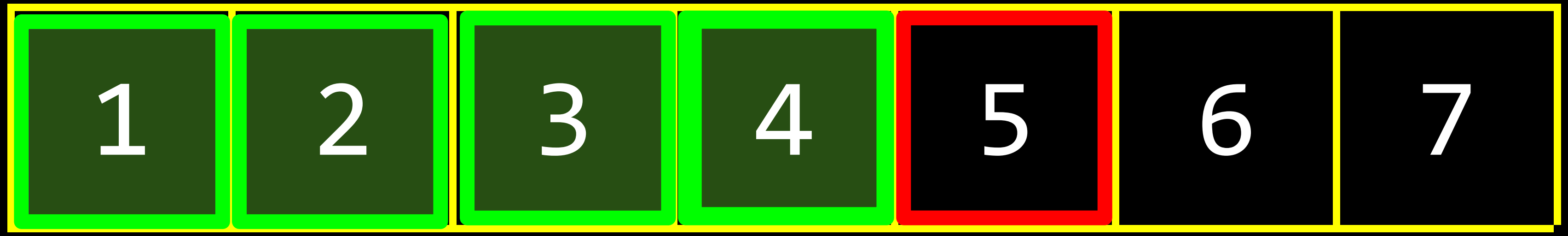
SO FAR: 4



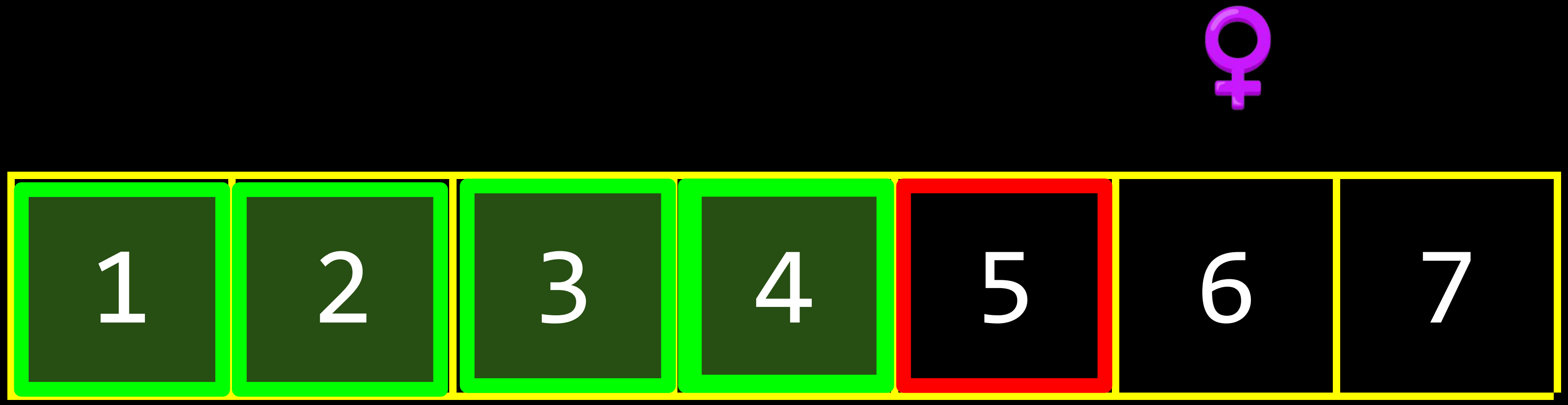
SO FAR: 4



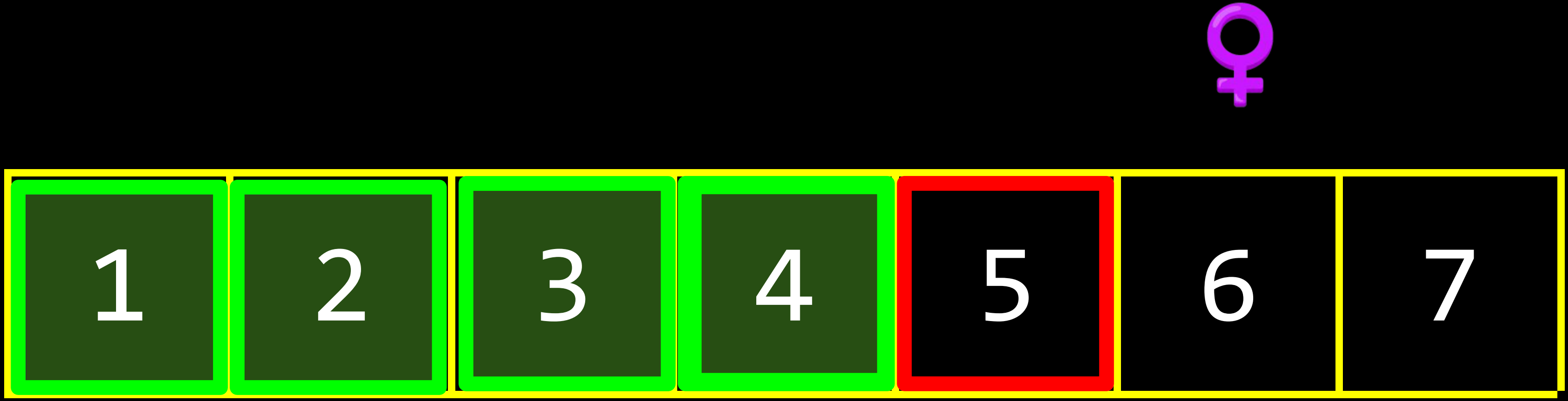
SMALLEST: 4



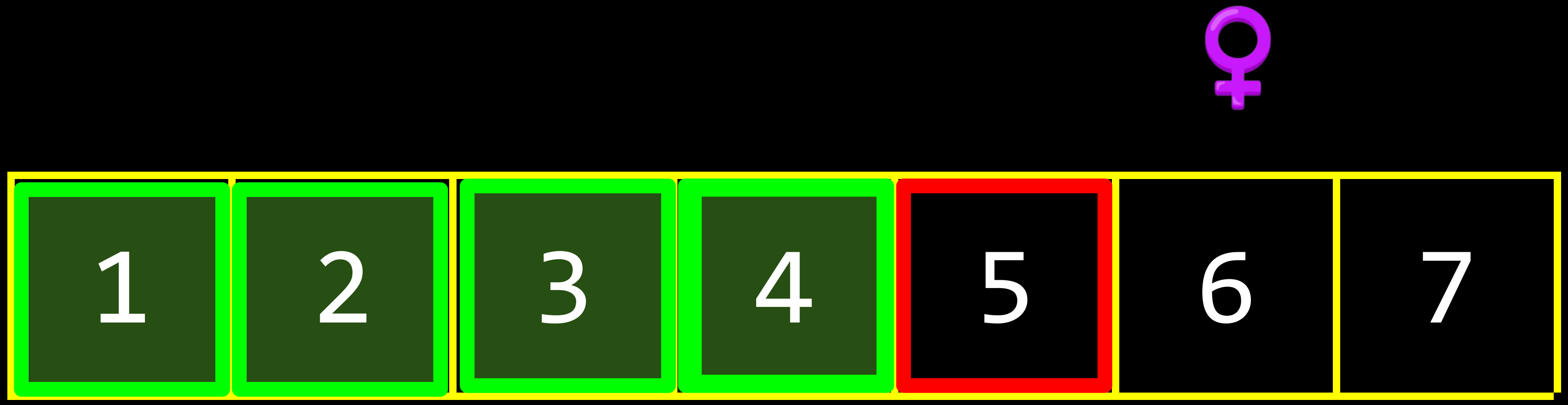
KEEP! (SWAP ITSELF)



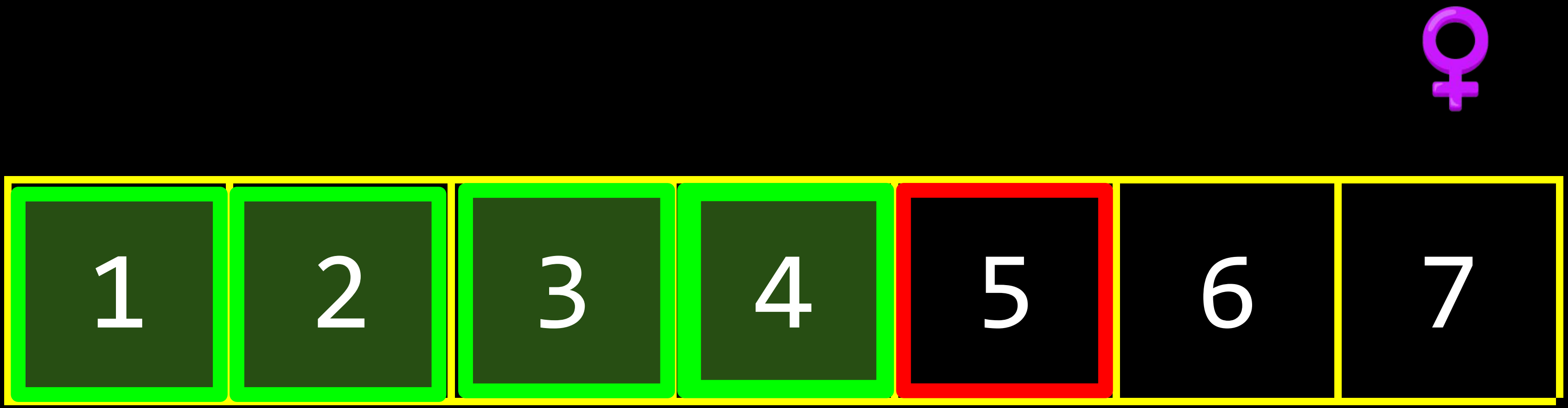
Start again…



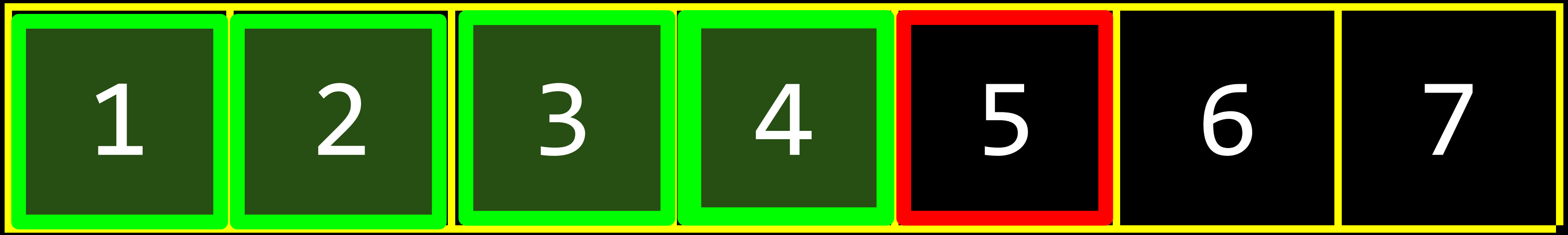
Check: smallest number in rest of the array …



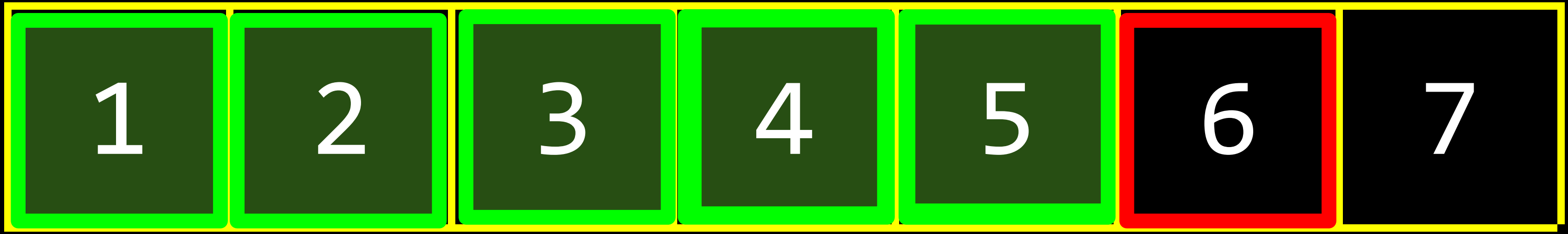
SO FAR: 5



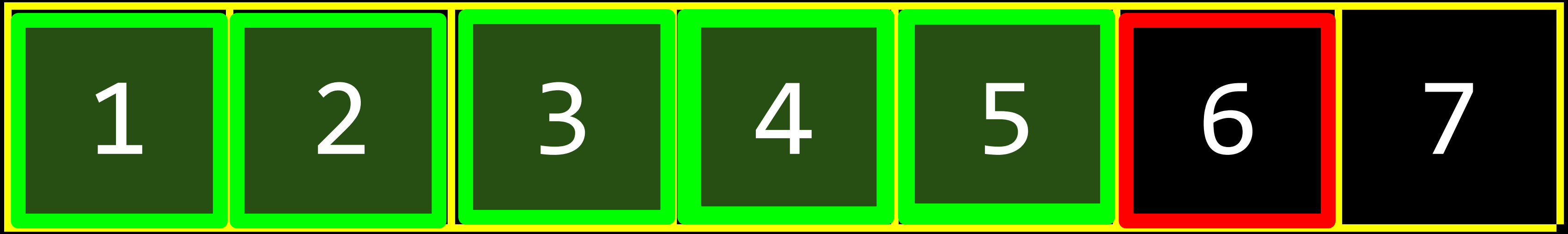
SO FAR: 5



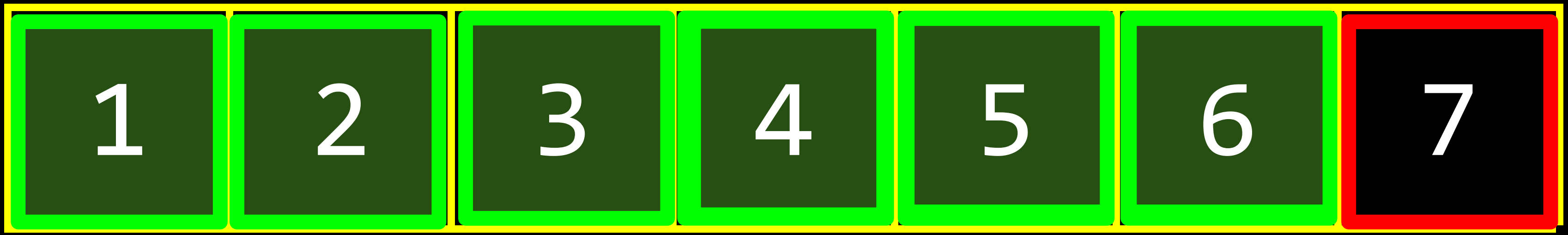
SMALLEST: 5



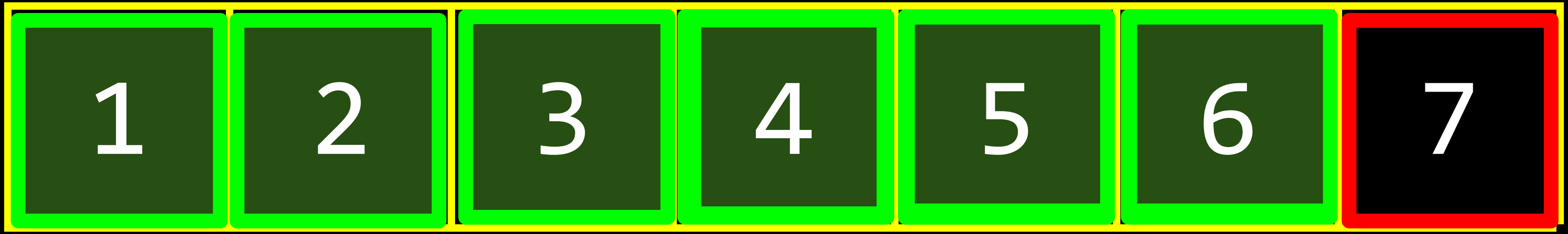
SMALLEST: 5



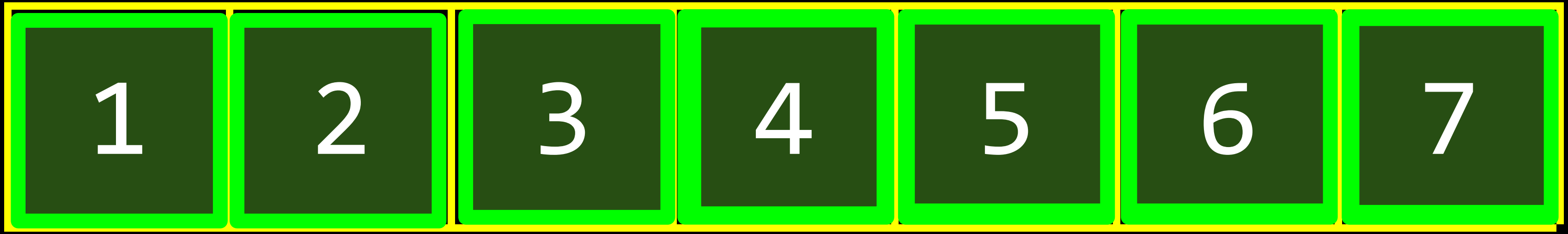
SMALLEST: 6



SMALLEST: 6



SMALLEST: 7



SORTED. ✅

SELECTION SORT

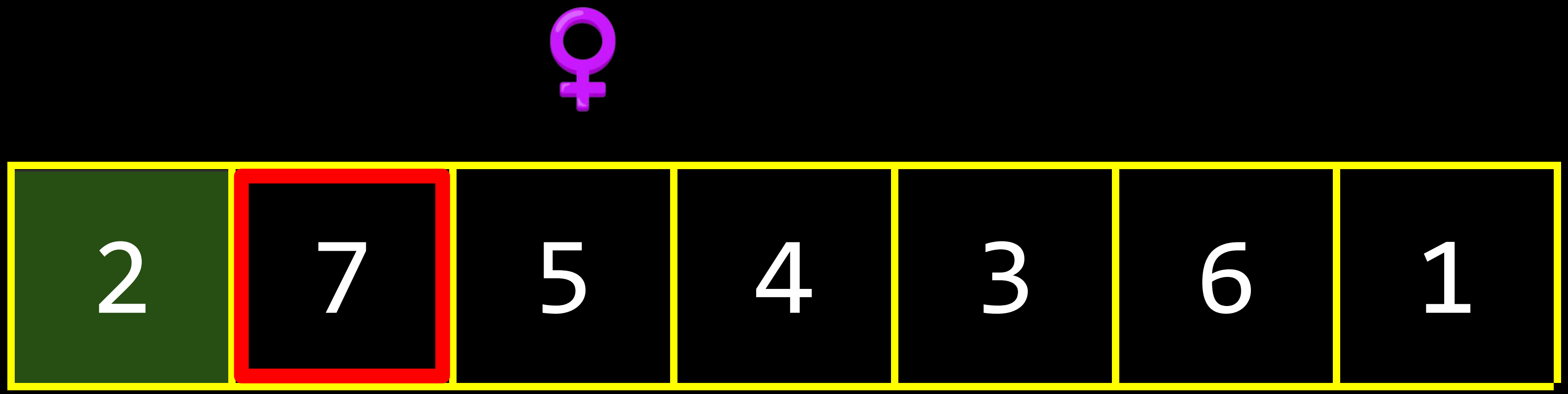
- Create a “**sorted**” and “**unsorted**” part of array
- Search and find smallest item of “**unsorted**” array
- Add this smallest item to “**sorted**” part by swapping with the beginning of “**unsorted**” part
- Continue throughout entire array

$O(n^2)$	selection sort
$O(n \log n)$	
$O(n)$	
$O(\log n)$	
$O(1)$	

INSERTION SORT

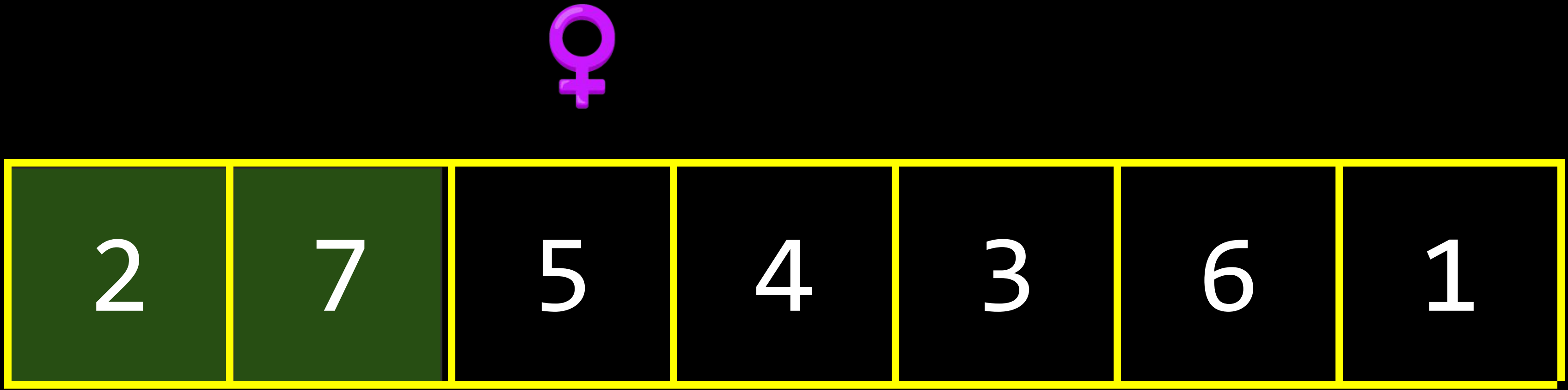
2	7	5	4	3	6	1

INSERTION SORT

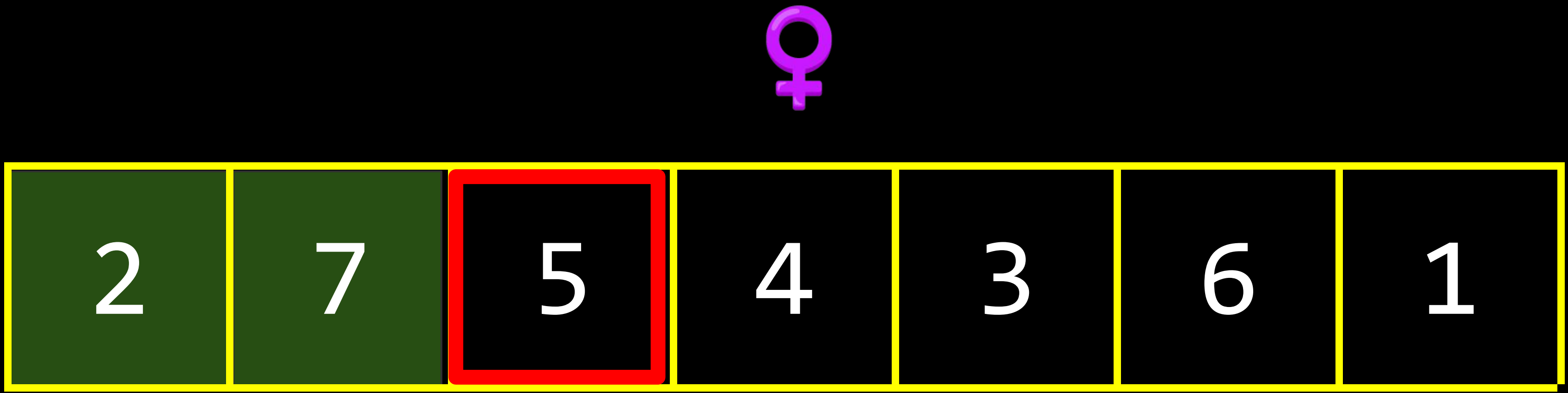


Check to the LEFT: is the one on the left smaller?

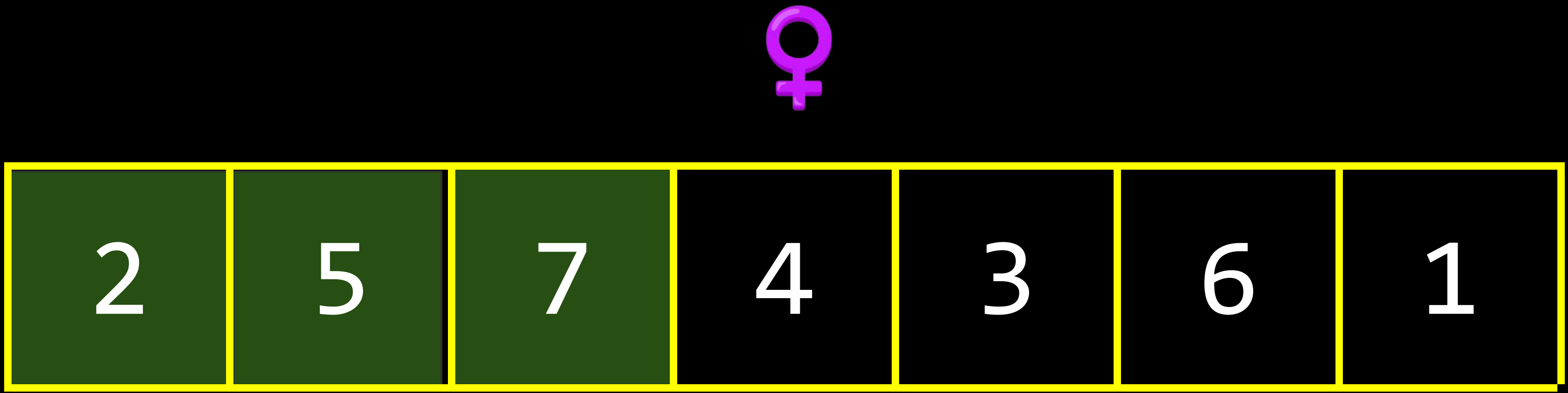
INSERTION SORT



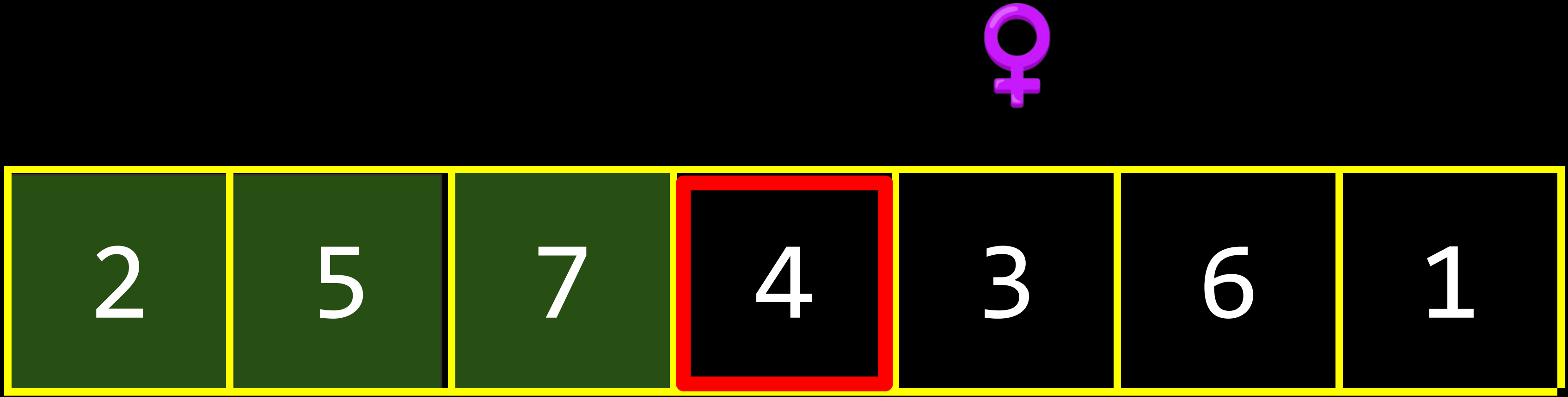
IF YES, move on …



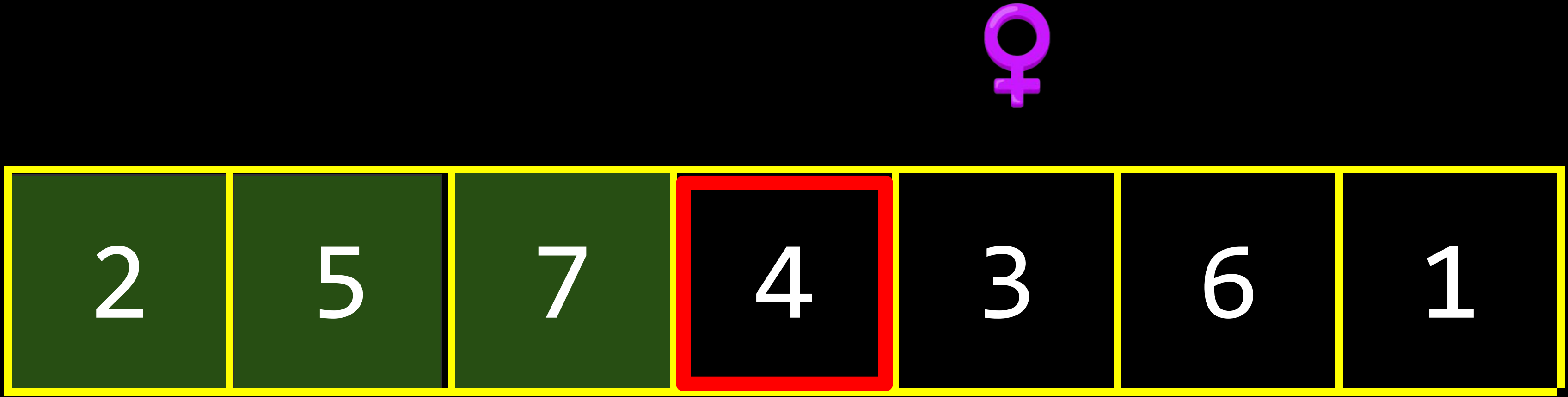
Check to the LEFT: is the one on the left smaller?



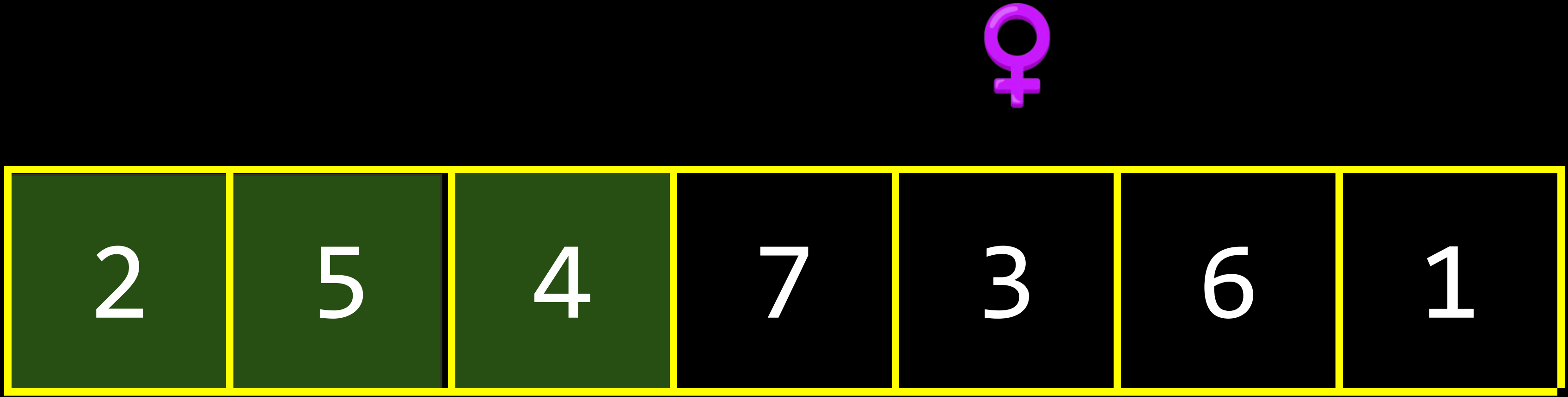
INSERT in correct spot!
(SWAP until the left is smaller)



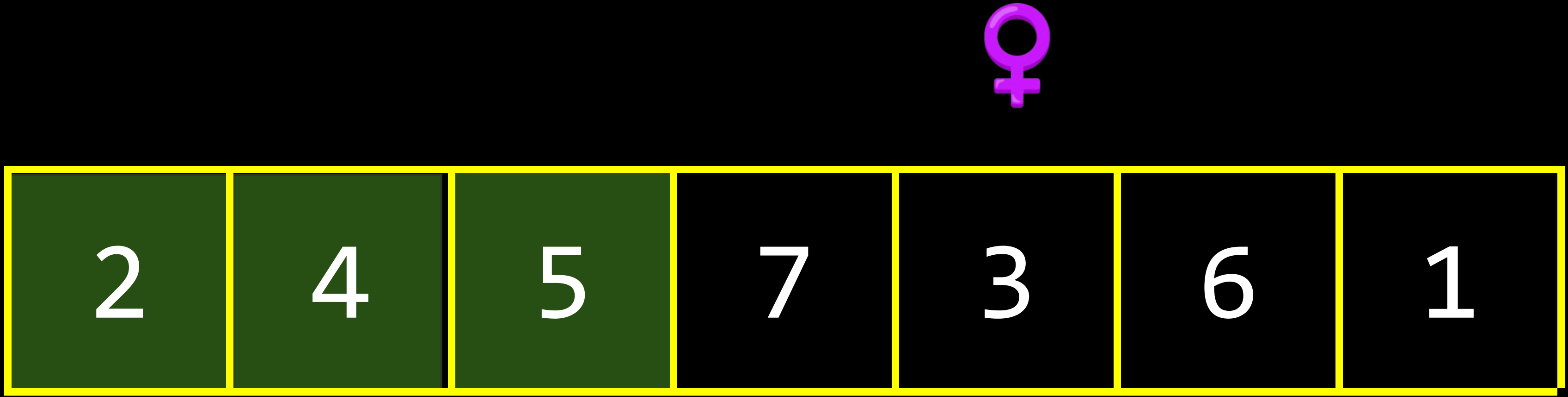
Check to the LEFT: is the one on the left smaller?



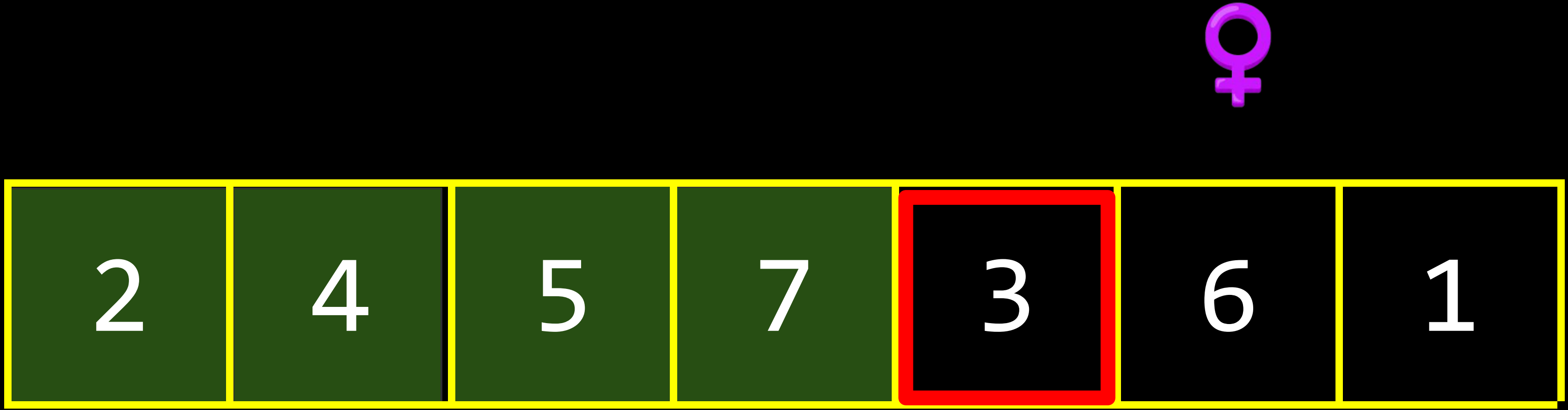
INSERT in correct spot!



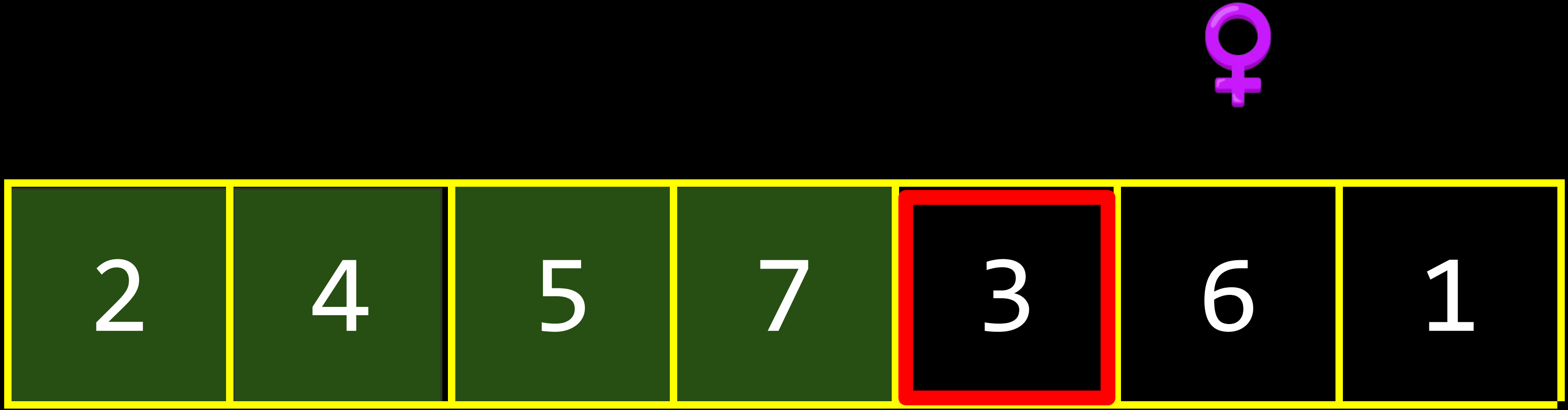
INSERT in correct spot!
(swap until in order)



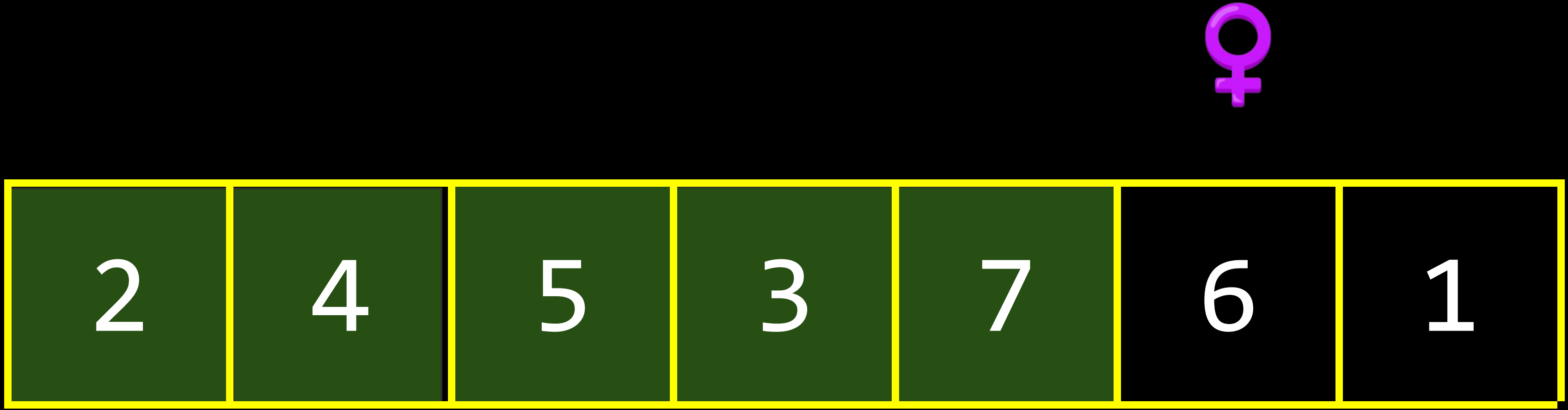
INSERT in correct spot!
(swap until in order)



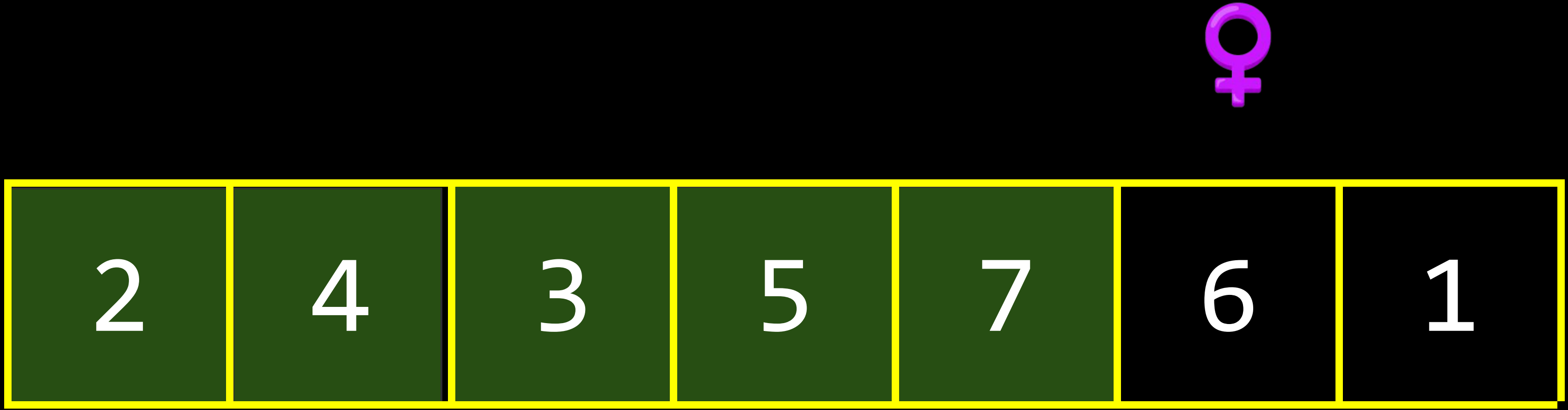
Check to the LEFT: is the one on the left smaller?



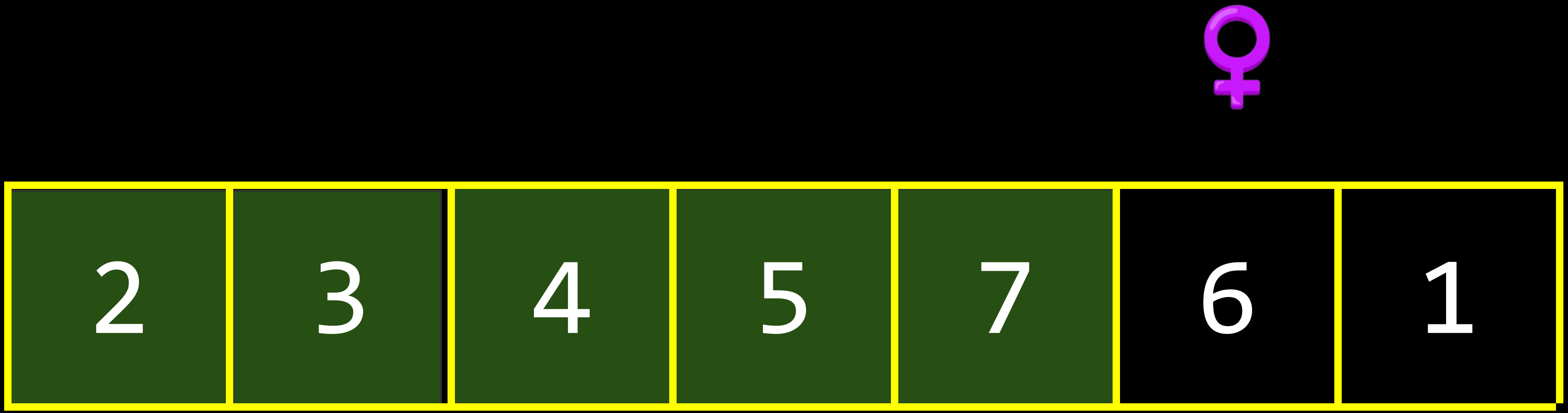
INSERT in correct spot!



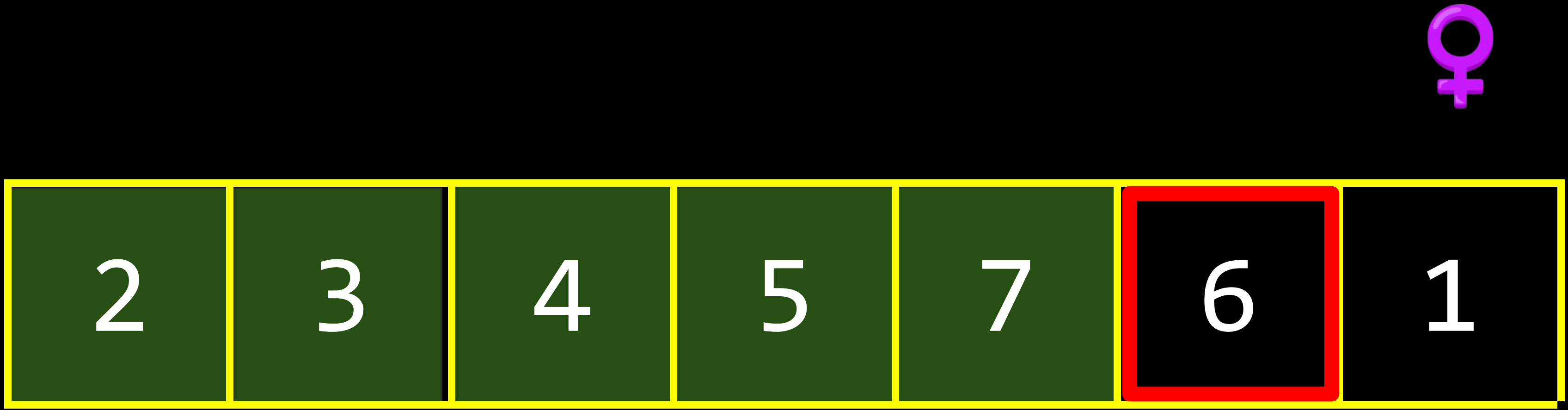
INSERT in correct spot!
(swap until in order)



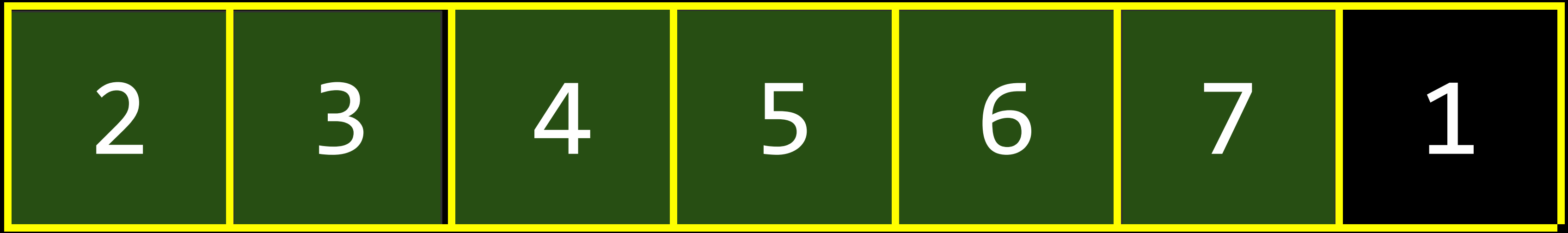
INSERT in correct spot!
(swap until in order)



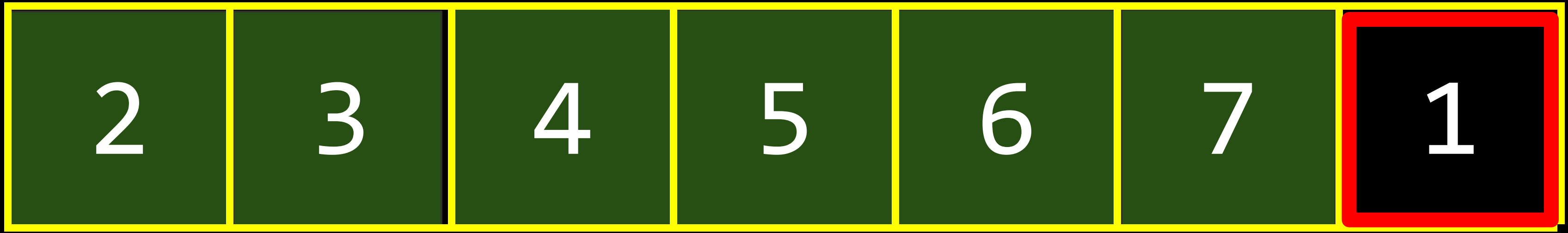
INSERT in correct spot!
(swap until in order)



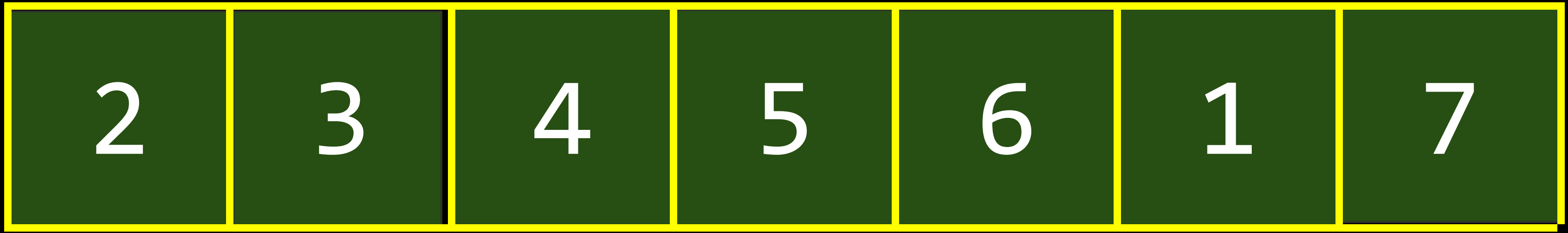
Check to the LEFT: is the one on the left smaller?



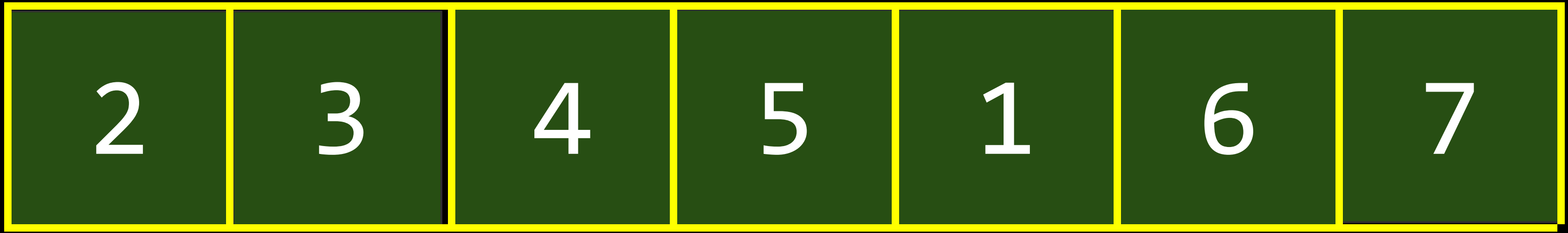
INSERT in correct spot!
(swap until in order)



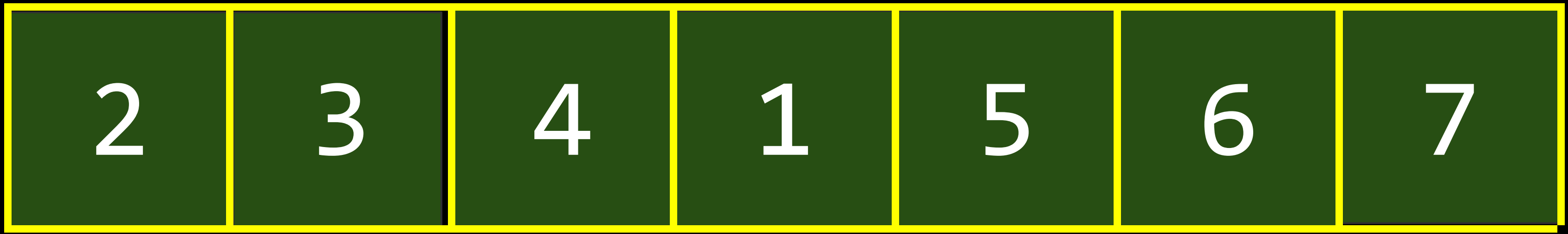
Check to the LEFT: is the one on the left smaller?



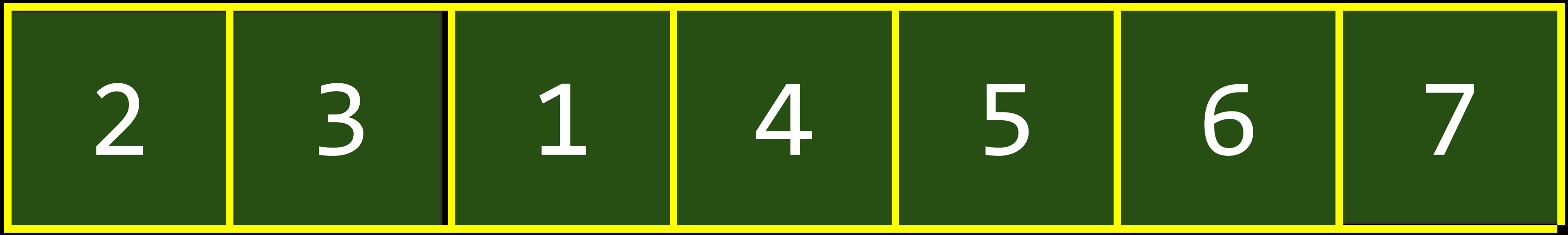
INSERT in correct spot!
(swap until in order)



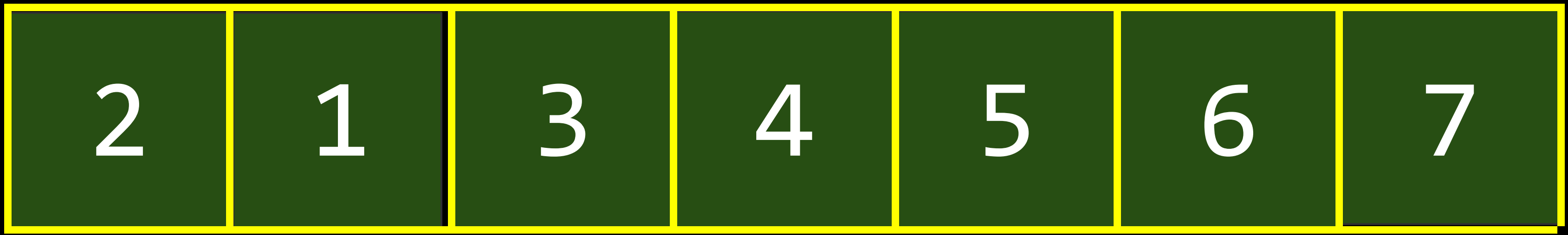
INSERT in correct spot!
(swap until in order)



INSERT in correct spot!
(swap until in order)



INSERT in correct spot!
(swap until in order)



INSERT in correct spot!
(swap until in order)

1	2	3	4	5	6	7

SORTED. ✅

INSERTION SORT

- ♠ Like sorting playing cards in your hand
- Create a “**sorted**” and “**unsorted**” part of array
- Compare the start of the “**unsorted**” part of the array on the right with the end of the “**sorted**” on the left - are they in the correct order?
- If not, swap those 2 values
- Continue swapping until the new value is in the correct order in the “**sorted**” array

$O(n^2)$	selection sort, insertion sort
$O(n \log n)$	
$O(n)$	
$O(\log n)$	
$O(1)$	

Classes of Running Times

Constant - $O(1)$

Logarithmic - $O(\log N)$

Linear - $O(N)$

Linearithmic - $O(N \log N)$

Polynomial - $O(N^2), O(N^3), O(N^4)$, etc.

Exponential - $O(2^N), O(3^N), O(4^N)$, etc.

SELECTION VS. INSERTION SORT

- Both OK for small lists!
- Both use nested loops - $O(n^2)$
- Insertion better for "**nearly sorted**" lists
- Ω (best case) is $O(n)$ for **insertion sort**
- Ω (best case) is still $O(n^2)$ for **selection sort** - we still have to go through all elements in the array left to right, repeatedly, to check if sorted

SELECTION VS. INSERTION SORT

- Ω (best case) for **insertion sort?**
- **It's already sorted! O(n)**
- O (worst case) for **insertion sort?**
- **It's sorted in reverse O(n^2)**

INSERTION SORT

7	6	5	4	3	2	1