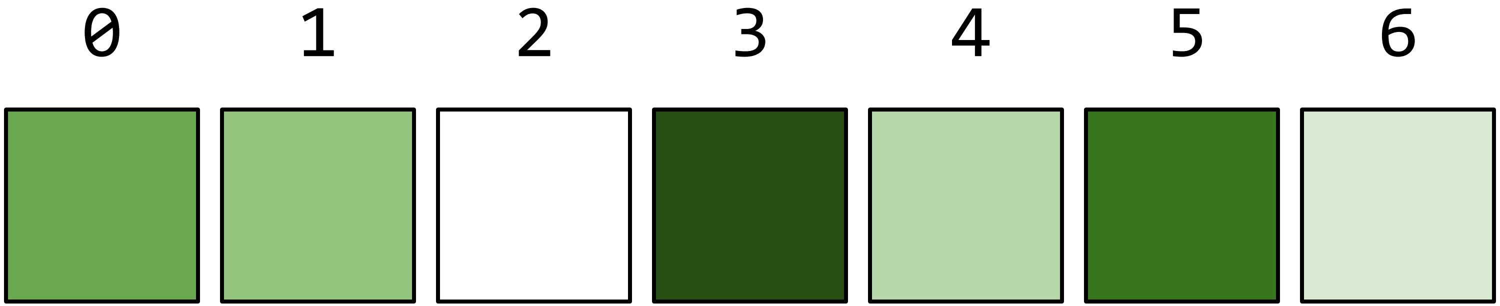
This is CS50

Think.

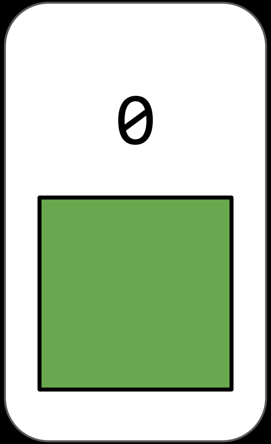
Pair.

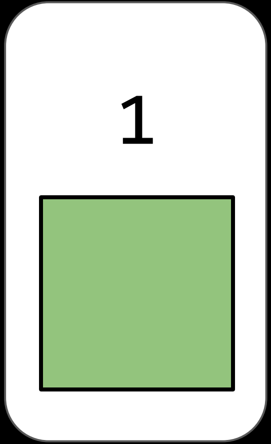
Share.

- How should we **compare** algorithms?
- When are **structs** useful?
- What is **recursion**?


0
1
2
3
4
5
6

Linear Search

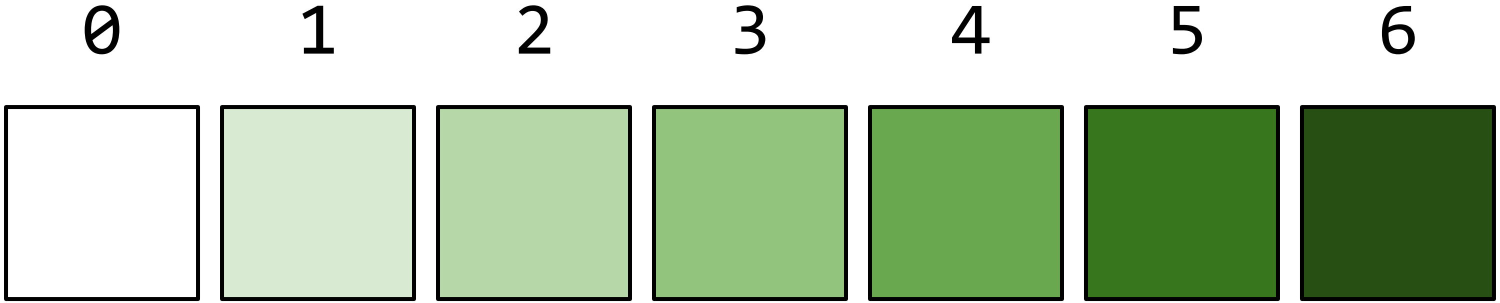

0



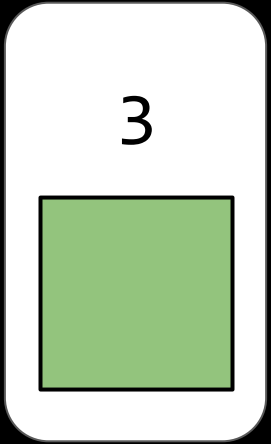
1

2

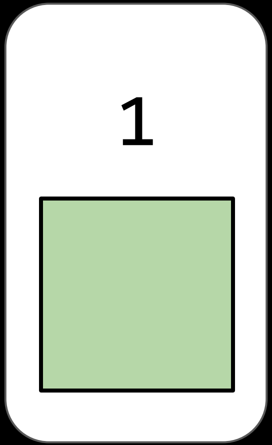
□

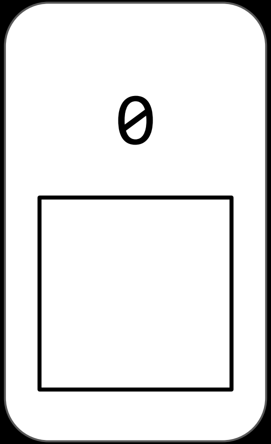

0
1
2
3
4
5
6

Binary Search



3


1



0

Running Time

	Linear Search	Binary Search
Number of Steps		

	Linear Search	Binary Search
Number of Steps	3 steps	

	Linear Search	Binary Search
Number of Steps	3 steps	3 steps

For any input, what is the **most** number of steps my algorithm will ever take?

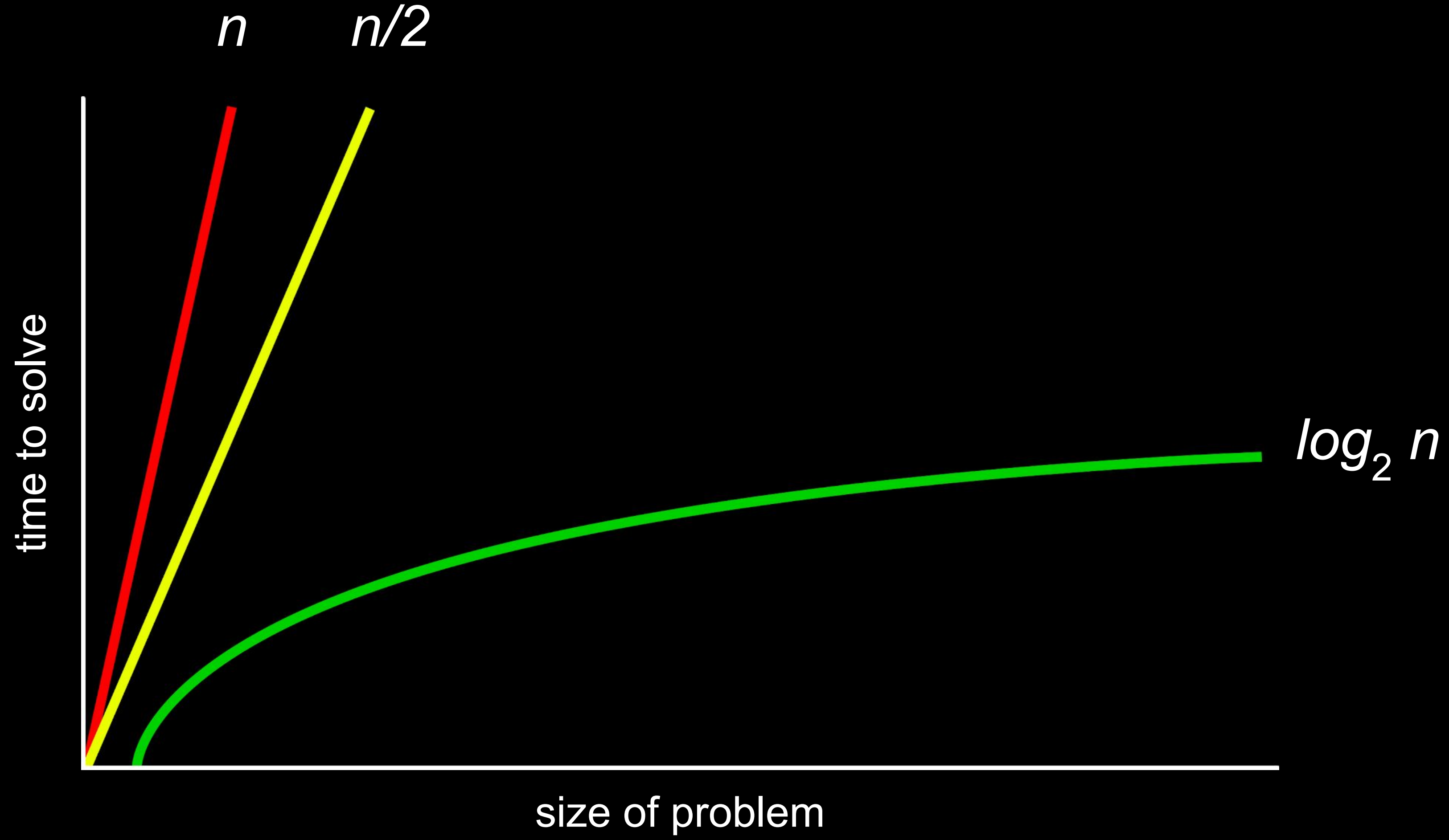
How many steps will my algorithm take for the very **worst case** input?

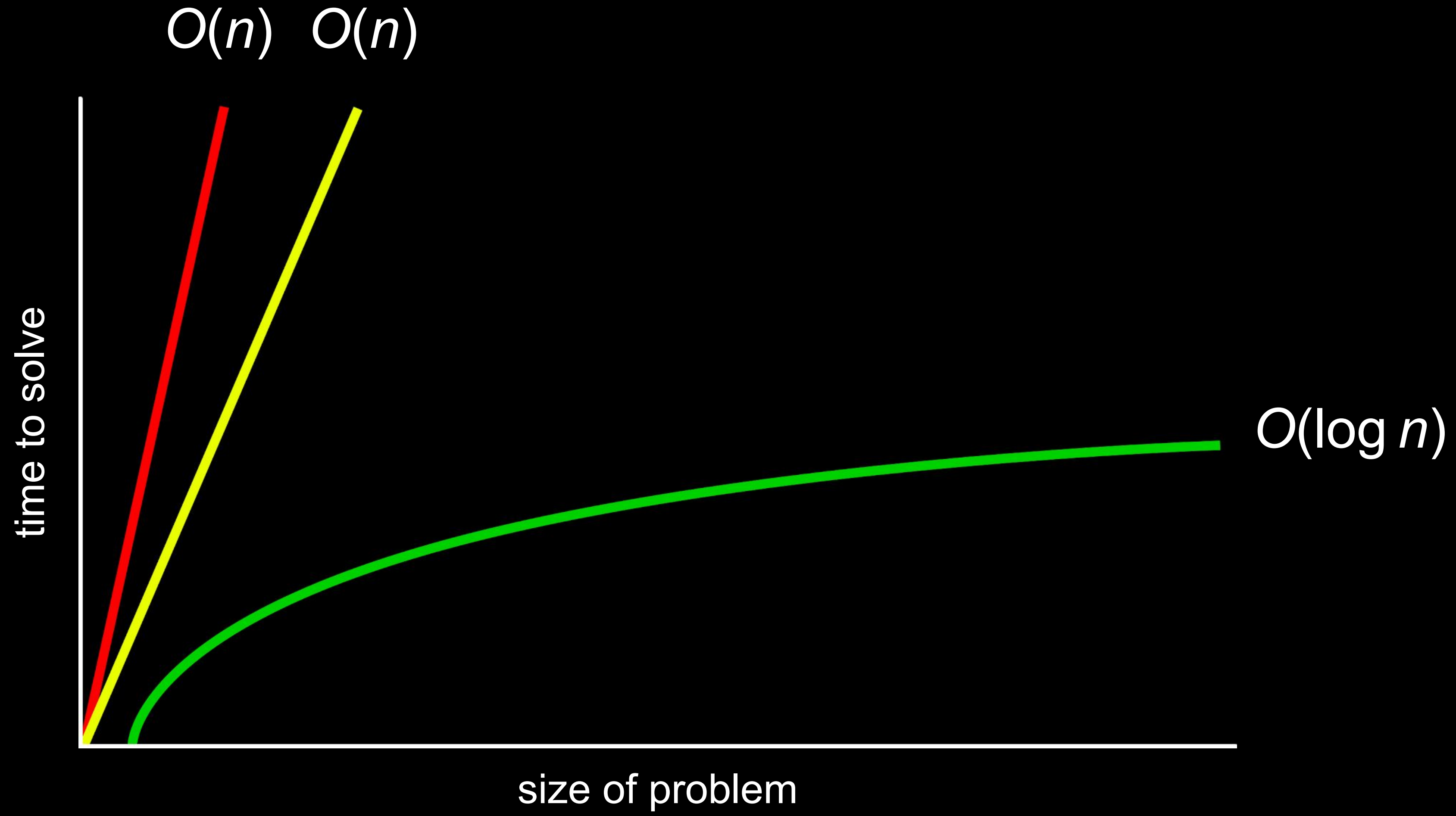
	Linear Search	Binary Search
Upper Bound		

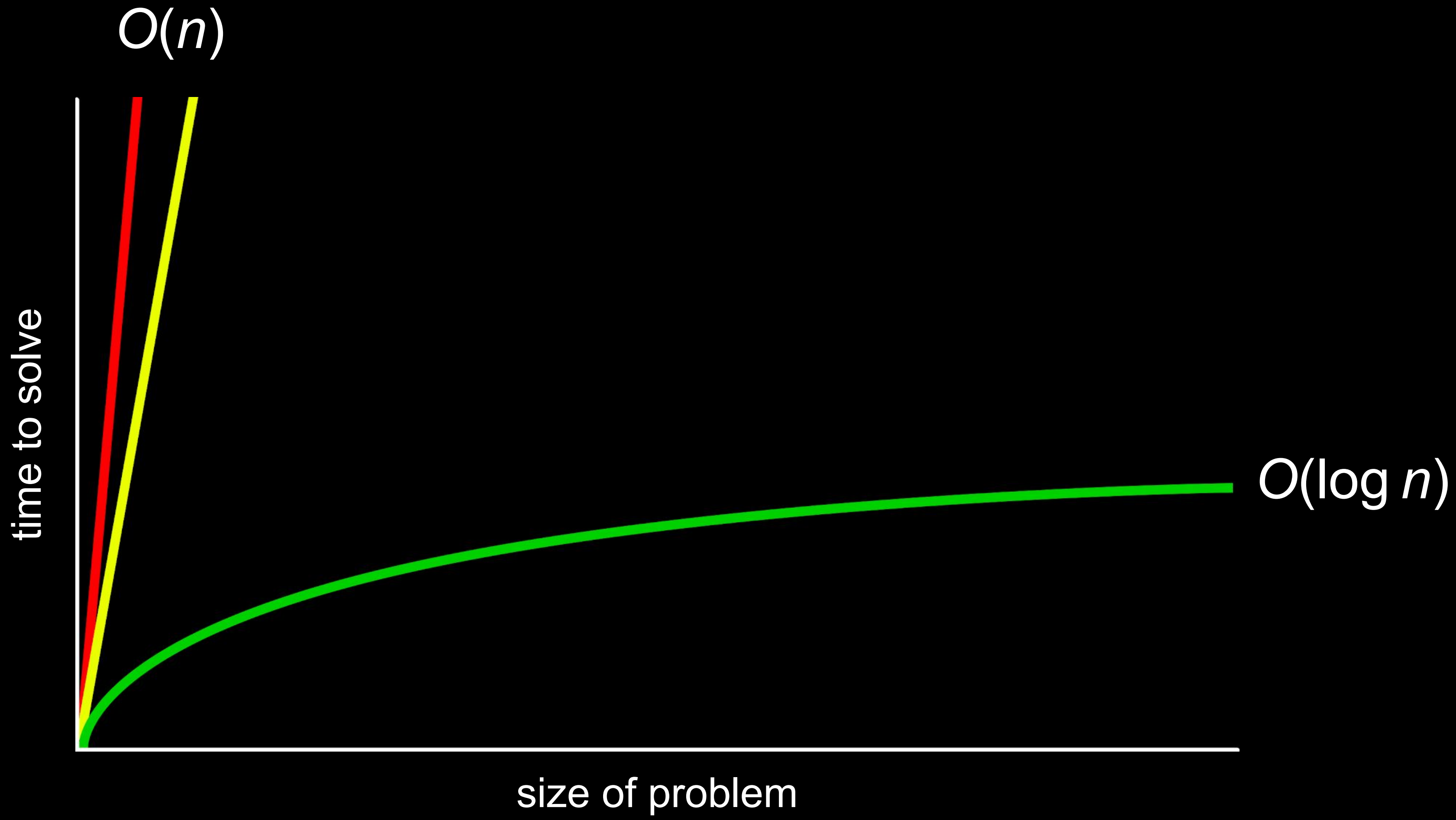
	Linear Search	Binary Search
Upper Bound	**n steps**	

	Linear Search	Binary Search
Upper Bound	**n steps**	**log n steps**

"On the order of…"


n
$n/2$
$\log_2 n$
time to solve
size of problem


$O(n)$
$O(n)$
$O(\log n)$
time to solve
size of problem


O(n)
O(log n)
time to solve
size of problem

	Linear Search	Binary Search
Upper Bound	***O*(n)**	***O*(log n)**

For any input, what is the **most** number of steps my algorithm will ever take?

For any input, what is the ~~**most**~~ number of steps my algorithm will ever take?

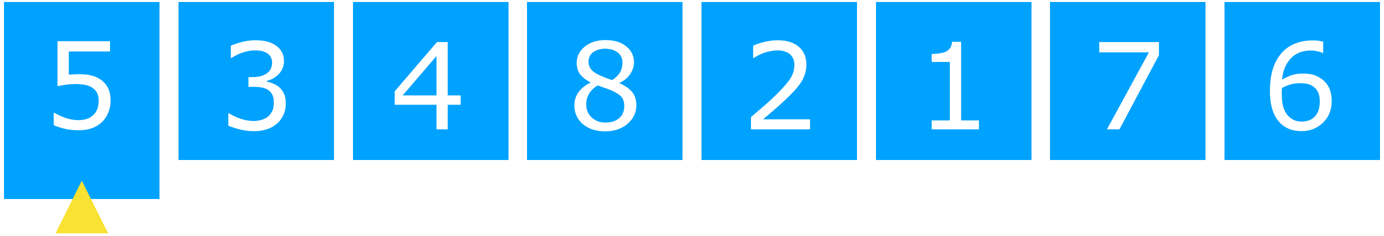
For any input, what is the **least** number of steps my algorithm will ever take?

How many steps will my algorithm take for the very **best case** input?

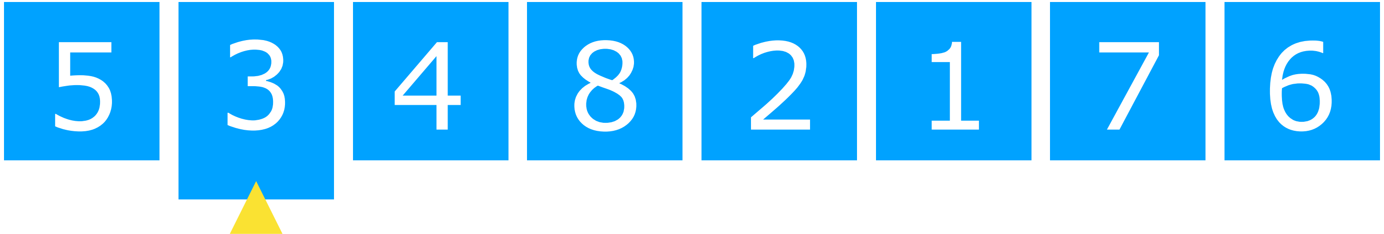
	Linear Search	Binary Search
Upper Bound	$O(n)$	$O(\log n)$
Lower Bound	1 step	1 step

	Linear Search	Binary Search
Upper Bound	$O(n)$	$O(\log n)$
Lower Bound	$\Omega(1)$	$\Omega(1)$

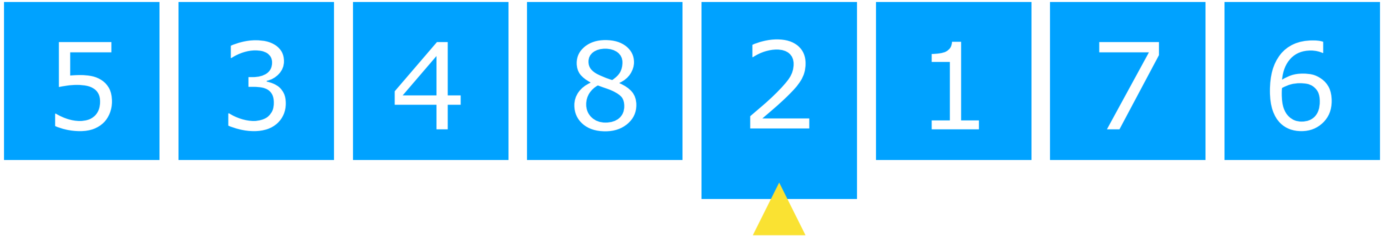
5	3	4	8	2	1	7	6



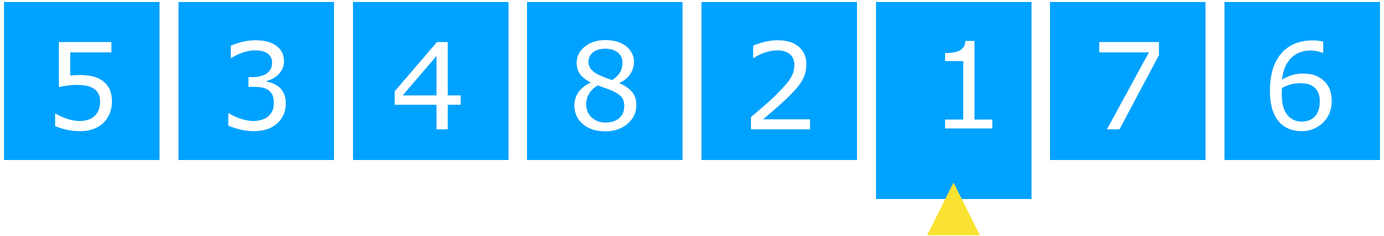
5	3	4	8	2	1	7	6



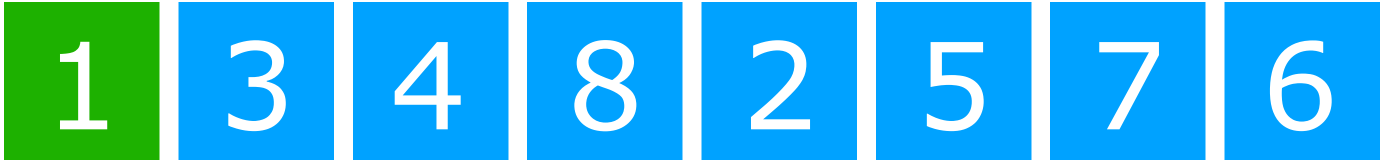
5	3	4	8	2	1	7	6



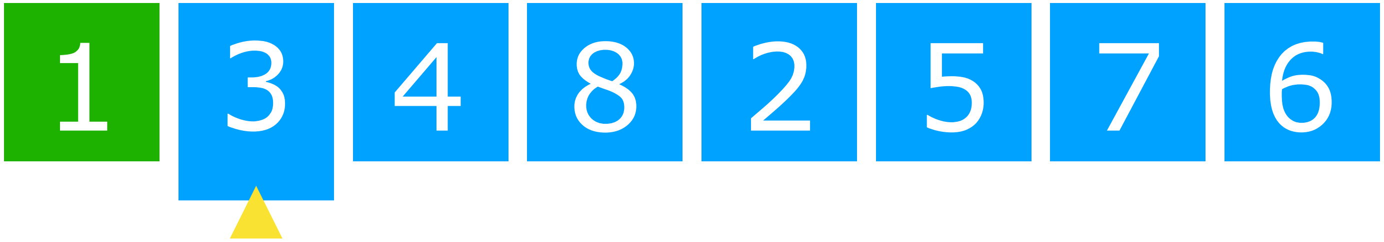
5 3 4 8 2 1 7 6



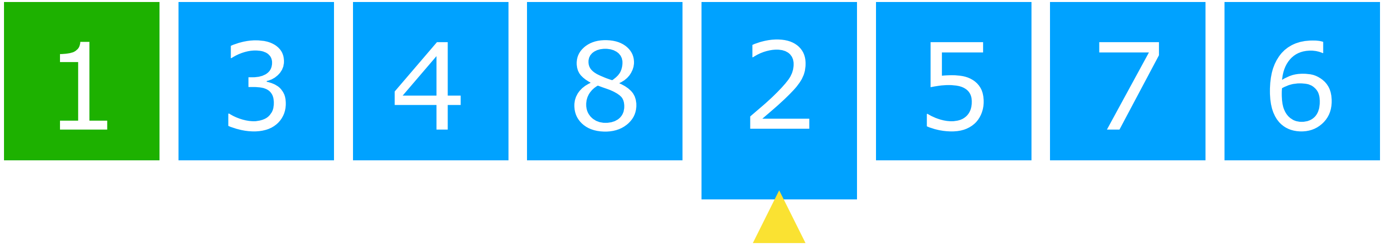
5	3	4	8	2	1	7	6



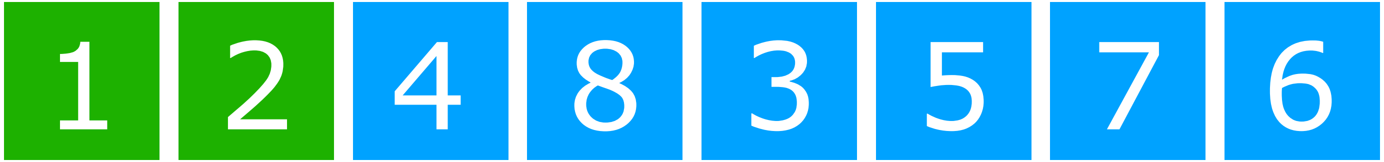
1	3	4	8	2	5	7	6



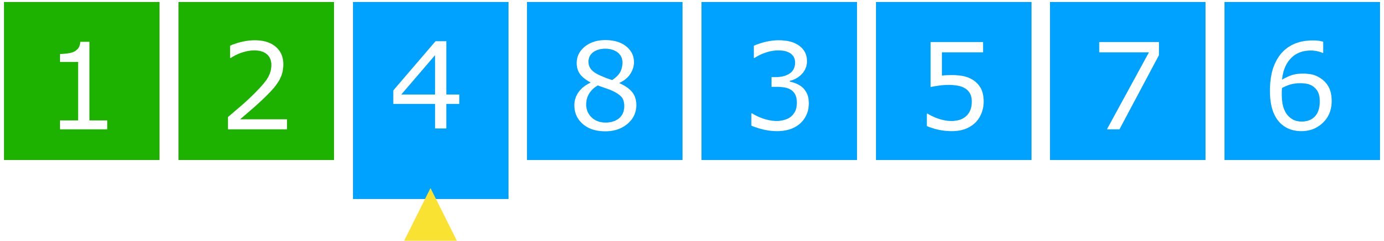
1 3 4 8 2 5 7 6



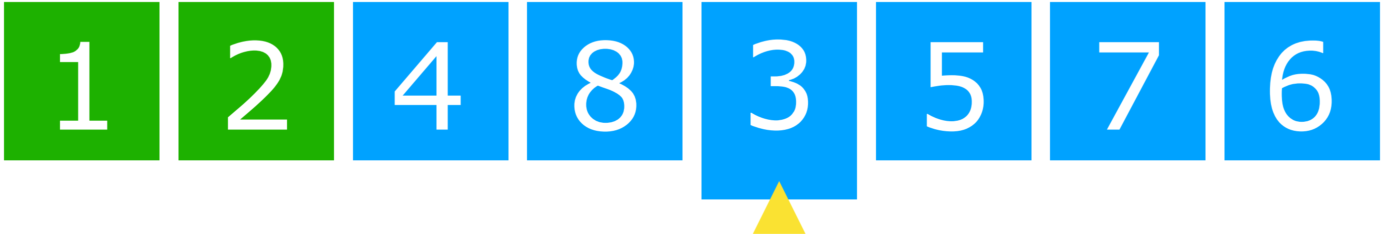
1	3	4	8	2	5	7	6



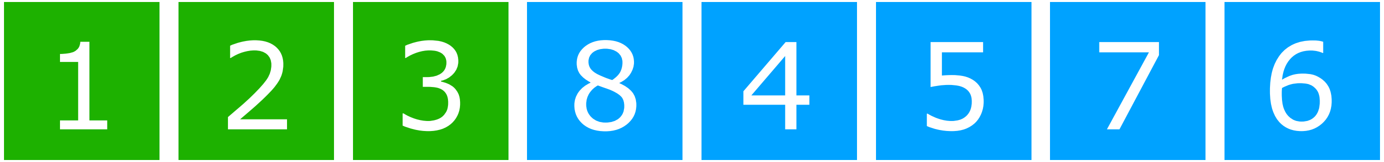
1 2 4 8 3 5 7 6



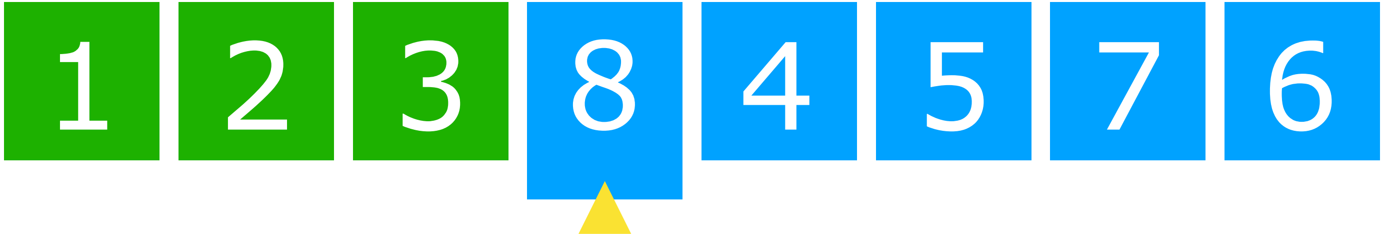
1 2 4 8 3 5 7 6

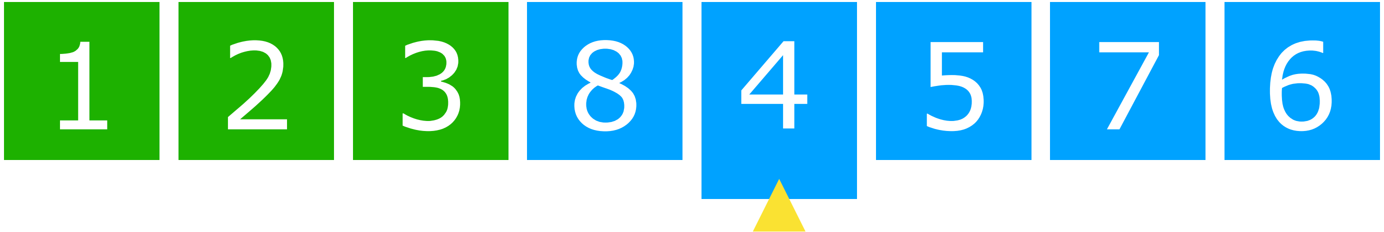


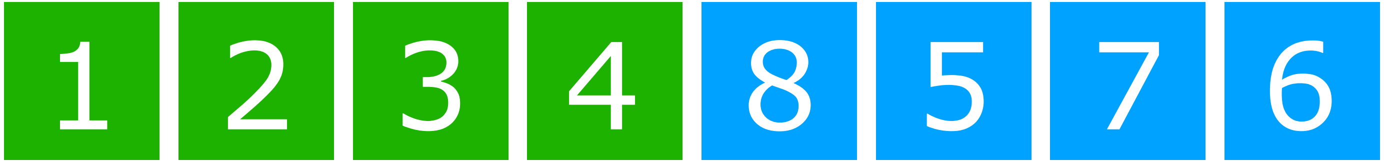
1	2	4	8	3	5	7	6



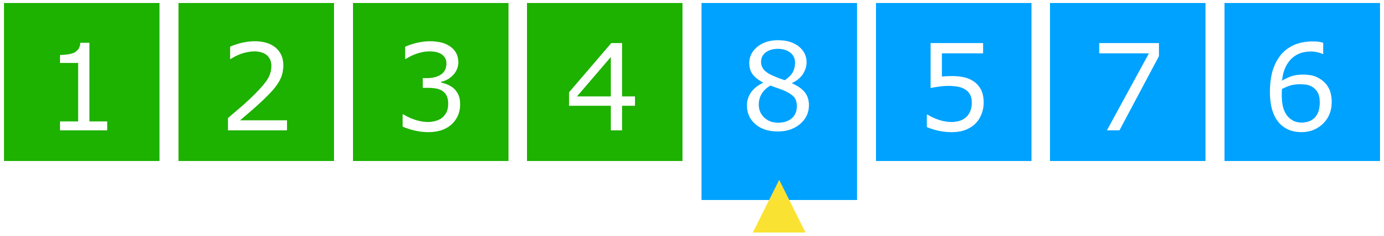
1 2 3 8 4 5 7 6


1
2
3
8
4
5
7
6

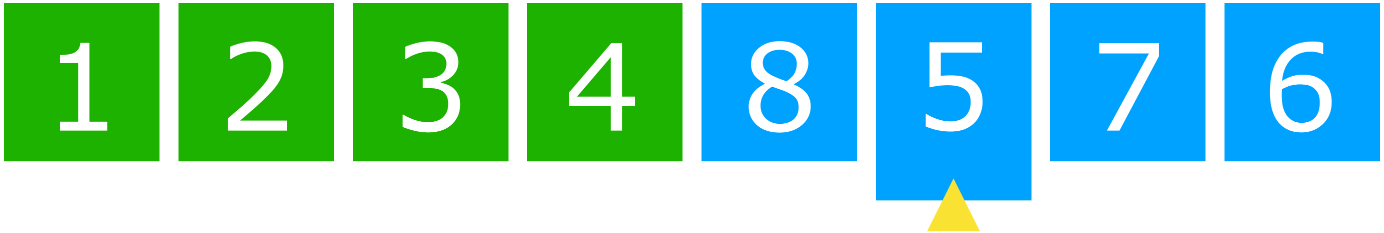

1
2
3
8
4
5
7
6



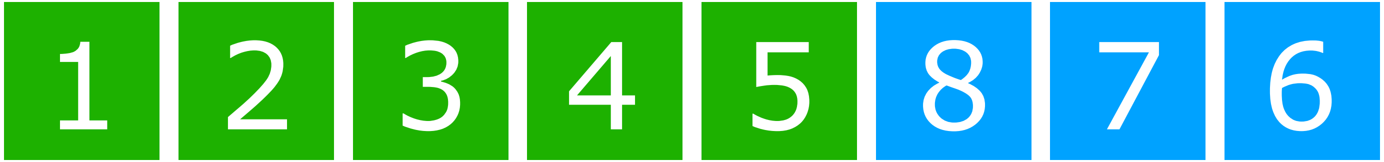
1	2	3	4	8	5	7	6



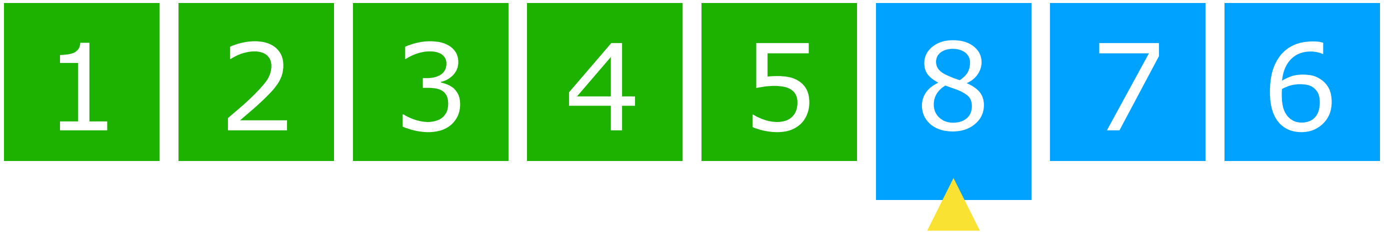
1 2 3 4 8 5 7 6



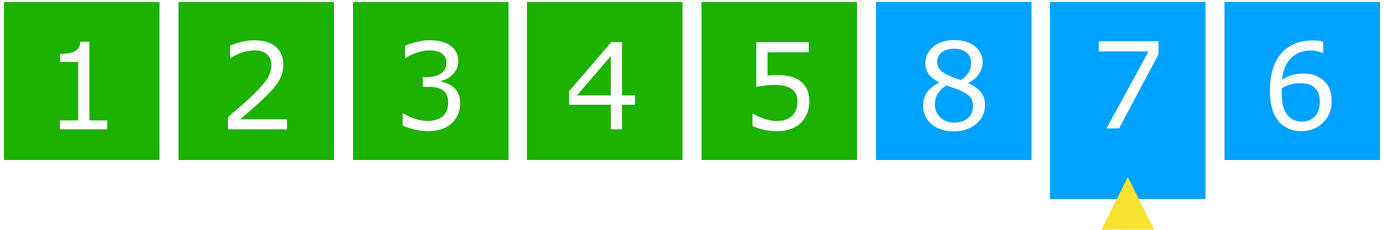
1 2 3 4 8 5 7 6

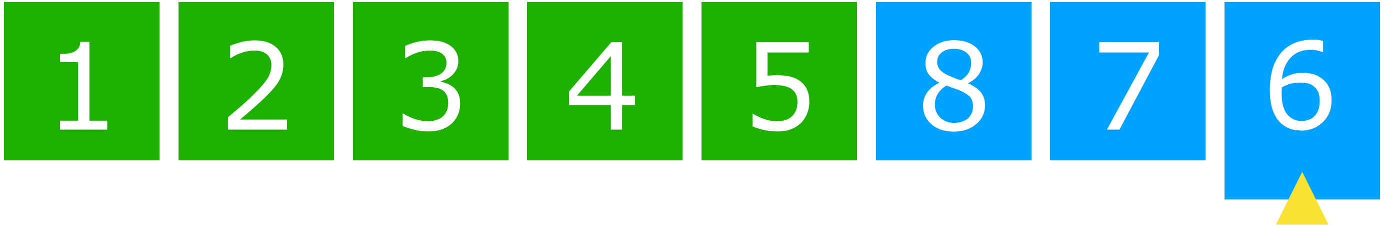


1	2	3	4	5	8	7	6

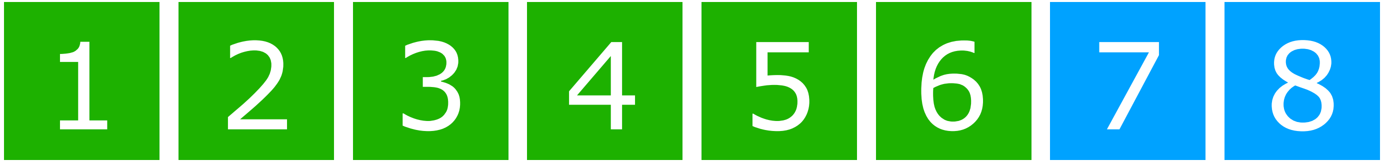


1 2 3 4 5 8 7 6

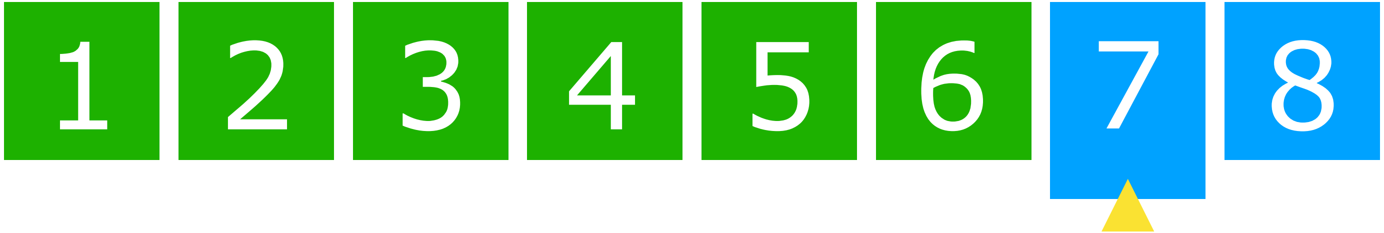

1
2
3
4
5
8
7
6

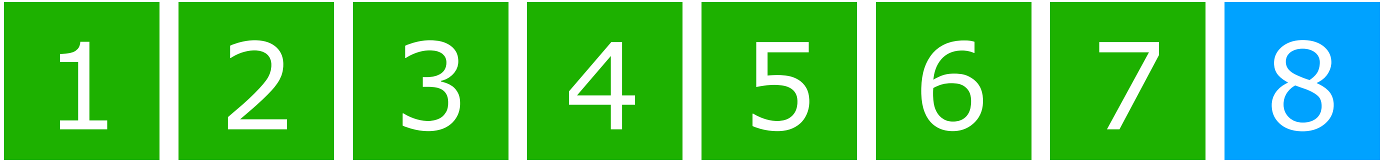


1 2 3 4 5 8 7 6



1 2 3 4 5 6 7 8


1
2
3
4
5
6
7
8



1 2 3 4 5 6 7 8

1 2 3 4 5 6 7 8

Selection Sort

Upper Bound

Selection Sort

Upper Bound $O(n^2)$

	Selection Sort
Upper Bound	$O(n^2)$
Lower Bound	

	Selection Sort
Upper Bound	$O(n^2)$
Lower Bound	$\Omega(n^2)$

Structs

```
typedef struct
{
    string name;
    int votes;
}
candidate;
```

```
typedef struct
{
    string name;
    int votes;
}
candidate;
```

```
typedef struct
{
    string name;
    int votes;
}
candidate;
```

```
typedef struct
{
    string name;
    int votes;
}
candidate;
```

```
candidate president;
candidate president;
president.name = "Alyssa";
president.votes = 10;
```

Structs and Functions Exercise

Create your own **get_candidate** function that prompts the user to input attributes for a candidate.

You may rely on **get_string**, **get_float**, etc.

Your function should return a candidate.

Arrays of Structs Exercise

Use your **get_candidate** function to create an array of three candidates, each of which should have attributes input by the user.

name	Alice	Bob	Charlie
votes	2	1	3

```
candidates[0];
```

name	Alice	Bob	Charlie
votes	2	1	3

```
candidates[0].name;
```

name	Alice	Bob	Charlie
votes	2	1	3

```
candidates[0].votes;
```

Recursion

Factorial

```
1! = 1
```

Factorial

```
1! = 1

2! = 2 * 1
```

Factorial

```
1!  =  1

2!  =  2  *  1

3!  =  3  *  2  *  1
```

Factorial

```
1! = 1

2! = 2 * 1

3! = 3 * 2 * 1

4! = 4 * 3 * 2 * 1
```

Factorial

```
            1! = 1
        2! = 2 * 1
    3! = 3 * 2 * 1
4! = 4 * 3 * 2 * 1
```

Factorial

```
4! = ?
```

Factorial

```
4! = 4 * 3!
```

Factorial

```
4! = 4 * 3!
```

"Recursive call"

Factorial

```
4! = 4 * 3!

3! = 3 * …
```

Factorial

4! = 4 * 3!

3! = 3 * 2!

Factorial

```
4! = 4 * 3!

3! = 3 * 2!

2! = 2 * 1!
```

Factorial

```
4! = 4 * 3!

3! = 3 * 2!

2! = 2 * 1!

1! = 1
```

Factorial

```
4! = 4 * 3!

3! = 3 * 2!

2! = 2 * 1!

1! = 1
```

← "Base case"

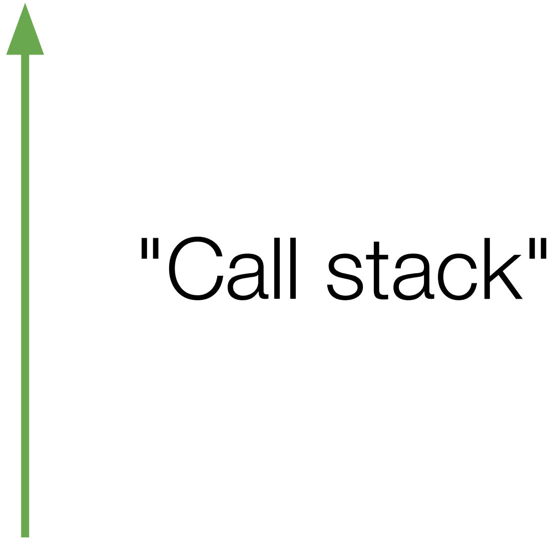
Factorial

```
4! = 4 * 3!

3! = 3 * 2!

2! = 2 * 1!

1! = 1
```



"Call stack"

Factorial

```
4! = 4 * 3!

3! = 3 * 2!

2! = 2 * 1
```

Factorial

```
4! = 4 * 3!

3! = 3 * 2 * 1
```

Factorial

```
4! = 4 * 3 * 2 * 1
```

Factorial

```
4! = 24
```

Factorial Exercise

Write your own recursive function called **factorial**.

factorial should take an **int** and return the factorial of the number as a parameter.

This was CS50