PSCS

Problem Solving for Computer Science

IS51021C

Goldsmiths Computing

March 8, 2021

Wk 8

Last week

Analysing Algorithms

1. RAM model
2. Big O notation
3. Worst-case Time Complexity
4. Problem 4

Last week

Analysing Algorithms

1. RAM model
2. Big O notation
3. Worst-case Time Complexity
4. Problem 4

Why do we use Big O notation?

To characterise the number of time-steps needed as the input size grows: (worst-case) time complexity

Big O

Bigger is **not** better in the study of algorithms

The larger the Big O class, the more operations needed

The larger the Big O class, the less efficient the algorithm is

We want the *smallest* Big O class containing the number of operations

$$O(n) < O(n^2)$$

Which is better?

Last week

Analysing Algorithms

1. RAM model
2. Big O notation
3. **Worst-case Time Complexity**
4. Problem 4

Linear Search

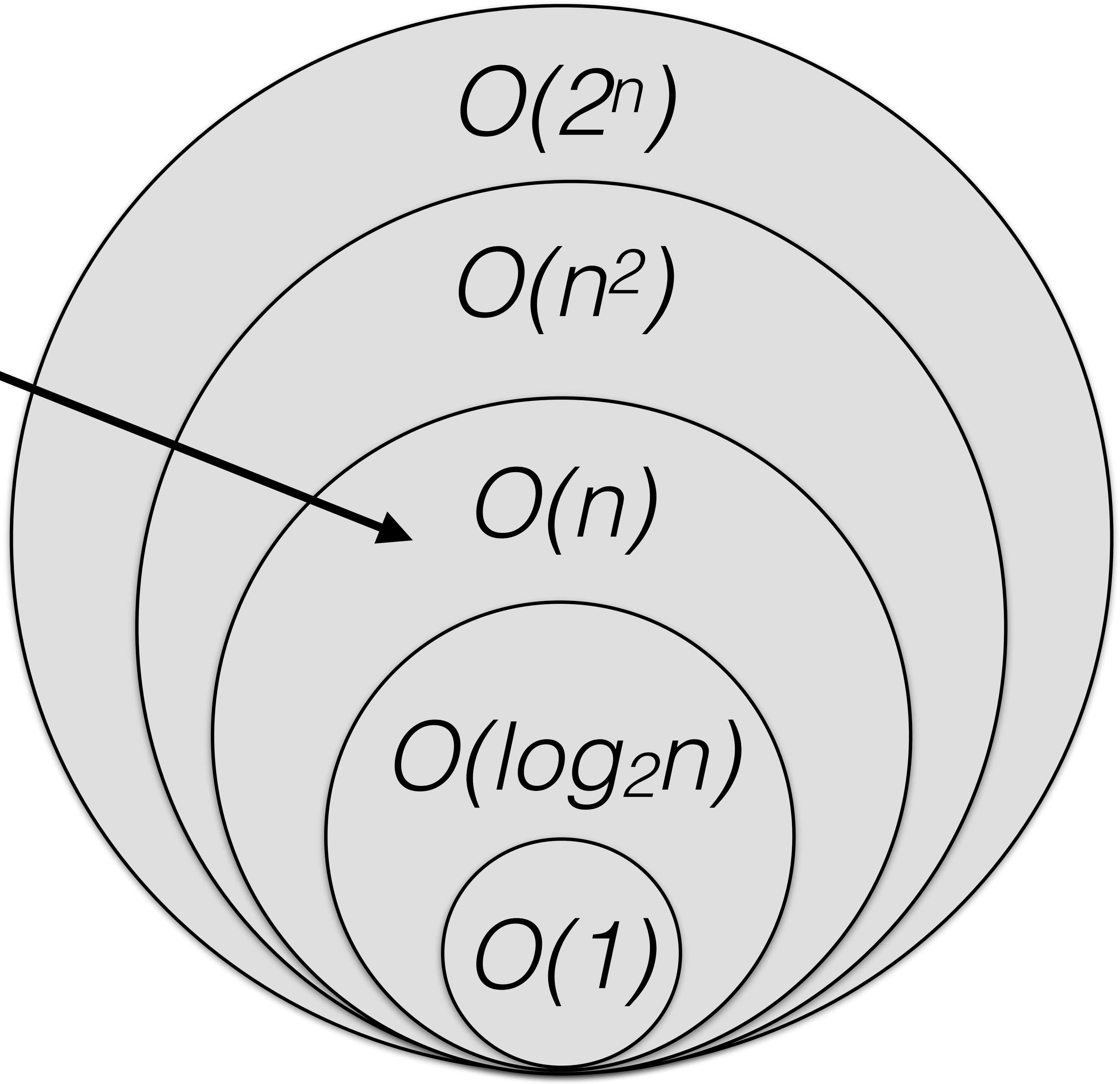
```
function linearSearch(array,x){

    var n = array.length;

    for (var i = 0; i < n; i++) {
        if (array[i] == x) {
            return true;
        }
    }

    return false;

}
```



Worst case: input where x is not in the array

Requires n iterations for length n
O(n) time-steps at most

Bubble Sort

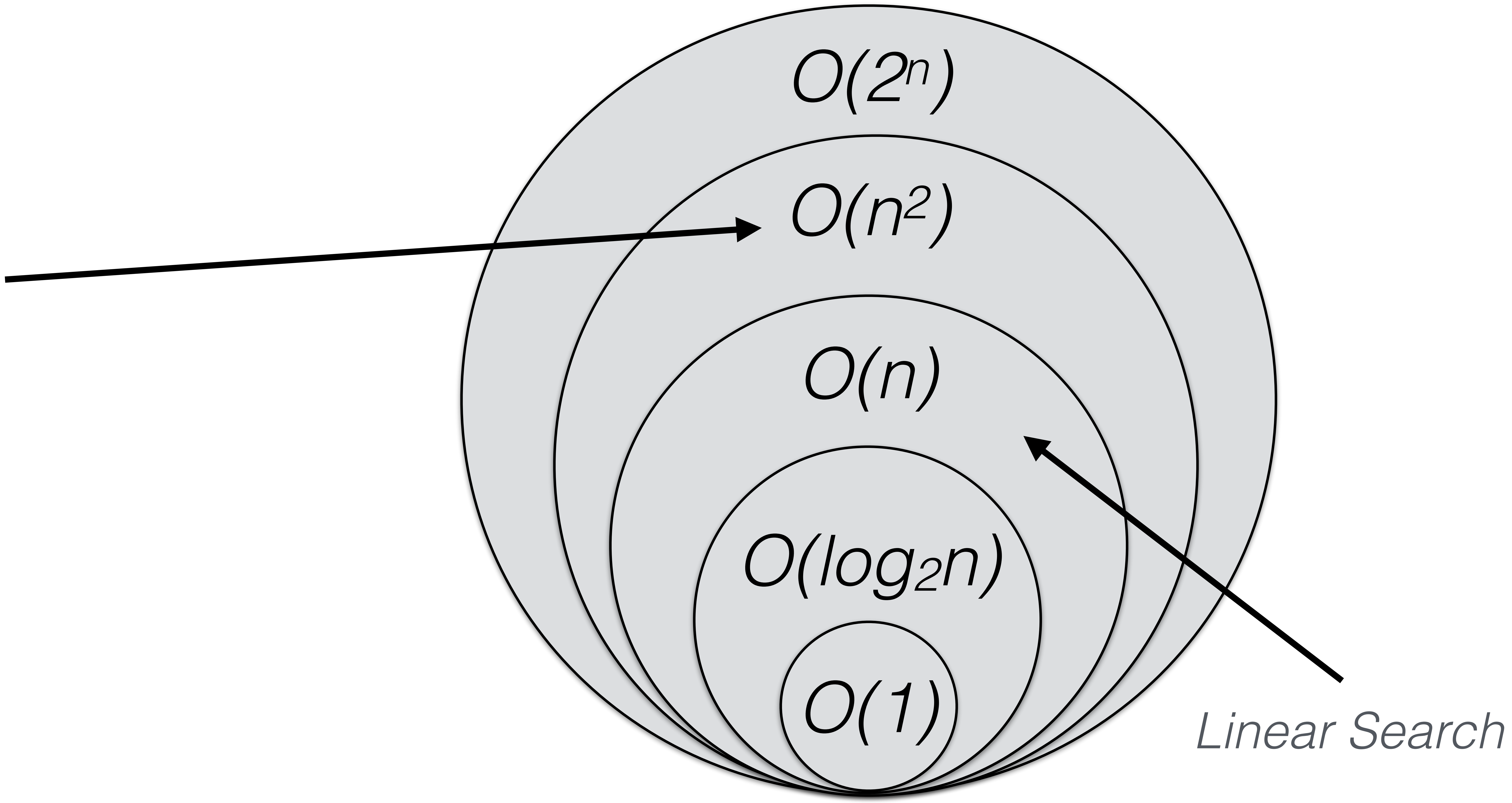
```
function swap(array,index1,index2) {
    var saveElement = array[index1];
    array[index1] = array[index2];
    array[index2] = saveElement;
    return array;
}

function bubbleSort(array) {
    var n = array.length;

    for (var i = 1; i < n; i++){
        var count = 0;
        for (var j = 0; j < n-1; j ++) {
            if (array[j+1] < array[j]) {
                count++;
                swap(array,j,j+1);
            }
        }
        if (count == 0) {
            break;
        }
    }

return array;

}
```



Worst case: array sorted in reverse

Requires (n-1) iterations, each with (n-1) iterations
$O(n^2)$ time-steps

Insertion Sort

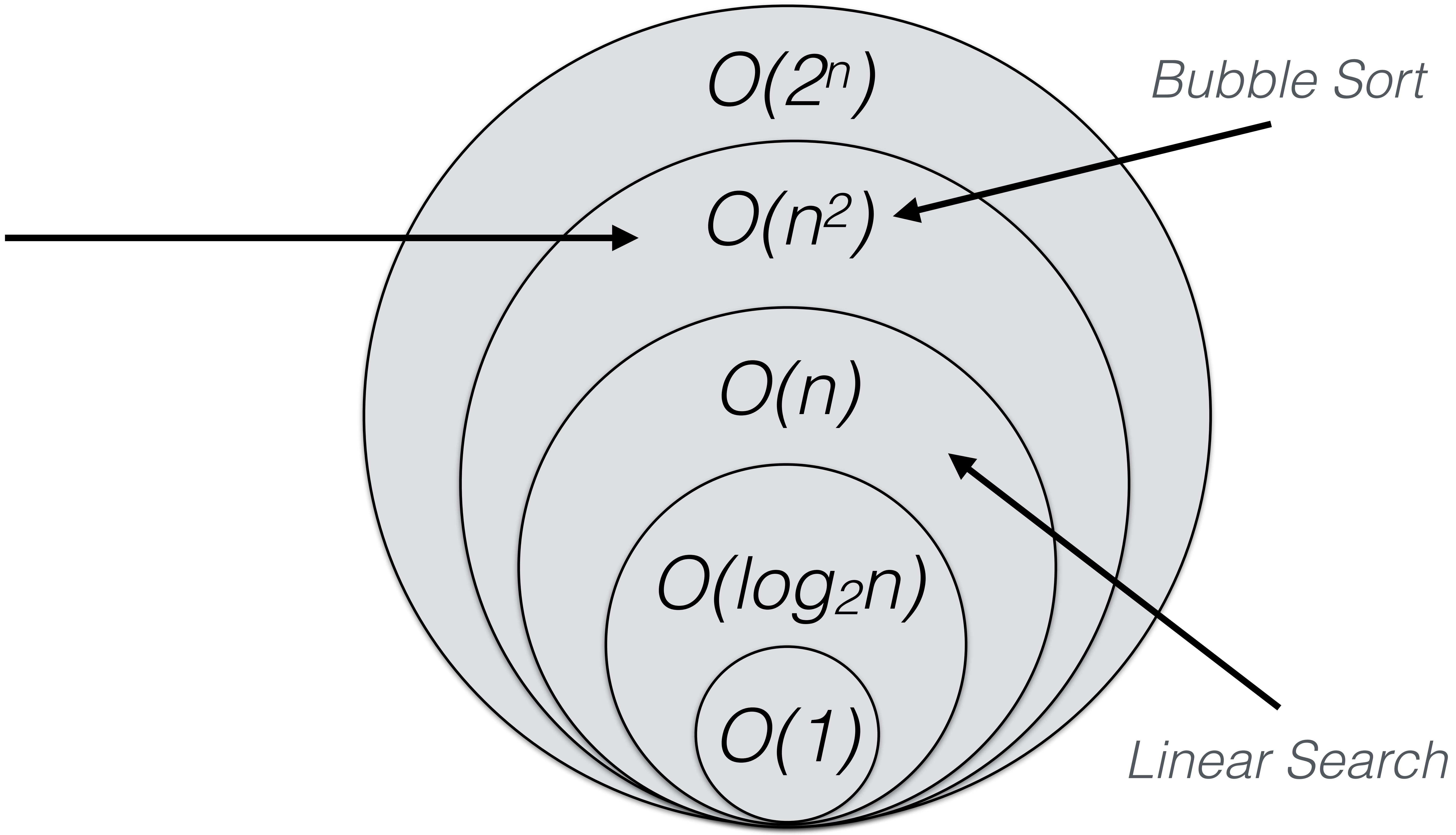
```
function swap(array,index1,index2) {
    var saveElement = array[index1];
    array[index1] = array[index2];
    array[index2] = saveElement;
    return array;
}

function insertionSort(array) {
    var n = array.length;

    for (var i = 1; i < n; i++) {
        var j = i;
        while ((j > 0) && (array[j-1]>array[j])) {
            swap(array,j,j-1);
            j--;
        }
    }

    return array;

}
```

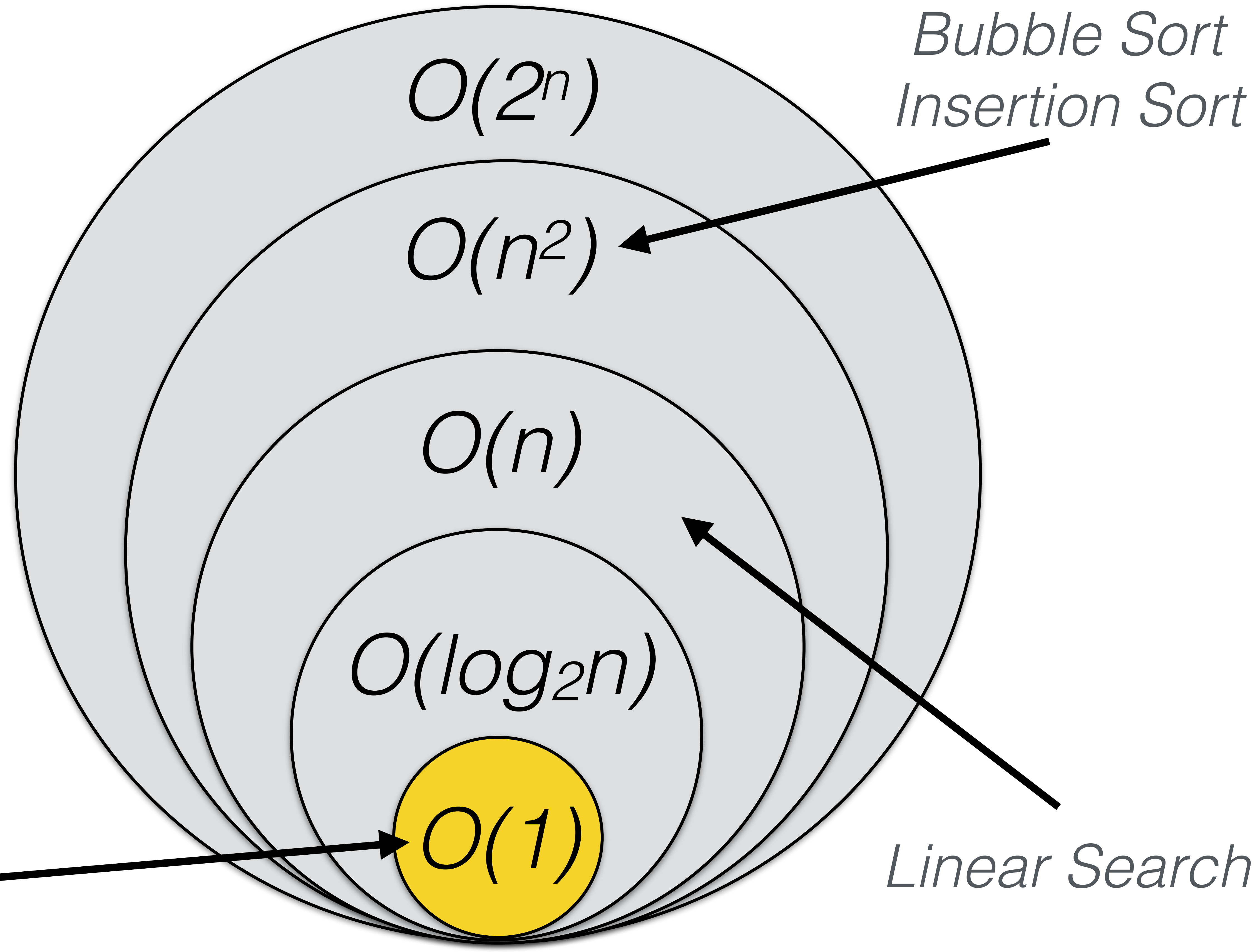


Worst case: array sorted in reverse

(n-1) iterations i, each with i swaps:

$1 + 2 + \ldots + (n-1) = n(n-1)/2$

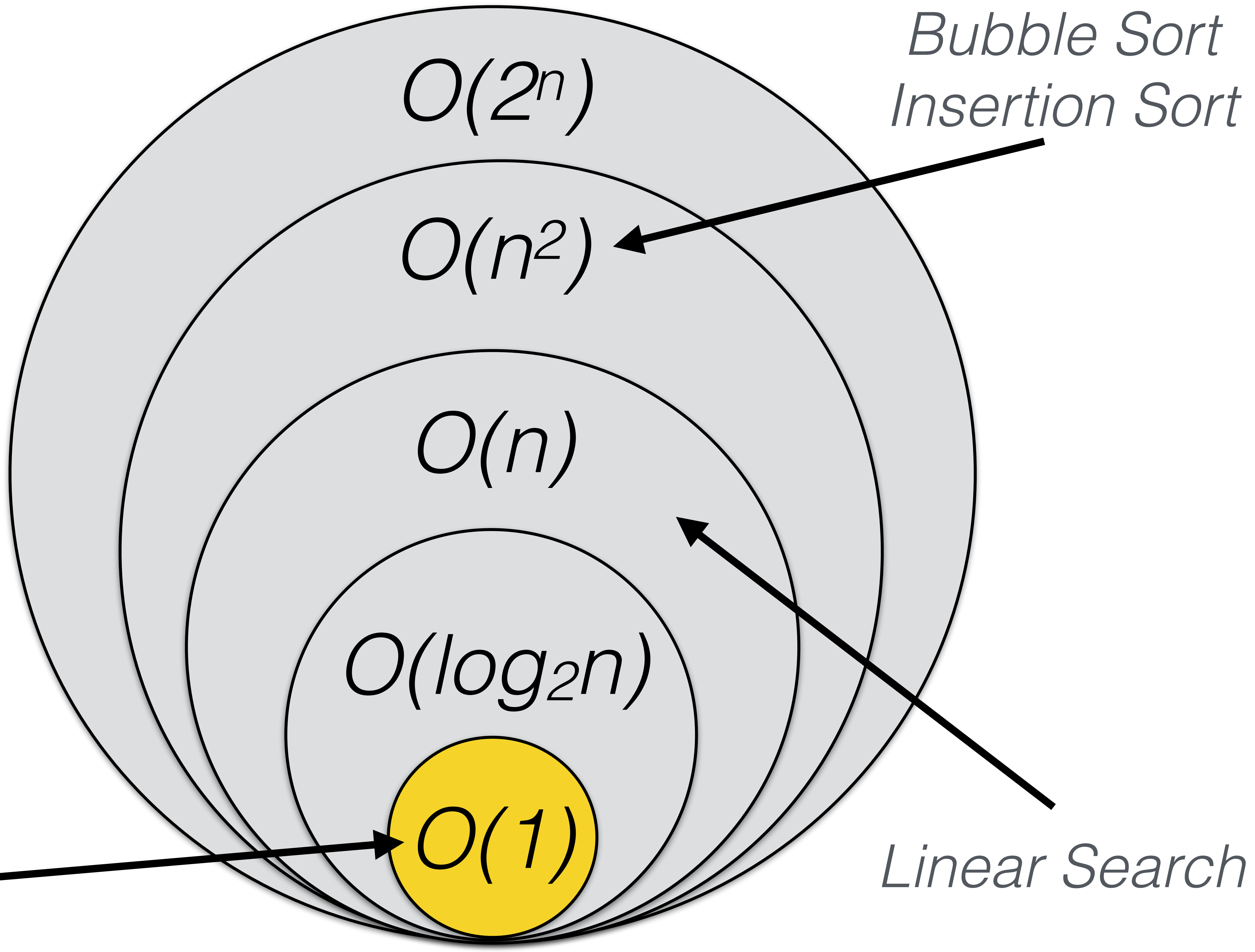
$O(n^2)$ *time-steps*

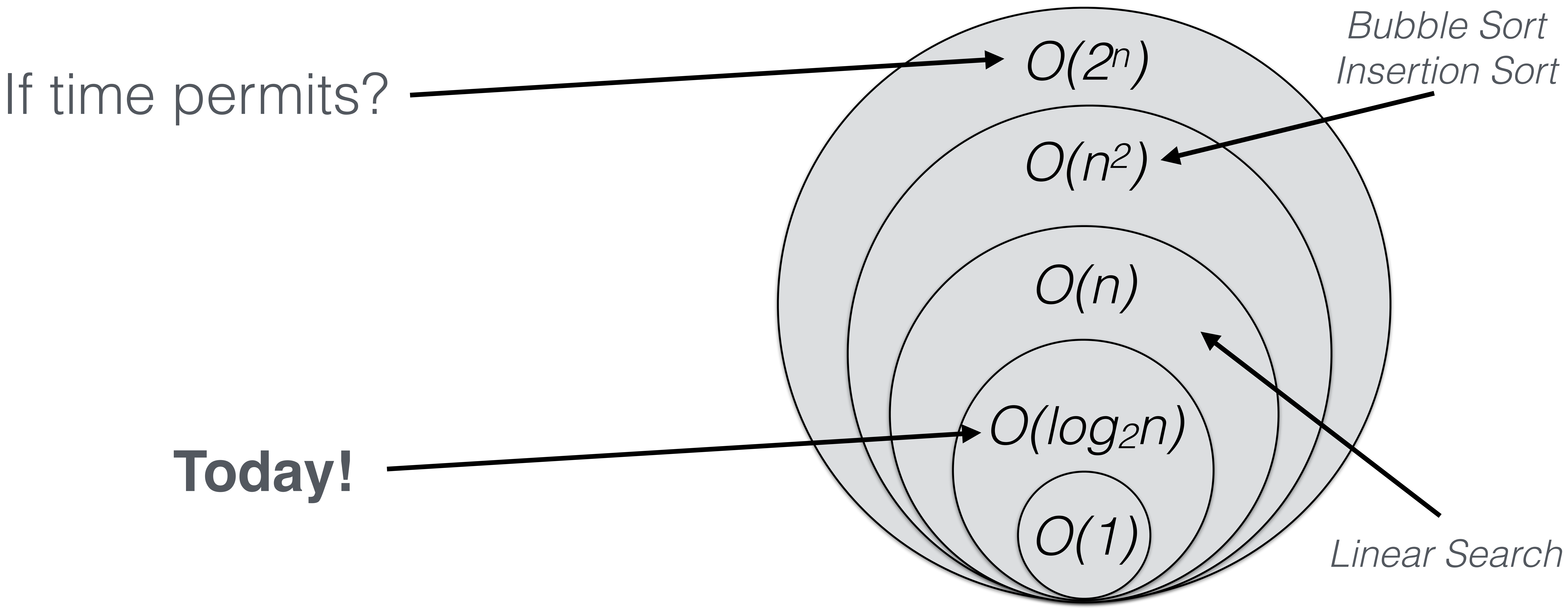


Can you come up with any algorithm that has a time complexity here?

```
function isEmpty(array){
    return array.length === 0;
}
```

Can you come up with any algorithm that has a time complexity here?



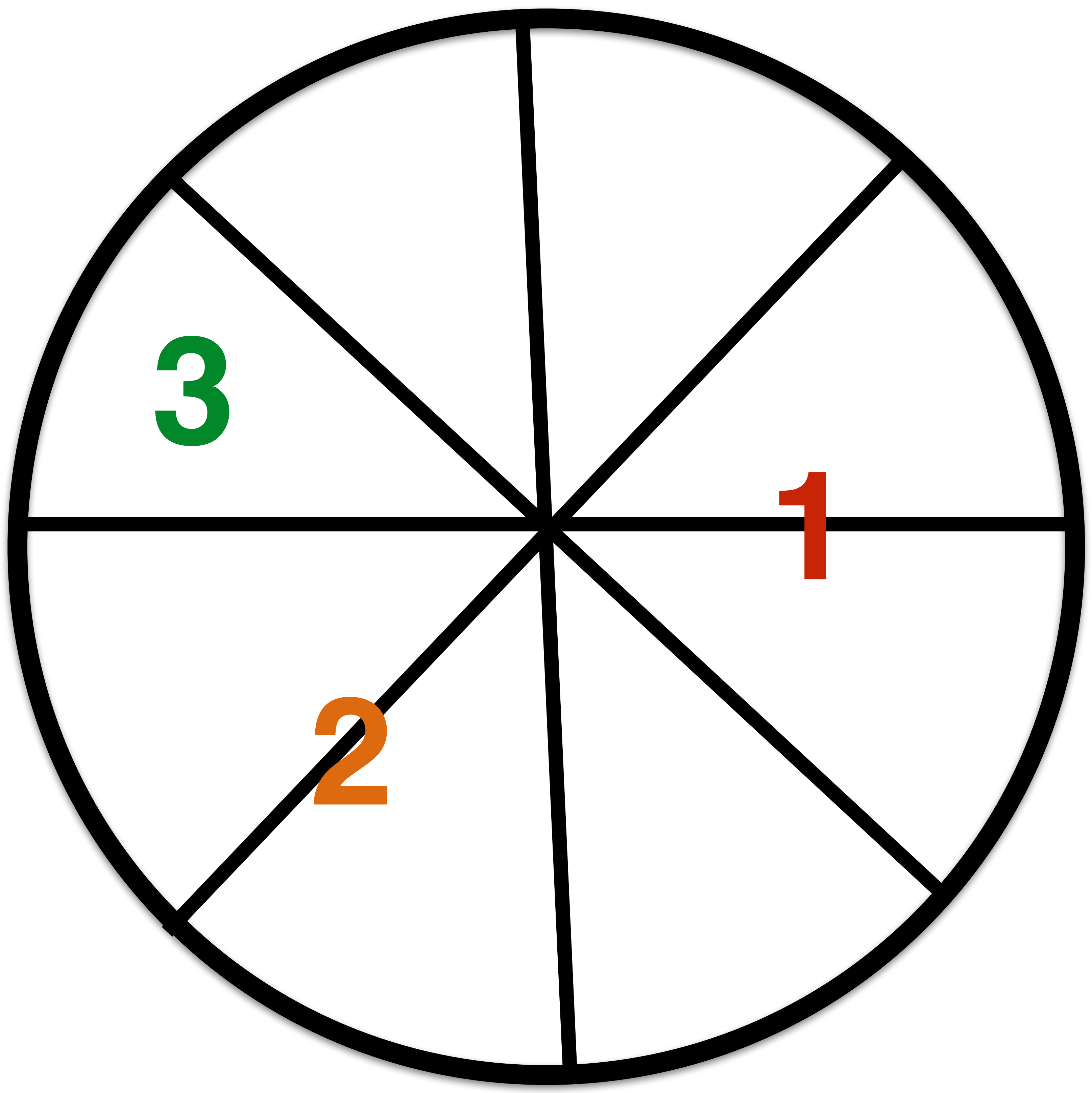

If time permits?
Today!
O(2n)
O(n2)
O(n)
O(log2n)
O(1)
Bubble Sort
Insertion Sort
Linear Search

Last week

Analysing Algorithms

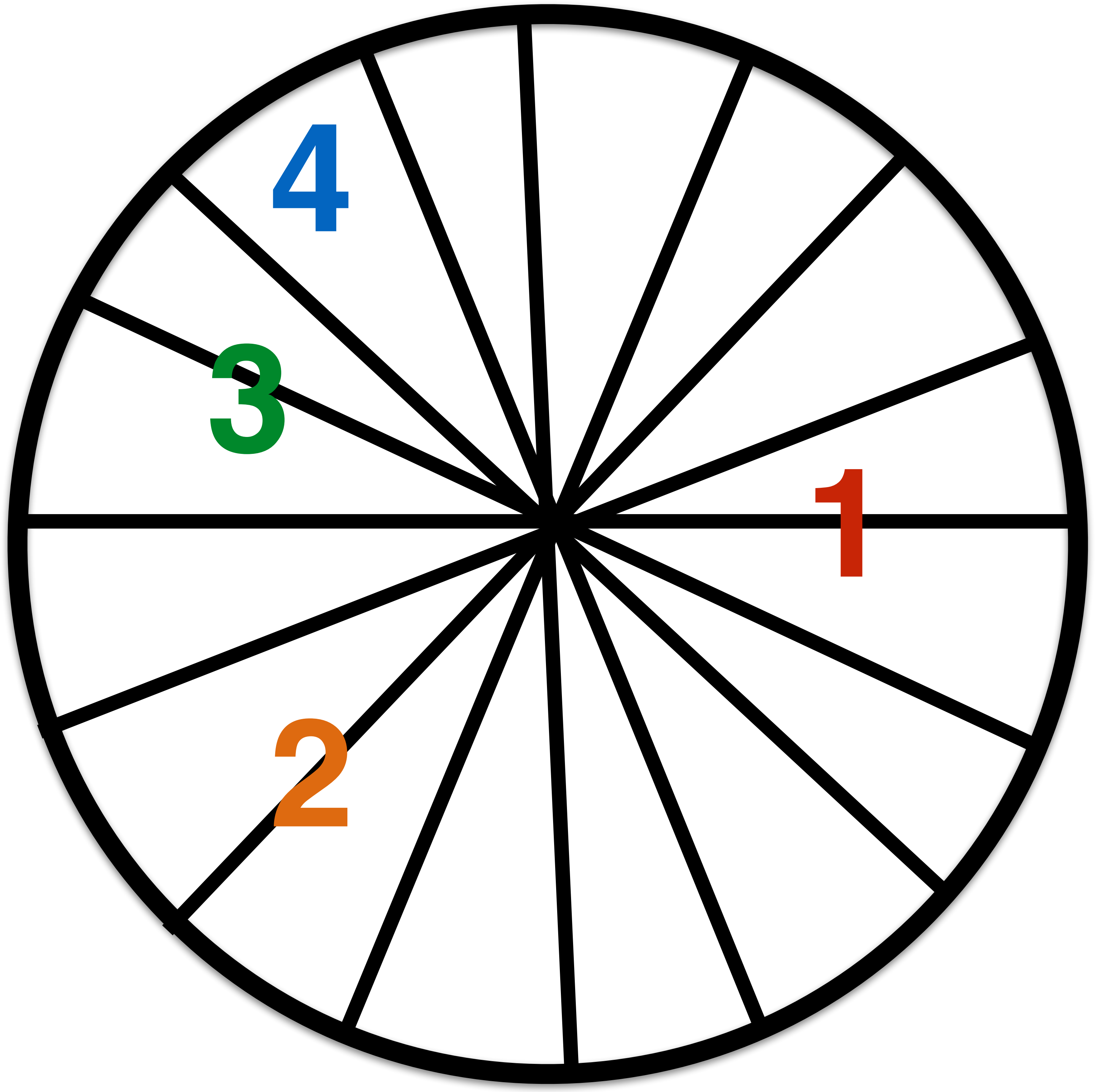
1. RAM model
2. Big O notation
3. Worst-case Time Complexity
4. **Problem 5**

$$8 * (1/2) * (1/2) * (1/2) = 1$$



For **8** slices we could accommodate **3** groups of friends

$$8 * (1/2) * (1/2) * (1/2) * (1/2) = 1$$



For **16** slices how many groups of friends?

4

For **n** slices how many groups of friends?

k = number of groups

For **n** slices how many groups of friends if they ask for two-thirds each time?

k = number of groups

For **n** slices how many groups of friends?

n	*k = number of groups*
1	2^0
2	2^1
3	2^1+1
4	2^2
5	2^2+1
6	2^2+2
7	2^2+3
8	2^3
9	2^3+1
10	2^3+2
11	2^3+3
12	2^3+4
13	2^3+5
14	2^3+6
15	2^3+7
16	2^4

Our examples

$2^{(\text{num of groups})}$

For **n** slices how many groups of friends?

k = number of groups

n	
1	2^0
2	2^1
3	2^1+1
4	2^2
5	2^2+1
6	2^2+2
7	2^2+3
8	2^3
9	2^3+1
10	2^3+2
11	2^3+3
12	2^3+4
13	2^3+5
14	2^3+6
15	2^3+7
16	2^4

Not all examples are powers of 2, or even

Let's say for each group we give away the ceiling of half the amount

We're generous

Our examples

$2^{(\text{num of groups})}$

For **n** slices how many groups of friends?

k = number of groups

n	
1	2^0
2	2^1
3	2^1+1
4	2^2
5	2^2+1
6	2^2+2
7	2^2+3
8	2^3
9	2^3+1
10	2^3+2
11	2^3+3
12	2^3+4
13	2^3+5
14	2^3+6
15	2^3+7
16	2^4

Every number **n** can be written as $2^m + p$ where $p < 2^m$

For **n** slices how many groups of friends?

k = number of groups

n	
1	2^0
2	2^1
3	2^1+1
4	2^2
5	2^2+1
6	2^2+2
7	2^2+3
8	2^3
9	2^3+1
10	2^3+2
11	2^3+3
12	2^3+4
13	2^3+5
14	2^3+6
15	2^3+7
16	2^4

Every number **n** can be written as $2^m + p$ where $p < 2^m$

Divide **n** in half: $2^{m-1} + p/2$
If p is not even, give away ceiling of **n/2**

For **n** slices how many groups of friends?

n	*k = number of groups*
1	2^0
2	2^1
3	2^1+1
4	2^2
5	2^2+1
6	2^2+2
7	2^2+3
8	2^3
9	2^3+1
10	2^3+2
11	2^3+3
12	2^3+4
13	2^3+5
14	2^3+6
15	2^3+7
16	2^4

Every number **n** can be written as $2^m + p$ where $p < 2^m$

Divide **n** in half: $2^{m-1} + p/2$
If p is not even, give away ceiling of **n/2**

New number **n'** can be written as $2^{m-1} + p'$ where $p' < 2^{m-1}$

For **n** slices how many groups of friends?

n	*k = number of groups*
1	2^0
2	2^1
3	2^1+1
4	2^2
5	2^2+1
6	2^2+2
7	2^2+3
8	2^3
9	2^3+1
10	2^3+2
11	2^3+3
12	2^3+4
13	2^3+5
14	2^3+6
15	2^3+7
16	2^4

New number **n'** can be written as $2^{m-1} + p'$ where $p' < 2^{m-1}$

Divide **n'** in half: $2^{m-1} + p/2$
If p is not even, give away ceiling of **n'/2**

New number of slices can be written as $2^{m-2} + p''$ where $p'' < 2^{m-2}$

For **n** slices how many groups of friends?

n	*k = number of groups*
1	2^0
2	2^1
3	2^1+1
4	2^2
5	2^2+1
6	2^2+2
7	2^2+3
8	2^3
9	2^3+1
10	2^3+2
11	2^3+3
12	2^3+4
13	2^3+5
14	2^3+6
15	2^3+7
16	2^4

Every number **n** can be written as $2^m + p$ where $p < 2^m$

We can divide in half m times to get 1 slice

$$2^{m-m} + p$$

where $p < 2^{m-m} = 1$
so $p = 0$

For **n** slices how many groups of friends?

k = number of groups

n	
1	2^0
2	2^1
3	2^1+1
4	2^2
5	2^2+1
6	2^2+2
7	2^2+3
8	2^3
9	2^3+1
10	2^3+2
11	2^3+3
12	2^3+4
13	2^3+5
14	2^3+6
15	2^3+7
16	2^4

Every number **n** can be written as $2^m + p$ where $p < 2^m$

We can divide in half ***m*** times to have 1 slice

$$k = m = \text{floor}(\log_2 n)$$

For **n** slices how many groups of friends?

k = number of groups

For **n** slices how many groups of friends if they ask for two-thirds each time?

k = number of groups

For **n** slices how many groups of friends if they ask for two-thirds each time?

k = number of groups

After each group we will be left with one-third of the pizza we had

Divide **n** by 3 multiple times…

For **n** slices how many groups of friends if they ask for two-thirds each time?

n	
1	3^0
2	2×3^0
3	3^1
4	$3^1 + 1$
5	$3^1 + 2$
6	2×3^1
7	$2 \times 3^1 + 1$
8	$2 \times 3^1 + 2$
9	3^2
10	$3^2 + 1$
11	$3^2 + 2$
12	$3^2 + 3$
13	$3^2 + 4$
14	$3^2 + 5$
15	$3^2 + 6$
16	$3^2 + 7$

Every number **n** can be written as $p3^m + q$ where $q < 3^m$ & $p < 3$

For **n** slices how many groups of friends if they ask for two-thirds each time?

n	
1	3^0
2	2×3^0
3	3^1
4	$3^1 + 1$
5	$3^1 + 2$
6	2×3^1
7	$2 \times 3^1 + 1$
8	$2 \times 3^1 + 2$
9	3^2
10	$3^2 + 1$
11	$3^2 + 2$
12	$3^2 + 3$
13	$3^2 + 4$
14	$3^2 + 5$
15	$3^2 + 6$
16	$3^2 + 7$

Every number **n** can be written as $p3^m + q$ where $q < 3^m$ & $p < 3$

We can divide **n** by three m times

$$p3^{m-m} + q3^{-m}$$
$$= p + 0 < 3$$

e.g. **n** = 6

For **n** slices how many groups of friends if they ask for two-thirds each time?

n	
1	3^0
2	2×3^0
3	3^1
4	$3^1 + 1$
5	$3^1 + 2$
6	2×3^1
7	$2 \times 3^1 + 1$
8	$2 \times 3^1 + 2$
9	3^2
10	$3^2 + 1$
11	$3^2 + 2$
12	$3^2 + 3$
13	$3^2 + 4$
14	$3^2 + 5$
15	$3^2 + 6$
16	$3^2 + 7$

Every number **n** can be written as $p3^m + q$ where $q < 3^m$ & $p < 3$

We can divide **n** by three ***m*** times

$$p3^{m-m} + q3^{-m}$$
$$= p + 0 < 3$$

We can end up with two slices at the end!

For **n** slices how many groups of friends if they ask for two-thirds each time?

n	
1	3^0
2	2×3^0
3	3^1
4	$3^1 + 1$
5	$3^1 + 2$
6	2×3^1
7	$2 \times 3^1 + 1$
8	$2 \times 3^1 + 2$
9	3^2
10	$3^2 + 1$
11	$3^2 + 2$
12	$3^2 + 3$
13	$3^2 + 4$
14	$3^2 + 5$
15	$3^2 + 6$
16	$3^2 + 7$

Every number **n** can be written as $p3^m + q$ where $q < 3^m$ & $p < 3$

We can divide **n** by three ***m*** times

$$k = m = floor(\log_3 n)$$

For **n** slices how many groups of friends?

$floor(log_2 n)$

For **n** slices how many groups of friends if they ask for two-thirds each time?

$floor(log_3 n)$

For **n** slices how many groups of friends?

$\boldsymbol{floor(\log_2 n)}$

For **n** slices how many groups of friends if they ask for two-thirds each time?

$\boldsymbol{floor(\log_3 n)}$

What are the “Big O” classes for these functions?

$$O(\log_3 n) = O(\log_2 n) = O(\log n)$$

What is the "Big O" class for this function?

$$O(log_3 n) = O(log_2 n)$$

$$log_3 n = log_2 n \, / \, log_2 3$$

Constant

$$log_3 n = c \times log_2 n$$

$c = 1 / \log_2 3$

$$O(log_3 n) = O(log_2 n)$$

All multiplying constants are "set equal to 1"

What is the “Big O” class for this function?

$$O(log_3 n) = O(log_2 n)$$

$$log_3 n = log_2 n \;/\; log_2 3$$

Constant

$$log_3 n = c \times log_2 n$$

$c = 1 / \log_2 3$

$$O(log_3 n) = O(log_2 n)$$

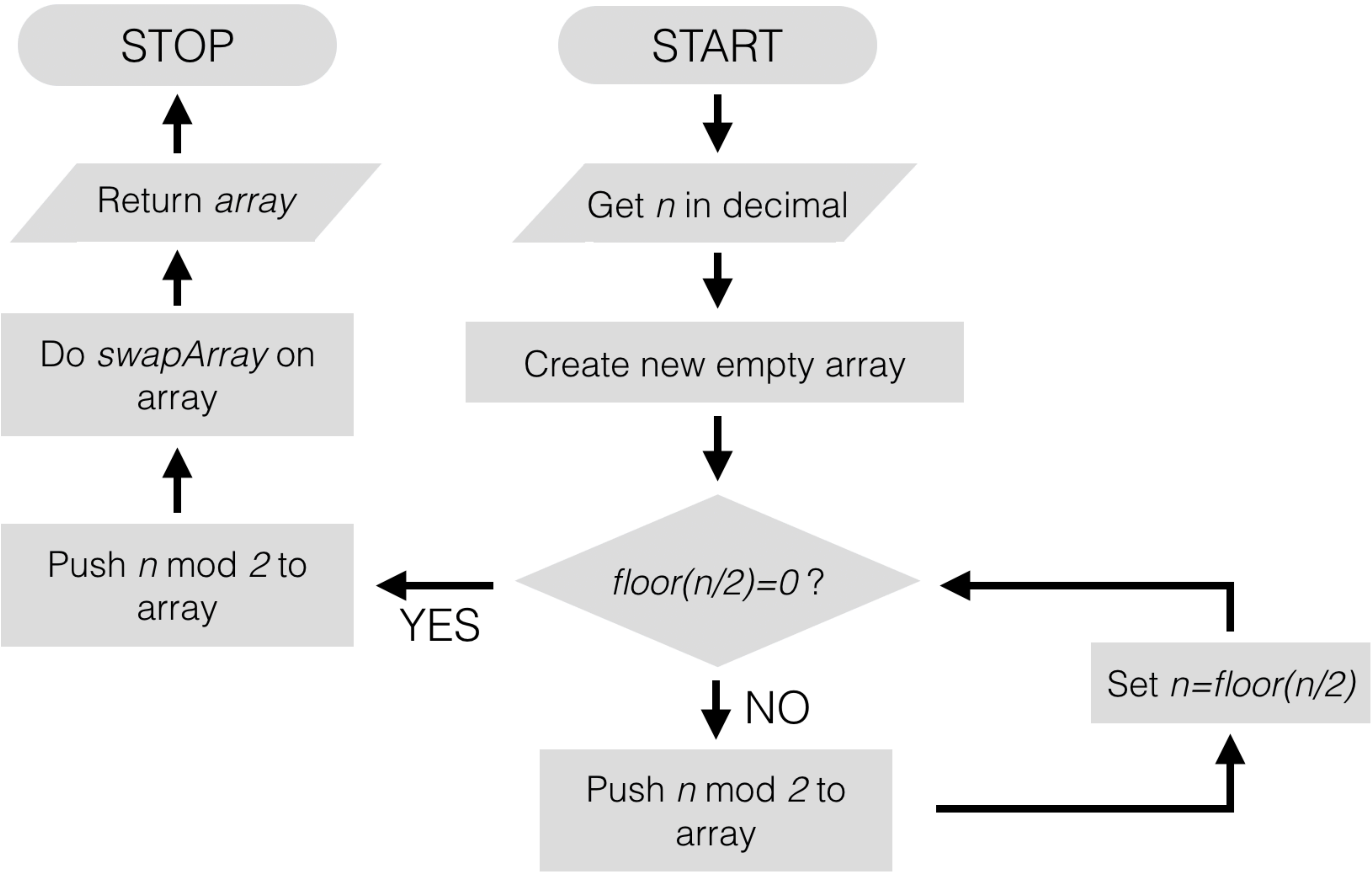
Intuitively: giving away 1/2 or 2/3 of pizza, the number of groups will change by constant (~0.63)

For **n** slices how many groups of friends?

floor($\log_2 n$)

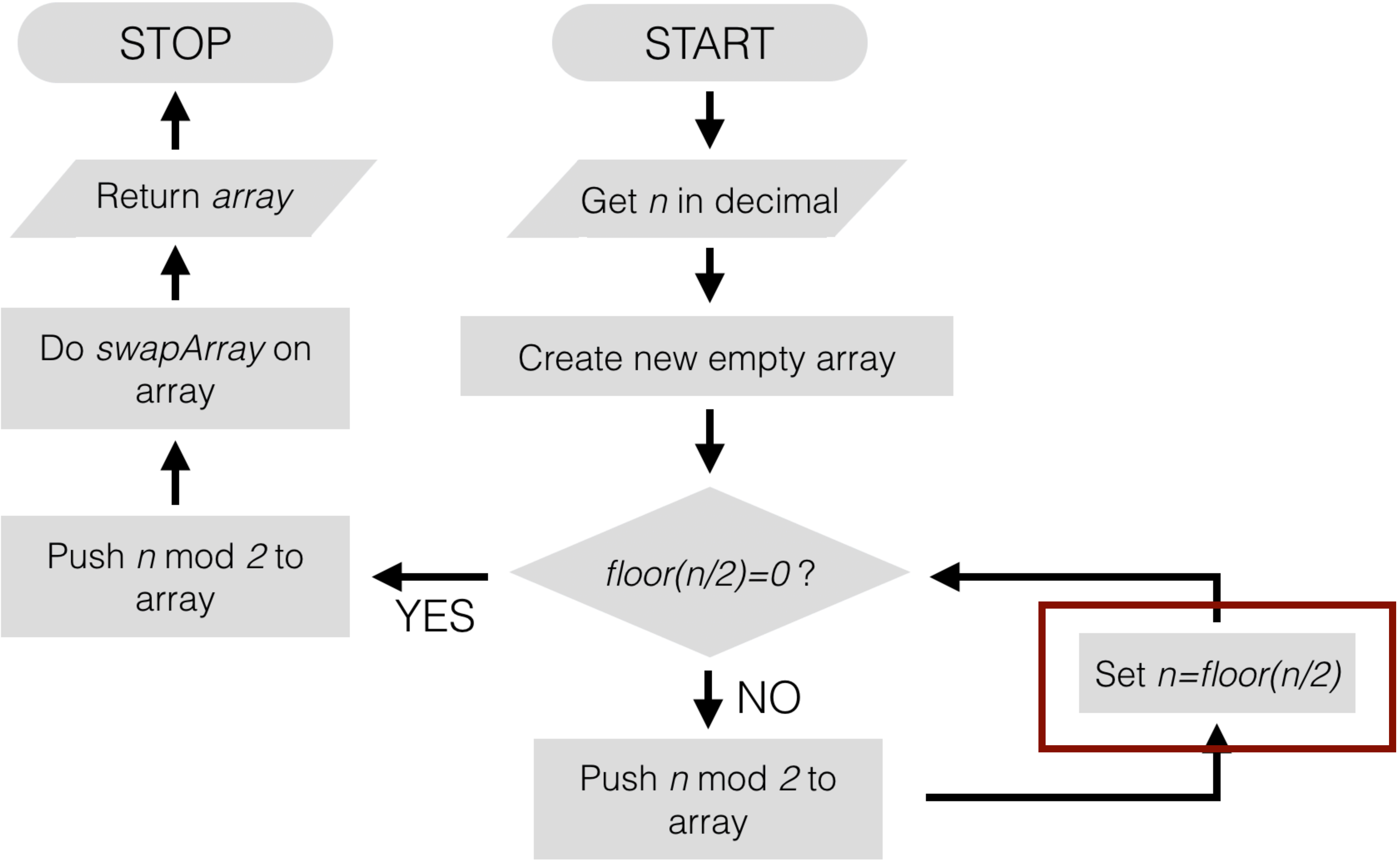
Where have we seen a process of repeatedly dividing in half?

Worksheet 1

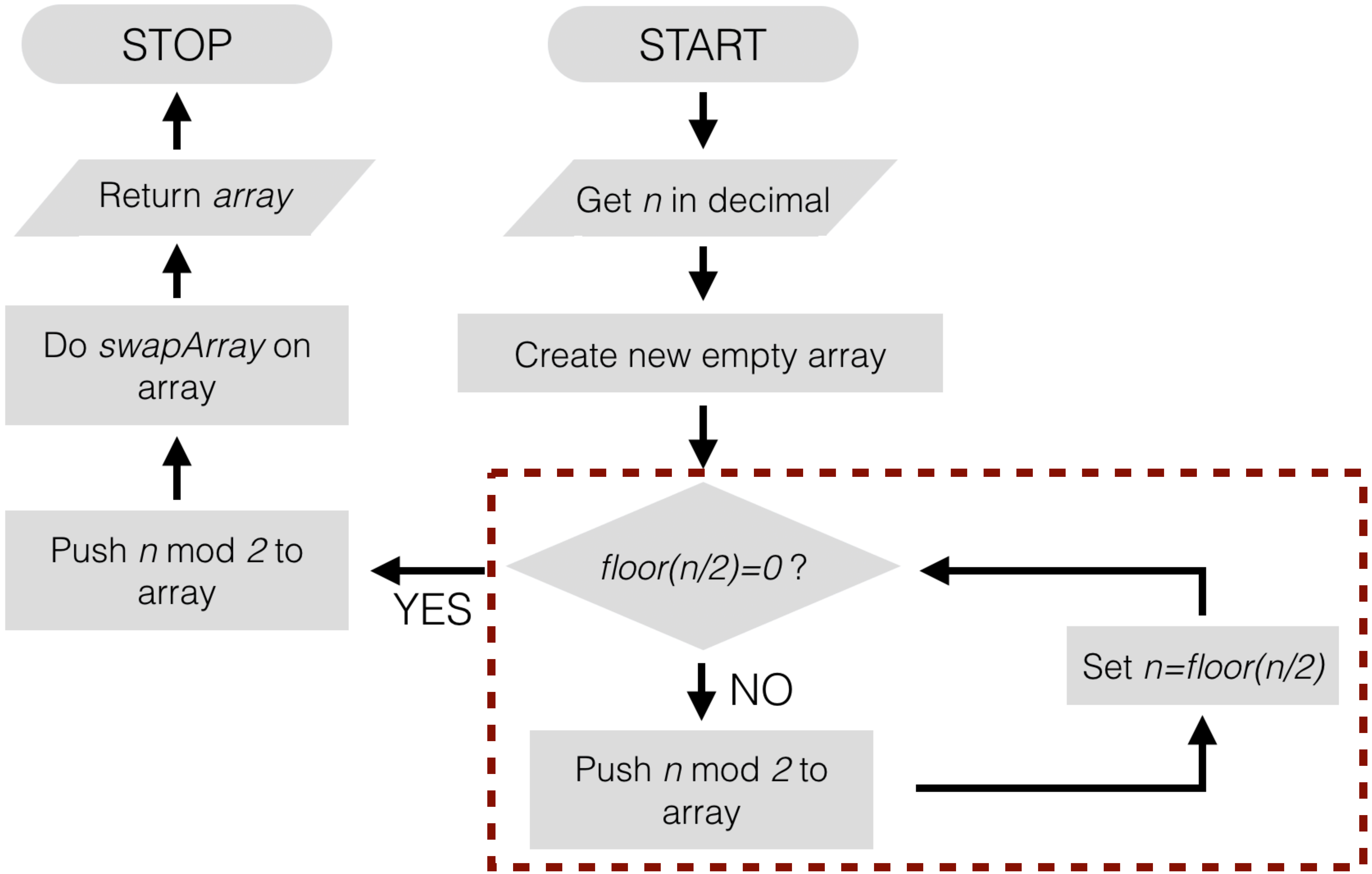


STOP
START
Return *array*
Get *n* in decimal
Do *swapArray* on array
Create new empty array
Push *n* mod *2* to array
YES
floor(n/2)=0 ?
Set *n=floor(n/2)*
NO
Push *n* mod *2* to array

Worksheet 1


STOP
Return *array*
Do *swapArray* on array
Push *n* mod *2* to array
YES
START
Get *n* in decimal
Create new empty array
floor(n/2)=0 ?
NO
Push *n* mod *2* to array
Set *n=floor(n/2)*

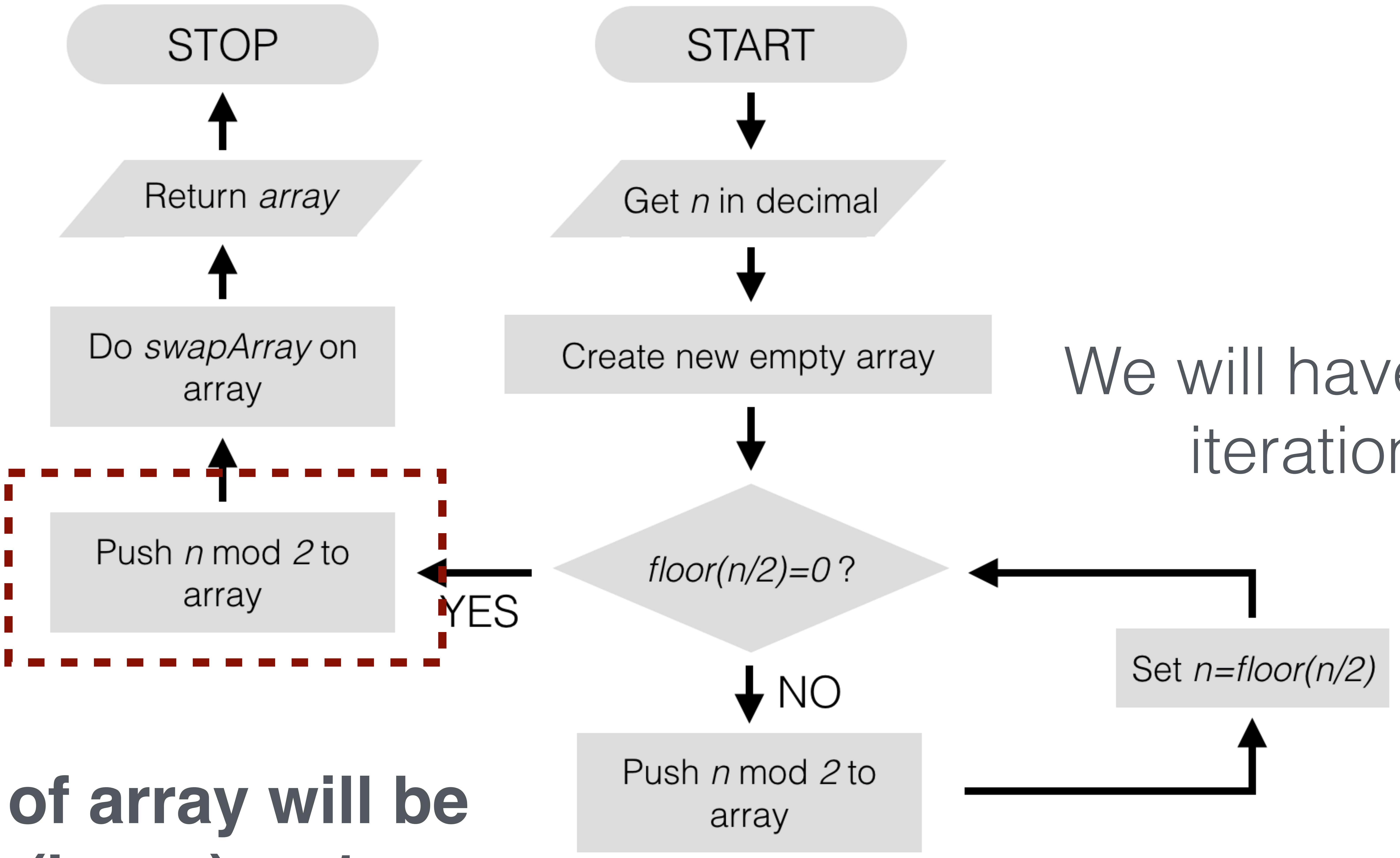
Worksheet 1



We will have **floor($\log_2$n)** iterations of loop

Since **floor(1/2) = 0** for “last slice of pizza”

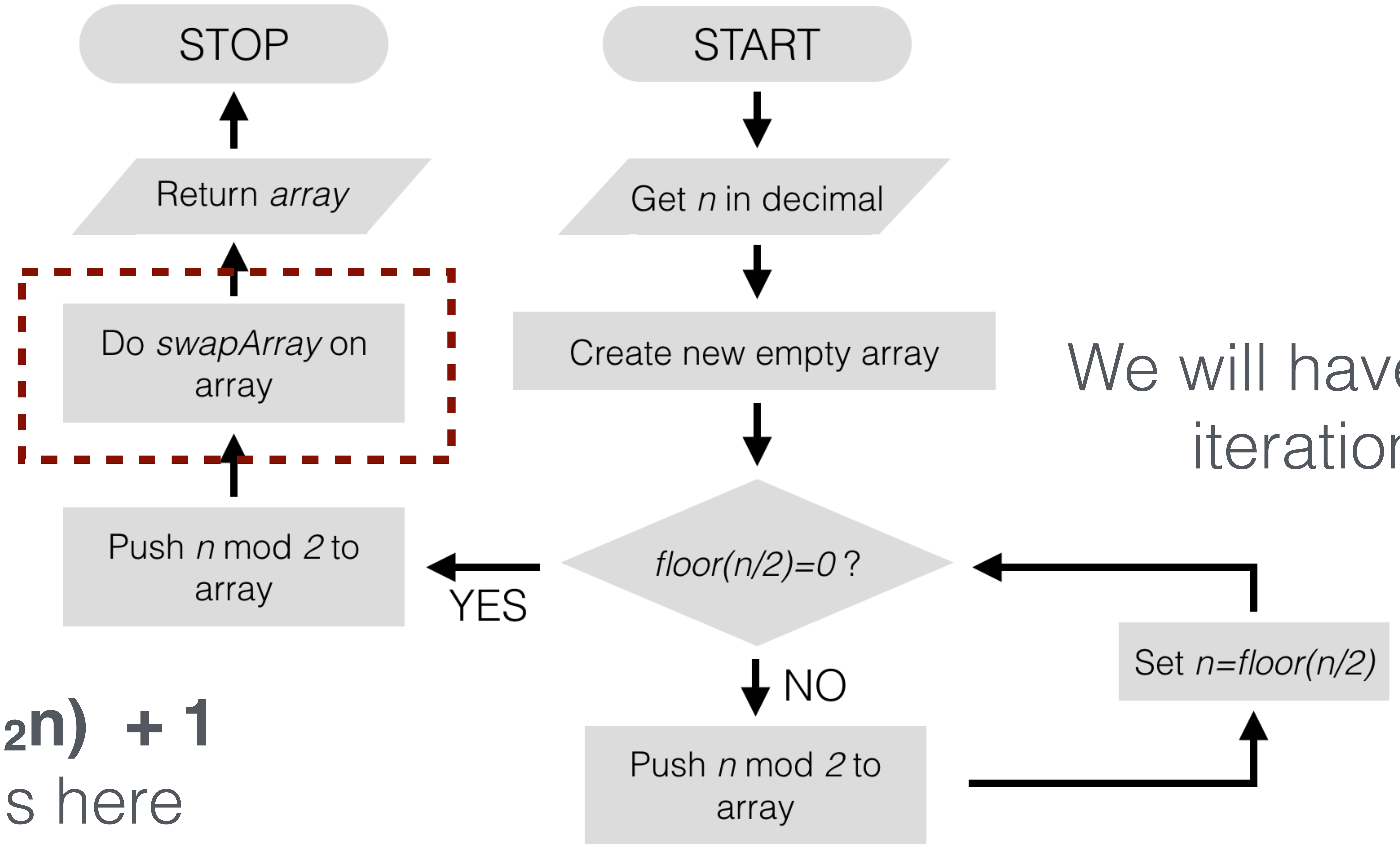
Worksheet 1



We will have **floor(log$_2$n)** iterations of loop

Length of array will be floor(log$_2$n) + 1

Worksheet 1

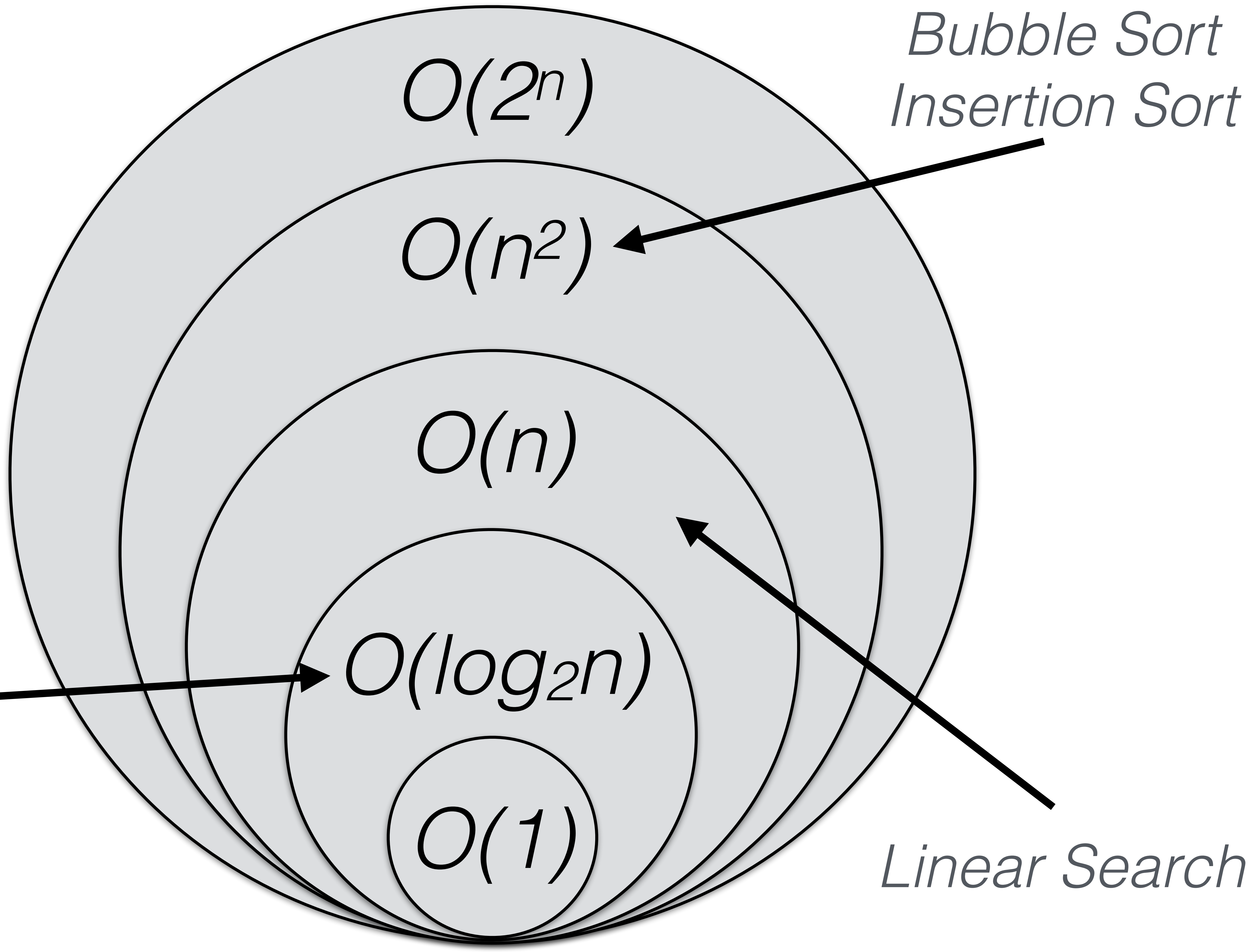


We will have **floor($\log_2$n)** iterations of loop

floor($\log_2$n) + 1 iterations here

Total: **2floor($\log_2$n) + 2** operations

O(log n)



Converting from decimal to binary

Second half of lecture: Searching in Logarithmic Time

Admin

- Sixth quiz available today at 4pm
  - Fifth quiz deadline next Monday at 4pm

- Sudoku assignment
  - Cut-off date is **15th March 4pm**
  - No more help with Sudoku assignment in Virtual Contact Hours from now on
  - Book me for office hours if you need help

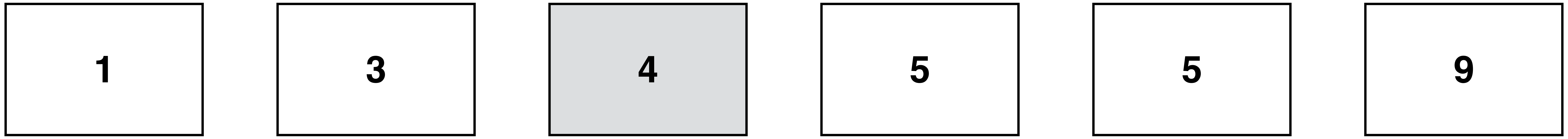
- Primes assignment
  - Worksheet 6 made **available TODAY at 11am**
  - Only involves programming tasks and submission of single js file
  - This week's VCH devoted to this assignment
  - Deadline **15th March 4pm**
  - Cut-off date **29th March 4pm**

Today

Binary Search

What’s the point in sorting something if it’s not going to be useful

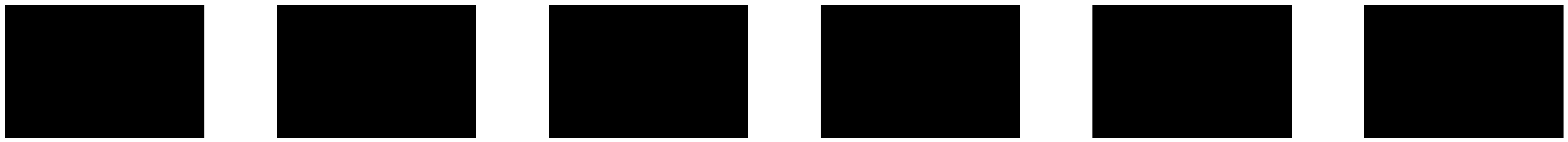
Start with sorted array



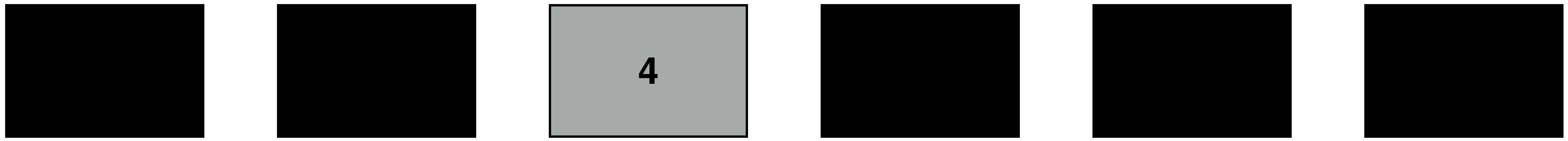
Every element either has:

1. All smaller or equal values to the left
2. All larger or equal values to the right

Search for value 5

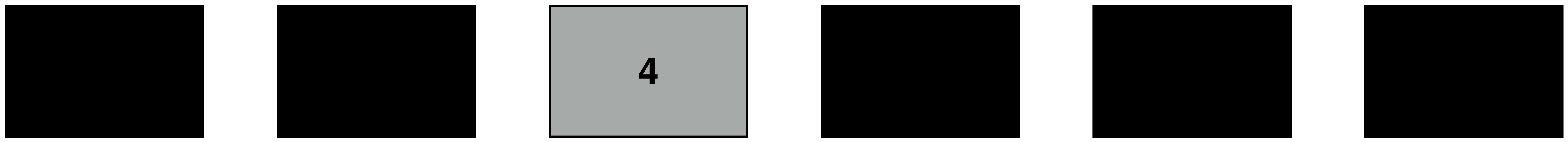


Search for value 5



Randomly pick element

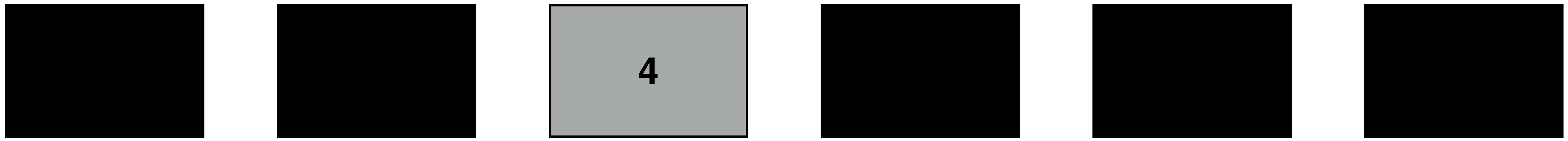
Search for value 5



Randomly pick element

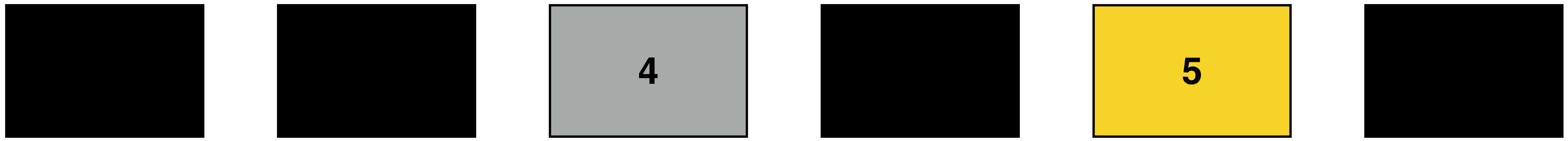
If it is not 5, then it must be to the right

Search for value 5



Randomly pick element to the right of previous element

Search for value 5



Randomly pick element to the right of previous element (right sub-array)

Found it in 2 look operations

Linear search requires 5

Binary Search

Replace the random picks with looking at mid-point element of each sub-array

Binary Search

Mid-point of an array:

floor of (left + right)/2

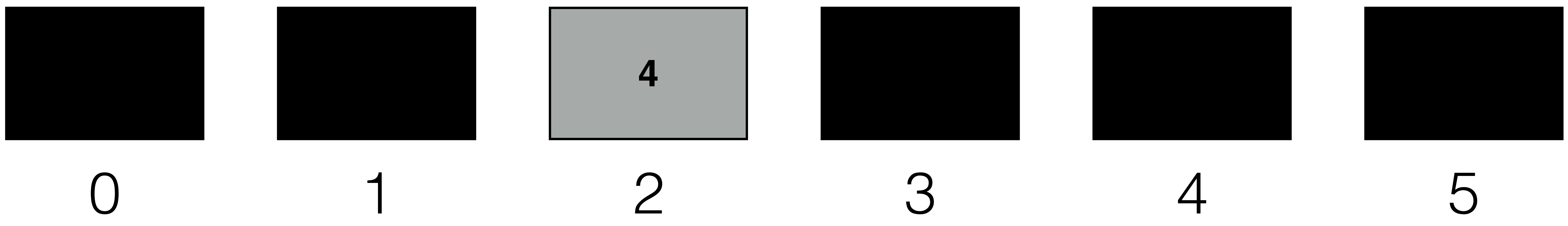
Binary Search

Mid-point of an array

floor of (left + right)/2

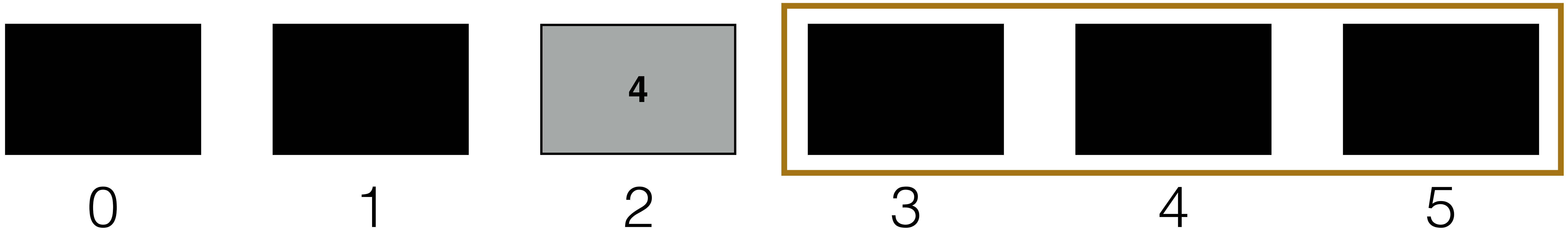
Search for value 5

Search for value 5



Mid-point

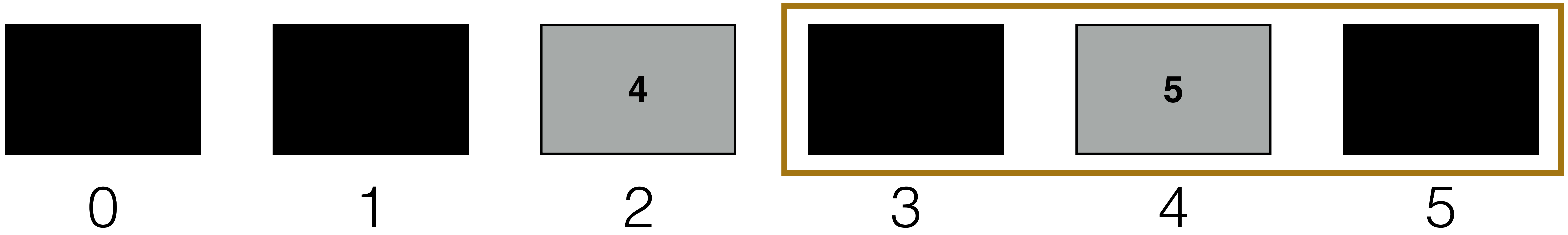
Search for value 5



Pick mid-point of sub-array to the right

floor of (3 + 5)/2

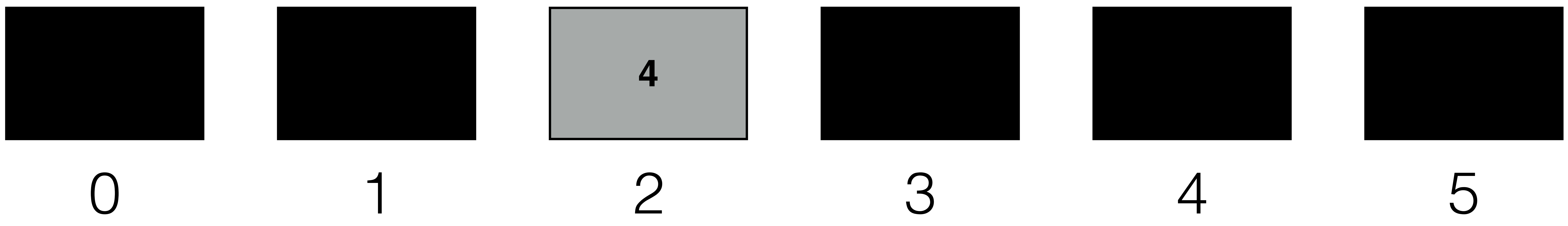
Search for value 5



Found the value

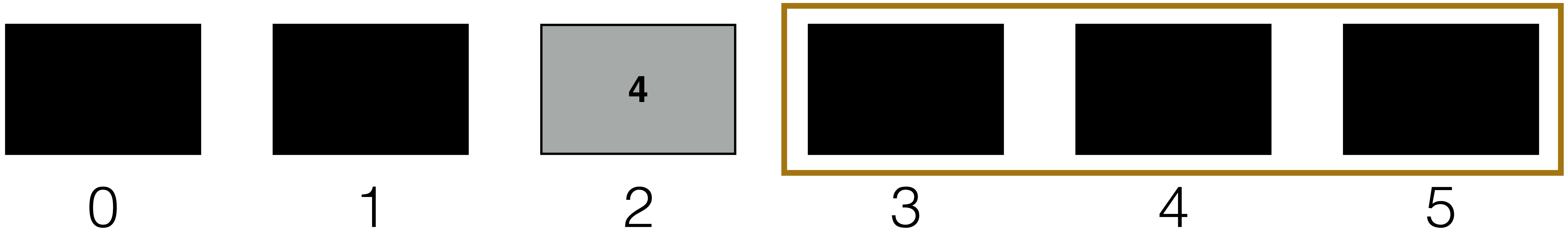
Search for value 8

Search for value 8



Mid-point

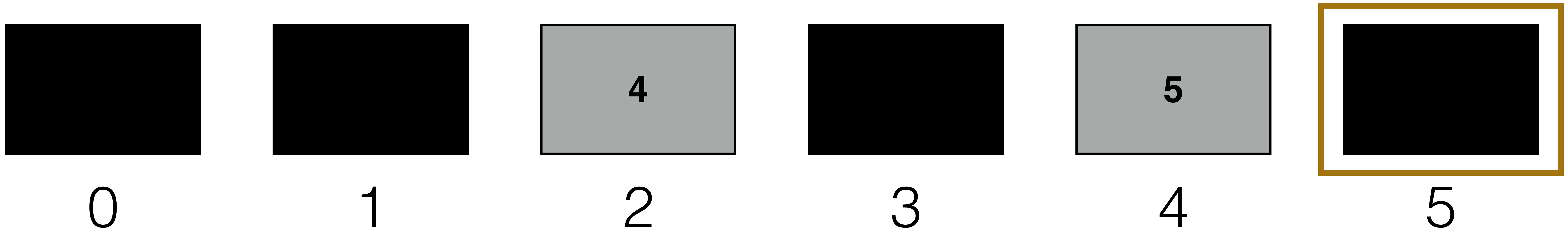
Search for value 8



Pick mid-point of sub-array to the right

floor of (3 + 5)/2

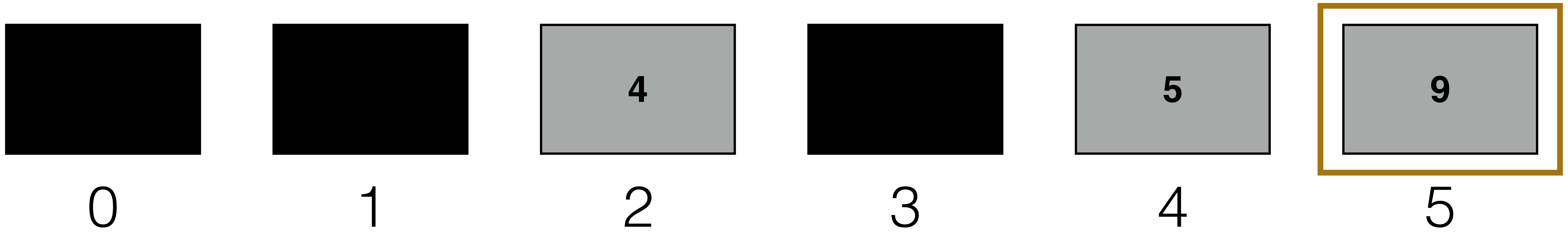
Search for value 8



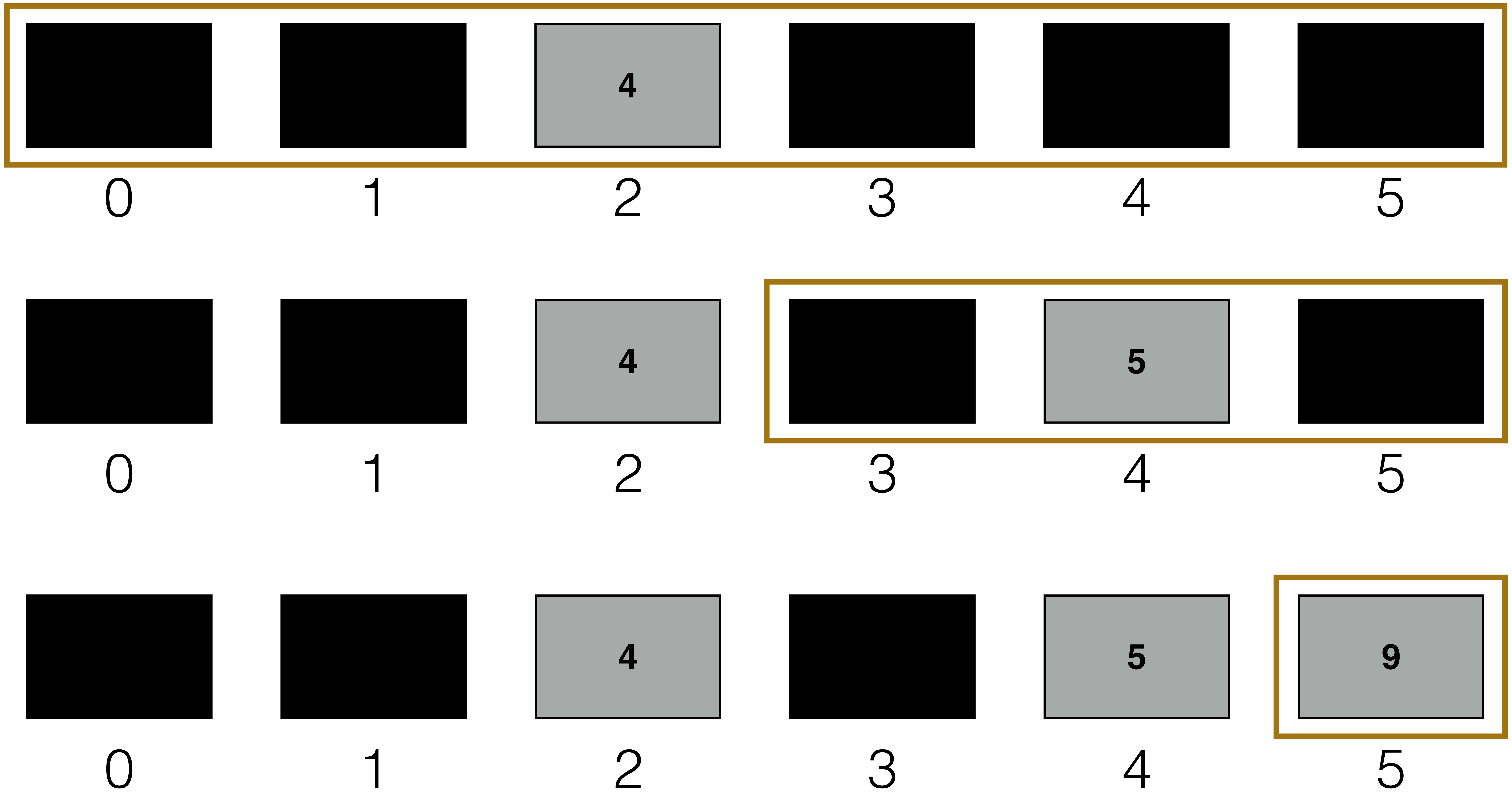
Need to consider sub-array to the right

Mid-point is (5+5)/2

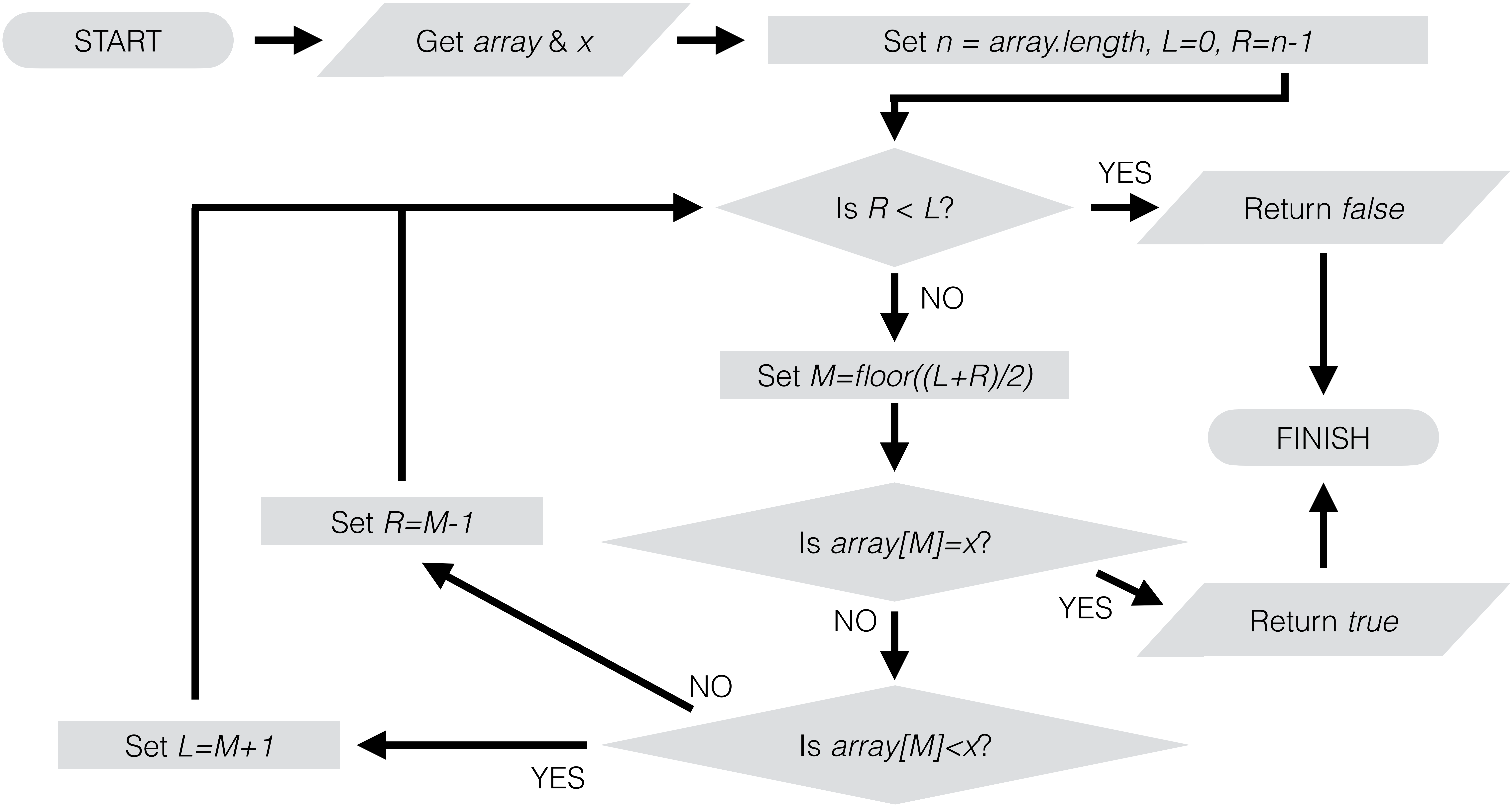
Search for value 8

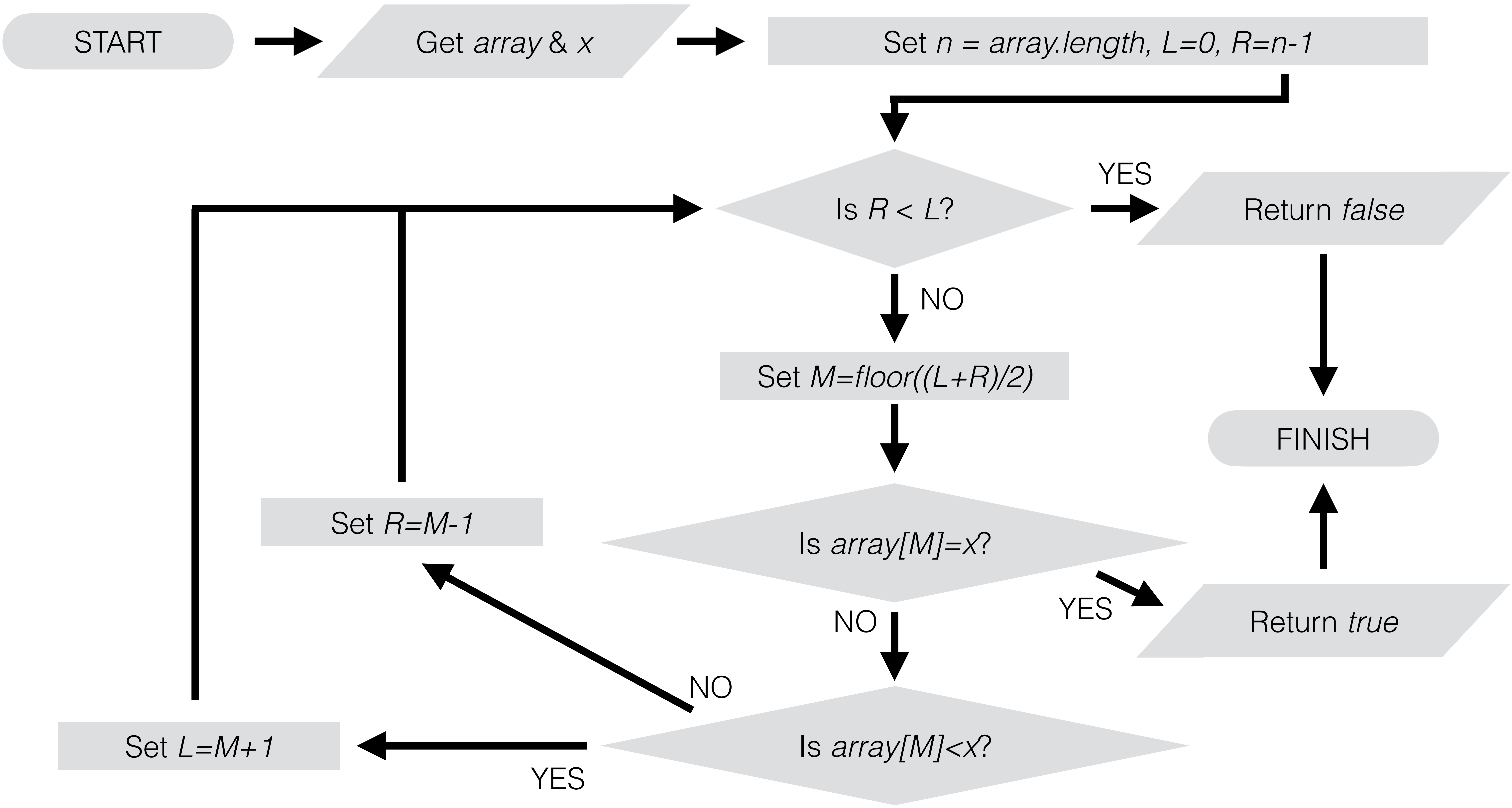


Value too large


4
0
1
2
3
4
5
4
5
0
1
2
3
4
5
4
5
9
0
1
2
3
4
5

Binary Search





Let’s implement this in JavaScript

Research task for Review Seminar

Until around 2006 nearly all Binary Search implementations in language libraries (e.g. Java) were broken

Find out why, and how you can have a better implementation of the algorithm

Binary Search

What is the best case input array and value?

What is the worst case input array and value?

Binary Search

What is the best-case input array and value?

Value is stored at mid-point of array

What is the worst-case input array and value?

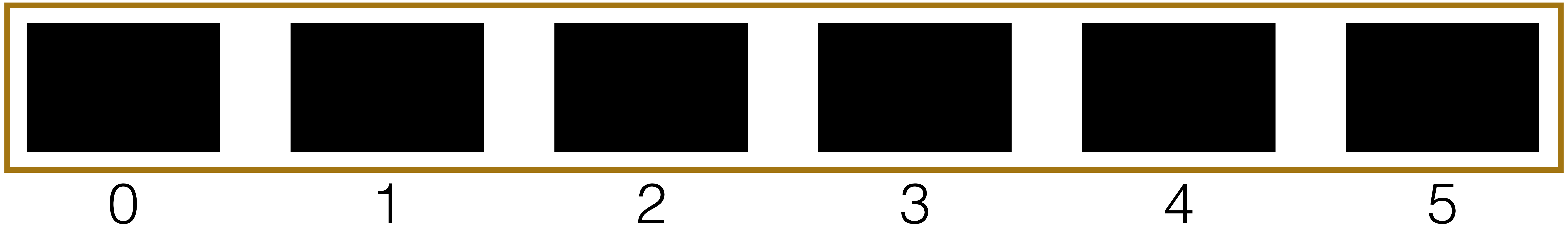
Value is stored at end of array
Value is not stored in array

Binary Search

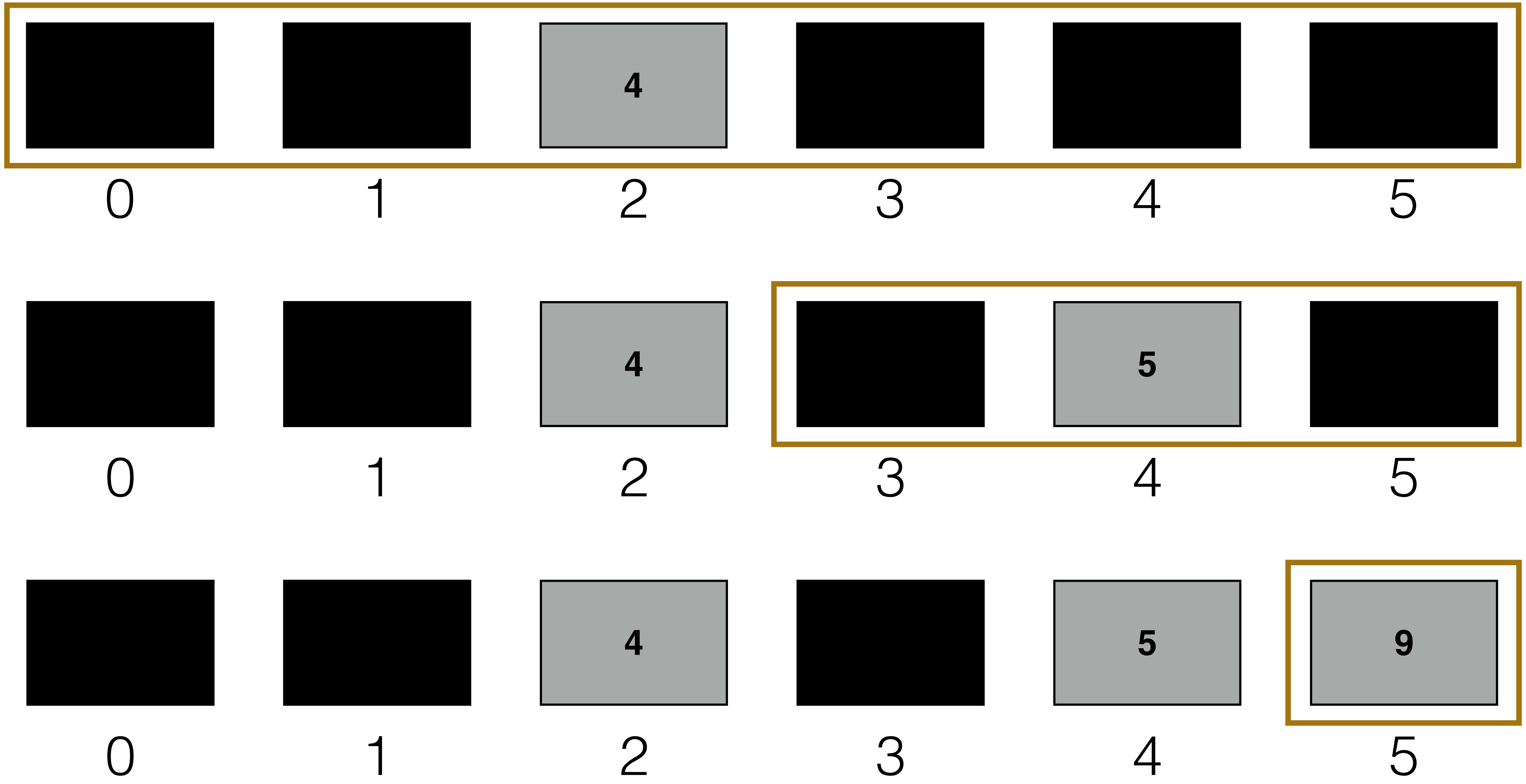
The worst-case time complexity is $O(log_2 n)$

Why?

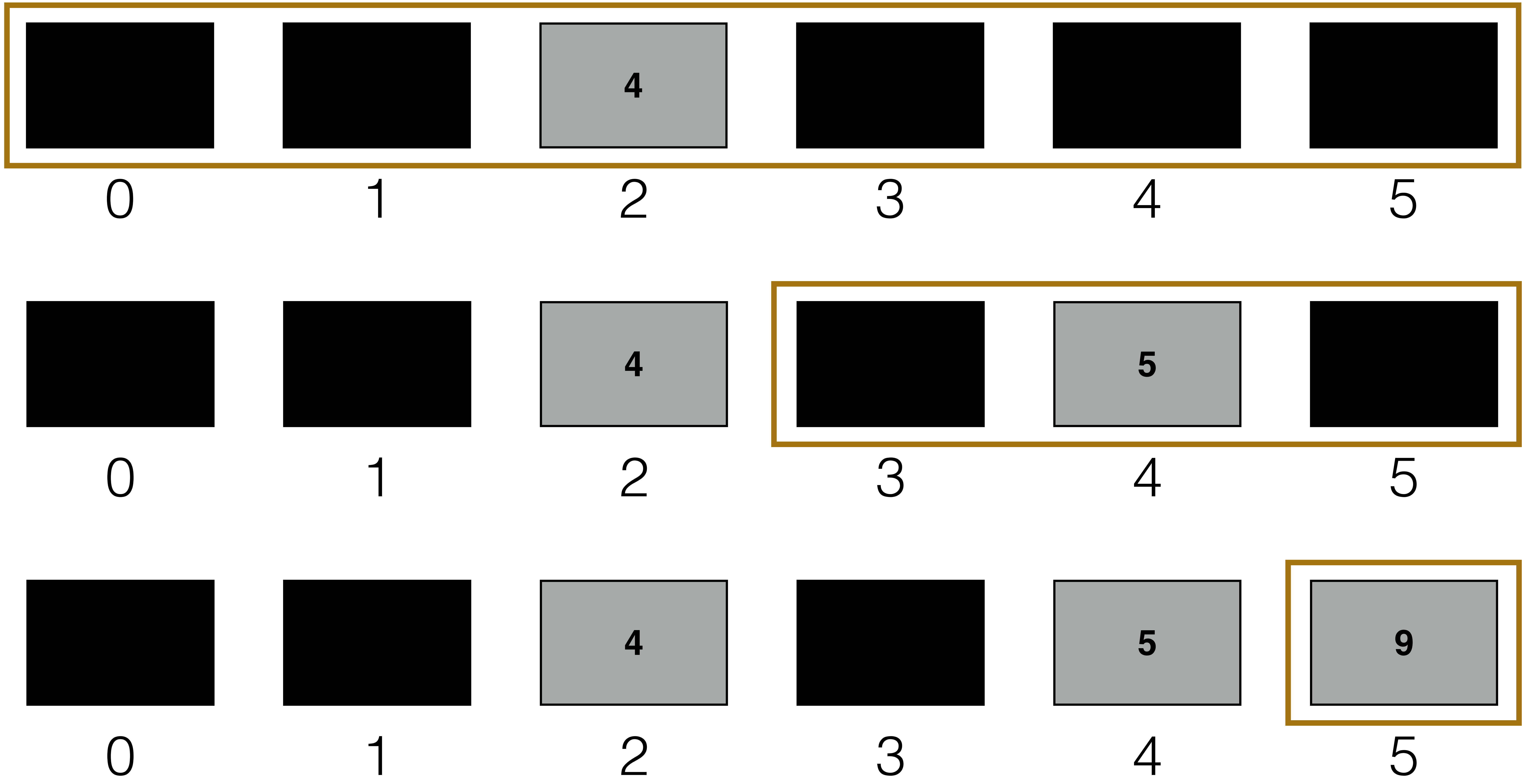
Search for value 9



Search for value 9

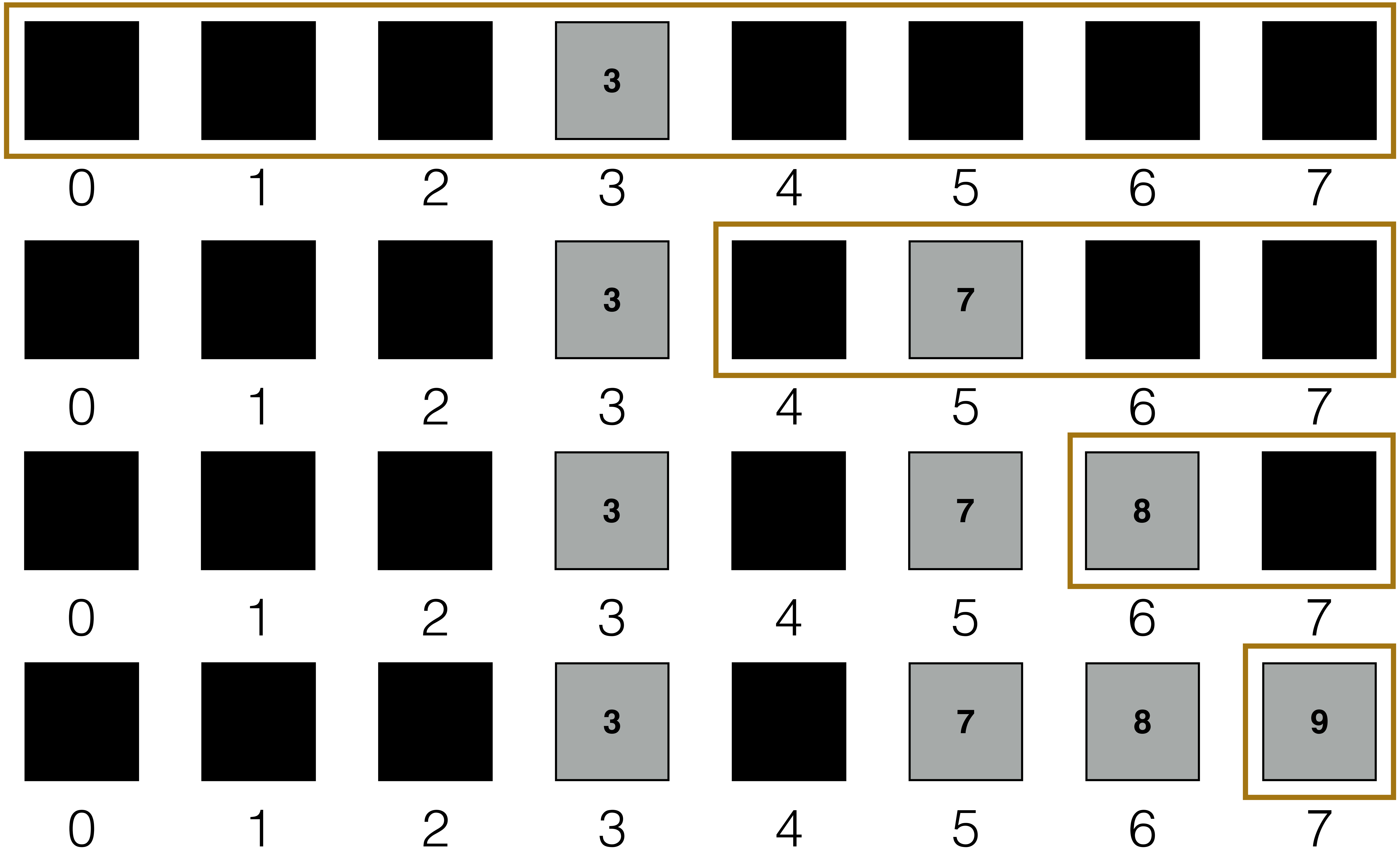


Search for value 9



In each iteration “halve” the array we consider until left with one element that we check

Search for value 9

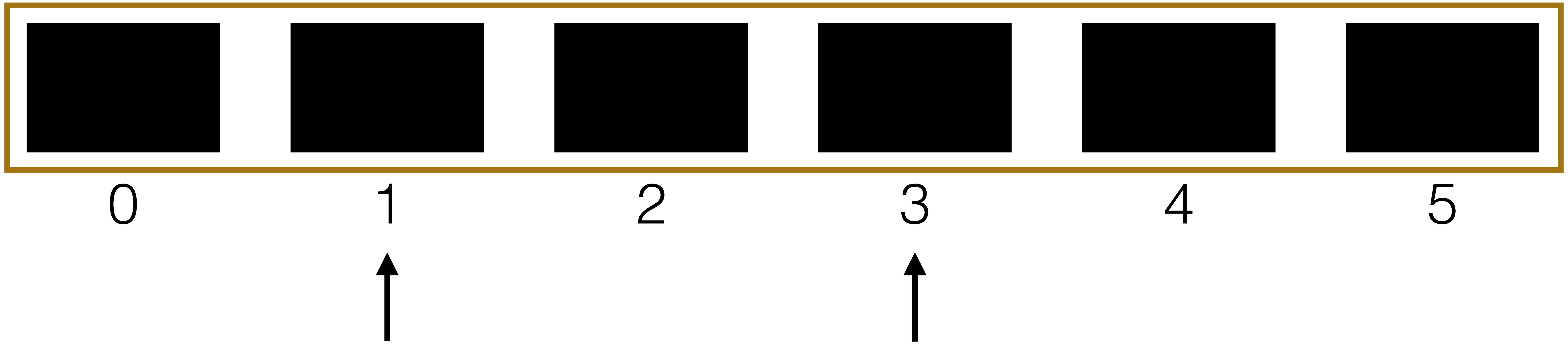


cf. Pizza problem: number of groups + 1

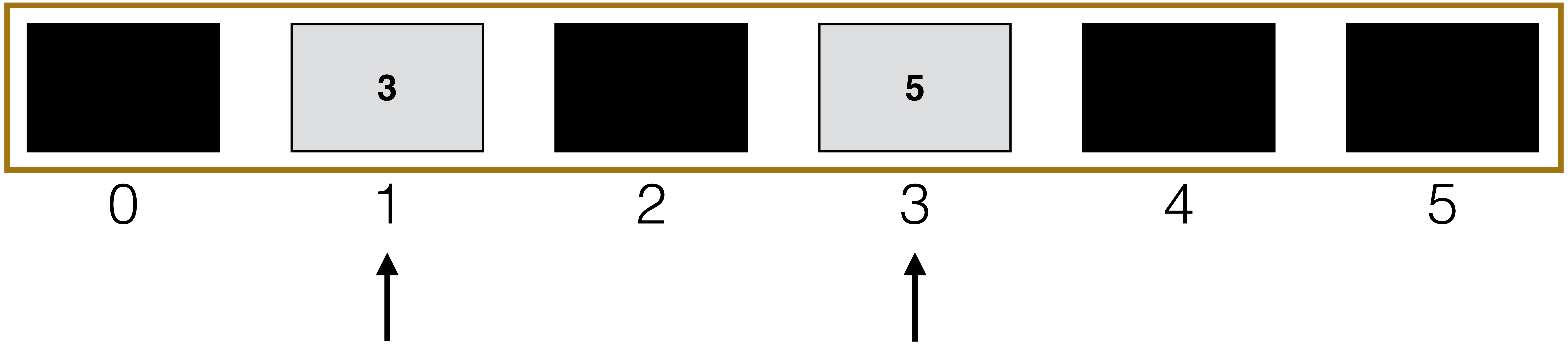
Ternary Search?

Instead of inspecting a single mid-point -
- why not look at two?

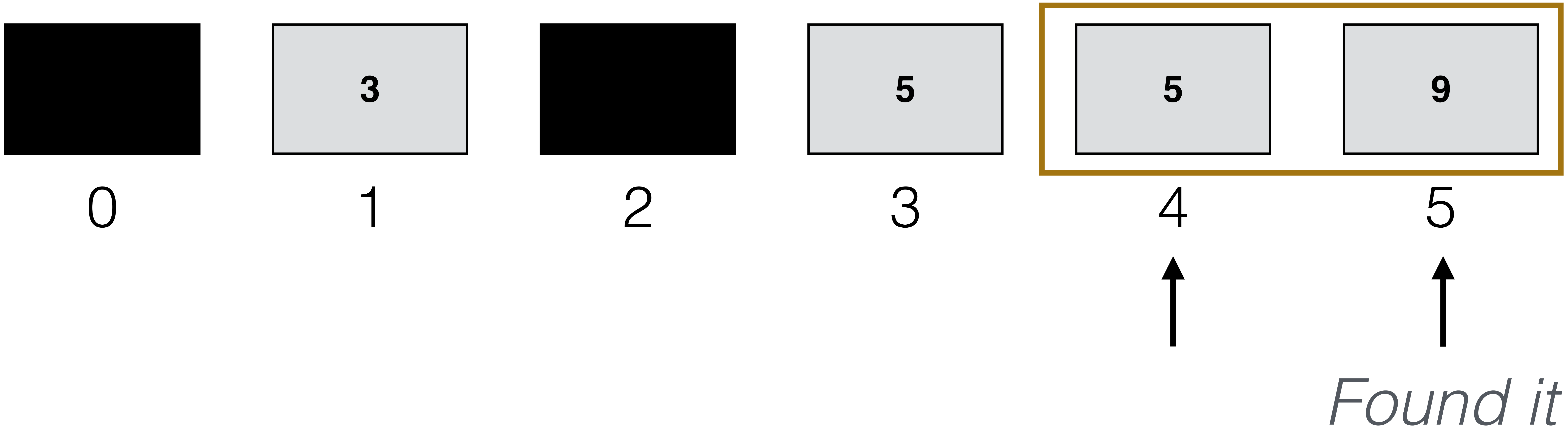
Search for value 9



Search for value 9



Search for value 9



```javascript
function ternarySearch(arr, item) {

    var left = 0;
    var right = arr.length - 1;

    while (left<=right) {

        var fir = left + Math.floor((right - left) / 3);
        var sec = left + Math.floor(2 * (right - left) / 3);

        if ((arr[fir] == item) || (arr[sec] == item)) {
            if (arr[fir] == item) {
                return fir;
            } else {
                return sec;
            }
        } else if (item < arr[fir]) {
            right = fir - 1;
        } else if (arr[sec] < item) {
            left = sec + 1;
        } else {
            left = fir + 1;
            right = sec - 1;
        }

    }

    return false;
}
```

Ternary Search

What is the worst-case input?

What is the worst-case time complexity?

Ternary Search

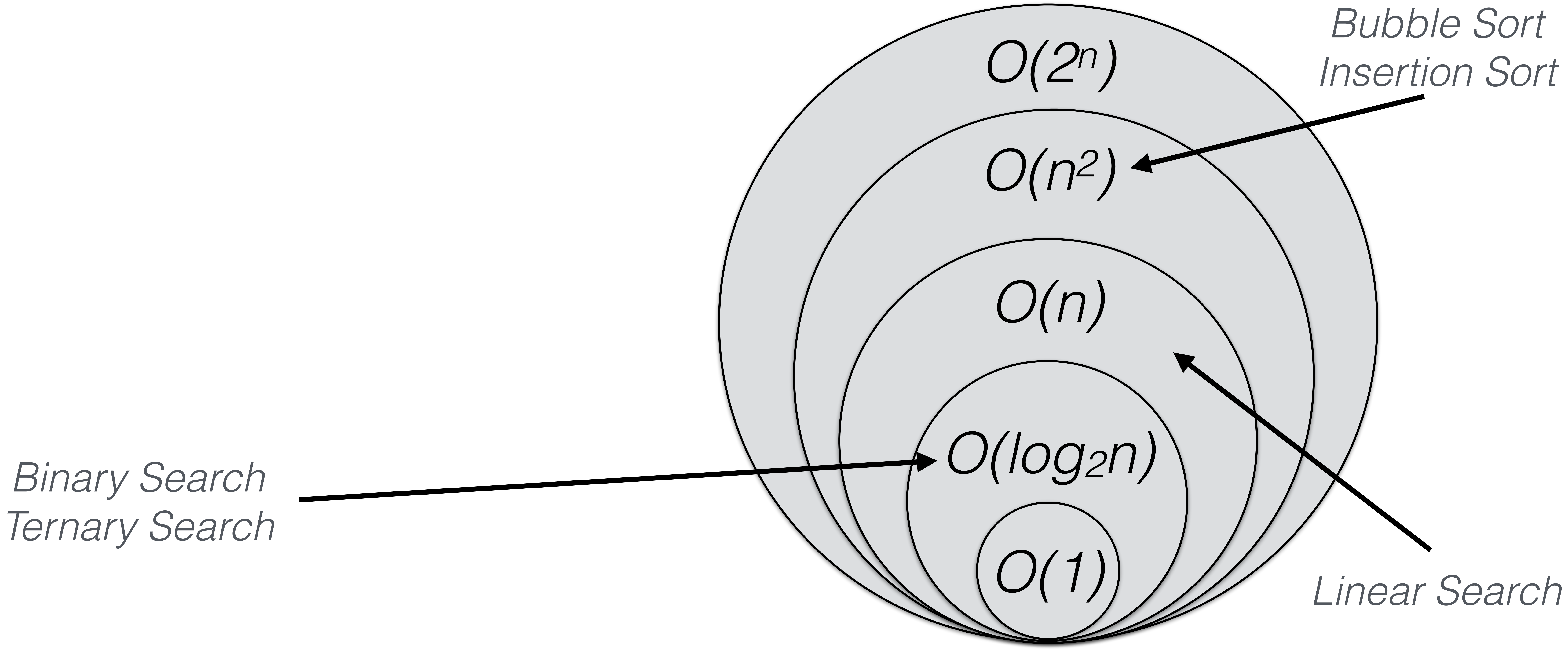
What is the worst-case input?

Same as in Binary Search

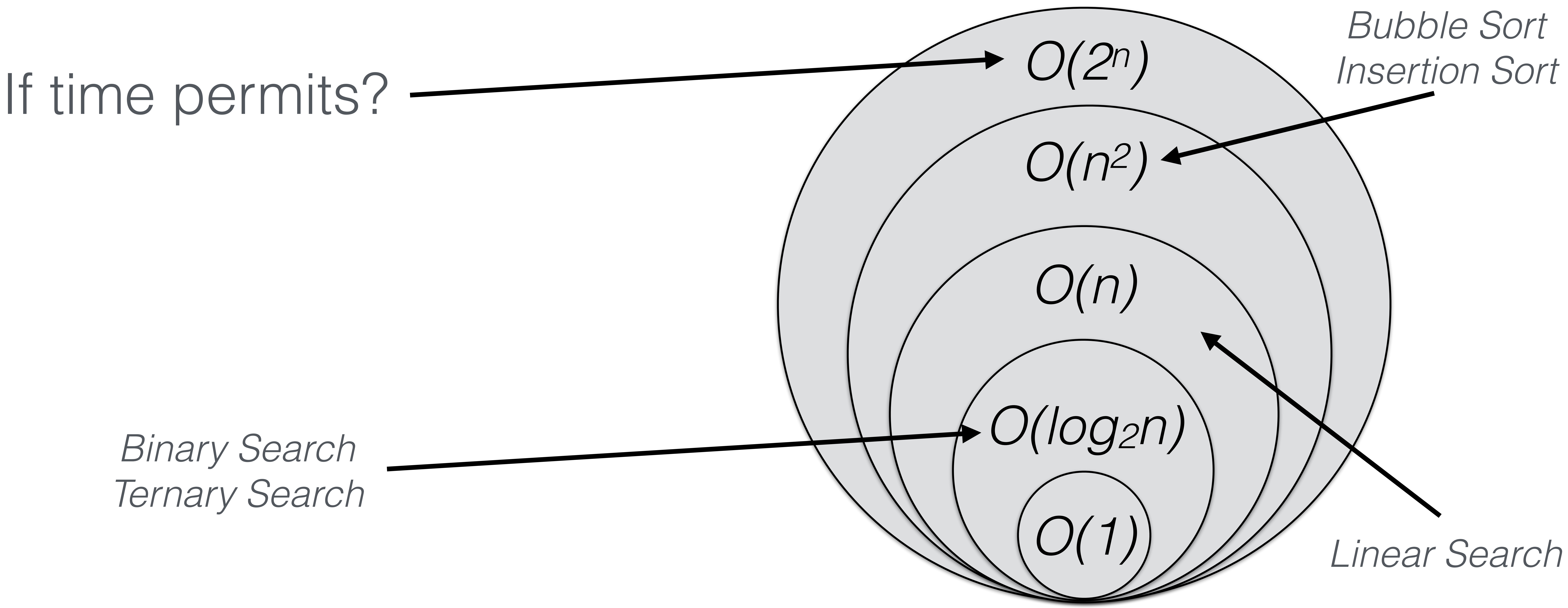
What is the worst-case time complexity?

$O(\log_3 n) = O(\log n)$

We are dividing the array into three sub-arrays



O(2^n)
O(n^2)
O(n)
O($\log_2 n$)
O(1)
Bubble Sort
Insertion Sort
Linear Search
Binary Search
Ternary Search

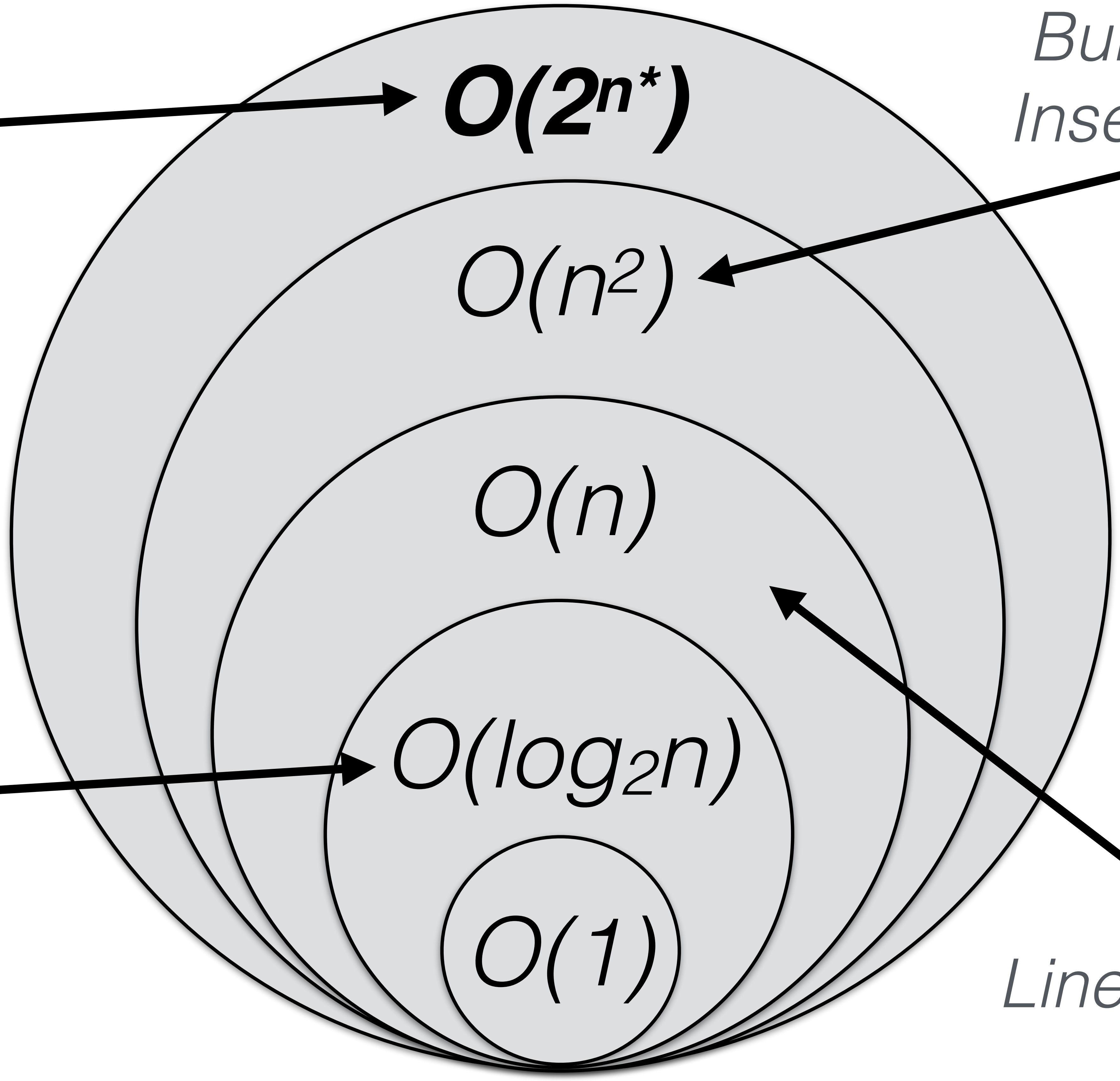


If time permits?
O(2^n)
Bubble Sort
Insertion Sort
O(n^2)
O(n)
O(log2n)
O(1)
Binary Search
Ternary Search
Linear Search

If time permits

$n^* = n + \log_2 n$

Binary Search
Ternary Search



Bubble Sort
Insertion Sort

Linear Search

Given an array of length *n* storing numbers, and a target value

[1, 2, 4, 6] and 10

[1, 3, 5, 8] and 10

Given subset of these numbers that sum up to give target value?

[4, 6]

[]

```javascript
function addOneBinary(binaryNumber) {
    var j = binaryNumber.length - 1;
    while (binaryNumber[j] > 0 && j >= 0){
        binaryNumber[j] = 0;
        j--;
    }
    if (j == -1) {
        binaryNumber.unshift(1);
    } else {
        binaryNumber[j] = 1;
    }
    return binaryNumber;
}
```

Given an array of length *n* storing numbers, and a target value

Is there combination of the numbers that sum up to give target value?

```javascript
function sumOfElements(array, targetNumber) {
    var subsetChoice = [];
    for (var i = 0; i < array.length; i++) {
        subsetChoice.push(0);
    }
    for (var i = 0; i < 2**(array.length); i++) {
        var total = 0;
        for (var j = 0; j < array.length; j++) {
            total = total + array[j]*subsetChoice[j];
        }
        if (total === targetNumber) {
            return subsetChoice;
        }
        addOneBinary(subsetChoice);
    }
    return [];

}
```

```javascript
function addOneBinary(binaryNumber) {
    var j = binaryNumber.length - 1;
    while (binaryNumber[j] > 0 && j >= 0){
        binaryNumber[j] = 0;
        j--;
    }
    if (j == -1) {
        binaryNumber.unshift(1);
    } else {
        binaryNumber[j] = 1;
    }
    return binaryNumber;
}
```

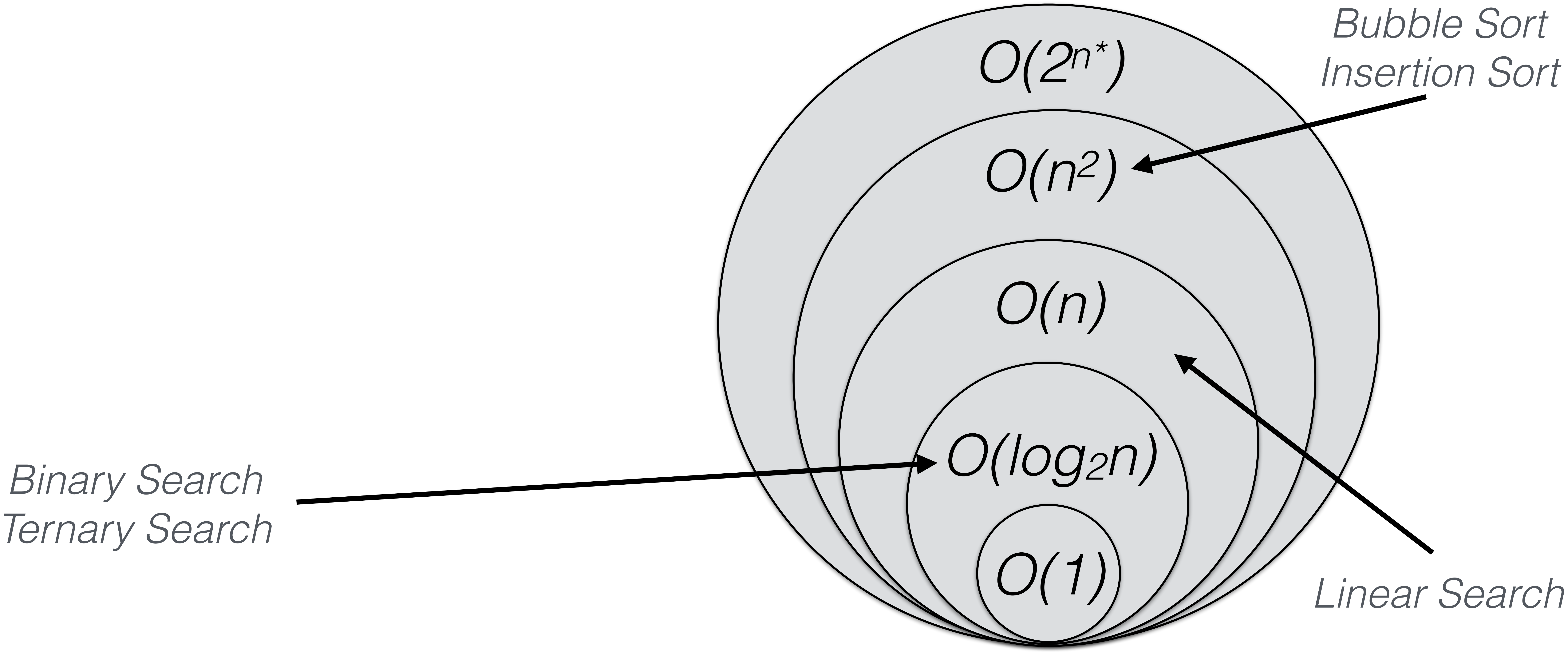
- Adds one to a bit-string array
- Worst-case $O(n)$ time where length of array is n

- Generates all possible bit-string arrays of length n
- Describes whether to include (1) numbers in combination or not (0)
- Worst-case $O(n2^n)$ time, which is $O(2^{n^*})$

$n^* = n + \log_2 n$

```javascript
function sumOfElements(array, targetNumber) {
    var subsetChoice = [];
    for (var i = 0; i < array.length; i++) {
        subsetChoice.push(0);
    }
    for (var i = 0; i < 2**(array.length); i++) {
        var total = 0;
        for (var j = 0; j < array.length; j++) {
            total = total + array[j]*subsetChoice[j];
        }
        if (total === targetNumber) {
            return subsetChoice;
        }
        addOneBinary(subsetChoice);
    }
    return [];
}
```

Review Seminar



Problem 6:

You’ve been given the task of helping to build a “pre-compiler” for a JavaScript teaching tool

This will conduct preliminary checks on code to look for syntax errors to avoid compile errors

Your part in this project is to write code that checks for bracketing errors, both {} and ()

Describe an algorithm and/or JavaScript implementation that flags an error when there is a bracketing error in the code