PSCS

Problem Solving for Computer Science

IS51021B/C

Goldsmiths Computing

March 22, 2021

Wk 10

Problem 7:

In Worksheet 3 you learnt about the Ceasar cipher

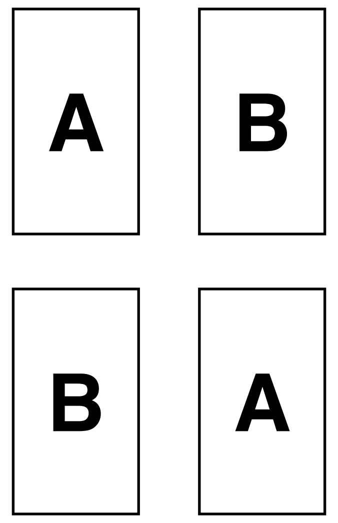
Letters in the alphabet are *cyclically* permuted

A more secure encryption scheme is to apply *any* permutation of the alphabet (26! many)

abcd

Can you find a ***recursive*** method to put all possible permutations of these four letters in an array?

Permutations of Two Letters A and B



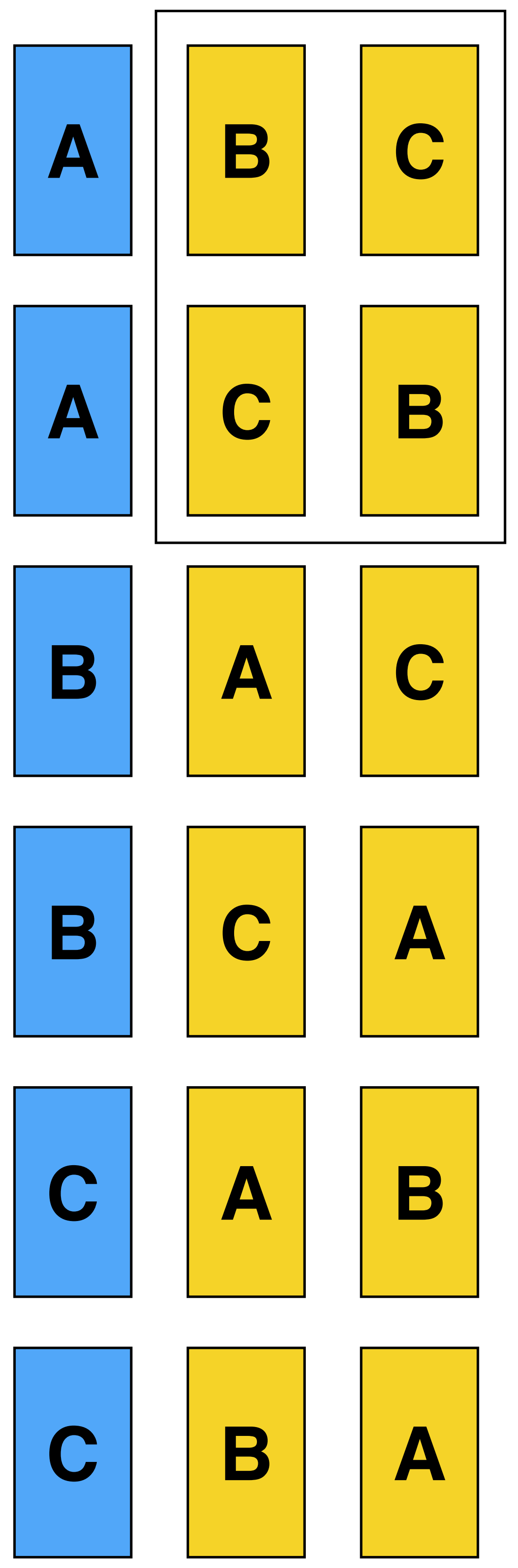
Permutations of Three Letters A, B and C

A	B	C
A	C	B
B	A	C
B	C	A
C	A	B
C	B	A

Permutations of Three Letters A, B and C

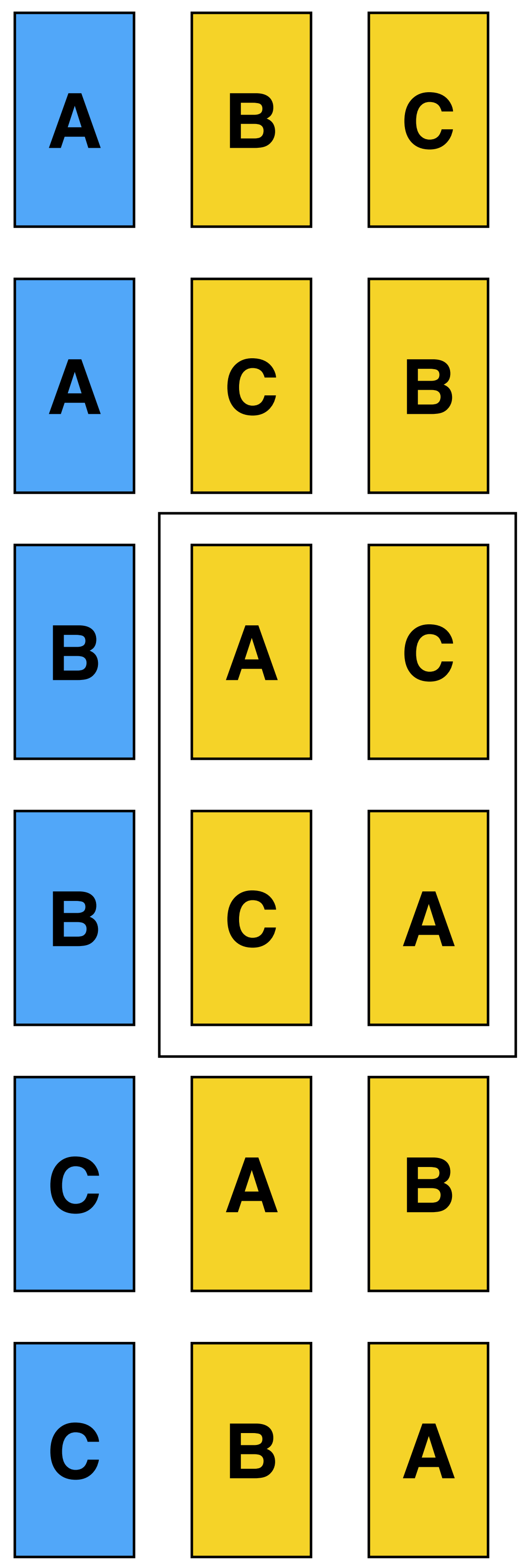
A	B	C
A	C	B
B	A	C
B	C	A
C	A	B
C	B	A

Permutations of Three Letters A, B and C



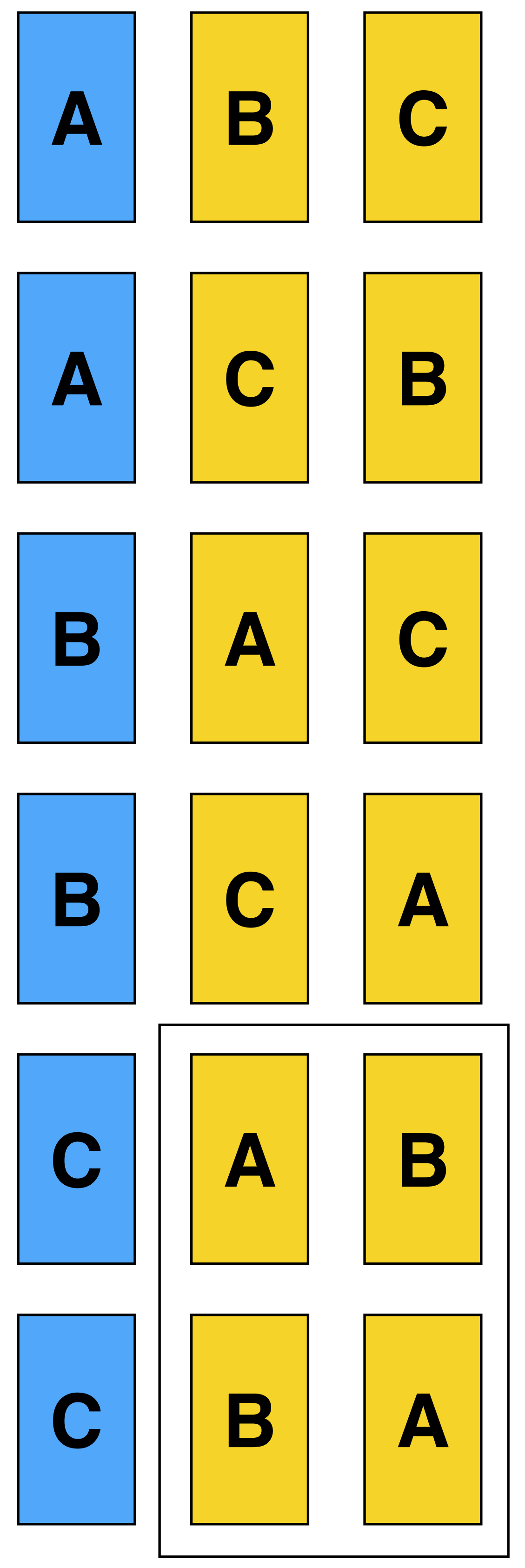
Permutations of B and C

Permutations of Three Letters A, B and C



Permutations of A and C

Permutations of Three Letters A, B and C



A	B	C
A	C	B
B	A	C
B	C	A
C	A	B
C	B	A

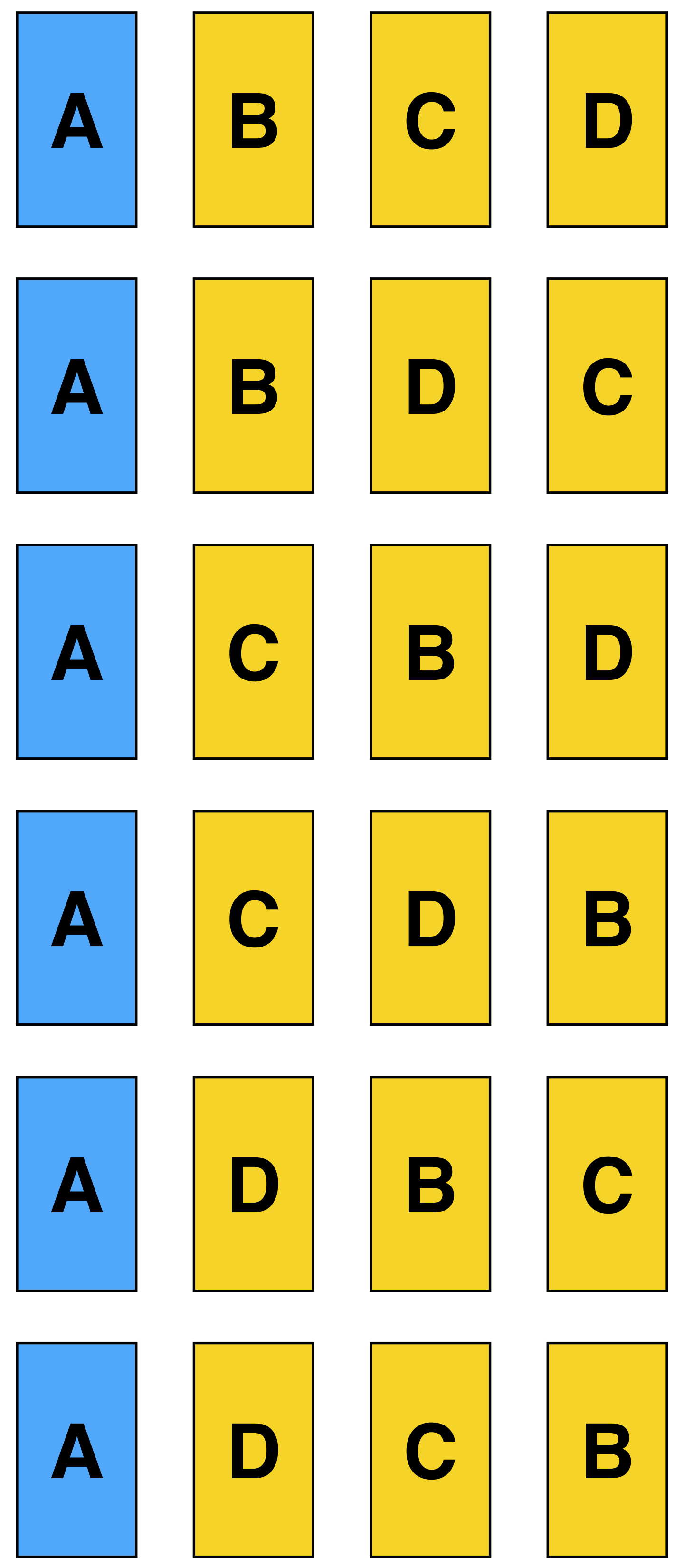
Permutations of A and B

Permutations of Four Letters A, B, C and D

A	B	C	D
A	B	D	C
A	C	B	D
A	C	D	B
A	D	B	C
A	D	C	B

...

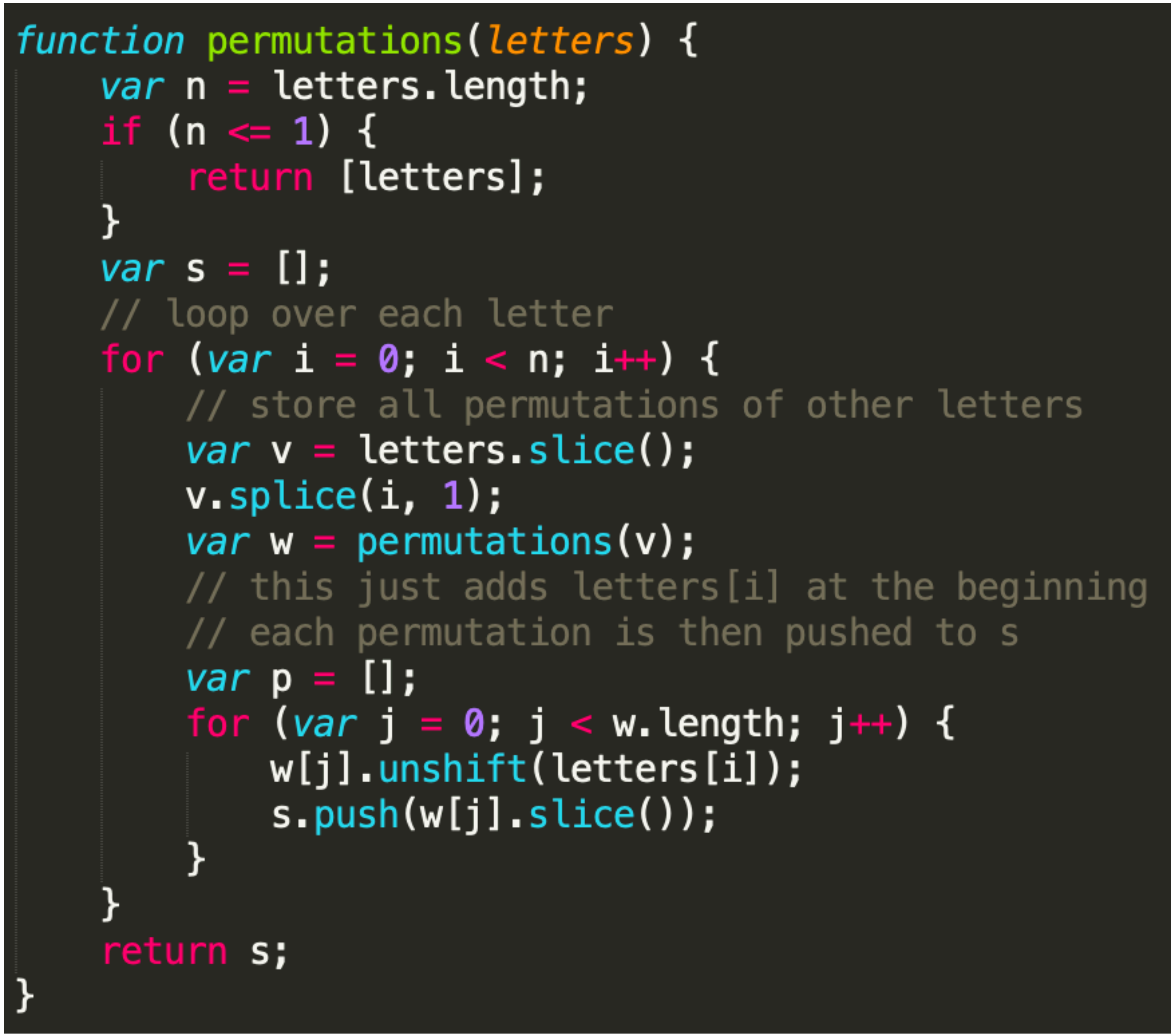
Permutations of Four Letters A, B, C and D



. . .

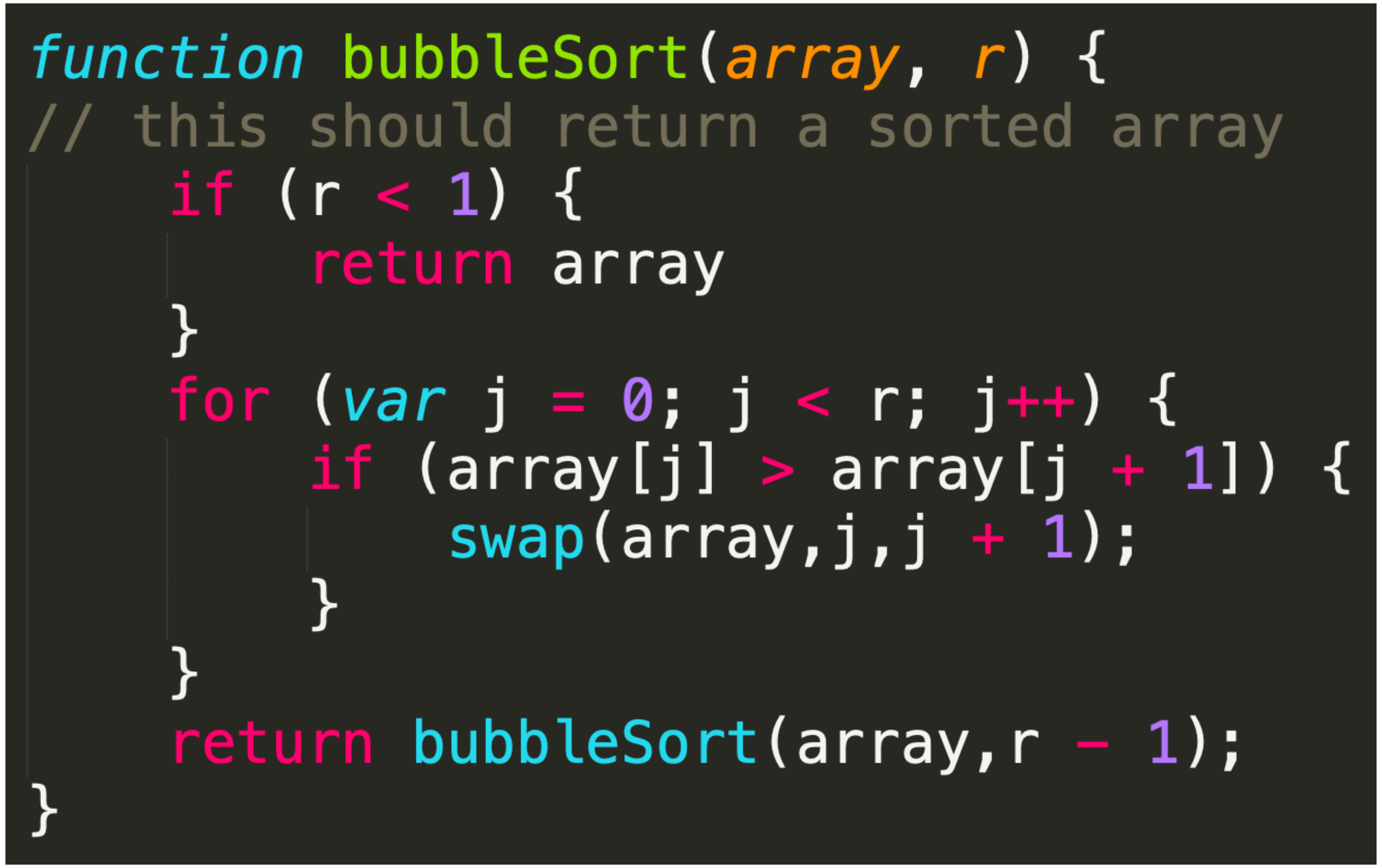
For each letter, generate all permutations of the remaining letters

Use recursion



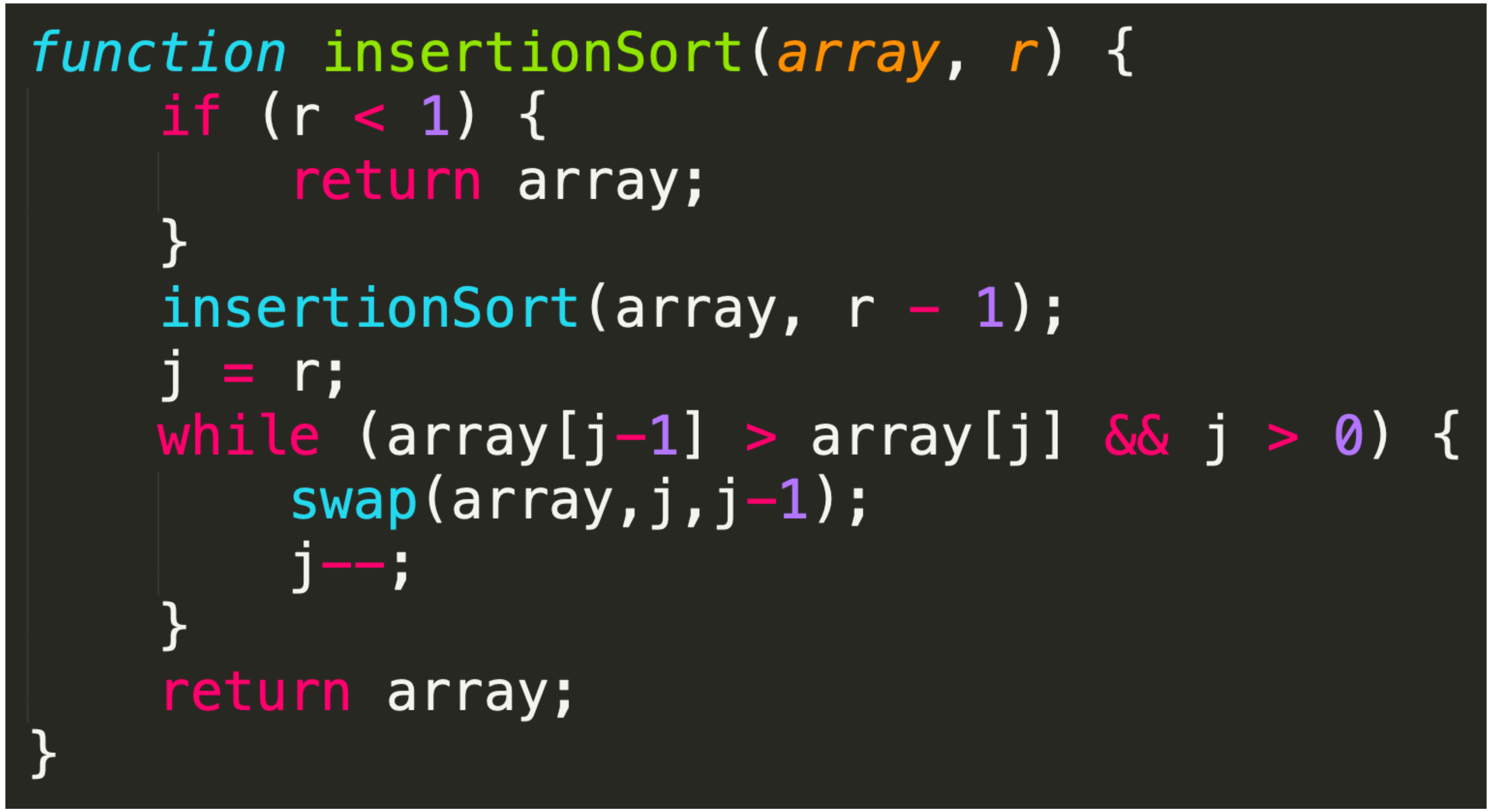
```
function permutations(letters) {
    var n = letters.length;
    if (n <= 1) {
        return [letters];
    }
    var s = [];
    // loop over each letter
    for (var i = 0; i < n; i++) {
        // store all permutations of other letters
        var v = letters.slice();
        v.splice(i, 1);
        var w = permutations(v);
        // this just adds letters[i] at the beginning
        // each permutation is then pushed to s
        var p = [];
        for (var j = 0; j < w.length; j++) {
            w[j].unshift(letters[i]);
            s.push(w[j].slice());
        }
    }
    return s;
}
```

Algorithms can be implemented with recursion



```
function bubbleSort(array, r) {
// this should return a sorted array
    if (r < 1) {
        return array
    }
    for (var j = 0; j < r; j++) {
        if (array[j] > array[j + 1]) {
            swap(array,j,j + 1);
        }
    }
    return bubbleSort(array,r - 1);
}
```

- *n - 1* recursive function calls for array of length *n*
- *r* iterations in each function call
- *(n - 1) + (n - 2) + … + 1* operations is in $O(n^2)$



```
function insertionSort(array, r) {
    if (r < 1) {
        return array;
    }
    insertionSort(array, r - 1);
    j = r;
    while (array[j-1] > array[j] && j > 0) {
        swap(array,j,j-1);
        j--;
    }
    return array;
}
```

- *n - 1* recursive function calls for array of length *n*
- *r* iterations in each function call
- *$(n - 1) + (n - 2) + \ldots + 1$* operations is in *$O(n^2)$*

Algorithms can be implemented with recursion

Recursion can be a **tool** for algorithm design

Divide and Conquer

Decrease and Conquer

- Take a problem, turn it into a single smaller problem (multiple times)
- Solve the smallest problem
- Combine solutions of smaller problems to give solution of original problem
- Use recursion

e.g. Binary Search

Divide and Conquer

- Take a problem, divide it up into two or more smaller problems (multiple times)
- Solve the smallest problems
- Combine solutions to give solution to original problem
- Use recursion

Divide and Conquer

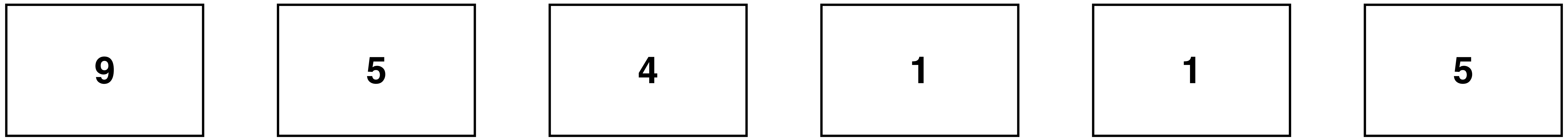
- Take a problem, divide it up into two or more smaller problems (multiple times)
- Solve the smallest problems
- Combine solutions to give solution to original problem
- Use recursion

e.g. Quicksort

Quicksort

Quicksort

- Divide-and-conquer algorithm for sorting dynamic arrays and vectors as implemented by JavaScript arrays
- “Divide up into half” using **a pivot**
- Apply Quicksort to the smaller arrays until we have array of one element (or none)



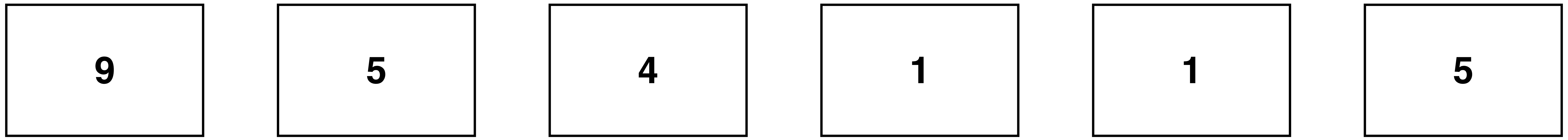
left = 0, right = 5

Pick a pivot index = floor((left + right)/2)

9	**5**	**4**	**1**	**1**	**5**

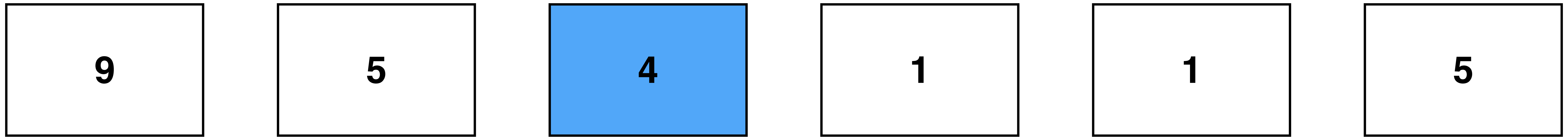
left = 0, right = 5
Pick a pivot index = floor((left + right)/2)

Other versions may have different pivot indices



left = 0, right = 5

Pick a pivot index = floor((left + right)/2)

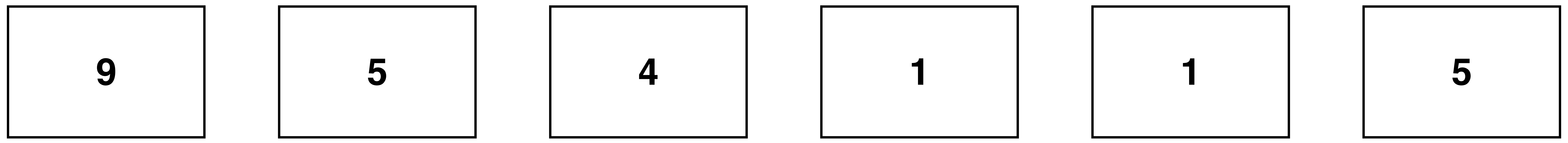


Partition array into two smaller arrays

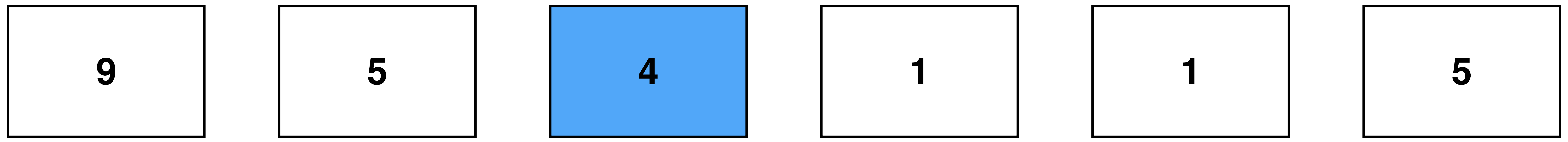
Array on left has values smaller or equal to pivot

Array on right has values larger or equal to pivot

Use a swap to do this

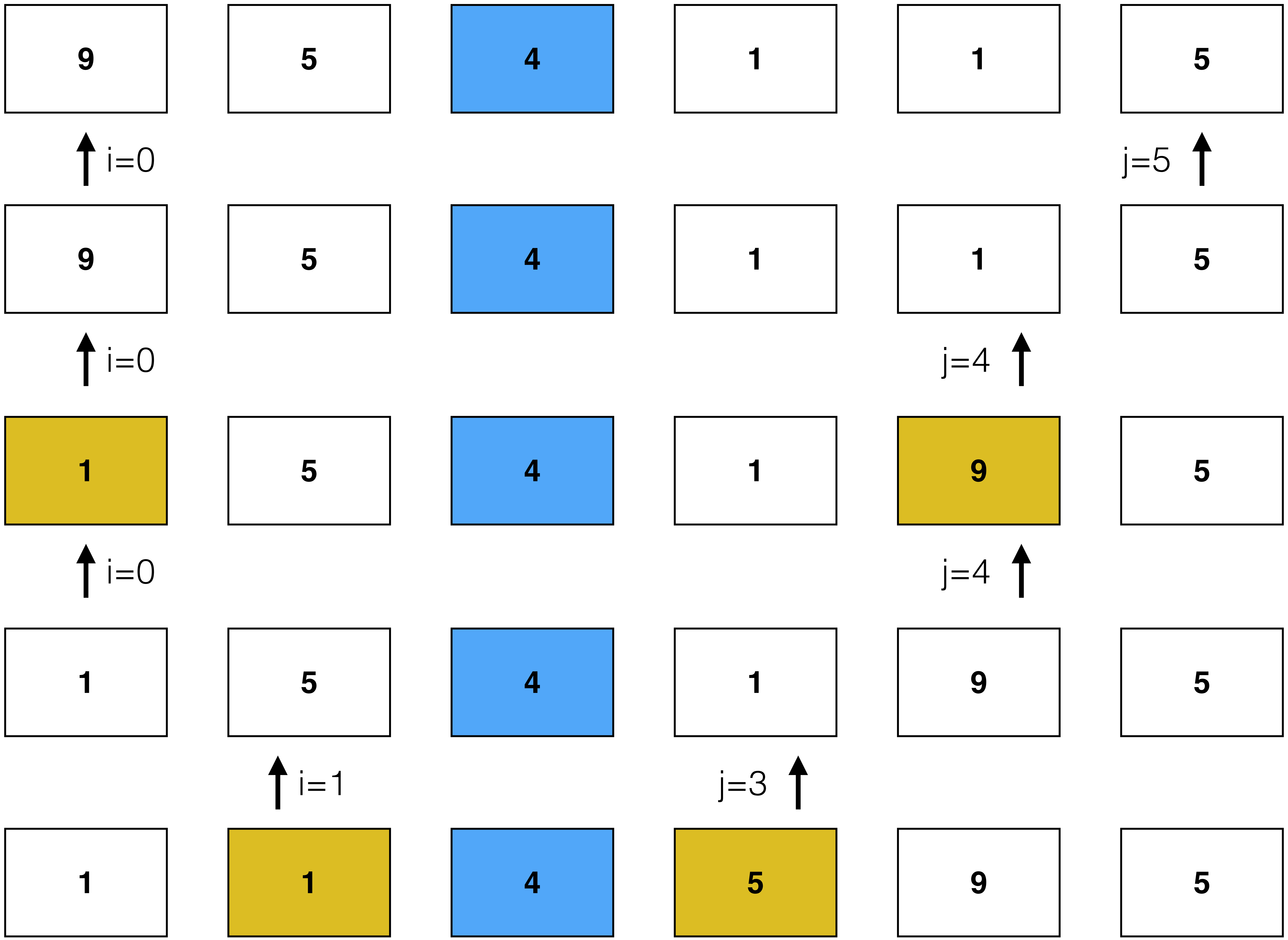


left = 0, right = 5
Pick a pivot index = floor((left + right)/2)

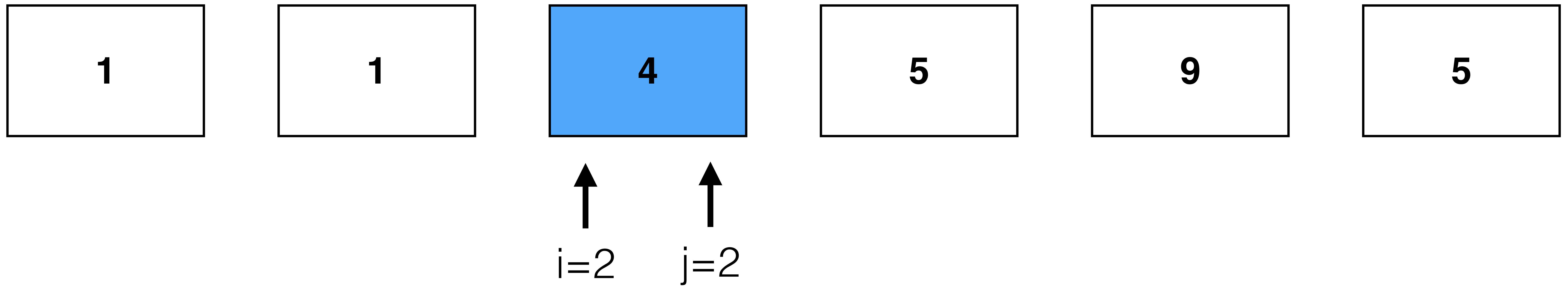


Partition array into two smaller arrays
Array on left has values smaller or equal to pivot
Array on right has values larger or equal to pivot
Use a swap to do this

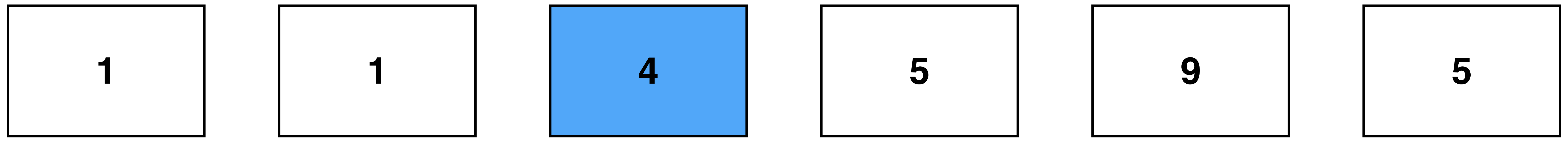
To partition we will use the Hoare partition algorithm



9 5 4 1 1 5
i=0
j=5
9 5 4 1 1 5
i=0
j=4
1 5 4 1 9 5
i=0
j=4
1 5 4 1 9 5
i=1
j=3
1 1 4 5 9 5



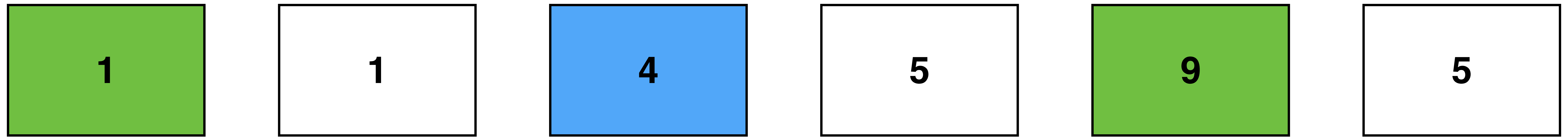
Finish partitioning when i=j



Left array Right array

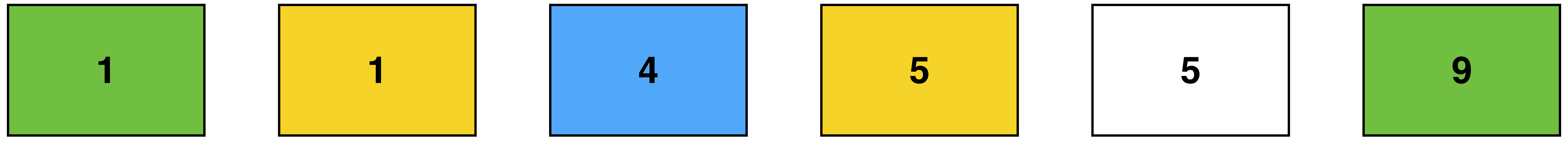
Now apply Quicksort to these individual arrays

Choose pivots

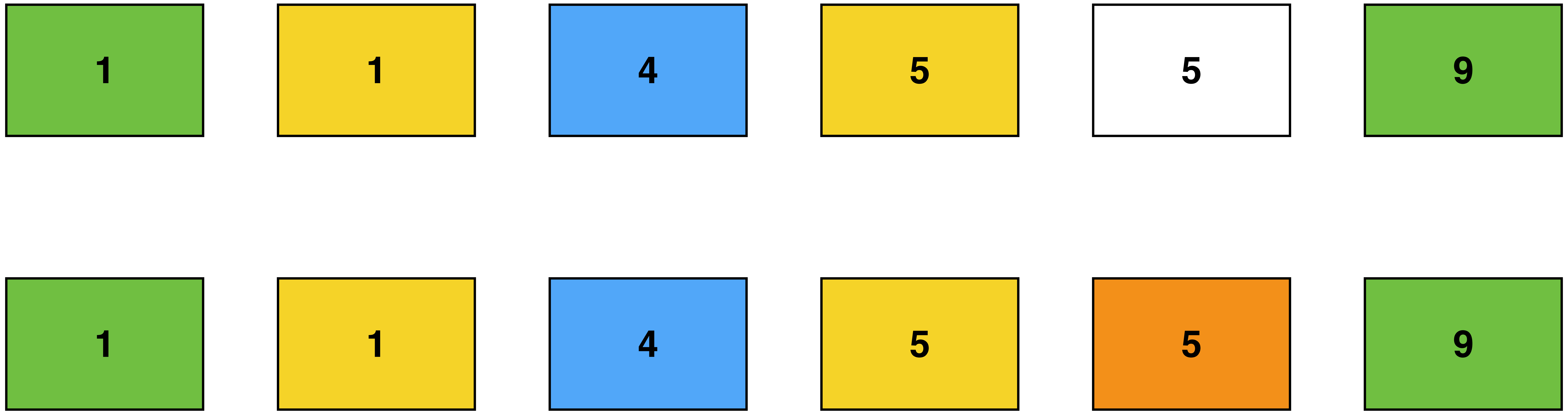


Generate new smaller arrays, apply Quicksort

Choose pivots…



…



When we hit base case of array of 1 element

Return array

In this example, many of the pivot values did not move

This might not happen in general

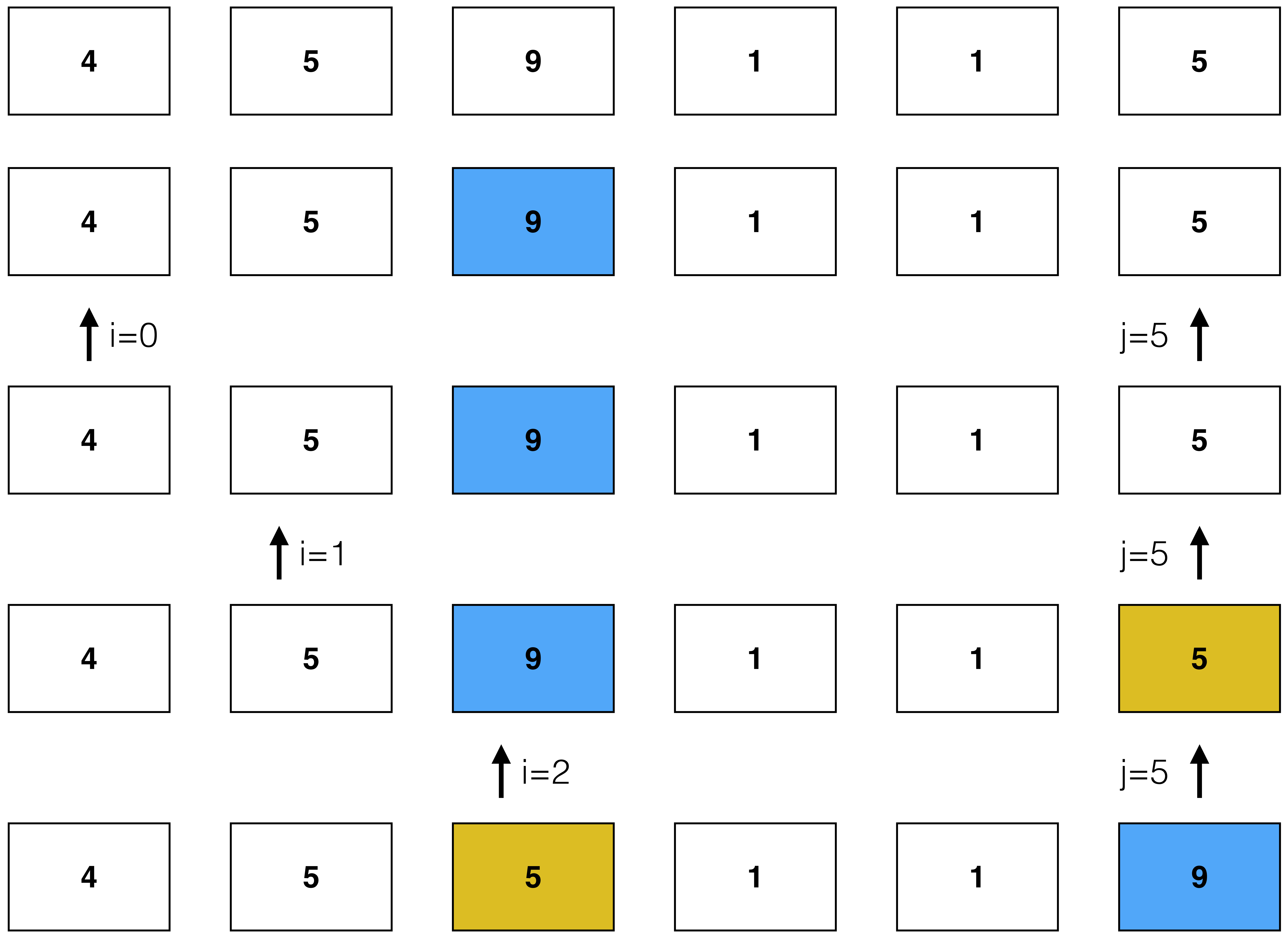
To call Quicksort on smaller arrays, we need final location of pivot

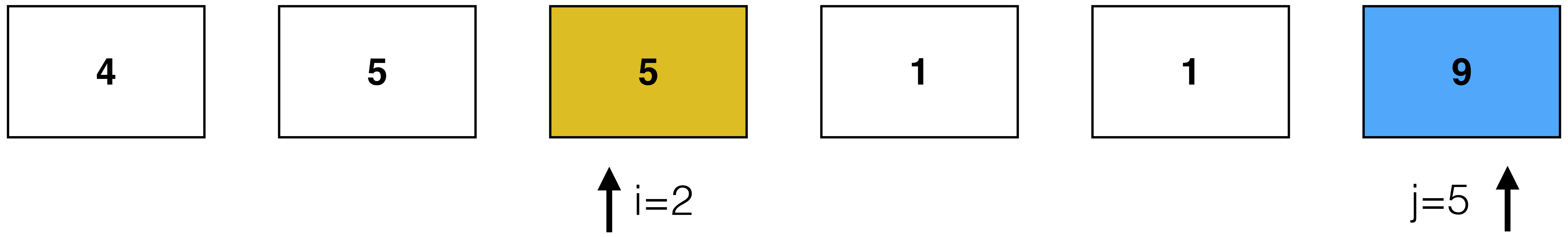
In this example, many of the pivot values did not move

This might not happen in general

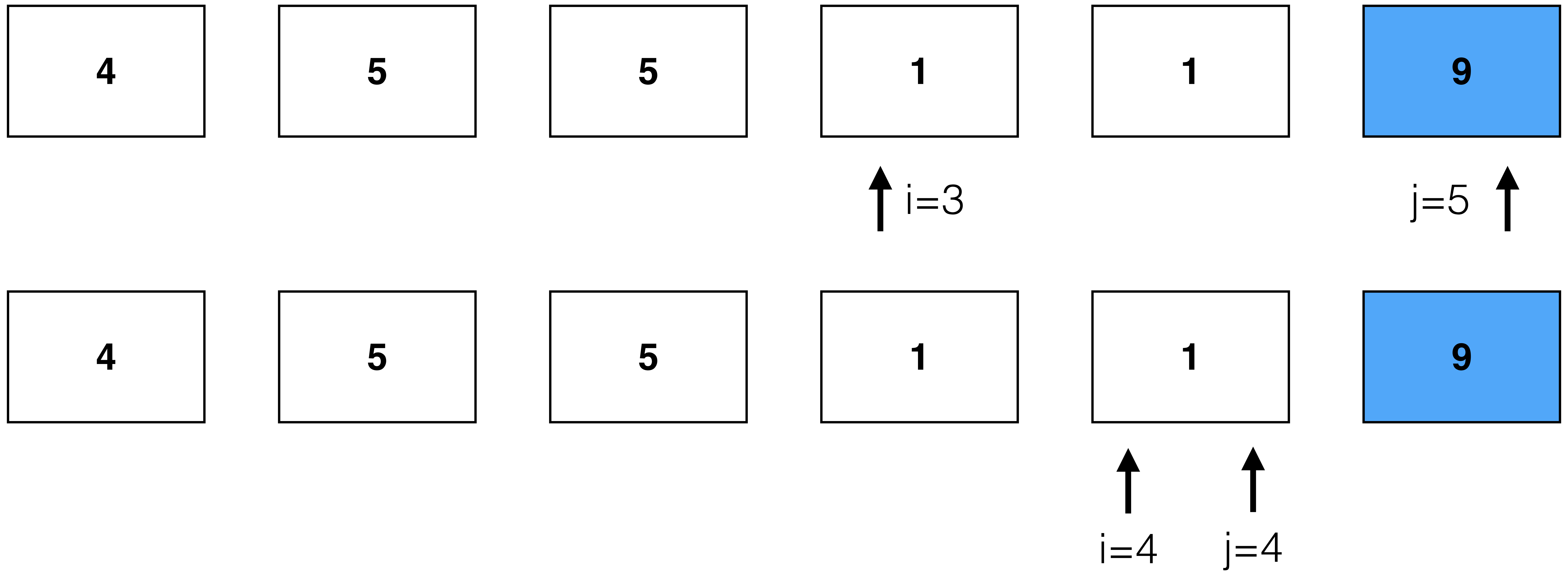
To call Quicksort on smaller arrays, we need final location of pivot

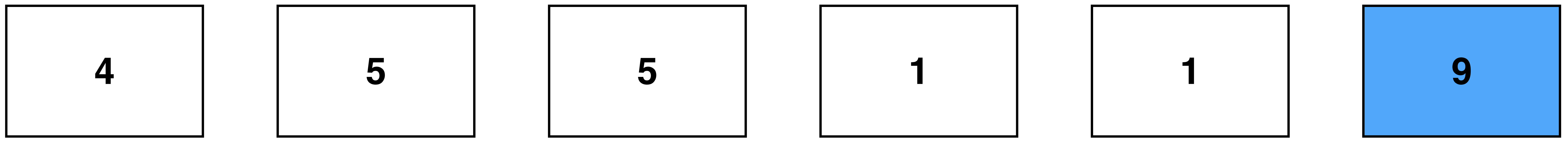
Another example of partitioning


4 5 9 1 1 5
4 5 9 1 1 5
i=0
j=5
4 5 9 1 1 5
i=1
j=5
4 5 9 1 1 5
i=2
j=5
4 5 5 1 1 9

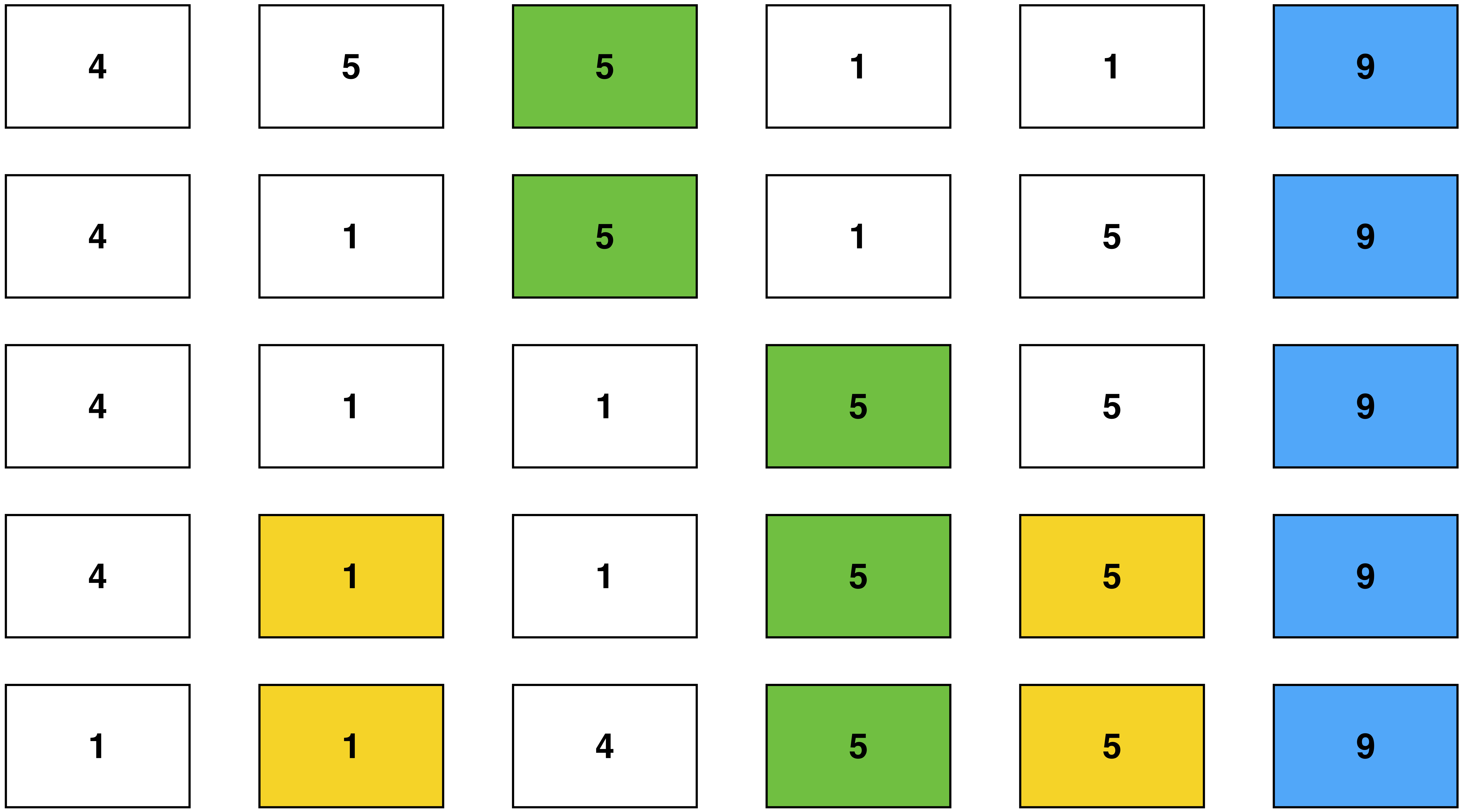


Make a note of new location of pivot value
Only increase i

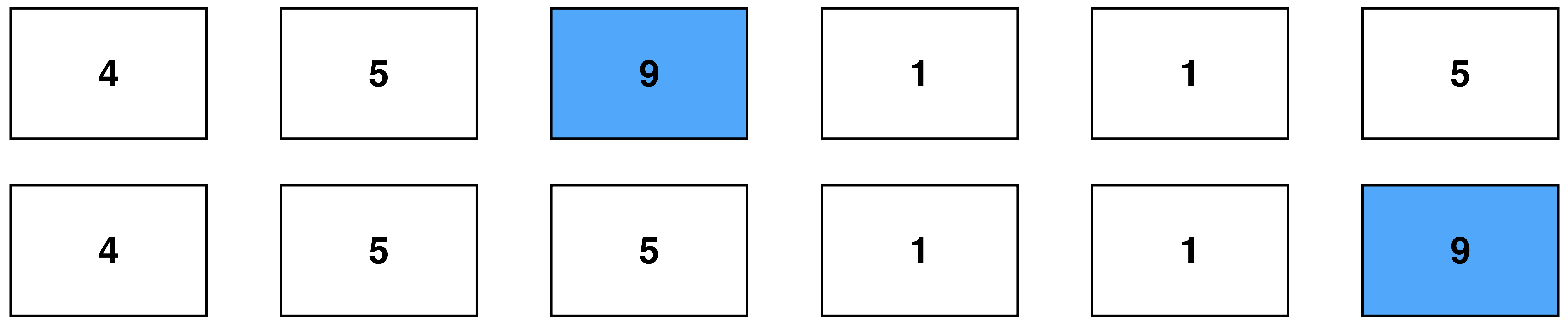


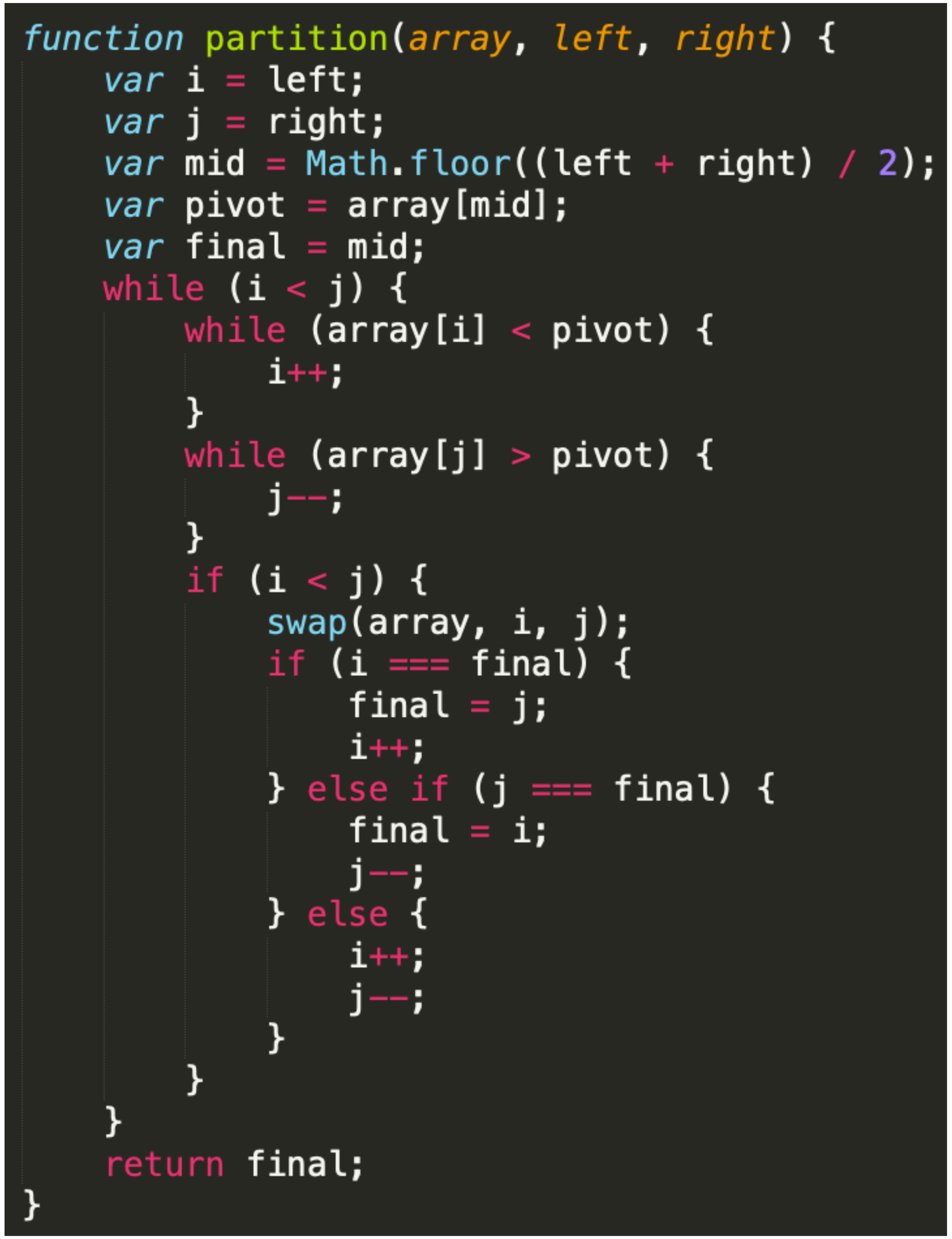


Look at array to the left of new location of pivot



Sometimes after partitioning we might have only one smaller array

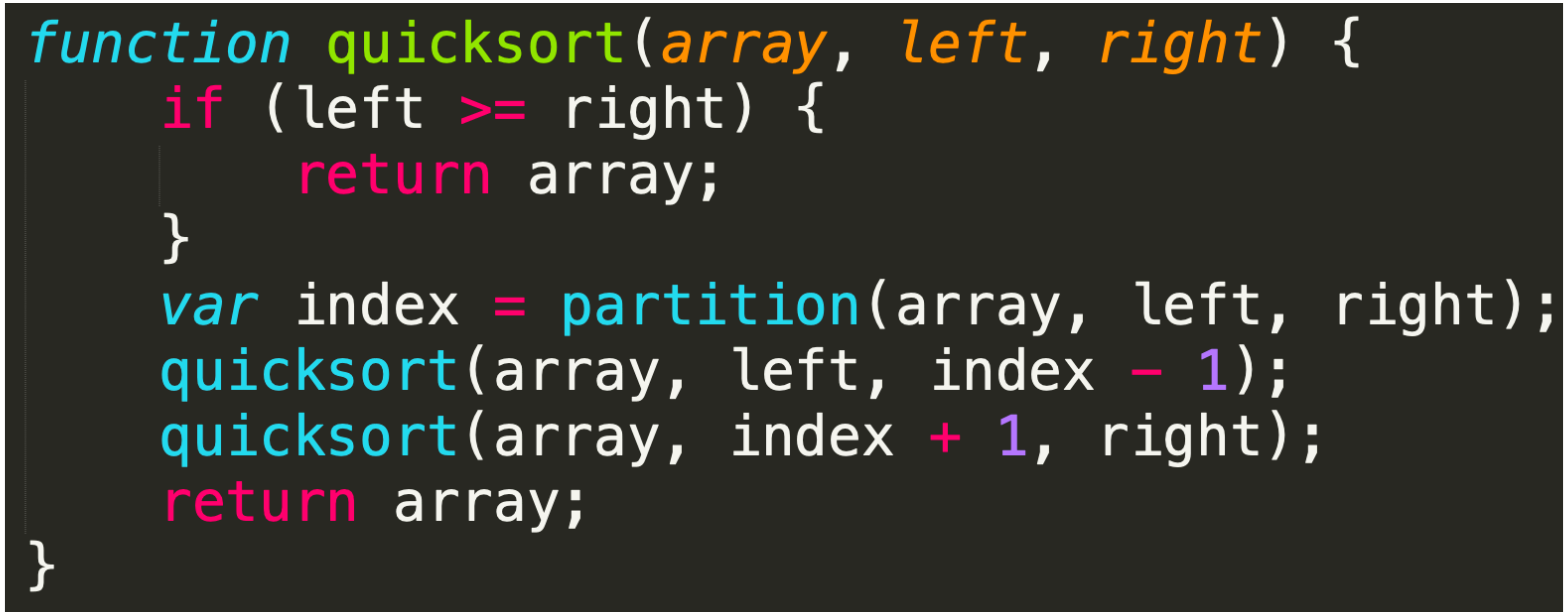




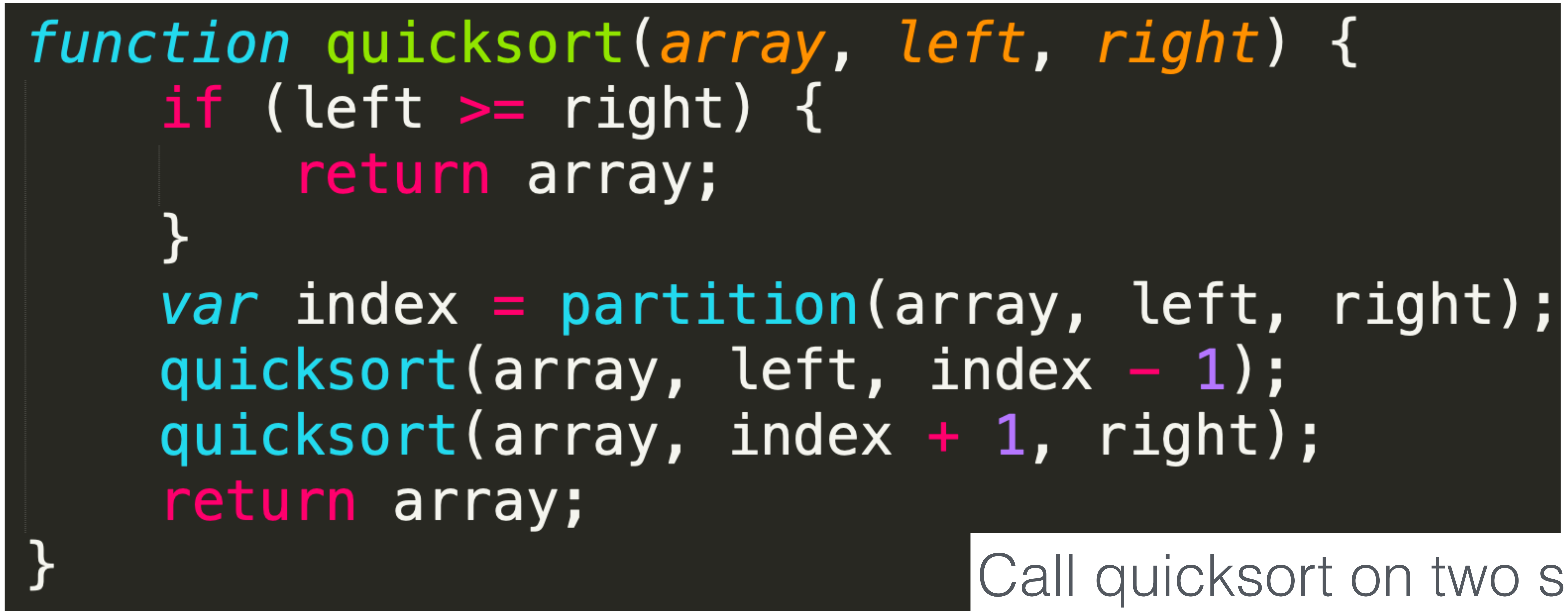
```
function partition(array, left, right) {
    var i = left;
    var j = right;
    var mid = Math.floor((left + right) / 2);
    var pivot = array[mid];
    var final = mid;
    while (i < j) {
        while (array[i] < pivot) {
            i++;
        }
        while (array[j] > pivot) {
            j--;
        }
        if (i < j) {
            swap(array, i, j);
            if (i === final) {
                final = j;
                i++;
            } else if (j === final) {
                final = i;
                j--;
            } else {
                i++;
                j--;
            }
        }
    }
    return final;
}
```

Variable final keeps track of final location of pivot value

If i or j points at where the pivot value is stored, we update final and increase/ decrease that variable



```javascript
function quicksort(array, left, right) {
    if (left >= right) {
        return array;
    }
    var index = partition(array, left, right);
    quicksort(array, left, index - 1);
    quicksort(array, index + 1, right);
    return array;
}
```



```
function quicksort(array, left, right) {
    if (left >= right) {
        return array;
    }
    var index = partition(array, left, right);
    quicksort(array, left, index - 1);
    quicksort(array, index + 1, right);
    return array;
}
```

Call quicksort on two smaller arrays

Admin

Second half of the lecture:

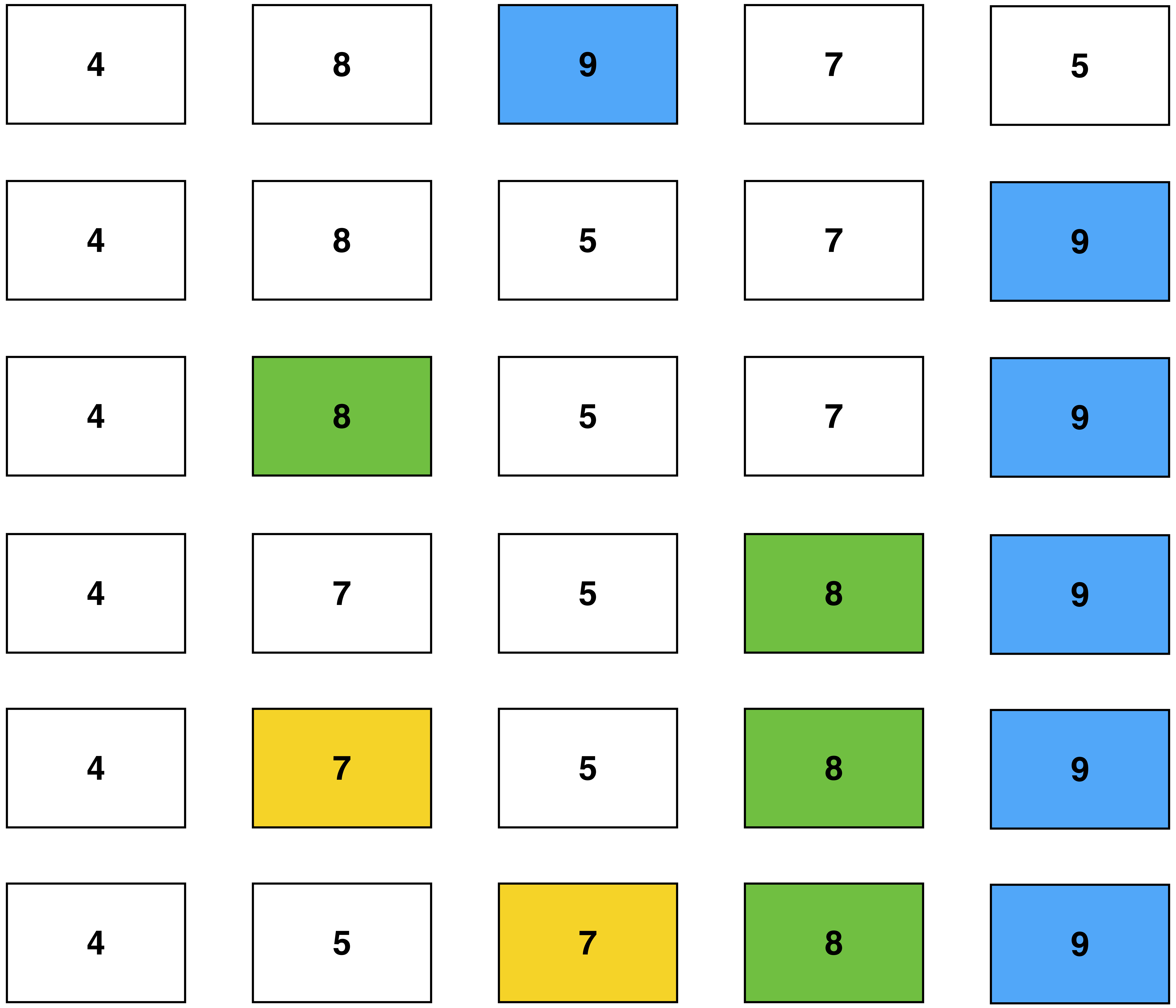
- Worst-case time complexity of Quicksort
- Space complexity
- General admin
- Mock Online Test ‘walkthrough’
- Randomised Quicksort

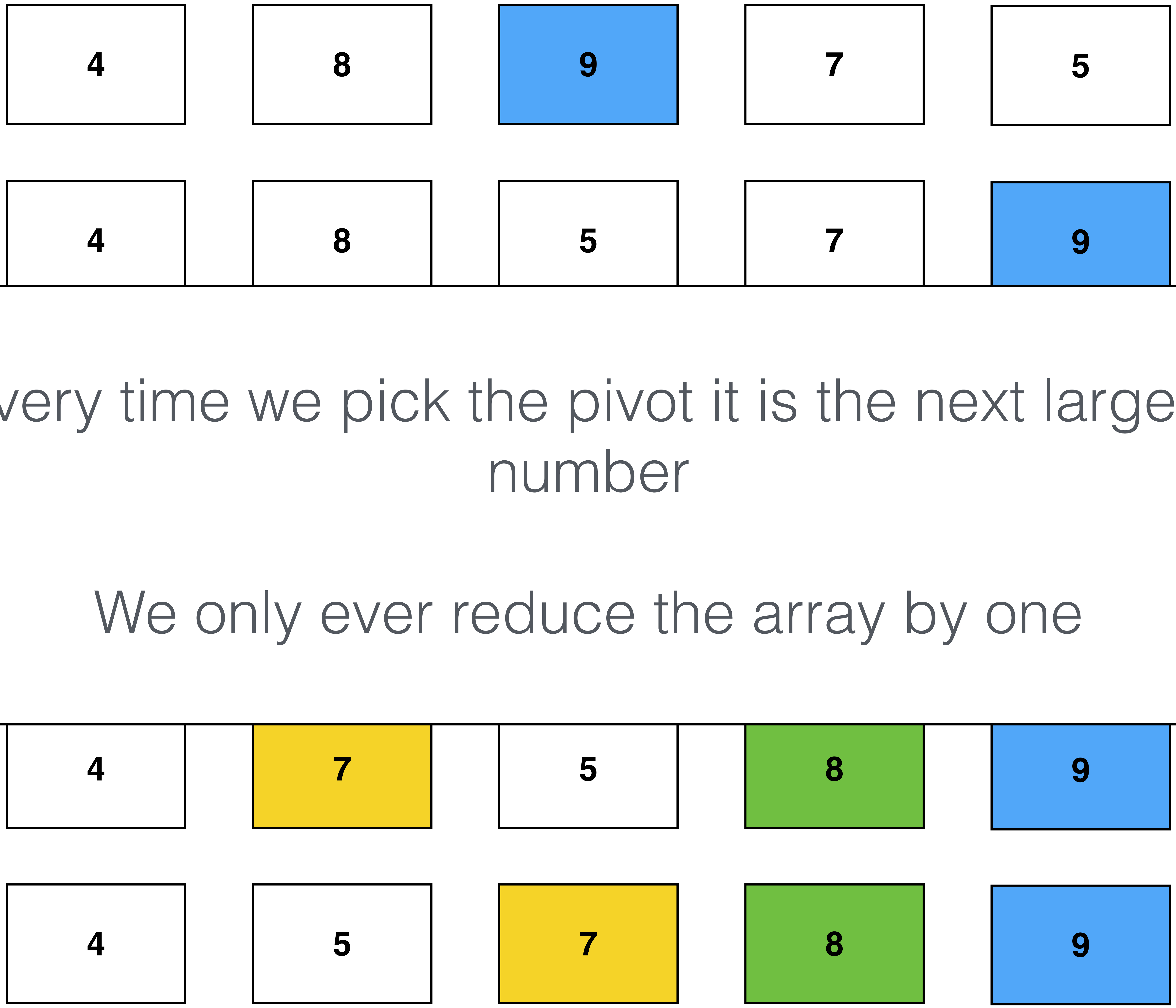
Admin

Second half of the lecture:

- Worst-case time complexity of Quicksort
- Space complexity
- General admin
- Mock Online Test ‘walkthrough’
- Randomised Quicksort

Worst-case Time Complexity of Quicksort


4 8 9 7 5
4 8 5 7 9
4 8 5 7 9
4 7 5 8 9
4 7 5 8 9
4 5 7 8 9



Every time we pick the pivot it is the next largest number

We only ever reduce the array by one

Worst-case Time Complexity

If every time we partition we only have one array left over

This array is of size n-1

Therefore we have to partition n-1 times

In every partition of an array of length m requires O(m) comparisons with the value of the pivot

Worst-case time complexity is sum over all m from 1 to n - 1 — similar to Insertion Sort

O(n^2)

Average-case Time Complexity

Why bother with Quicksort if it has the same worst-case time complexity as Bubble Sort and Insertion Sort?

Getting the worst-case array is **rare** among all possible arrays

It has an **average-case** time complexity of ***O(nlog n)***

Average-case Time Complexity

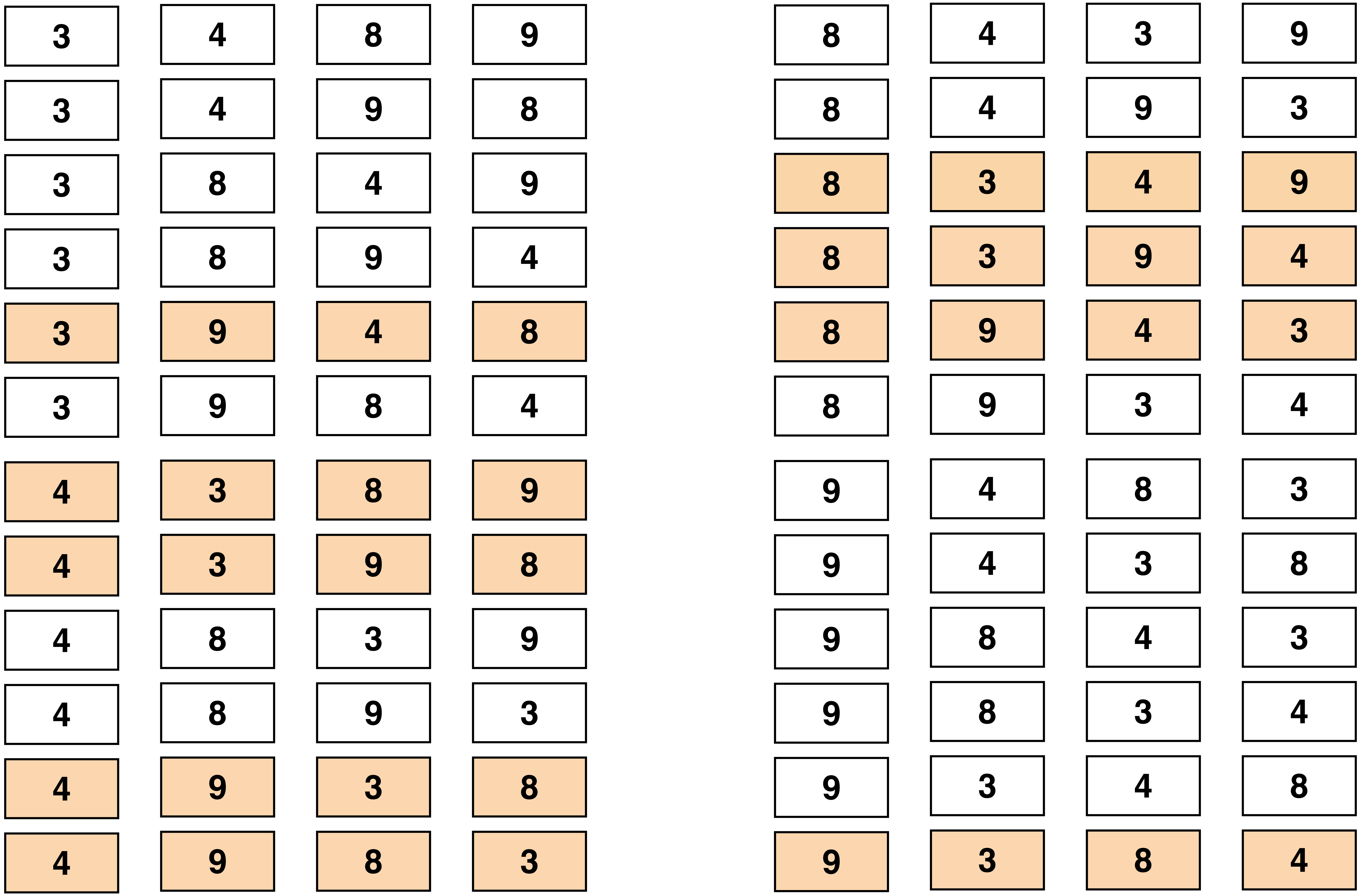
Quicksort has an **average-case** time complexity of ***O(nlog n)***

Consider all arrays of length 4 with values 3, 4, 8, 9

3	4	8	9
3	4	9	8
3	8	4	9
3	8	9	4
3	9	4	8
3	9	8	4

. . .

4! possibilities = 24



Most of the time (15/24) there will be some partitioning into multiple smaller arrays

Average-case Time Complexity

More efficient *in practice*

lecture10.js

Average-case Time Complexity

If all possible n! permutations are equally likely to be the input, the worst case input array becomes less and less likely

For majority of the inputs there is some division into two smaller arrays

(Number of times we can halve) *x* (Final number of arrays)
= O(nlog n)

More efficient *in practice*

Quicksort

Divide and conquer is a ***general approach*** to algorithms

Based on ***recursion***

Quicksort is a classic example

More efficient in practice than Bubble and Insertion Sort

Algorithm	Worst-case Time Complexity	Average-case Time Complexity
Bubble Sort	$O(n^2)$	$O(n^2)$
Insertion Sort	$O(n^2)$	$O(n^2)$
Quicksort	$O(n^2)$	$O(n\log n)$

Algorithm	Worst-case Time Complexity	Average-case Time Complexity
Bubble Sort	$O(n^2)$	$O(n^2)$
Insertion Sort	$O(n^2)$	$O(n^2)$
Quicksort	$O(n^2)$	$O(n\log n)$
Merge Sort	$O(n\log n)$	$O(n\log n)$

Space Complexity

Space Complexity

The focus of this module in analysis algorithms is time complexity: number of operations

Another resource other than time is space: how much memory is used in implementation of algorithms

In RAM model a memory unit stores an integer

A primitive data variable takes up constant space

Implementations of data structures take up multiple memory units

Space Complexity

Linear Search, Bubble Sort, Insertion Sort and Binary Search only ever create a constant number of variables

The worst-case space complexity: maximum amount of memory units in Big O notation in n needed for all input arrays of length n

Above algorithms only need constant space in RAM model in the worst case

Space Complexity

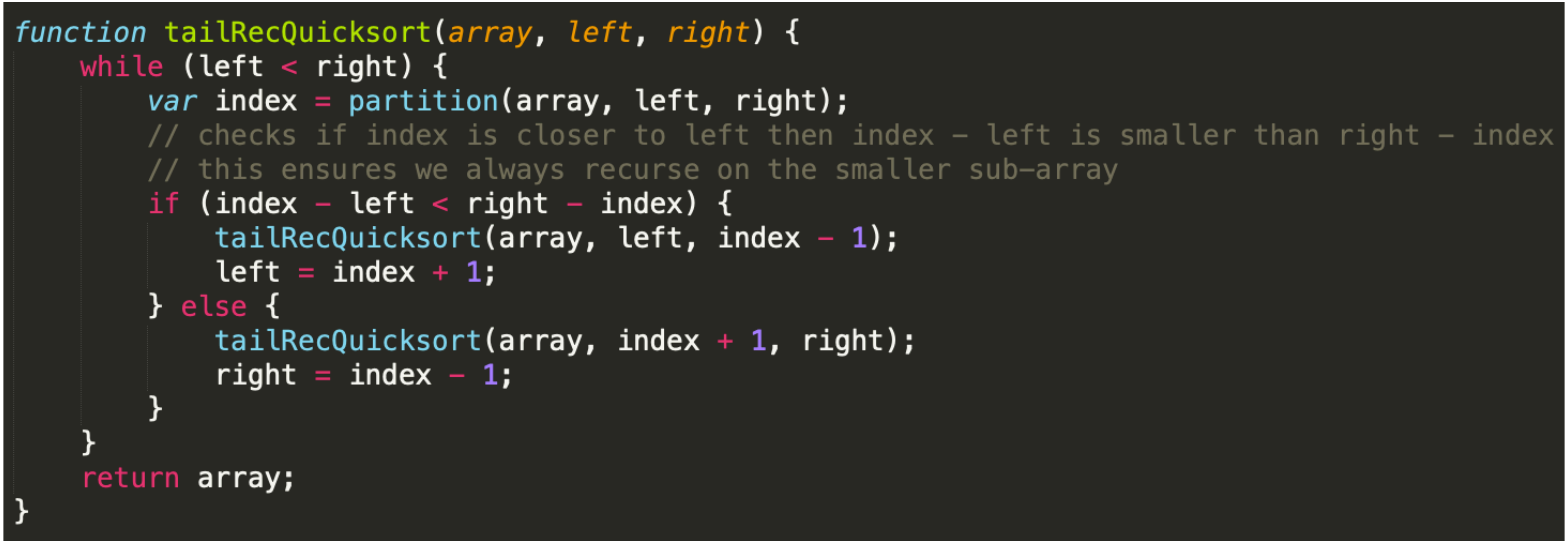
For recursion, every time we have a function call a new element is created of the call stack - we need extra space

Number of recursive function calls indicates the space complexity

Worst-case space complexity of Quicksort we've seen is O(n) for array of length n: n function calls, with constant number of variables created in each call

We can do better! Uses something called tail recursion

Space-efficient Quicksort



```
function tailRecQuicksort(array, left, right) {
    while (left < right) {
        var index = partition(array, left, right);
        // checks if index is closer to left then index - left is smaller than right - index
        // this ensures we always recurse on the smaller sub-array
        if (index - left < right - index) {
            tailRecQuicksort(array, left, index - 1);
            left = index + 1;
        } else {
            tailRecQuicksort(array, index + 1, right);
            right = index - 1;
        }
    }
    return array;
}
```

- In the worst-case input, we never get any recursive function calls!
- Still has quadratic worst-case time complexity
- Worst-case space complexity is now O(log n)

Admin

- Sixth quiz deadline today at 4pm
  - Seventh quiz deadline **29th March** at 4pm

- Primes assignment
  - Cut-off date **29th March 4pm**
  - Help with Primes Assignment in VCH

- New worksheet released today
  - Help in VCH

- Review Seminar this week - Q&A with second-year students

- Mock Online Test (walkthrough now)

Randomised Quicksort

Reminder: Average-case Time Complexity

If all possible n! permutations are equally likely to be the input, the worst case input array becomes less and less likely to be given

For majority of the inputs there is some division into two smaller arrays

(Number of times we can halve) *x* (Final number of arrays) = O(nlog n)

More efficient *in practice*

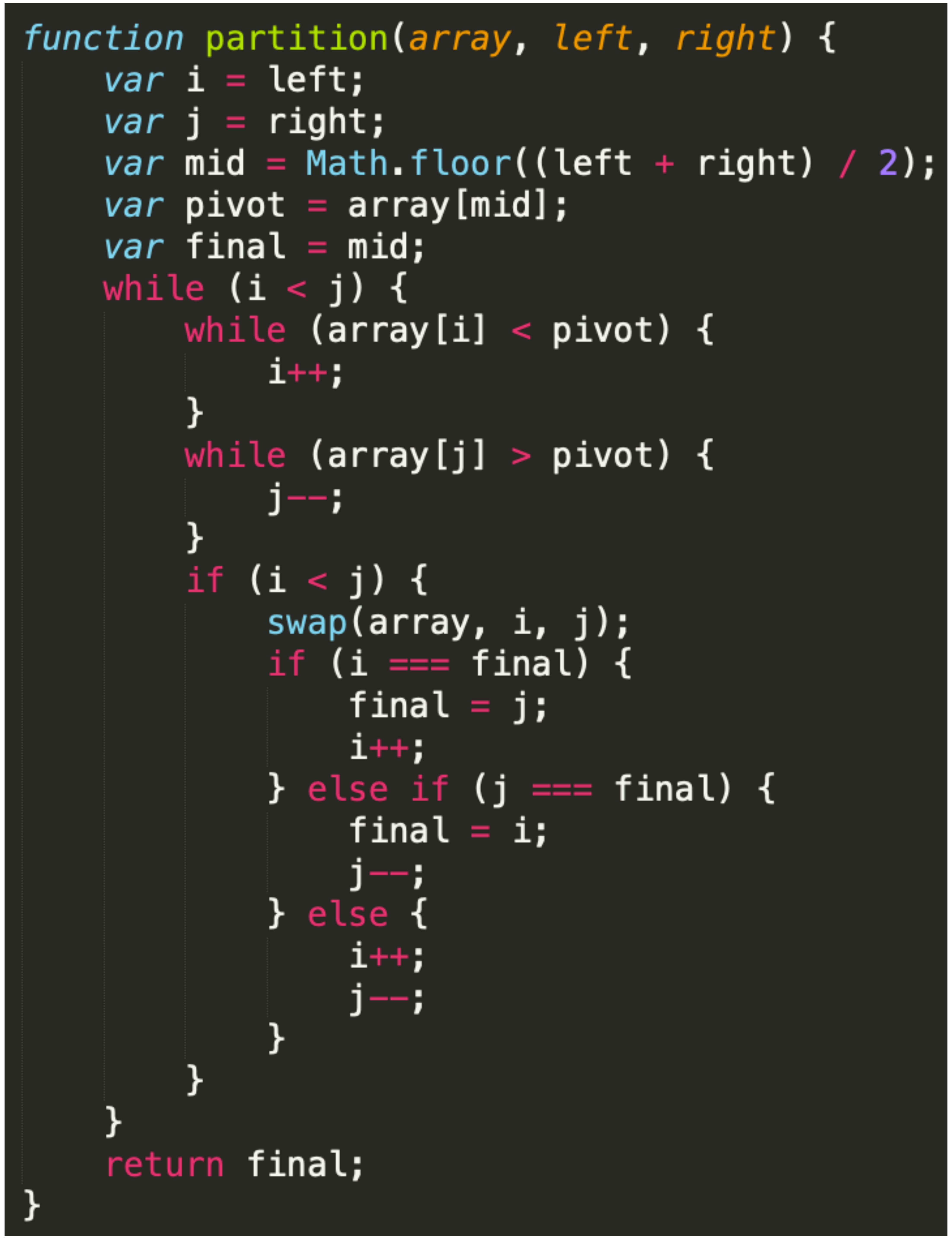
Reminder: Average-case Time Complexity

If all possible n! permutations are equally likely to be the input, the worst case input array becomes **less and less likely to be given**

For majority of the inputs there is some division into two smaller arrays

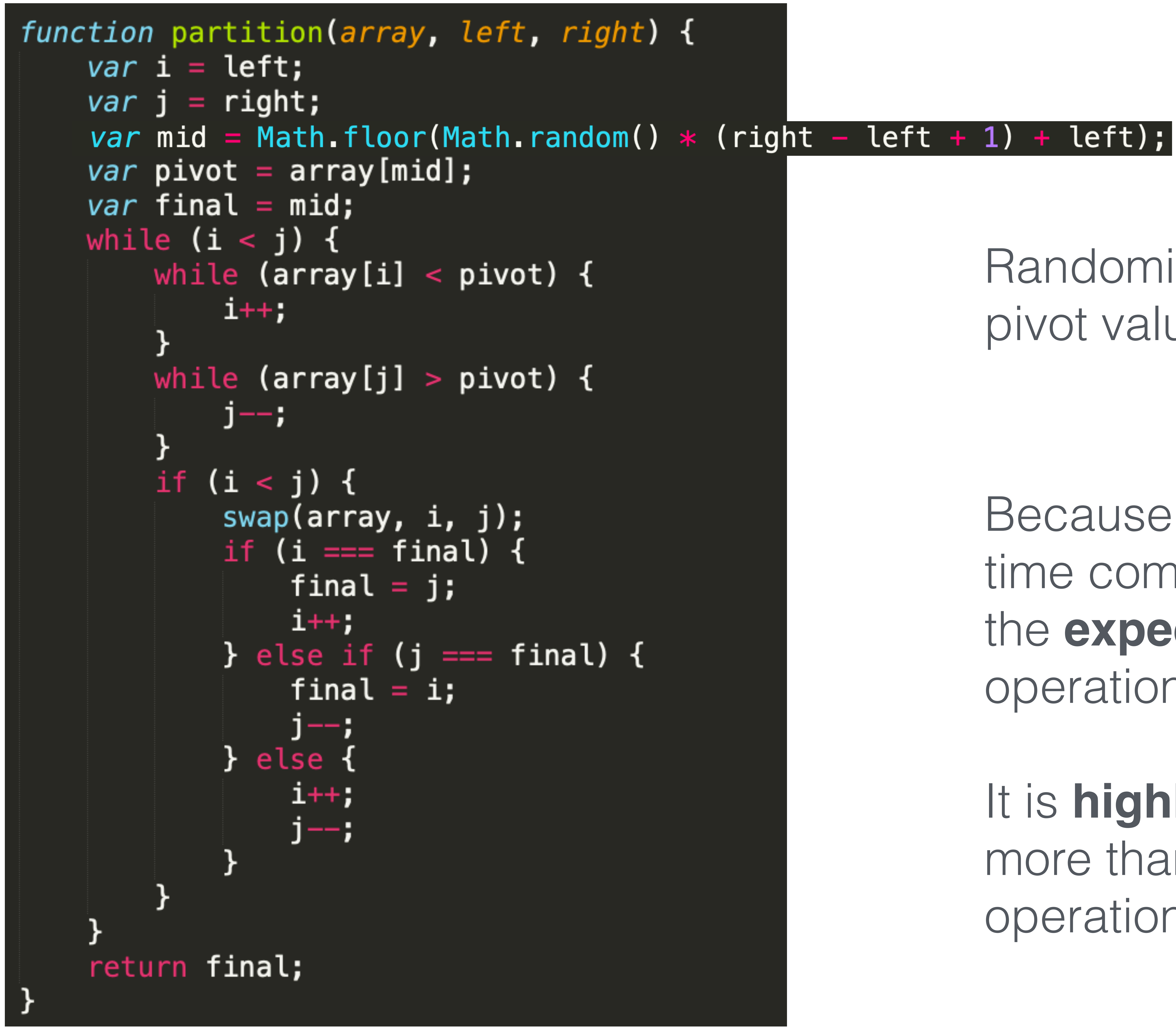
It is improbable that we will find the worst-case input as n gets larger

We can make it “less likely” by introducing some randomness…



```
function partition(array, left, right) {
    var i = left;
    var j = right;
    var mid = Math.floor((left + right) / 2);
    var pivot = array[mid];
    var final = mid;
    while (i < j) {
        while (array[i] < pivot) {
            i++;
        }
        while (array[j] > pivot) {
            j--;
        }
        if (i < j) {
            swap(array, i, j);
            if (i === final) {
                final = j;
                i++;
            } else if (j === final) {
                final = i;
                j--;
            } else {
                i++;
                j--;
            }
        }
    }
    return final;
}
```

Randomise the choice of pivot value!

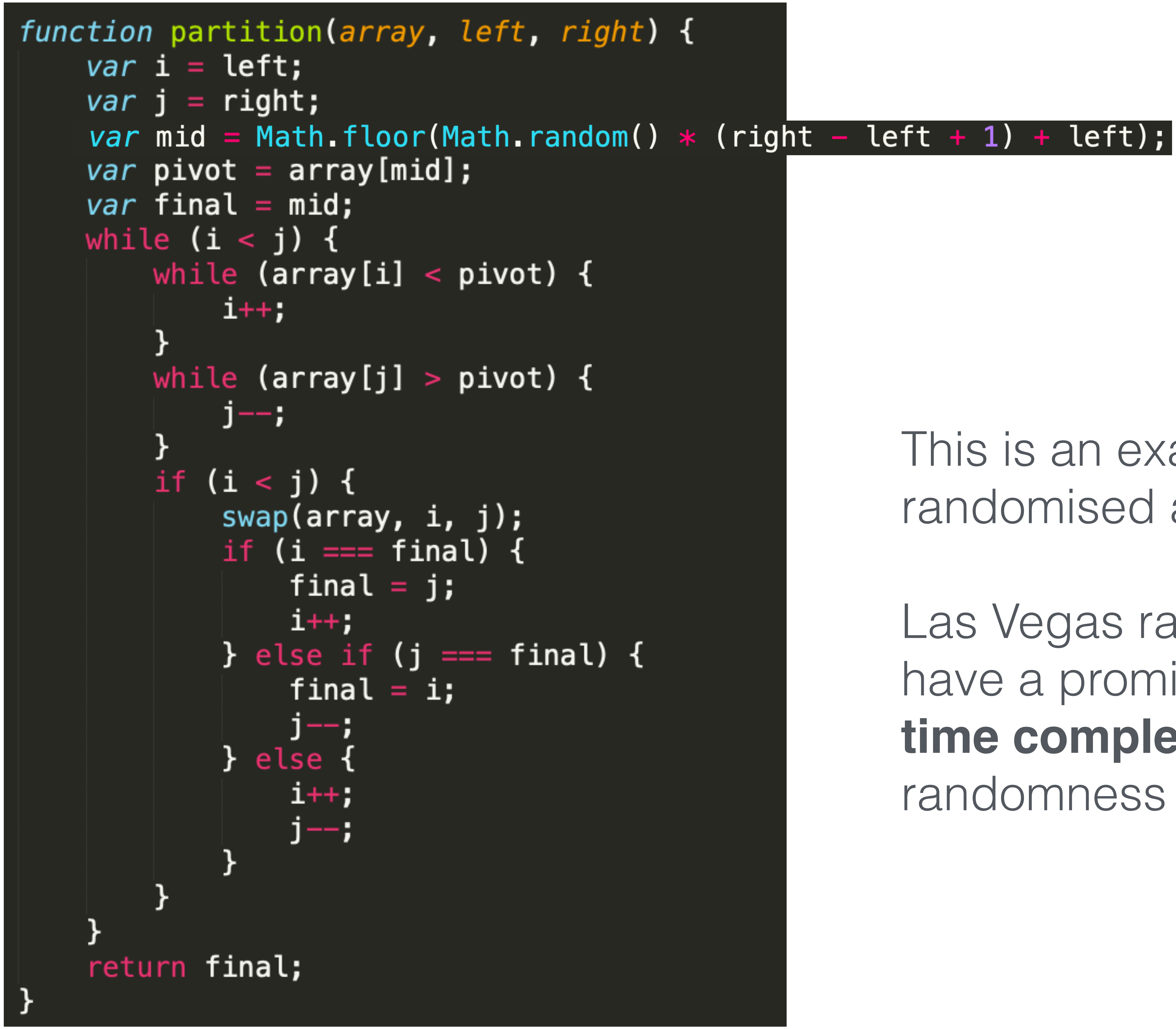


```
function partition(array, left, right) {
    var i = left;
    var j = right;
    var mid = Math.floor(Math.random() * (right - left + 1) + left);
    var pivot = array[mid];
    var final = mid;
    while (i < j) {
        while (array[i] < pivot) {
            i++;
        }
        while (array[j] > pivot) {
            j--;
        }
        if (i < j) {
            swap(array, i, j);
            if (i === final) {
                final = j;
                i++;
            } else if (j === final) {
                final = i;
                j--;
            } else {
                i++;
                j--;
            }
        }
    }
    return final;
}
```

Randomise the choice of pivot value!

Because the average-case time complexity is *O(nlog n)*, the **expected** number of operations is *O(nlog n)*

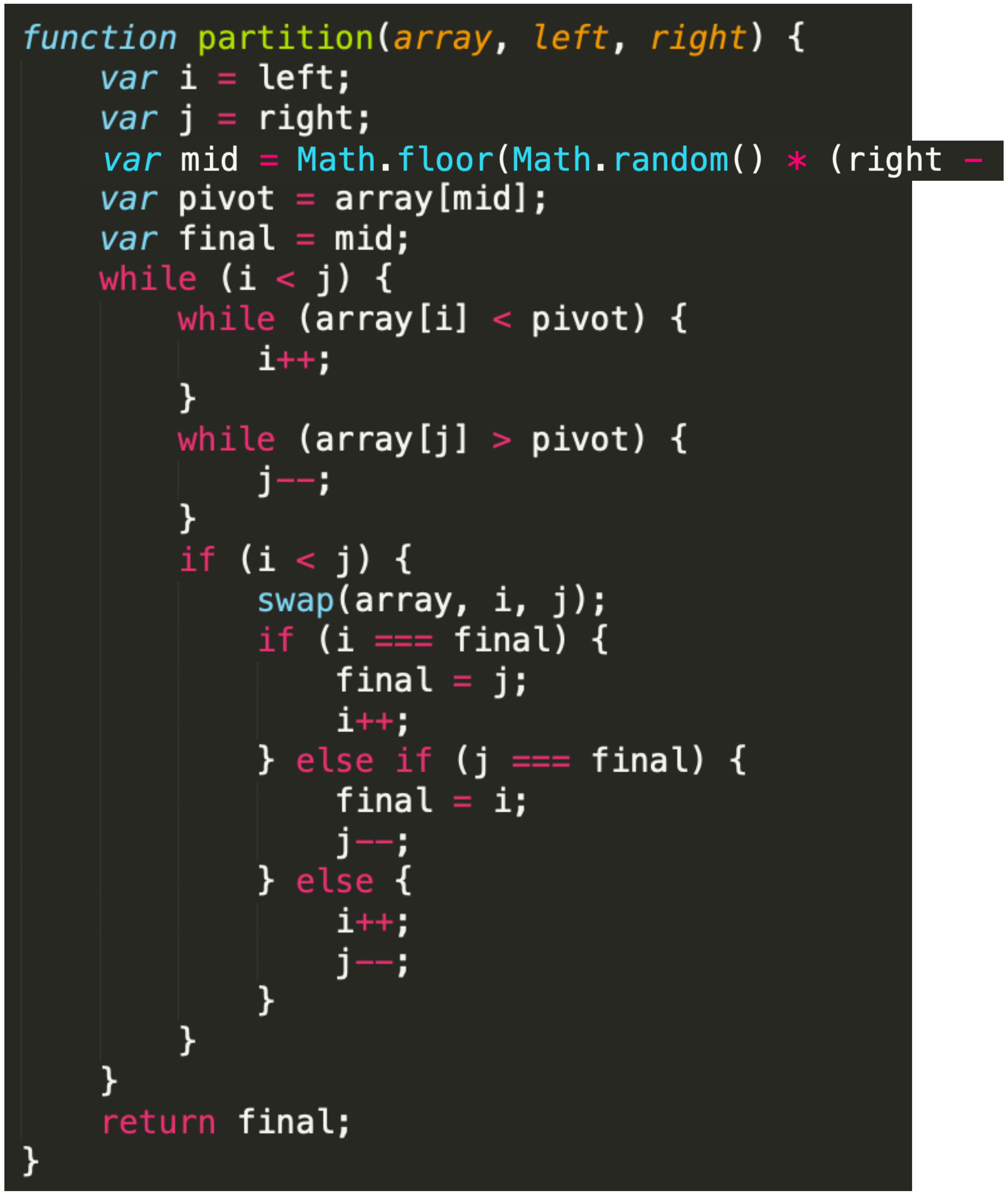
It is **highly unlikely** to need more than *O(nlog n)* operations



```
function partition(array, left, right) {
    var i = left;
    var j = right;
    var mid = Math.floor(Math.random() * (right - left + 1) + left);
    var pivot = array[mid];
    var final = mid;
    while (i < j) {
        while (array[i] < pivot) {
            i++;
        }
        while (array[j] > pivot) {
            j--;
        }
        if (i < j) {
            swap(array, i, j);
            if (i === final) {
                final = j;
                i++;
            } else if (j === final) {
                final = i;
                j--;
            } else {
                i++;
                j--;
            }
        }
    }
    return final;
}
```

This is an example of a Las Vegas randomised algorithm

Las Vegas randomised algorithms have a promise on the **expected time complexity** using randomness



```
function partition(array, left, right) {
    var i = left;
    var j = right;
    var mid = Math.floor(Math.random() * (right -
    var pivot = array[mid];
    var final = mid;
    while (i < j) {
        while (array[i] < pivot) {
            i++;
        }
        while (array[j] > pivot) {
            j--;
        }
        if (i < j) {
            swap(array, i, j);
            if (i === final) {
                final = j;
                i++;
            } else if (j === final) {
                final = i;
                j--;
            } else {
                i++;
                j--;
            }
        }
    }
    return final;
}
```

In addition to Las Vegas algorithms, there are Monte Carlo algorithms

Monte Carlo algorithms allow for the possibility of giving the wrong answer, but with a very small probability

If we can be wrong **we can use fewer operations**

Monte Carlo version of Randomised Quicksort in quicksort.js

Guaranteed to run in O(nlog n) time but small probability of incorrect output

I hope this is the beginning in your algorithms journey

It’s a voyage of human ingenuity and creativity that continues to have impacts on our day-to-day life

Understanding the limitations of algorithms is just as important as understanding what they can do

ALGORITH
!!!
YOUR ALGORITHM DOESN'T KNOW ME
YOUR ALGORITHM DOESN'T KNOW ME