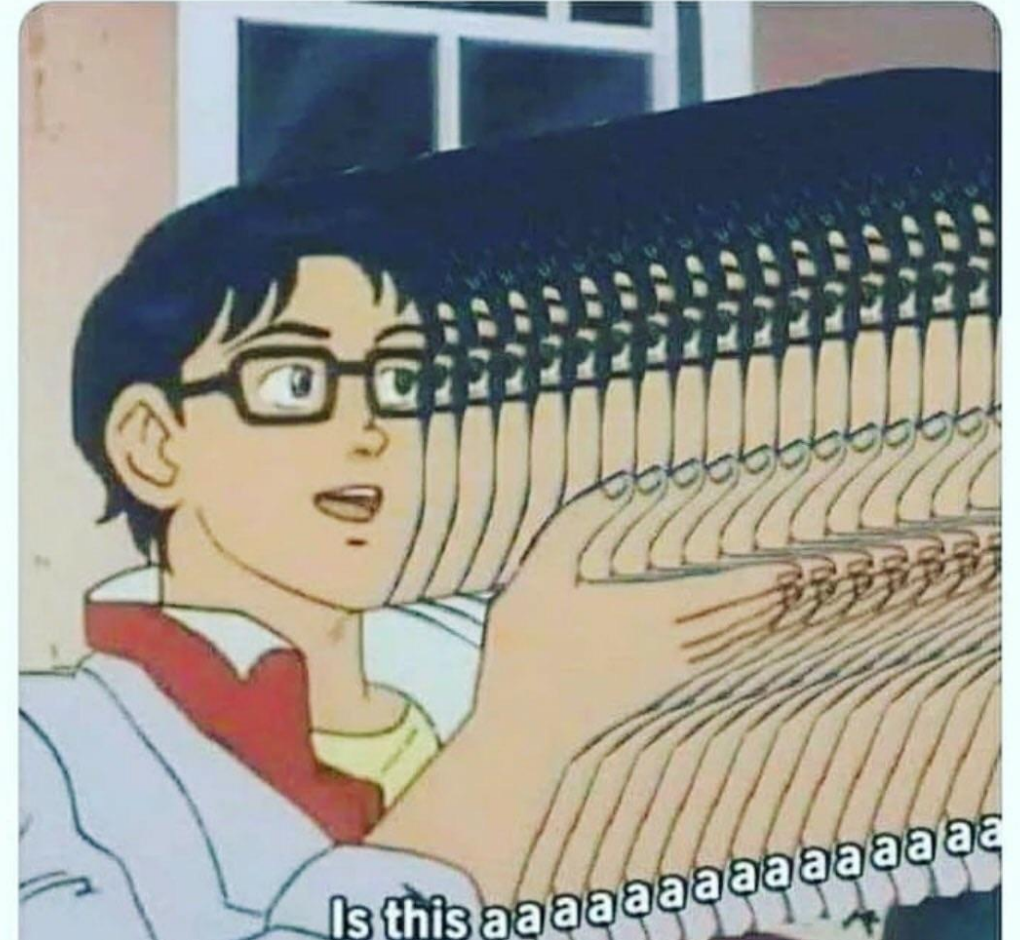# CSCI 132:
Basic Data Structures and Algorithms

Sorting (Quick Sort)

Reese Pearsall & Iliana Castillon

Fall 2024

Announcements

Program 4 due **tonight** at 11:59 PM

When your CS instructor is teaching you about recursive functions and you think you found one

Is this aaaaaaaaaaaaaaa

Quick Sort is a sorting algorithm that works by <u>partitioning</u> an array around a certain element in the array, called a <u>pivot</u>. This is a recursive method that then sorts the sections of the array to the left of the pivot, and to the right of the pivot.

Quick sort is a **Divide and Conquer** algorithm, which involves dividing the problem into smaller sub-problems (divide), recursively solving the smaller problems (conquer), and combining the sub problems to get the final solution for the original problem

Quick sort is rather complex. I don't expect you to memorize the code, and if you don't fully understand the code, *that is fine!*

You should, however, be able to describe how quick sort works from a high level, and be able to draw out the steps if given an example array
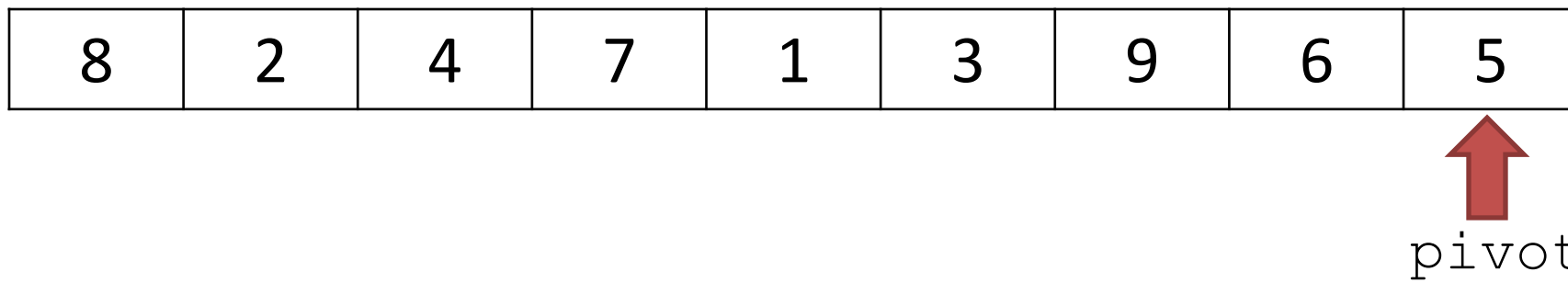
You should also know the time complexity
of the sorting algorithms that we talk about

8	2	4	7	1	3	9	6	5

The first step of Quick Sort is to select a pivot, and to get the pivot sorted to is correct position

→ This step is known as **partitioning**

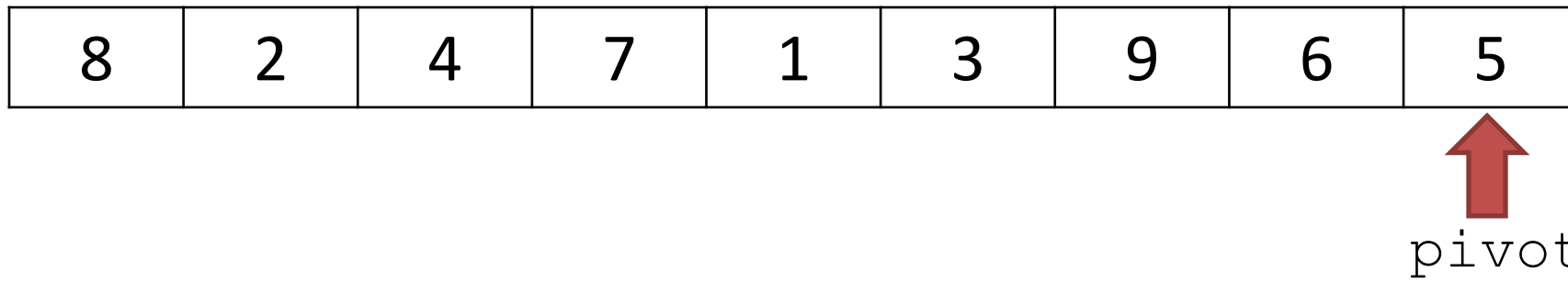| 8	2	4	7	1	3	9	6	5

pivot

The first step of Quick Sort is to select a pivot, and to get the pivot sorted to is correct position
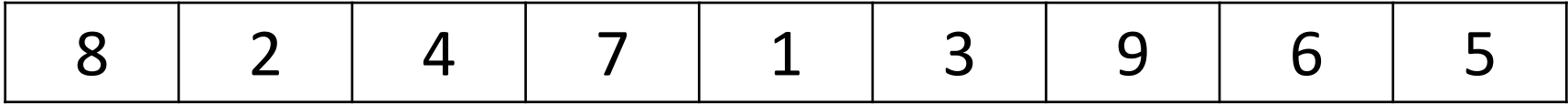
→ This step is known as **partitioning**

There are many ways to select a pivot, but to keep things simple, the last element of the array will be the pivot
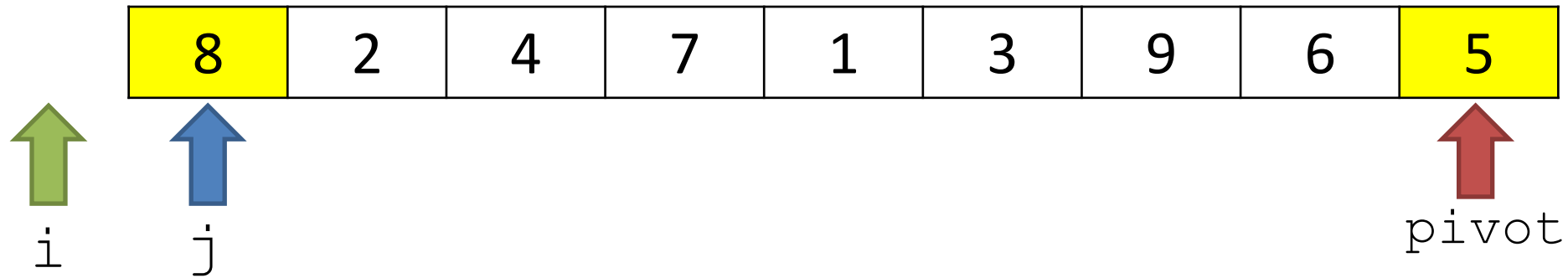
8	2	4	7	1	3	9	6	5

↑
pivot

We are going to define two pointers `i` and `j`, that will help us get the pivot sorted correctly

8	2	4	7	1	3	9	6	5

i   j
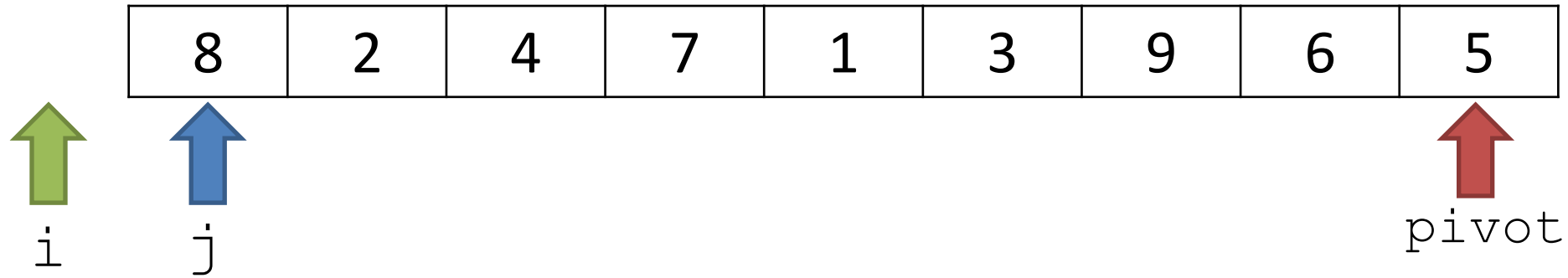
pivot

`j` will be defined to be the starting point of the array (0),
and `i` will be defined to be `(j-1)`

8	2	4	7	1	3	9	6	5

i j

pivot

`j` will now iterate through the array until it reaches the `pivot`

8	2	4	7	1	3	9	6	5

i   j   pivot

`j` will now iterate through the array until it reaches the `pivot`

We will check if index `j` is less than the pivot

8	2	4	7	1	3	9	6	5

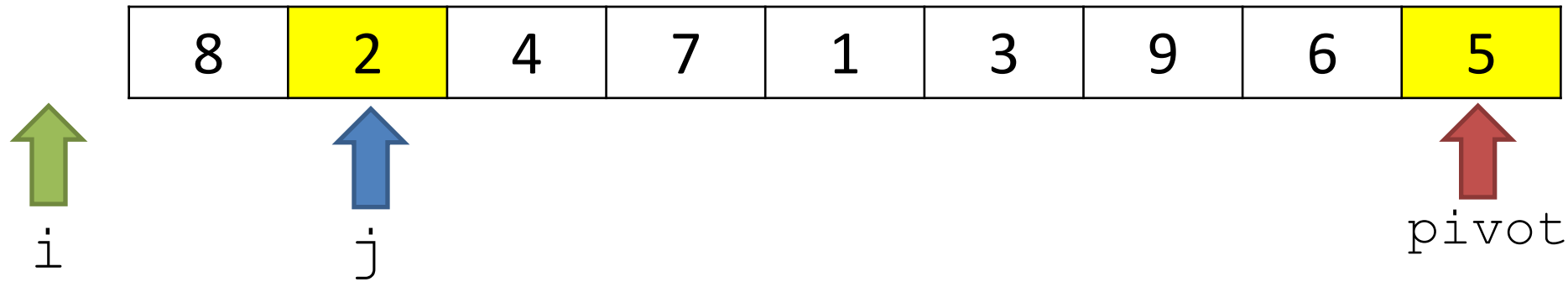i   j   pivot

`j` will now iterate through the array until it reaches the `pivot`

We will check if index `j` is less than the pivot

If so, we will increase `i` by 1, and then swap the elements located at index `i` and `j`

If not, increase `j` by 1

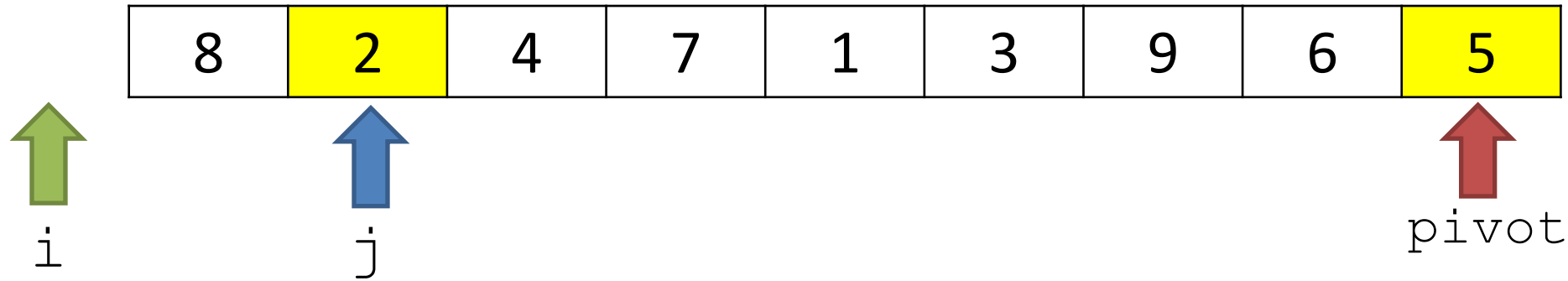| 8	2	4	7	1	3	9	6	5

i

j

pivot

j will now iterate through the array until it reaches the pivot

We will check if index j is less than the pivot

If so, we will increase i by 1, and then swap the elements located at index i and j

If not, increase j by 1

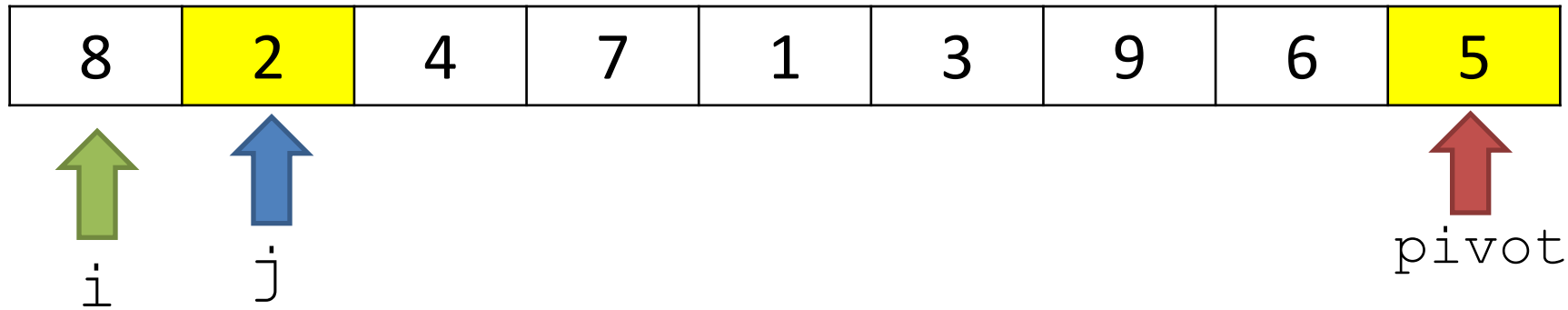| 8	2	4	7	1	3	9	6	5

i   j   pivot

2 is less than 5, so we increase `i` and then swap!

`j` will now iterate through the array until it reaches the `pivot`

We will check if index `j` is less than the pivot

If so, we will increase `i` by 1, and then swap the elements located at index `i` and `j`

If not, increase `j` by 1

8	2	4	7	1	3	9	6	5

i

j

pivot

2 is less than 5, so we increase i and then swap!

`j` will now iterate through the array until it reaches the `pivot`

We will check if index `j` is less than the pivot

If so, we will increase `i` by 1, and then swap the elements located at index `i` and `j`

If not, increase `j` by 1

2	8	4	7	1	3	9	6	5

i　　j　　　　　　　　　　　　　　　　pivot

2 is less than 5, so we increase i and then swap!

j will now iterate through the array until it reaches the pivot

We will check if index j is less than the pivot

If so, we will increase i by 1, and then swap the elements located at index i and j
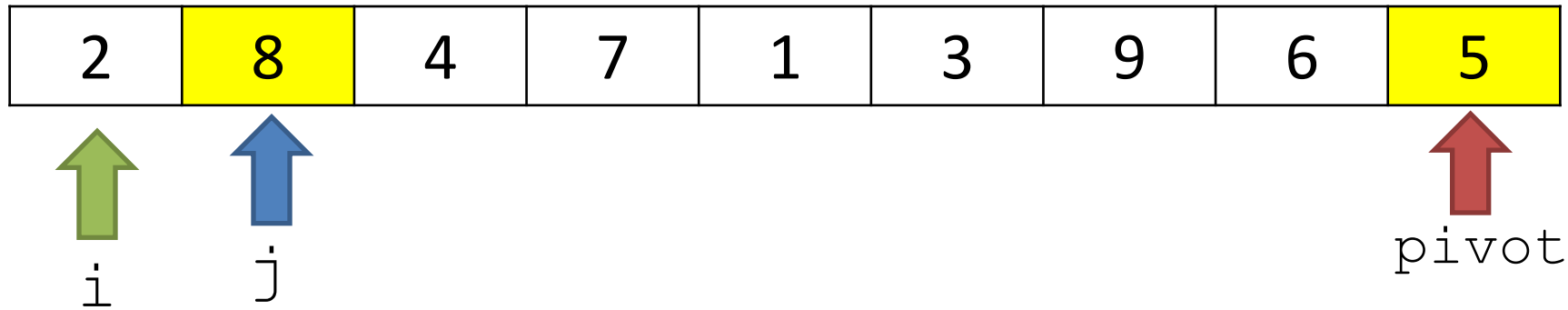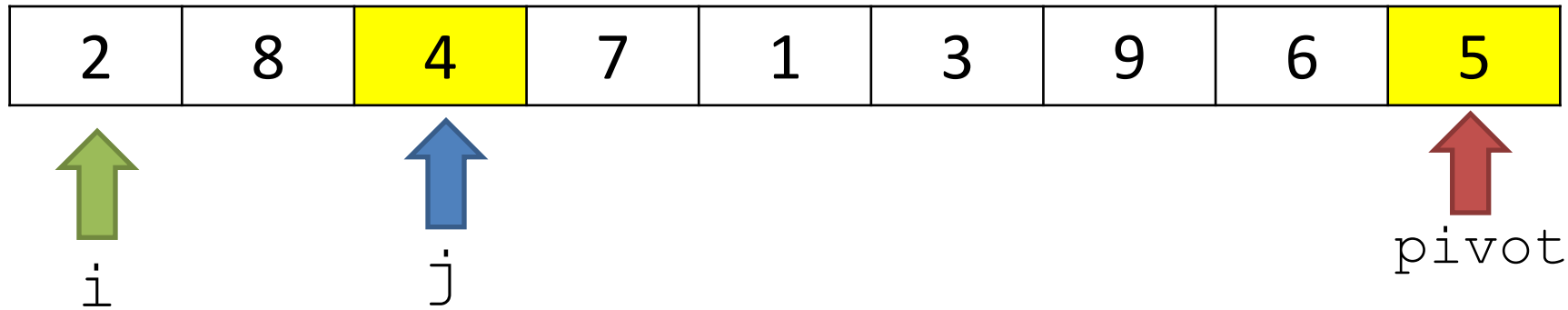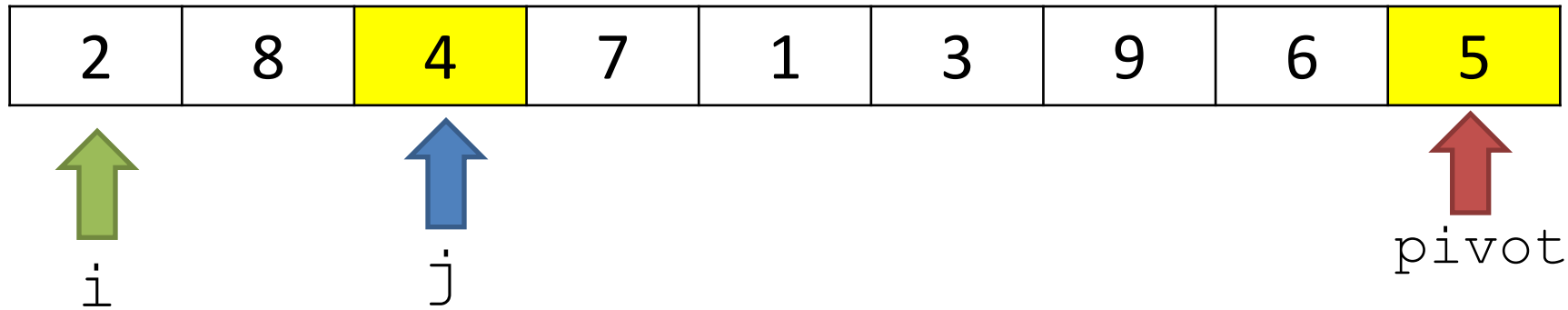
If not, increase j by 1
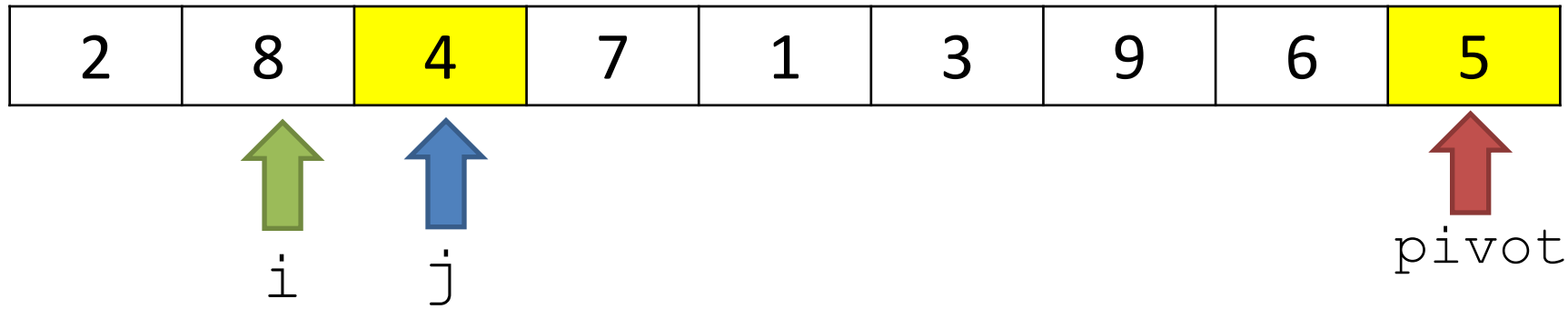
j will now iterate through the array until it reaches the `pivot`

We will check if index j is less than the pivot

If so, we will increase i by 1, and then swap the elements located at index i and j

If not, increase j by 1

2	8	4	7	1	3	9	6	5

i   j   pivot

4 is less than 5, so we increase `i` and then swap!

`j` will now iterate through the array until it reaches the `pivot`

We will check if index `j` is less than the pivot

If so, we will increase `i` by 1, and then swap the elements located at index `i` and `j`

If not, increase `j` by 1

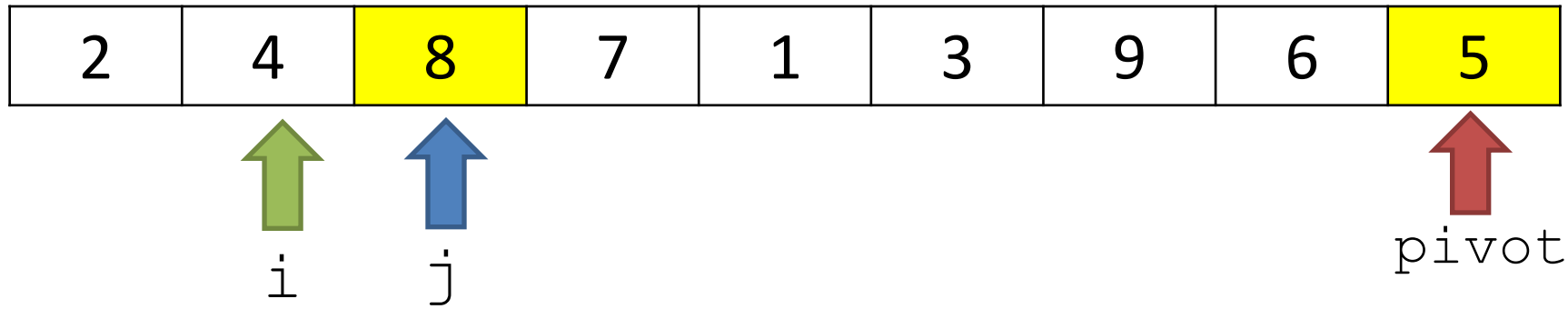| 2	8	4	7	1	3	9	6	5

i   j

pivot

4 is less than 5, so we increase `i` and then swap!

`j` will now iterate through the array until it reaches the `pivot`

We will check if index `j` is less than the pivot

If so, we will increase `i` by 1, and then swap the elements located at index `i` and `j`

If not, increase `j` by 1

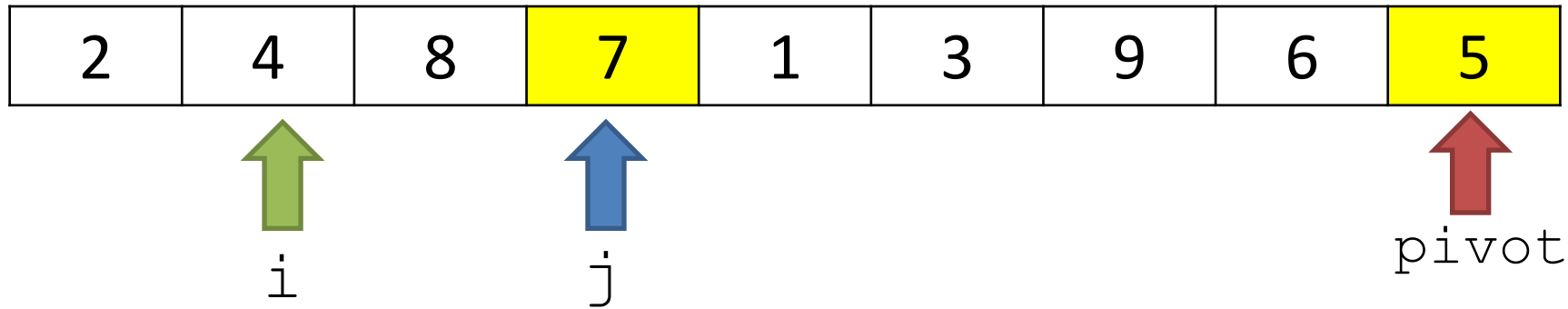| 2	4	8	7	1	3	9	6	5

i   j

pivot

4 is less than 5, so we increase `i` and then swap!

`j` will now iterate through the array until it reaches the `pivot`

We will check if index `j` is less than the pivot

If so, we will increase `i` by 1, and then swap the elements located at index `i` and `j`

If not, increase `j` by 1

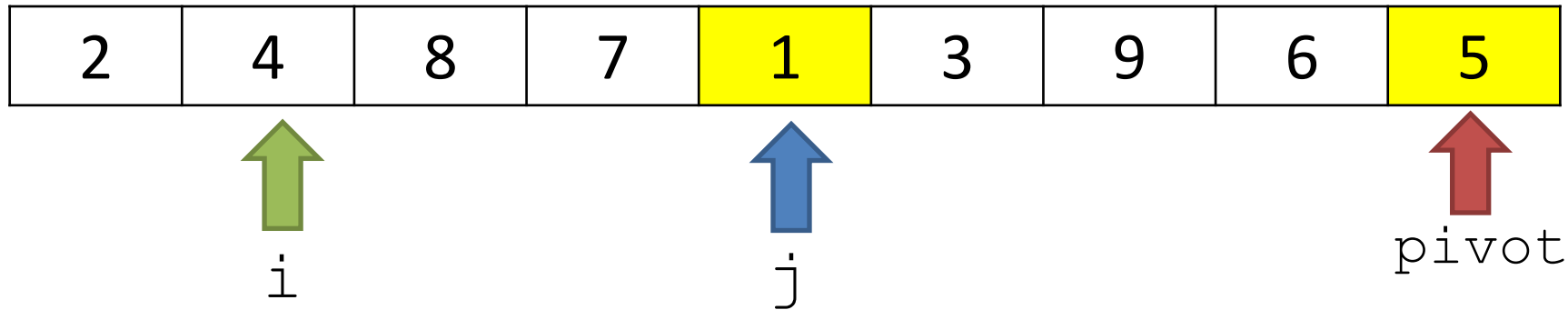| 2	4	8	7	1	3	9	6	5

i   j   pivot

`j` will now iterate through the array until it reaches the `pivot`

We will check if index `j` is less than the pivot

If so, we will increase `i` by 1, and then swap the elements located at index `i` and `j`

If not, increase `j` by 1

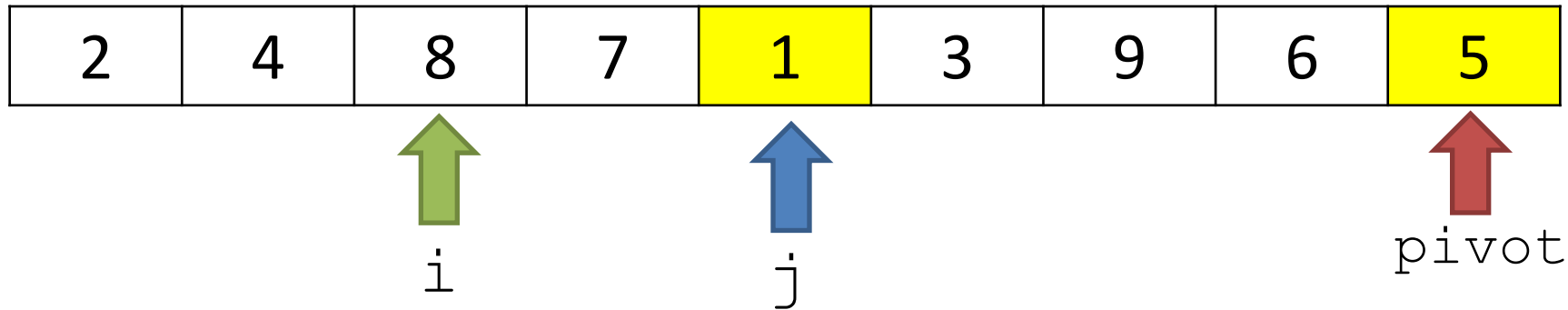| 2	4	8	7	1	3	9	6	5

i   j   pivot

`j` will now iterate through the array until it reaches the `pivot`

We will check if index `j` is less than the pivot

If so, we will increase `i` by 1, and then swap the elements located at index `i` and `j`

If not, increase `j` by 1

2	4	8	7	1	3	9	6	5

i   j   pivot
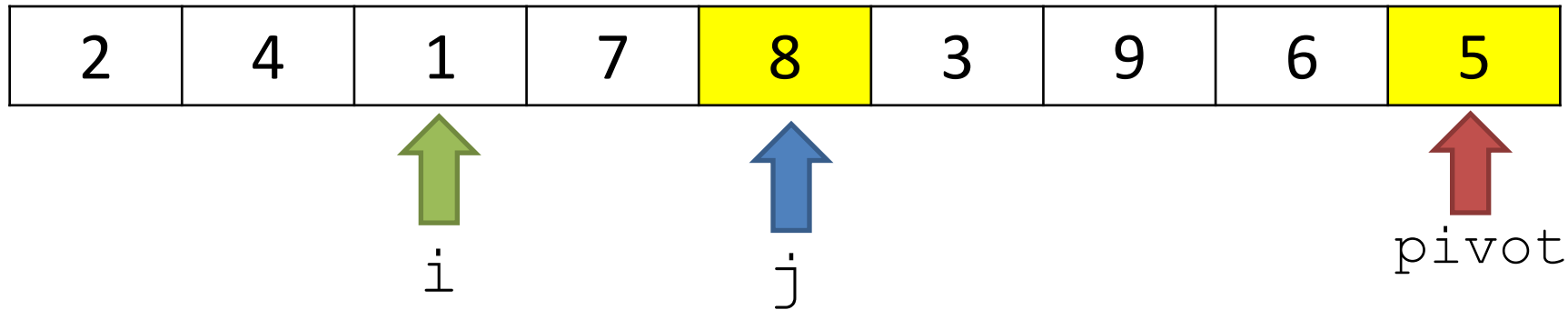
`j` will now iterate through the array until it reaches the `pivot`

We will check if index `j` is less than the pivot

If so, we will increase `i` by 1, and then swap the elements located at index `i` and `j`

If not, increase `j` by 1

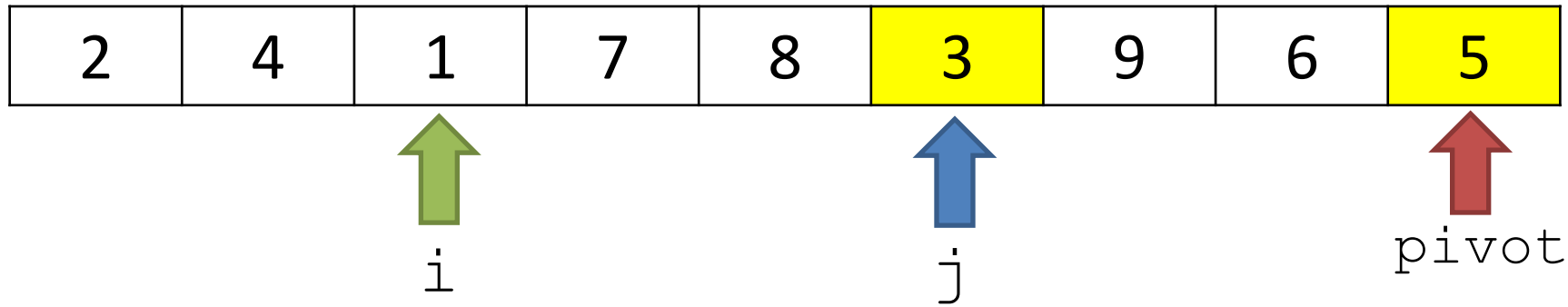| 2	4	1	7	8	3	9	6	5

i   j   pivot

`j` will now iterate through the array until it reaches the `pivot`

We will check if index `j` is less than the pivot

If so, we will increase `i` by 1, and then swap the elements located at index `i` and `j`

If not, increase `j` by 1

| 2 | 4 | 1 | 7 | 8 | 3 | 9 | 6 | 5 |

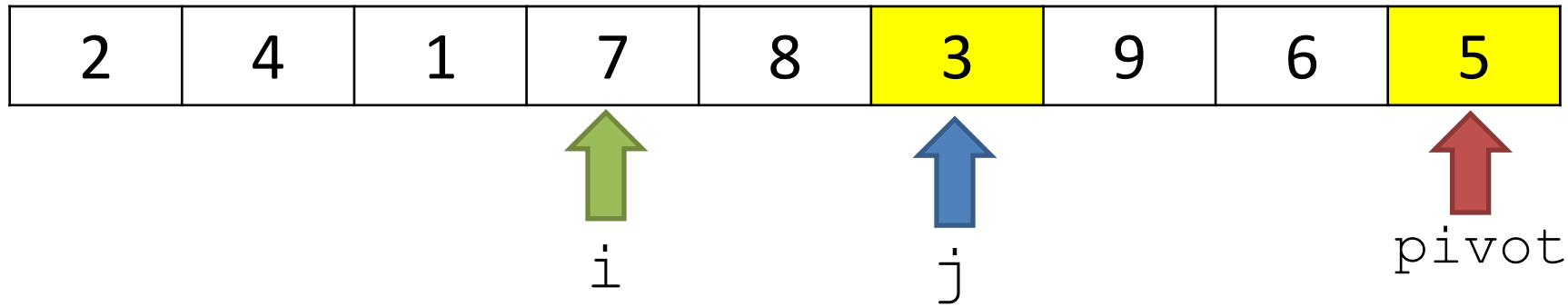i points to 1, j points to 3, pivot points to 5.

j will now iterate through the array until it reaches the `pivot`

We will check if index j is less than the pivot

If so, we will increase i by 1, and then swap the elements located at index i and j

If not, increase j by 1

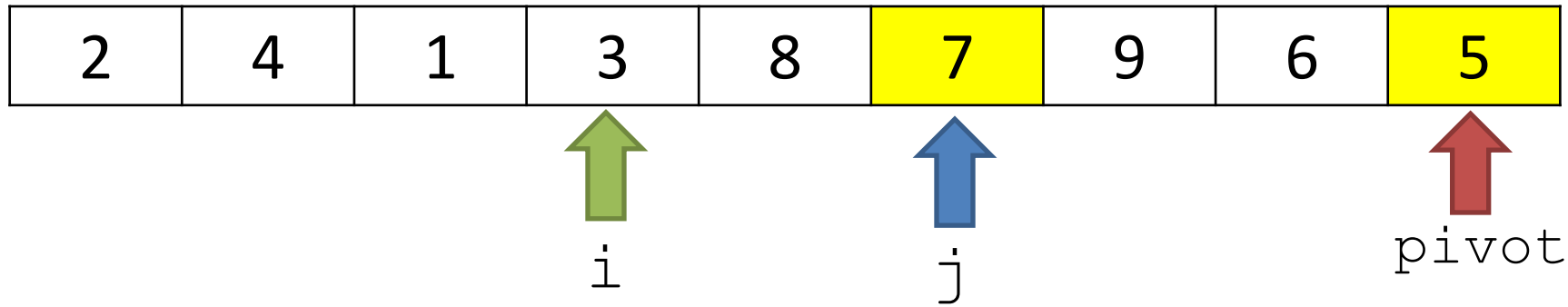| 2 | 4 | 1 | 7 | 8 | 3 | 9 | 6 | 5 |

i   j   pivot

`j` will now iterate through the array until it reaches the `pivot`

We will check if index `j` is less than the pivot

If so, we will increase `i` by 1, and then swap the elements located at index `i` and `j`

If not, increase `j` by 1

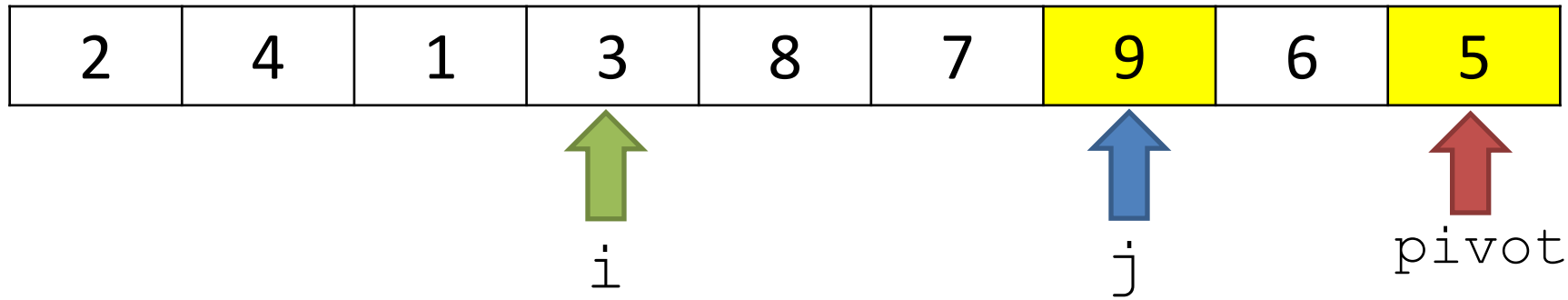| 2	4	1	3	8	7	9	6	5

i   j   pivot

`j` will now iterate through the array until it reaches the `pivot`

We will check if index `j` is less than the pivot

If so, we will increase `i` by 1, and then swap the elements located at index `i` and `j`

If not, increase `j` by 1

2	4	1	3	8	7	9	6	5

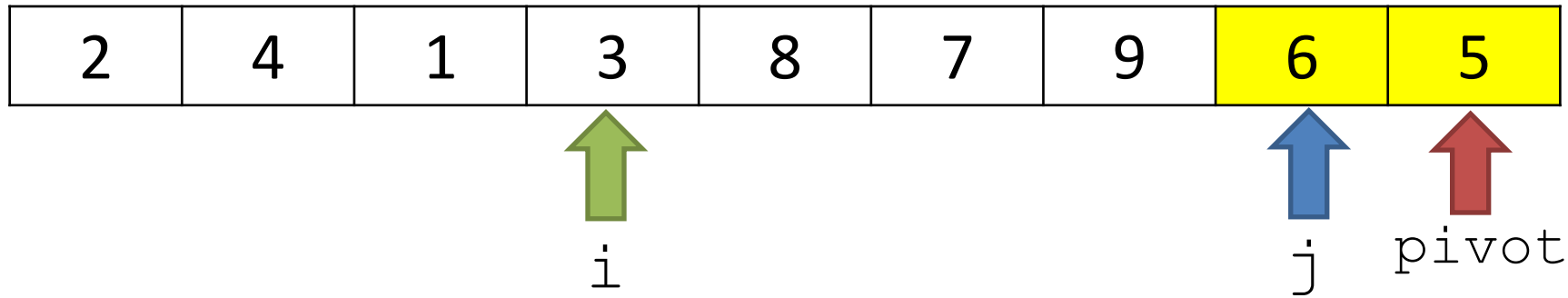i   j   pivot

`j` will now iterate through the array until it reaches the `pivot`

We will check if index `j` is less than the pivot

If so, we will increase `i` by 1, and then swap the elements located at index `i` and `j`

If not, increase `j` by 1

2	4	1	3	8	7	9	6	5

i   j   pivot

`j` will now iterate through the array until it reaches the `pivot`

We will check if index `j` is less than the pivot

If so, we will increase `i` by 1, and then swap the elements located at index `i` and `j`

If not, increase `j` by 1

2	4	1	3	8	7	9	6	5

i

j pivot

j will now iterate through the array until it reaches the pivot

We will check if index j is less than the pivot

If so, we will increase i by 1, and then swap the elements located at index i and j
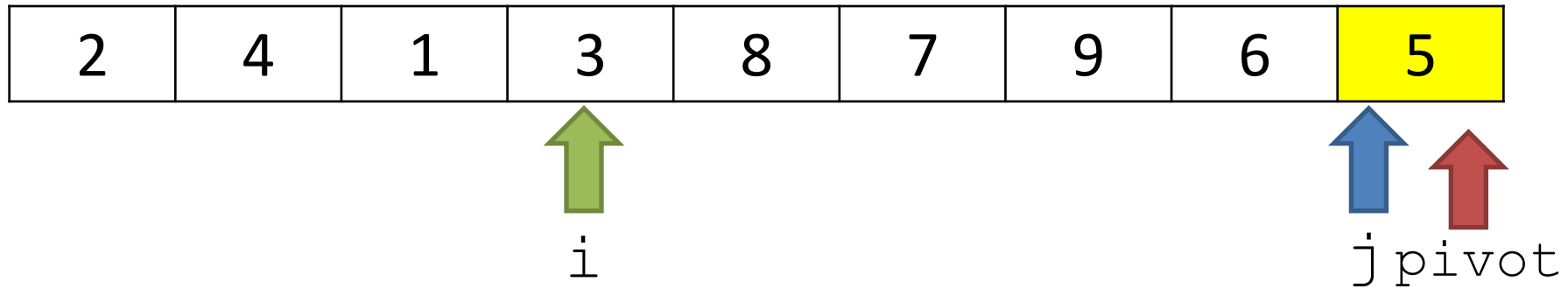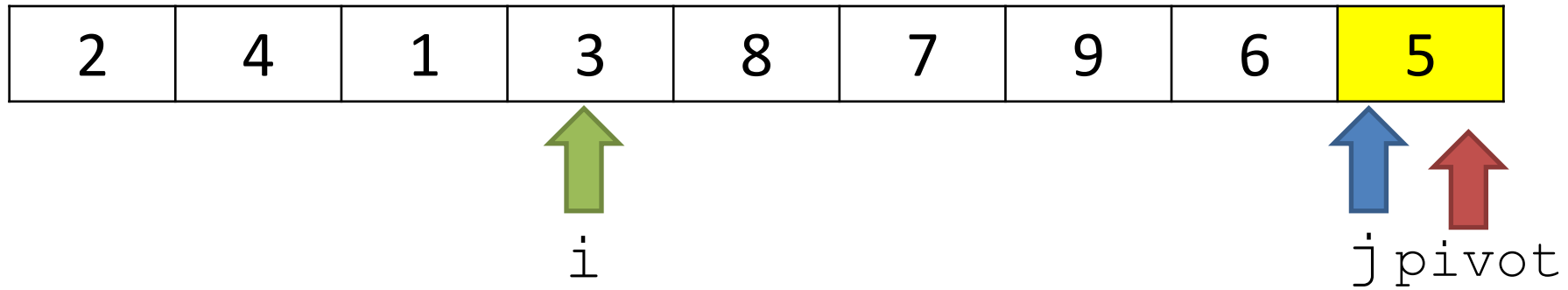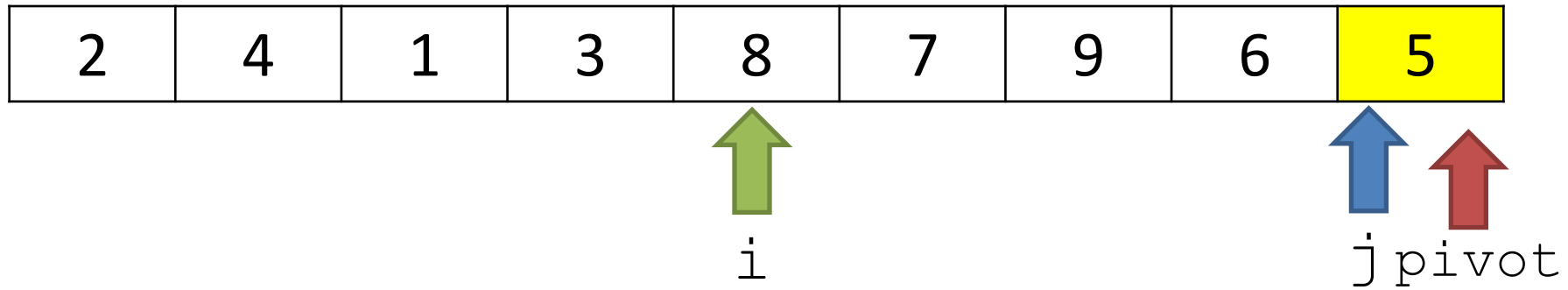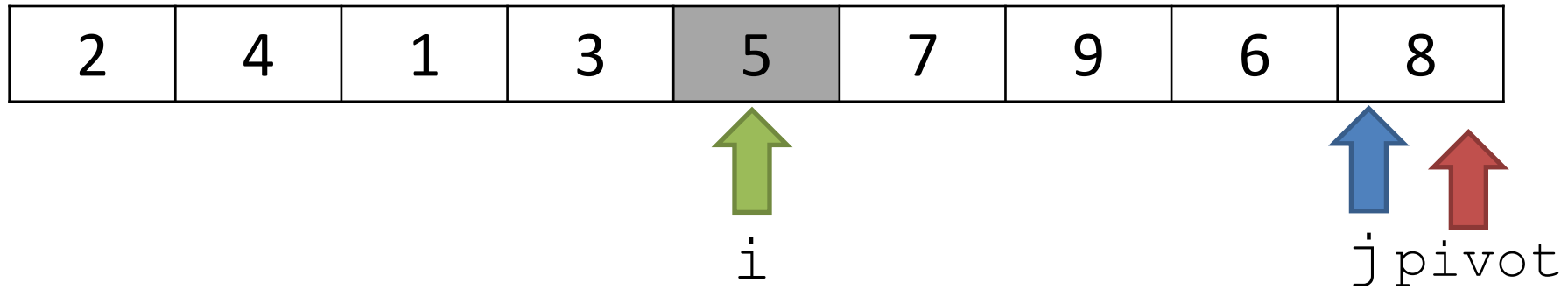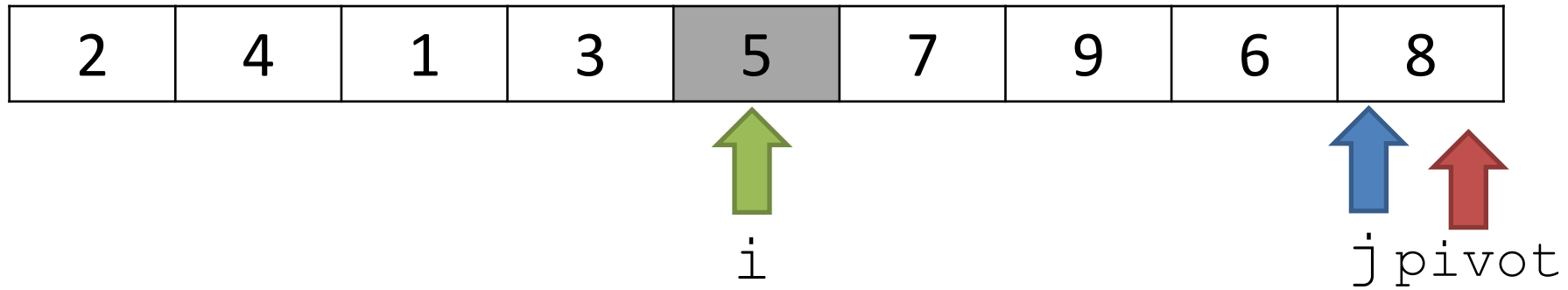
If not, increase j by 1

2	4	1	3	8	7	9	6	5

i   j pivot

Once `j` reaches the pivot, we will increase `i` by 1, and
then swap the pivot with the element located at index `i`

2	4	1	3	8	7	9	6	5

i

j pivot

Once `j` reaches the pivot, we will increase `i` by 1, and then swap the pivot with the element located at index `i`

2	4	1	3	5	7	9	6	8

i

j pivot

Once j reaches the pivot, we will increase i by 1, and then swap the pivot with the element located at index i

2	4	1	3	5	7	9	6	8

i

j pivot

Once `j` reaches the pivot, we will increase `i` by 1, and then swap the pivot with the element located at index `i`

???

2	4	1	3	5	7	9	6	8

i   j pivot

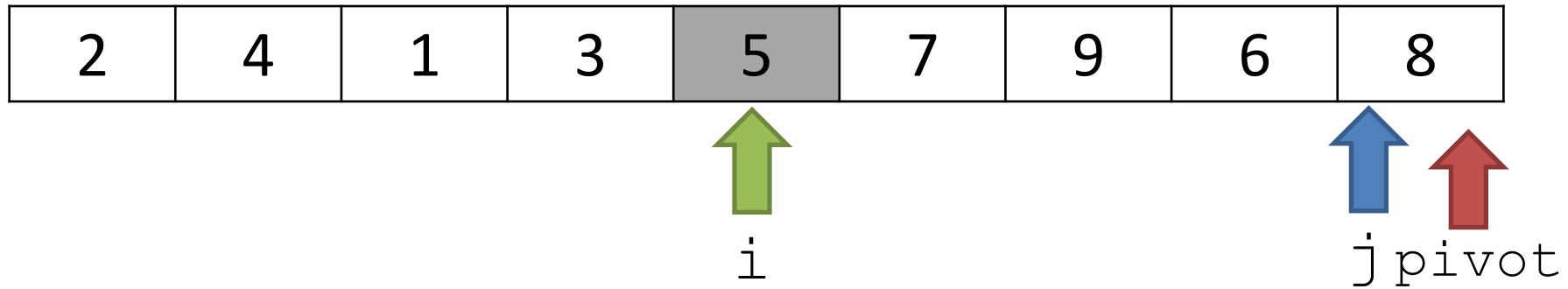Once `j` reaches the pivot, we will increase `i` by 1, and then swap the pivot with the element located at index `i`

Our pivot is now in the correct spot!

Everything to the left is less than the pivot, everything to the right is greater than the pivot

2	4	1	3	5	7	9	6	8

This step is known as **partitioning**

| 2 | 4 | 1 | 3 | 5 | 7 | 9 | 6 | 8 |

Now, we will recursively call `quick_sort` on the left section of the array, and the right section of the array

| 2 | 4 | 1 | 3 | 5 | 7 | 9 | 6 | 8 |

Now, we will recursively call `quick_sort` on the left section of the array, and the right section of the array

quick_sort(| 2 | 4 | 1 | 3 |)   quick_sort(| 7 | 9 | 6 | 8 |)

2	4	1	3	5	7	9	6	8

Now, we will recursively call `quick_sort` on the left section of the array, and the right section of the array

quick_sort(

2	4	1	3

)   quick_sort(

7	9	6	8

)

Unlike Merge Sort, these are **not** new arrays, these are just "sections" of the original array

2	4	1	3	5	7	9	6	8

Now, we will recursively call `quick_sort` on the left section of the array, and the right section of the array

quick_sort(| 2 | 4 | 1 | 3 |)   quick_sort(| 7 | 9 | 6 | 8 |)

Unlike Merge Sort, these are **not** new arrays, these are just "sections" of the original array

Due to how we call our recursive methods, we will always prioritize the "left tree" of the array

2	4	1	3	5	7	9	6	8

quick_sort(| 2 | 4 | 1 | 3 |)
|---|---|---|---|

Now we partition this section!

2	4	1	3	5	7	9	6	8

quick_sort(| 2 | 4 | 1 | 3 |)

i   j   pivot

2	4	1	3	5	7	9	6	8

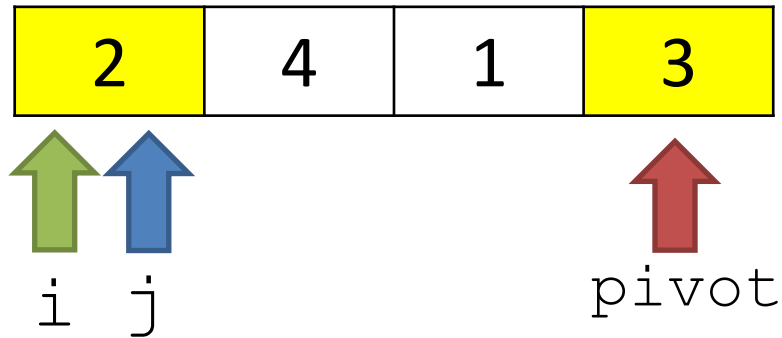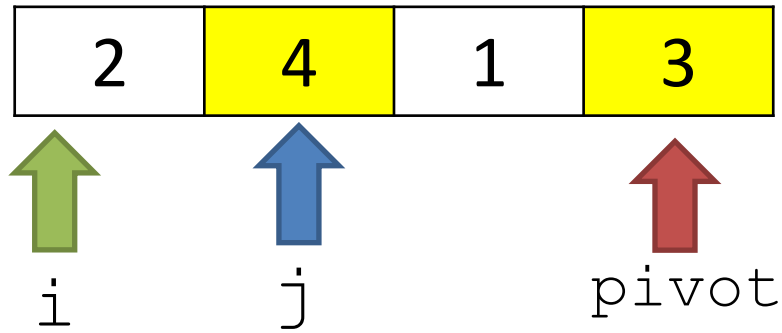| 2	4	1	3

i j

pivot

j will now iterate through the array until it reaches the `pivot`

We will check if index j is less than the pivot

If so, we will increase i by 1, and then swap the elements located at index i and j

If not, increase j by 1

2	4	1	3	5	7	9	6	8

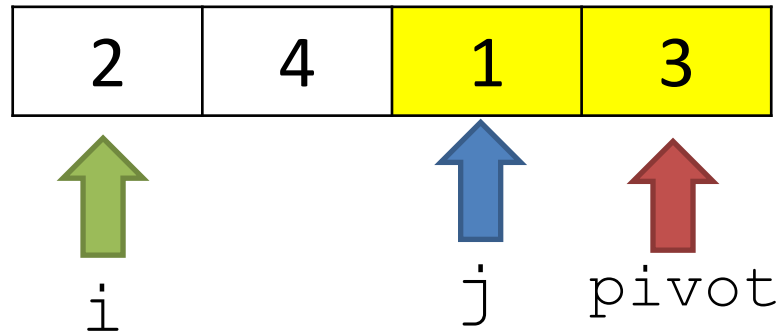| 2	4	1	3

i j   pivot

`j` will now iterate through the array until it reaches the `pivot`

We will check if index `j` is less than the pivot

If so, we will increase `i` by 1, and then swap the elements located at index `i` and `j`

If not, increase `j` by 1

2	4	1	3	5	7	9	6	8

2	4	1	3

i   j   pivot

`j` will now iterate through the array until it reaches the `pivot`

We will check if index `j` is less than the pivot

    If so, we will increase `i` by 1, and then swap the elements located at index `i` and `j`

    If not, increase `j` by 1

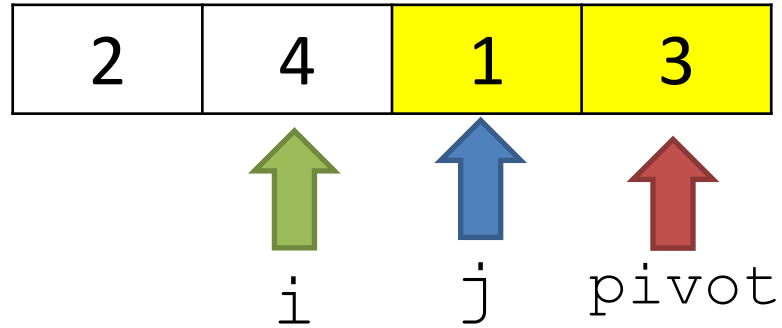| 2 | 4 | 1 | 3 | 5 | 7 | 9 | 6 | 8 |

| 2 | 4 | 1 | 3 |

i   j   pivot

`j` will now iterate through the array until it reaches the `pivot`

We will check if index `j` is less than the pivot

    If so, we will increase `i` by 1, and then swap the elements located at index `i` and `j`

    If not, increase `j` by 1

2	4	1	3	5	7	9	6	8

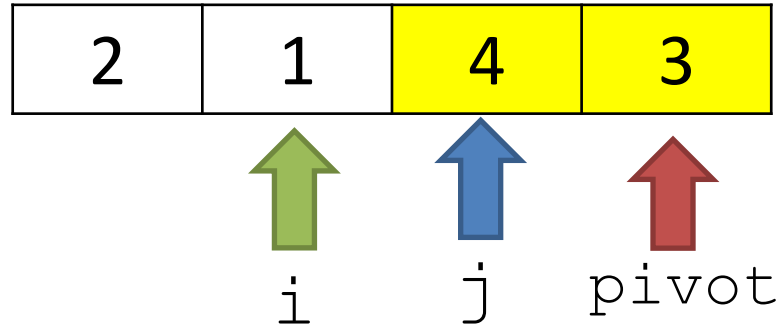| 2	4	1	3

i   j   pivot

j will now iterate through the array until it reaches the $pivot$

We will check if index j is less than the pivot

    If so, we will increase i by 1, and then swap the elements located at index i and j

    If not, increase j by 1

2	4	1	3	5	7	9	6	8

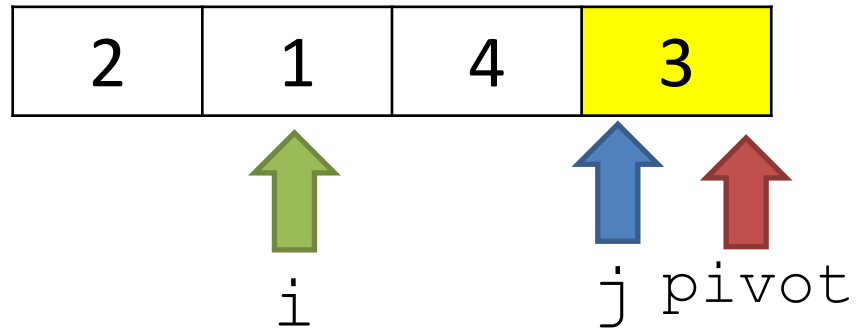| 2	1	4	3

i   j   pivot

`j` will now iterate through the array until it reaches the `pivot`

We will check if index `j` is less than the pivot

    If so, we will increase `i` by 1, and then swap the elements located at index `i` and `j`

    If not, increase `j` by 1

2	4	1	3	5	7	9	6	8

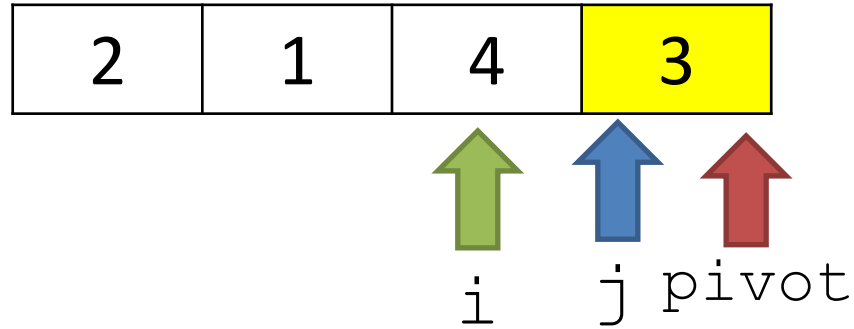| 2	1	4	3

i   j pivot

j will now iterate through the array until it reaches the $pivot$

We will check if index j is less than the pivot

If so, we will increase i by 1, and then swap the elements located at index i and j

If not, increase j by 1

2	4	1	3	5	7	9	6	8

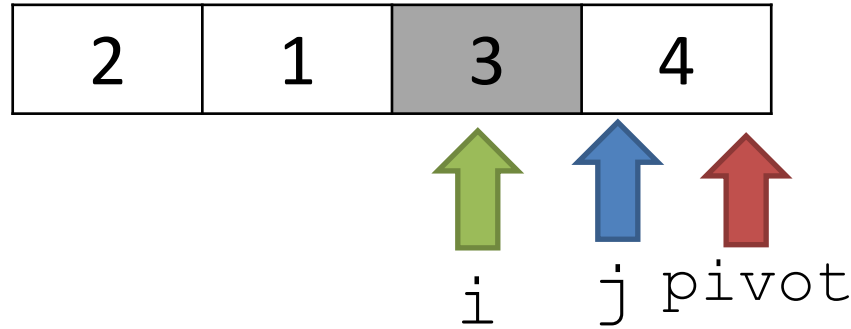| 2	1	4	3

i   j pivot

`j` will now iterate through the array until it reaches the `pivot`

We will check if index `j` is less than the pivot

    If so, we will increase `i` by 1, and then swap the elements located at index `i` and `j`

    If not, increase `j` by 1

2	4	1	3	5	7	9	6	8

2	1	3	4

i   j pivot

j will now iterate through the array until it reaches the `pivot`

We will check if index j is less than the pivot

    If so, we will increase i by 1, and then swap the elements located at index i and j

    If not, increase j by 1

2	4	1	3	5	7	9	6	8

2	1	3	4

2	1

4

2	4	1	3	5	7	9	6	8

2	1	3	4

2	1

4

i j pivot

2	4	1	3	5	7	9	6	8

2	1	3	4

2	1		4

i   j pivot

2	4	1	3	5	7	9	6	8

2	1	3	4

2	1		4
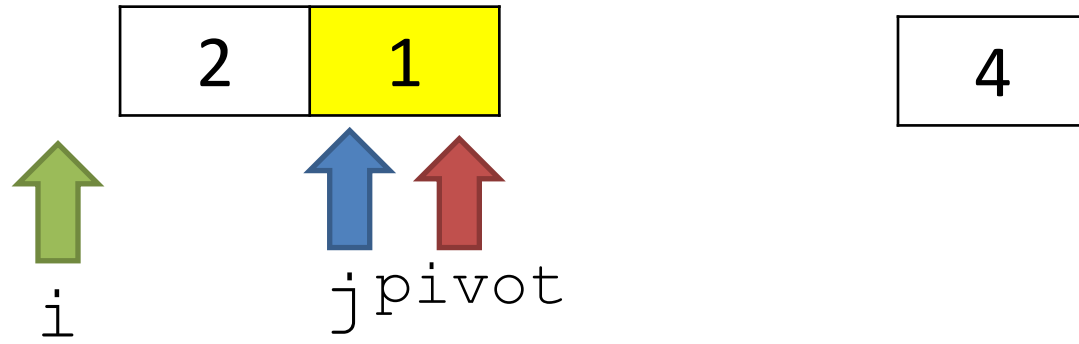
i j pivot

2	4	1	3	5	7	9	6	8

2	1	3	4

1	2		4

i j pivot

2	4	1	3	5	7	9	6	8

2	1	3	4

1	2

4

2	4	1	3	5	7	9	6	8

2	1	3	4

1	2

4

2

If the size of our "array section" is 1, then it's already sorted!

| 2 | 4 | 1 | 3 | 5 | 7 | 9 | 6 | 8 |

| 2 | 1 | 3 | 4 |

| 1 | 2 | | 4 |

| 2 |

If the size of our "array section" is 1, then it's already sorted!

2	4	1	3	5	7	9	6	8

2	1	3	4

1	2

4

If the size of our "array section" is 1, then it's already sorted!

2	4	1	3	5	7	9	6	8

1	2	3	4

4

If the size of our "array section" is 1, then it's already sorted!

2	4	1	3	5	7	9	6	8

1	2	3	4

4

If the size of our "array section" is 1, then it's already sorted!

2	4	1	3	5	7	9	6	8

1	2	3	4

If the size of our "array section" is 1, then it's already sorted!

1	2	3	4	5	7	9	6	8

quick_sort(| 2 | 4 | 1 | 3 |)

quick_sort(| 7 | 9 | 6 | 8 |)

1	2	3	4	5	7	9	6	8

quick_sort(

7	9	6	8

)

1	2	3	4	5	7	9	6	8

7	9	6	8

i   j   pivot

1	2	3	4	5	7	9	6	8

7	9	6	8

i   j   pivot

| 1 | 2 | 3 | 4 | 5 | 7 | 9 | 6 | 8 |

| 7 | 9 | 6 | 8 |

i   j   pivot

1	2	3	4	5	7	9	6	8

7	9	6	8

i   j   pivot

1	2	3	4	5	7	9	6	8

7	6	9	8

i   j   pivot

1	2	3	4	5	7	9	6	8

7	6	9	8

i   j pivot

1	2	3	4	5	7	9	6	8

7	6	9	8

i   j pivot

1	2	3	4	5	7	9	6	8

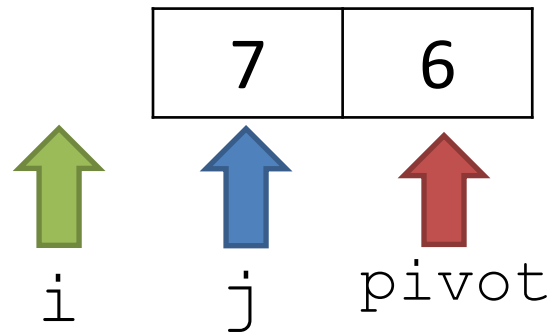| 7	6	8	9

7	6

9

Call quick sort, and give it the section of the array to the left of the pivot, and to the right of the pivot

1	2	3	4	5	7	9	6	8

7	6	8	9

7	6

9

i   j   pivot

1	2	3	4	5	7	9	6	8

7	6	8	9

7	6

9

i   j   pivot

1	2	3	4	5	7	9	6	8

7	6	8	9

7	6		9

i   j pivot

| 1 | 2 | 3 | 4 | 5 | 7 | 9 | 6 | 8 |

| 7 | 6 | 8 | 9 |

| 6 | 7 |

| 9 |

i j pivot

1	2	3	4	5	7	9	6	8

7	6	8	9

6	7

9

1	2	3	4	5	7	9	6	8

6	7	8	9

9

1	2	3	4	5	7	9	6	8

6	7	8	9

9

1	2	3	4	5	7	9	6	8

6	7	8	9

1	2	3	4	5	6	7	8	9

All done!!

1	2	3	4	5	6	7	8	9

All done!!

Let's code this!!!

```java
public static int[] quick_sort(int[] array, int start, int end) {
    if(end <= start) { //base case: array is size 1 or nothing
        return array;
    }
    int pivot = partition(array, start, end);
    quick_sort(array, start, pivot-1);
    quick_sort(array,pivot + 1, end);
    return array;
}
```

```java
public static int partition(int[] array, int start, int end) {
    int pivot = array[end];
    int i = start - 1;
    for(int j = start; j <= end - 1; j++) {
        if(array[j] < pivot) {
            i++;
            int temp = array[i];
            array[i] = array[j];
            array[j] = temp;
        }
    }
}
```

```java
public static int[] quick_sort(int[] array, int start, int end) {
    if(end <= start) { //base case: array is size 1 or nothing
        return array;
    }
    int pivot = partition(array, start, end);
    quick_sort(array, start, pivot-1);
    quick_sort(array,pivot + 1, end);
    return array;
}
```

```java
public static int partition(int[] array, int start, int end) {
    int pivot = array[end];
    int i = start - 1;
    for(int j = start; j <= end - 1; j++) {
        if(array[j] < pivot) {
            i++;
            int temp = array[i];
            array[i] = array[j];
            array[j] = temp;
        }
    }
}
```

Running time?

```java
public static int[] quick_sort(int[] array, int start, int end) {
    if(end <= start) { //base case: array is size 1 or nothing
        return array;
    }
    int pivot = partition(array, start, end);
    quick_sort(array, start, pivot-1);
    quick_sort(array,pivot + 1, end);
    return array;
}
```

```java
public static int partition(int[] array, int start, int end) {
    int pivot = array[end]; O(1)
    int i = start - 1; O(1)
    for(int j = start; j <= end - 1; j++) {  O(n)
        if(array[j] < pivot) { O(1)
            i++; O(1)
            int temp = array[i]; O(1)
            array[i] = array[j]; O(1)
            array[j] = temp;O(1)
        }
    }
}
```

```java
public static int[] quick_sort(int[] array, int start, int end) {
    if(end <= start) { //base case: array is size 1 or nothing
        return array;
    }
    int pivot = partition(array, start, end);
    quick_sort(array, start, pivot-1);
    quick_sort(array,pivot + 1, end);
    return array;
}
```

```java
public static int partition(int[] array, int start, int end) {
    int pivot = array[end]; O(1)
    int i = start - 1; O(1)
    for(int j = start; j <= end - 1; j++) {  O(n)
      if(array[j] < pivot) { O(1)
        i++; O(1)
        int temp = array[i]; O(1)
        array[i] = array[j]; O(1)
        array[j] = temp;O(1)
      }
    }
}
```

Running time of `partition` subroutine = **O(n)** where n= # of elements in array

```java
public static int[] quick_sort(int[] array, int start, int end) {
    if(end <= start) { //base case: array is size 1 or nothing   O(1)
        return array; O(1)
    }
    int pivot = partition(array, start, end); O(n)
    quick_sort(array, start, pivot-1); O(1)
    quick_sort(array,pivot + 1, end); O(1)
    return array; O(1)
}
```

Running time of quick sort method = **O(n)**
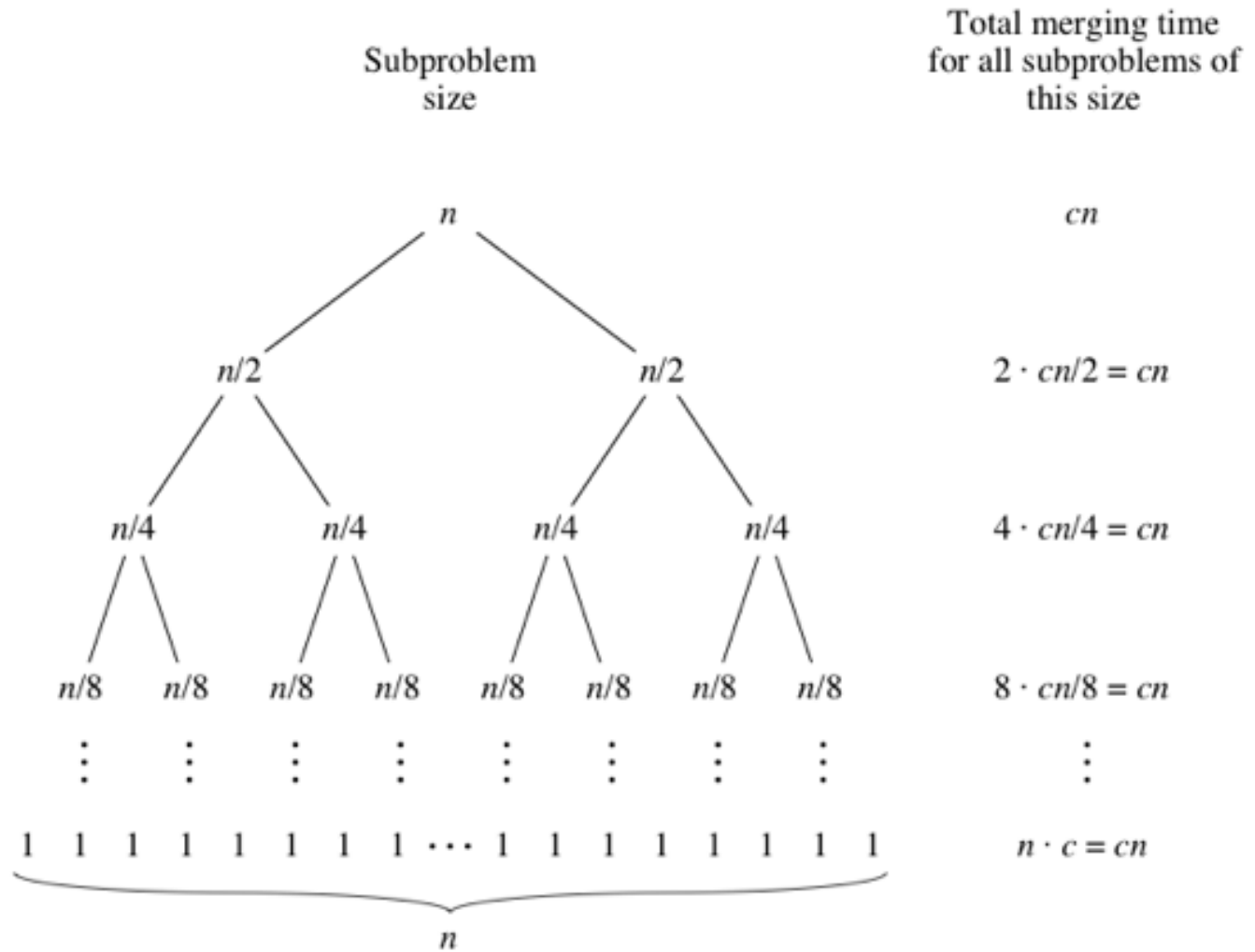
```java
public static int partition(int[] array, int start, int end) {
    int pivot = array[end]; O(1)
    int i = start - 1; O(1)
    for(int j = start; j <= end - 1; j++) {  O(n)
        if(array[j] < pivot) { O(1)
            i++; O(1)
            int temp = array[i]; O(1)
            array[i] = array[j]; O(1)
            array[j] = temp; O(1)
        }
    }
}
```
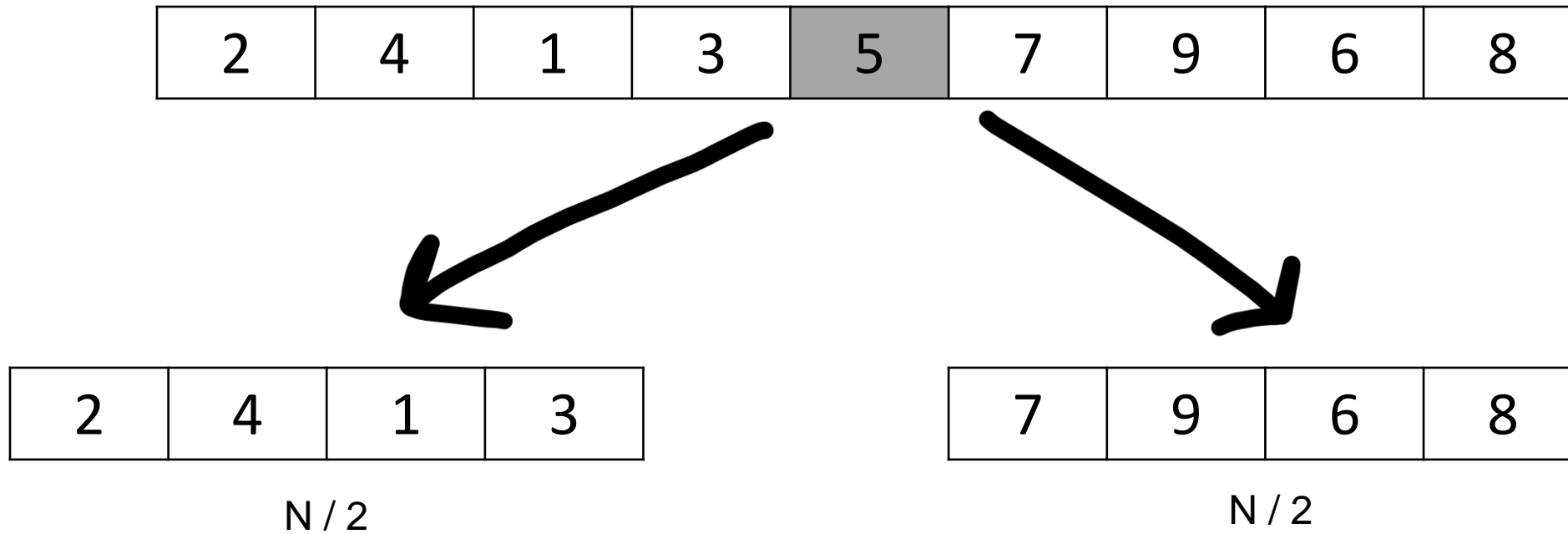
Running time of `partition` subroutine = **O(n)** where n= # of elements in array

We must now evaluate how often we recursively call the method, and the size of the problem we give that method

Reminder: Recursion tree for **merge sort**

Subproblem size

Total merging time for all subproblems of this size

n cn

$n/2$ $n/2$ $2 \cdot cn/2 = cn$

$n/4$ $n/4$ $n/4$ $n/4$ $4 \cdot cn/4 = cn$

$n/8$ $n/8$ $n/8$ $n/8$ $n/8$ $n/8$ $n/8$ $n/8$ $8 \cdot cn/8 = cn$

1 1 1 1 1 1 1 1 $\cdots$ 1 1 1 1 1 1 1 1 $n \cdot c = cn$

n

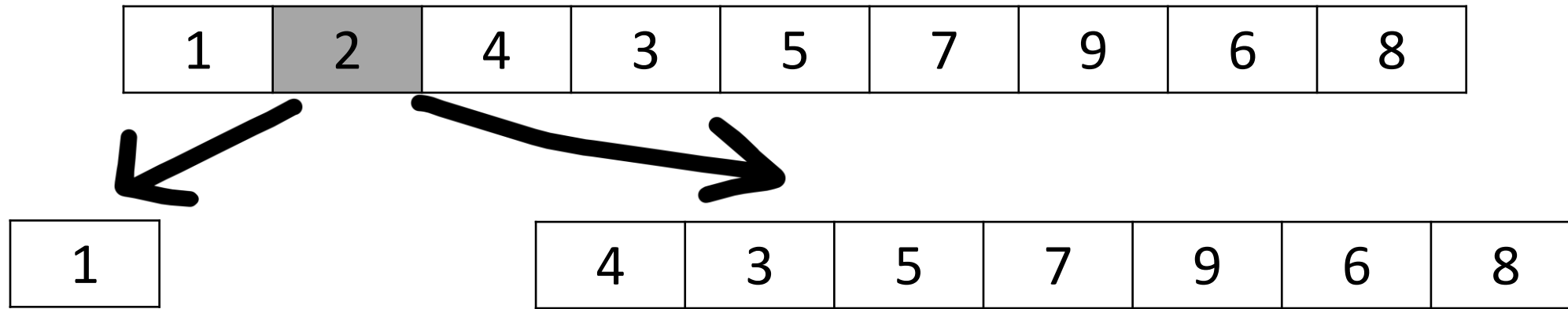| 2 | 4 | 1 | 3 | 5 | 7 | 9 | 6 | 8 |

| 2 | 4 | 1 | 3 |

N / 2

| 7 | 9 | 6 | 8 |

N / 2

Ideally, we want **balanced** partitions.

That way we are dividing the problem size by 2 (which gives us log n running time!)
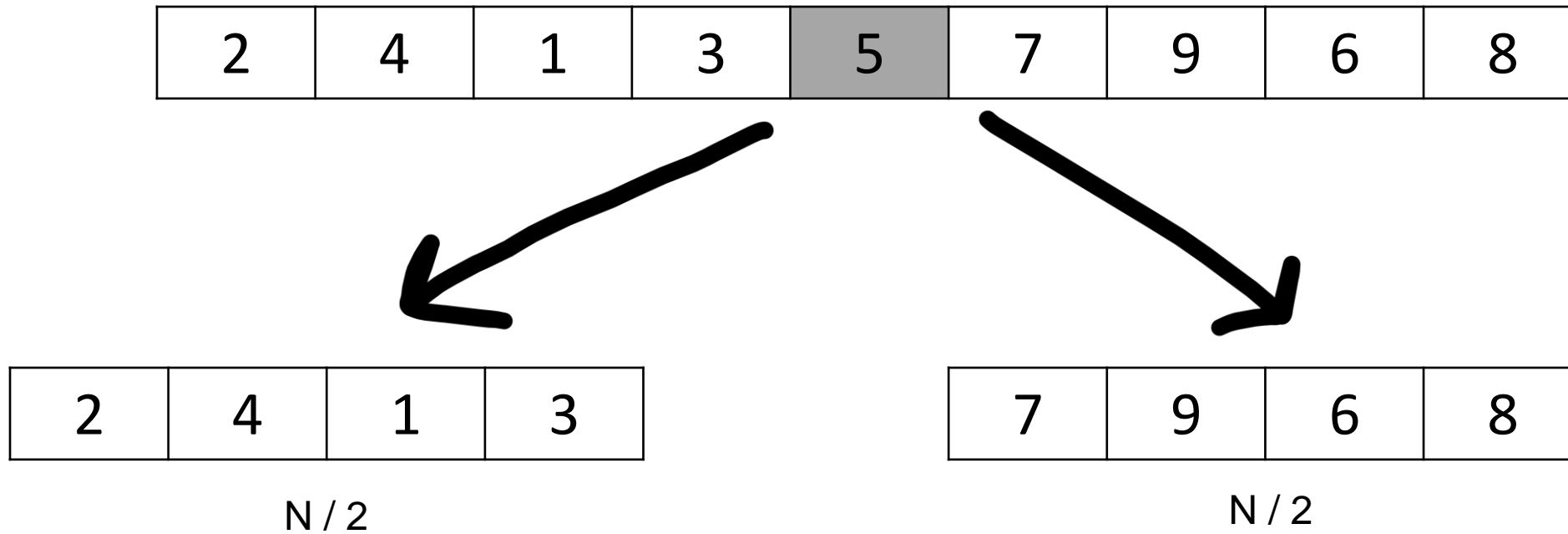
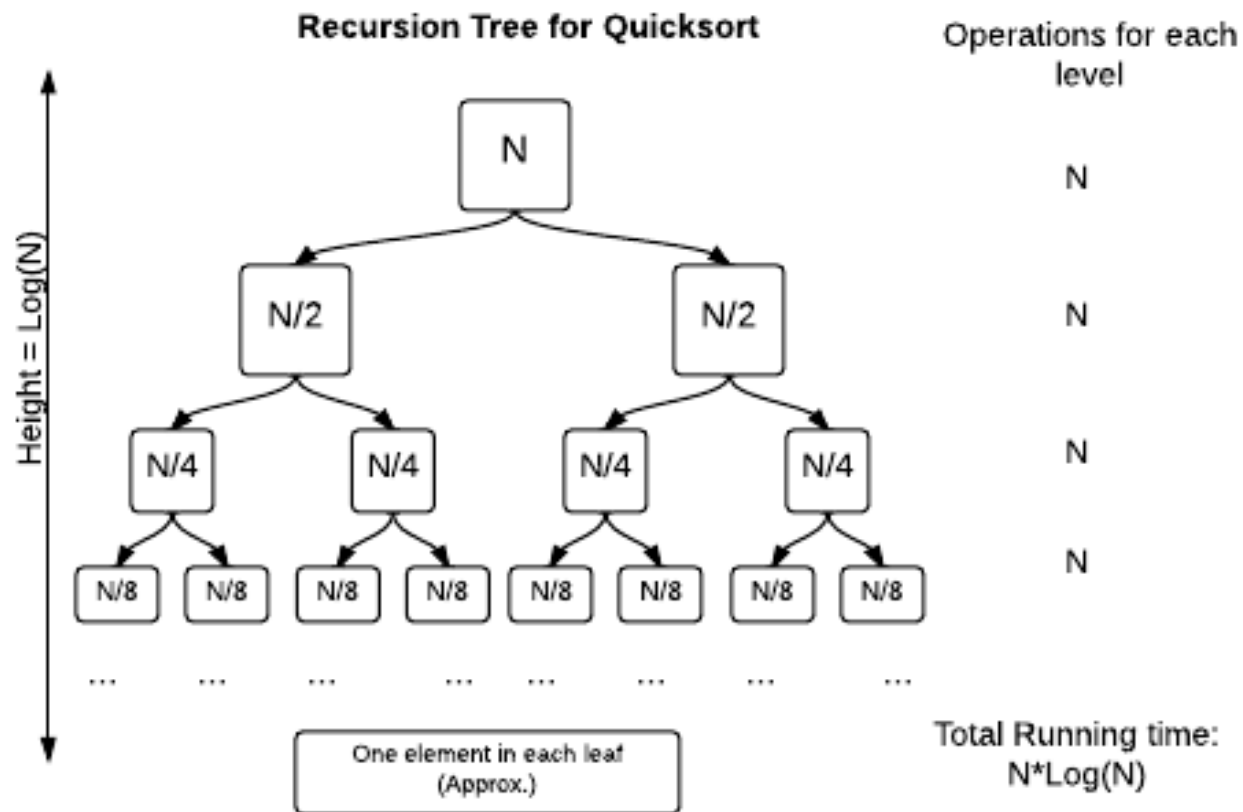| 2 | 4 | 1 | 3 | 5 | 7 | 9 | 6 | 8 |

| 2 | 4 | 1 | 3 |

N / 2

| 7 | 9 | 6 | 8 |

N / 2

Because our array is random, we don't know how balanced our partitions will be

| 1 | 2 | 4 | 3 | 5 | 7 | 9 | 6 | 8 |

| 1 |

| 4 | 3 | 5 | 7 | 9 | 6 | 8 |

Because our array is random, we don't know how balanced our partitions will be

(This is a less ideal case)

Recursion Tree for Quicksort

Height = Log(N)

Operations for each level

N — N
N/2   N/2 — N
N/4 N/4 N/4 N/4 — N
N/8 N/8 N/8 N/8 N/8 N/8 N/8 N/8 — N

One element in each leaf (Approx.)

Total Running time: N*Log(N)

This is the ideal situation…

This will give us **O(N * log(n))** running time

Subproblem sizes

n

0 $n-1$

0 $n-2$

0 $n-3$

2

0 1

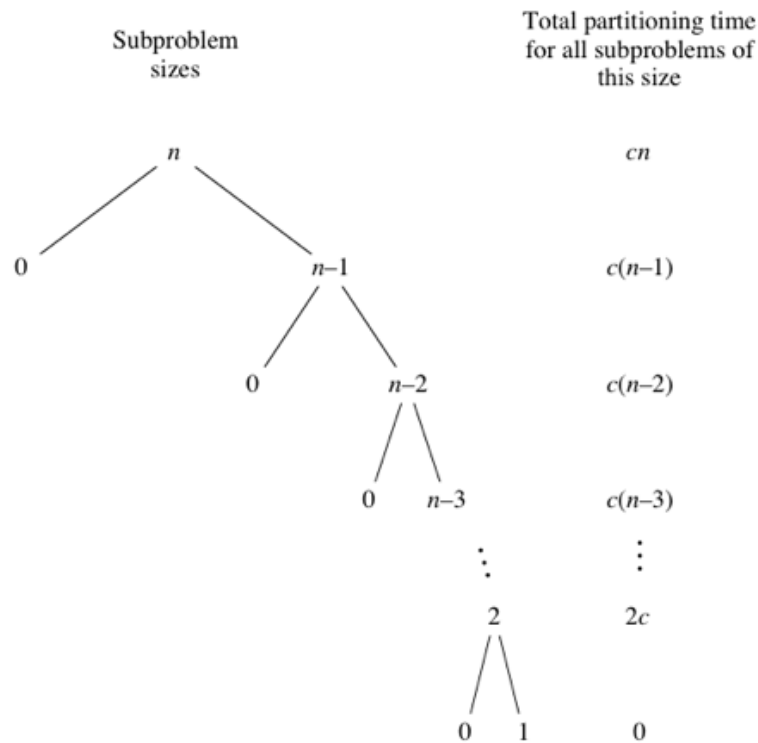**Total partitioning time for all subproblems of this size**

cn

$c(n-1)$

$c(n-2)$

$c(n-3)$

$2c$

0
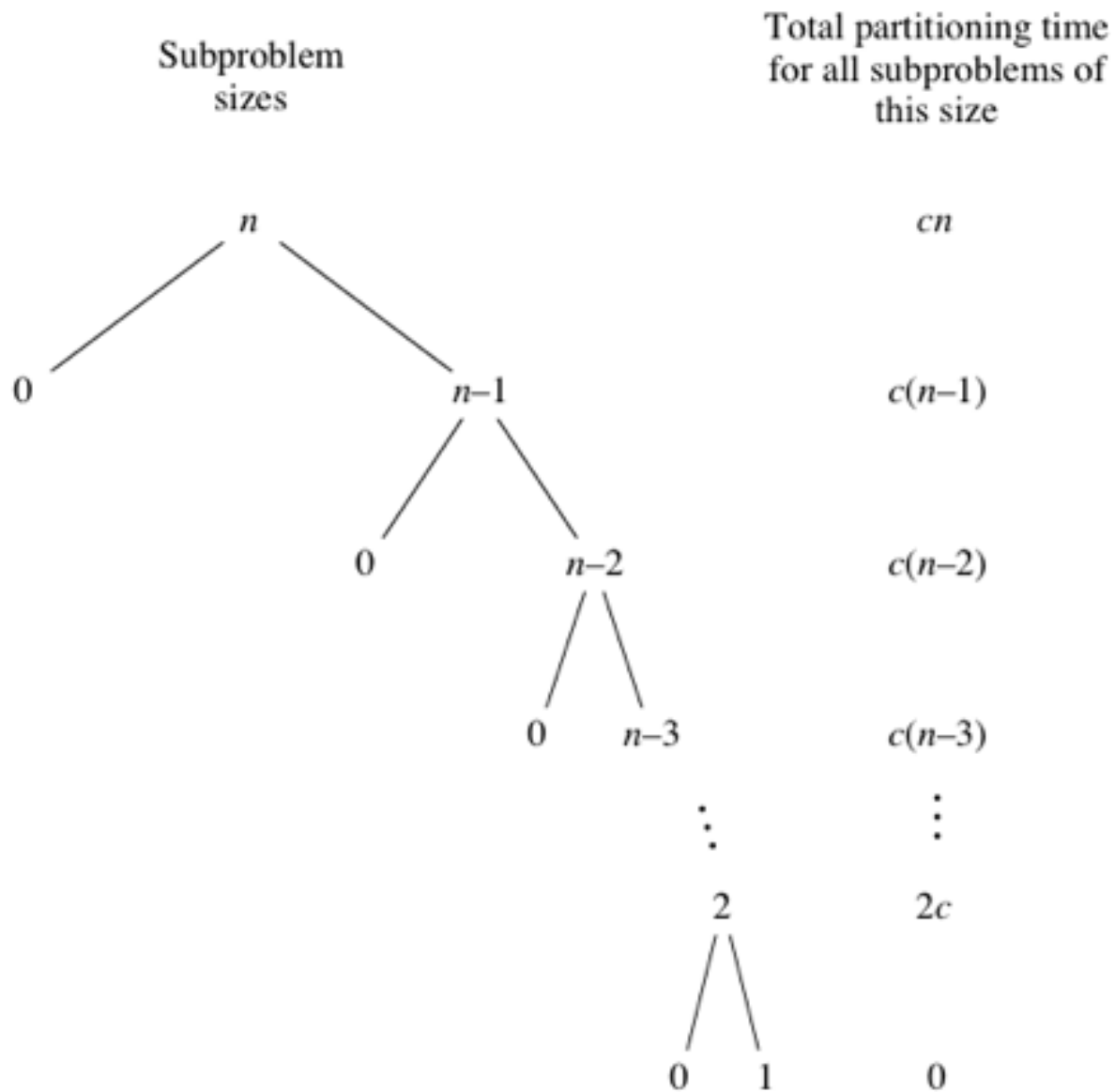
This is worst case scenario…

This will give us **O(N²)** running time

Subproblem sizes | Total partitioning time for all subproblems of this size

n — cn
0, $n-1$ — $c(n-1)$
0, $n-2$ — $c(n-2)$
0, $n-3$ — $c(n-3)$
⋮ — ⋮
2 — $2c$
0, 1 — 0

Recursion Tree for Quicksort

Height = Log(N)

N
N/2, N/2
N/4, N/4, N/4, N/4
N/8, N/8, N/8, N/8, N/8, N/8, N/8, N/8

One element in each leaf (Approx.)

Operations for each level

N
N
N
N

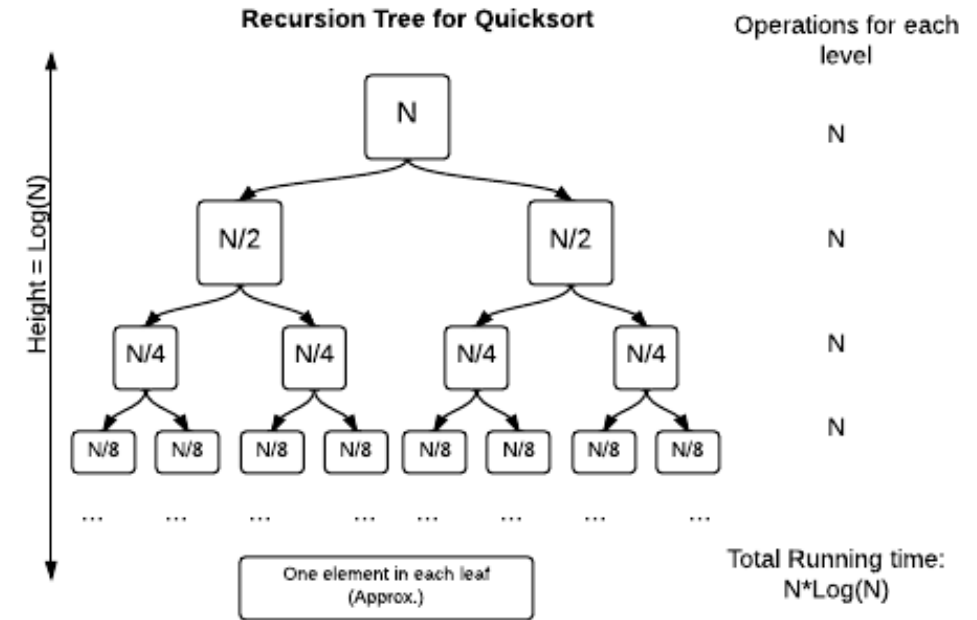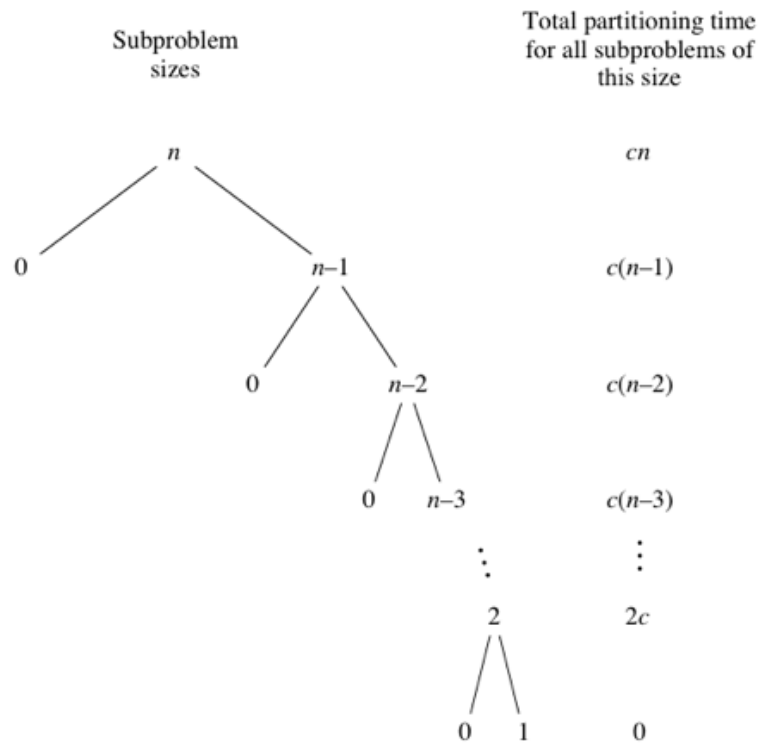Total Running time: N*Log(N)
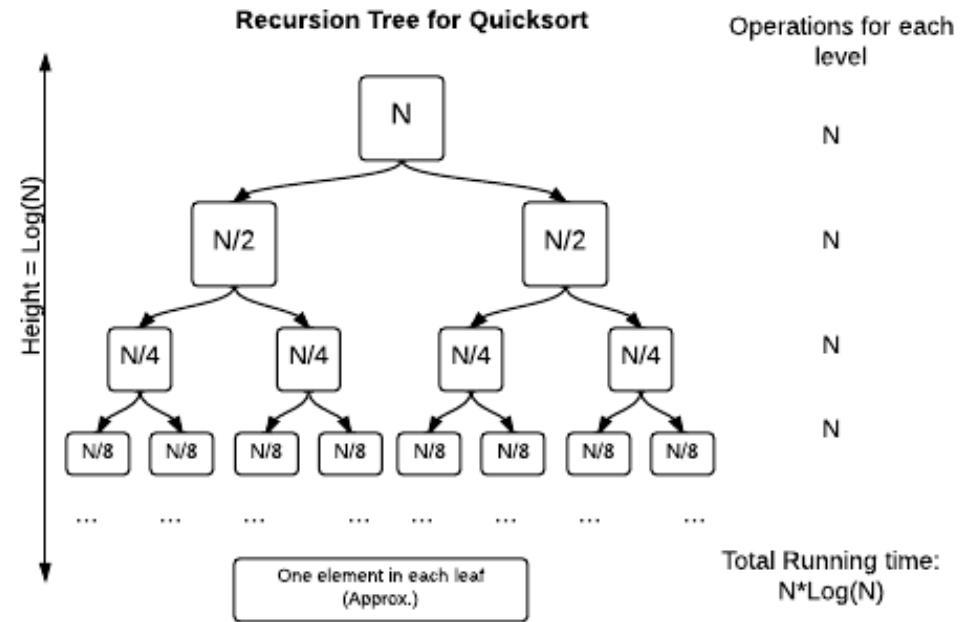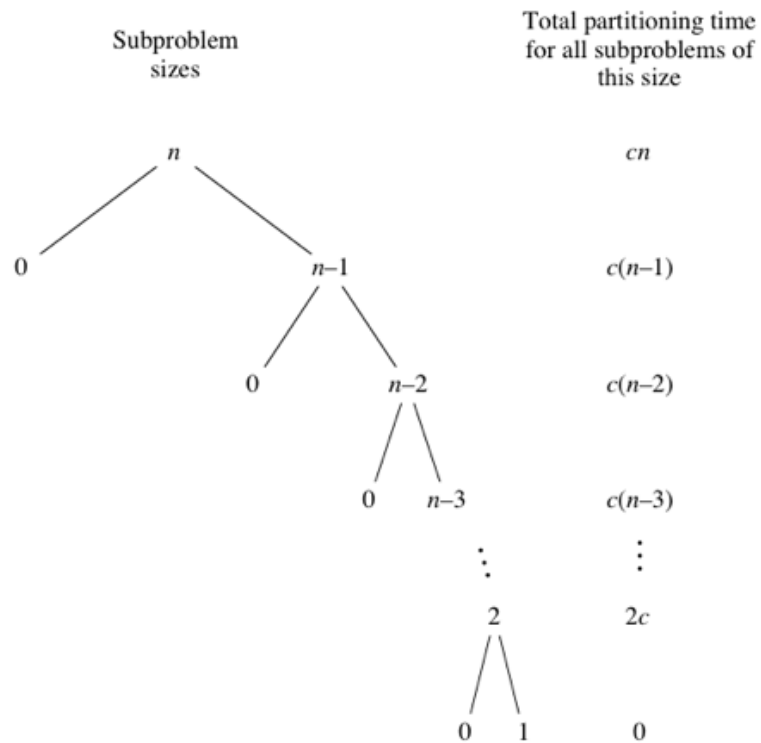
How balanced our partitions are depends on **how we select the pivot**
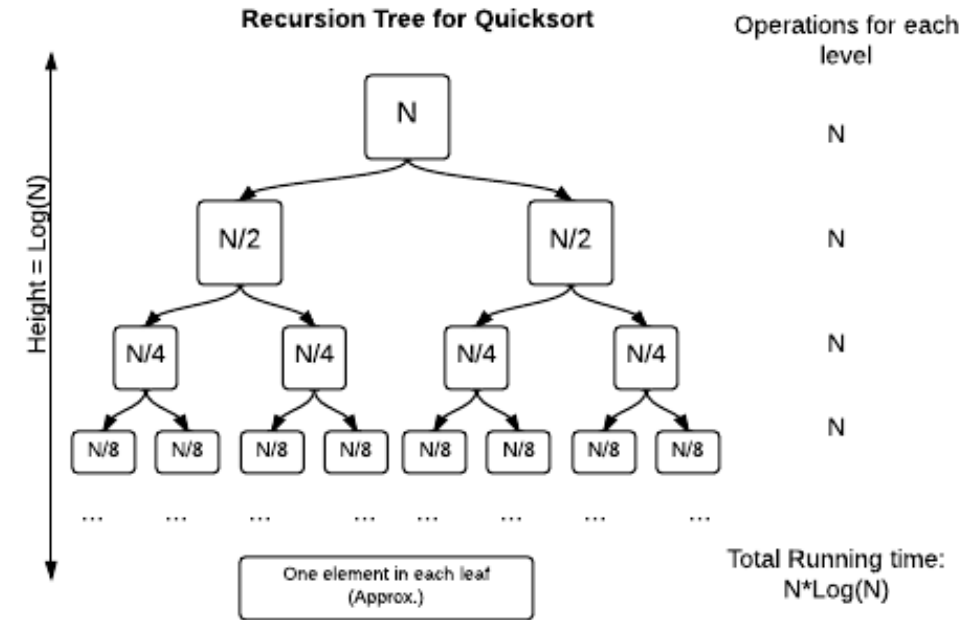
If we select the **median** value of the array as the pivot, that will always give us the optimal recursion tree and **O(n logn)** running time

Subproblem sizes / Total partitioning time for all subproblems of this size

Recursion Tree for Quicksort

If we select a **random value** as the pivot, that will actually give us a much better chance of **O(nlogn)** running time

Running time of Quick Sort

$O(n^2)$ worse case scenario   (n recursive calls, O(n) work at each level)

O(n * log n) on average   (logn recursive calls, O(n) work at each level)