O Notation 4

Quicksort

QuickSort Introduction

- Quicksort is a more efficient sorting algorithm than either selection or insertion sort
  - It sorts an array by repeatedly *partitioning* it
- Partitioning is the process of dividing an array into sections (partitions), based on some criteria
  - Big and small values
  - Negative and positive numbers
  - Names that begin with *a-m*, names that begin with *n-z*
  - Darker and lighter pixels

Partitioning an Array

Partition this array into *small* and *big* values using a partitioning algorithm

31	12	07	23	93	02	11	18

Partitioning an Array

Partition this array into *small* and *big* values using a partitioning algorithm

We will partition the array around the last value (18), we'll call this value the *pivot*

Use two indices, one at each end of the array, call them *low* and *high*

31	12	07	23	93	02	11	18

Partitioning an Array

Partition this array into *small* and *big* values using a partitioning algorithm

We will partition the array around the last value (18), we'll call this value the *pivot*

Use two indices, one at each end of the array, call them *low* and *high*

31	12	07	23	93	02	11	18

arr[*low*] (31) is greater than the pivot and should be on the right, we need to swap it with something

Partitioning an Array

Partition this array into *small* and *big* values using a partitioning algorithm

We will partition the array around the last value (18), we'll call this value the *pivot*

Use two indices, one at each end of the array, call them *low* and *high*

31	12	07	23	93	02	11	18

arr[*low*] (31) is greater than the pivot and should be on the right, we need to swap it with something

arr[*high]* (11) is less than the pivot so swap with arr[*low*]

Partitioning an Array

Partition this array into *small* and *big* values using a partitioning algorithm

We will partition the array around the last value (18), we'll call this value the *pivot*

Use two indices, one at each end of the array, call them *low* and *high*

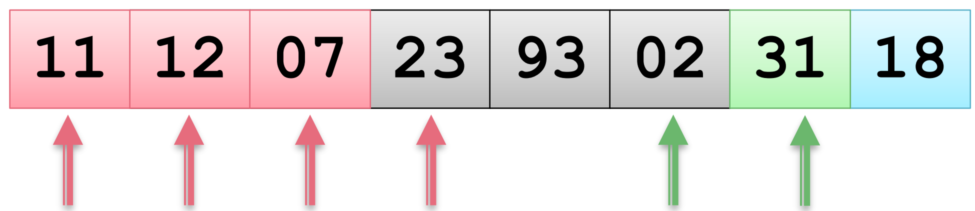
11	12	07	23	93	02	31	18

Partitioning an Array

Partition this array into *small* and *big* values using a partitioning algorithm

We will partition the array around the last value (18), we'll call this value the *pivot*

Use two indices, one at each end of the array, call them *low* and *high*



increment *low* until it needs to be swapped with something

then decrement *high* until it can be swapped with *low*

Partitioning an Array

Partition this array into *small* and *big* values using a partitioning algorithm

We will partition the array around the last value (18), we'll call this value the *pivot*

Use two indices, one at each end of the array, call them *low* and *high*

11	12	07	02	93	23	31	18

increment *low* until it needs to be swapped with something

then decrement *high* until it can be swapped with *low*

and then swap them

Partitioning Algorithm

Partition this array into *small* and *big* values using a partitioning algorithm

We will partition the array around the last value (18), we'll call this value the *pivot*

Use two indices, one at each end of the array, call them *low* and *high*

11	12	07	02	93	23	31	18

repeat this process until

high and *low* are the same

Partitioning an Array

Partition this array into *small* and *big* values using a partitioning algorithm

We will partition the array around the last value (18), we'll call this value the *pivot*

Use two indices, one at each end of the array, call them *low* and *high*

11	12	07	02	18	23	31	93

repeat this process until

high and *low* are the same

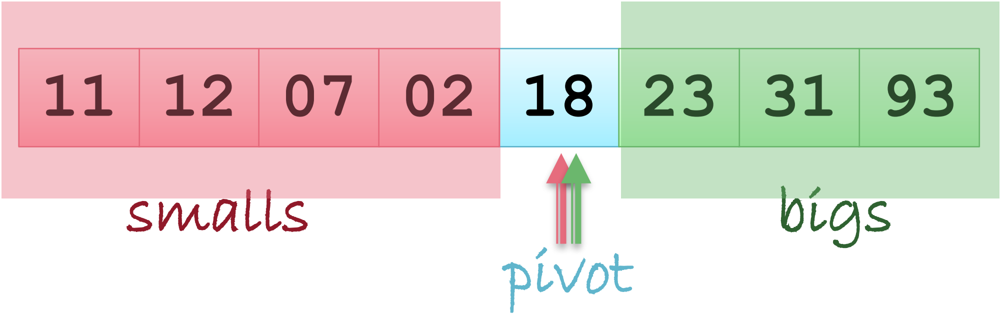
We'd like the pivot value to be in the centre of the array, so we will swap it with the first item greater than it

Partitioning an Array

Partition this array into *small* and *big* values using a partitioning algorithm

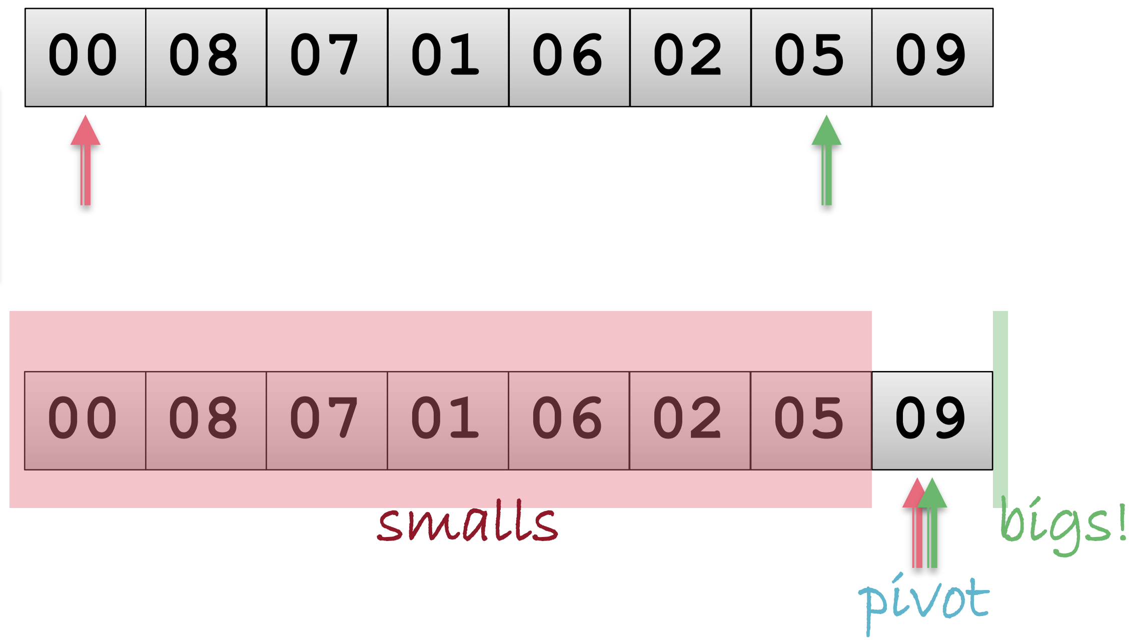
We will partition the array around the last value (18), we'll call this value the *pivot*

Use two indices, one at each end of the array, call them *low* and *high*



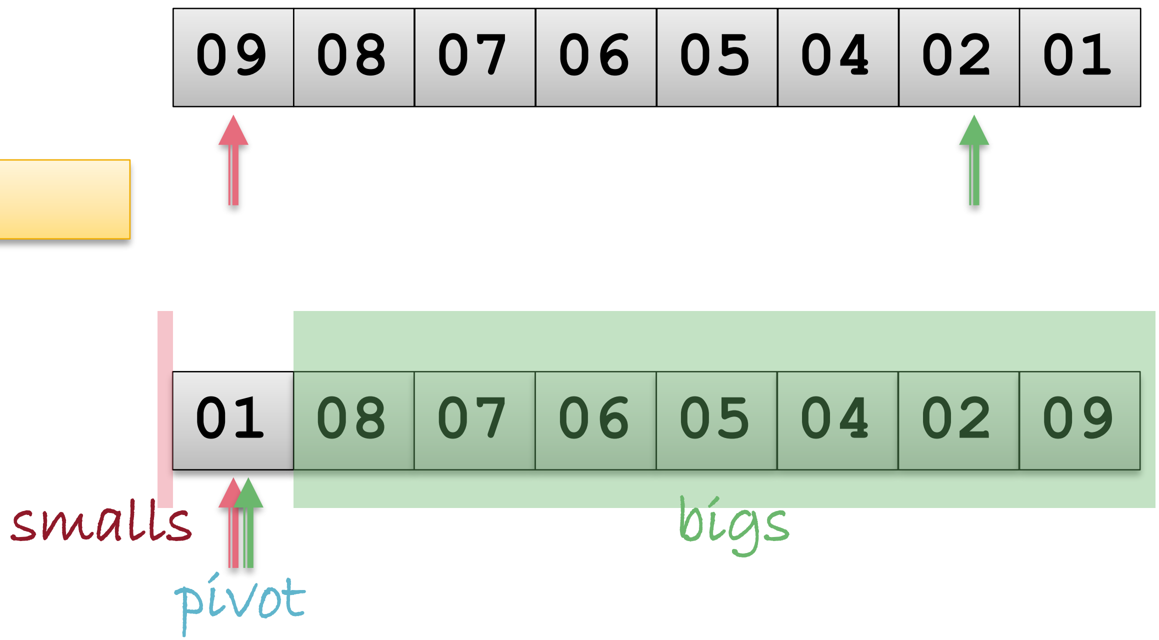
Partitioning Question

Use the same algorithm to partition this array into small and big values



Partitioning Question

Or this one:



Quicksort

- The quicksort algorithm works by *repeatedly partitioning* an array
- Each time a sub-array is partitioned there is
  - A sequence of *small* values,
  - A sequence of *big* values, and
  - A *pivot* value *which is in the correct position*
- Partition the small values, and the big values
  - Repeat the process until each sub-array being partitioned consists of just one element

Quicksort Algorithm

- The quicksort algorithm repeatedly partitions an array until it is sorted
  - Until all partitions consist of at most one element
- A simple iterative approach would halve each sub-array to get partitions
  - But partitions are not necessarily the same size
  - So the start and end indexes of each partition are not easily predictable

Uneven Partitions

47	70	36	97	03	61	29	11	48	09	53
36	09	29	48	03	11	47	53	97	61	70
36	09	03	11	29	47	48	53	61	70	97
08	03	11	29	36	47	48	53	61	70	97
09	03	11	29	36	47	48	53	61	70	97
03	09	11	29	36	47	48	53	61	70	97

Keeping Track of Indexes

- One way to implement quicksort might be to record the index of each new partition
- But this is difficult and requires space in memory
  - The goal is to record the start and end index of each partition
  - This can be achieved by making them the parameters of a recursive function

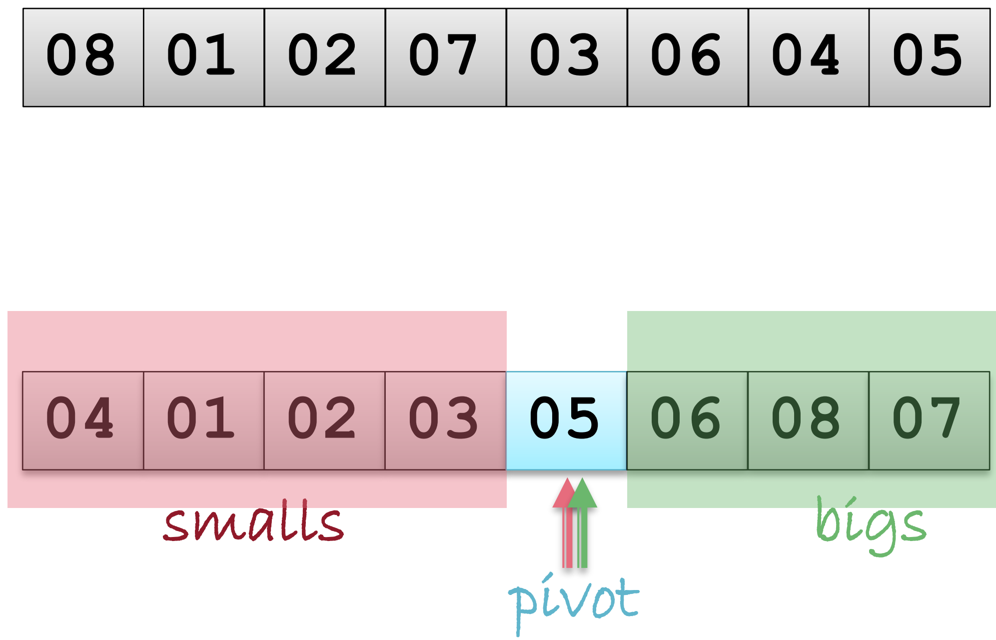
Recursive Quicksort

```
void quicksort(arr[], int low, int high){
  if (low < high){
    pivot = partition(arr, low, high);
    quicksort(arr, low, pivot – 1);
    quicksort(arr, pivot + 1, high);
  }
}
```

Quicksort Analysis

- How long does Quicksort take to run?
  - Let's consider the best and the worst case
  - These differ because the partitioning algorithm may not always do a good job
- Let's look at the best case first
  - Each time a sub-array is partitioned the pivot is the exact midpoint of the slice (or as close as it can get)
    - So, it is divided in half
  - What is the running time?

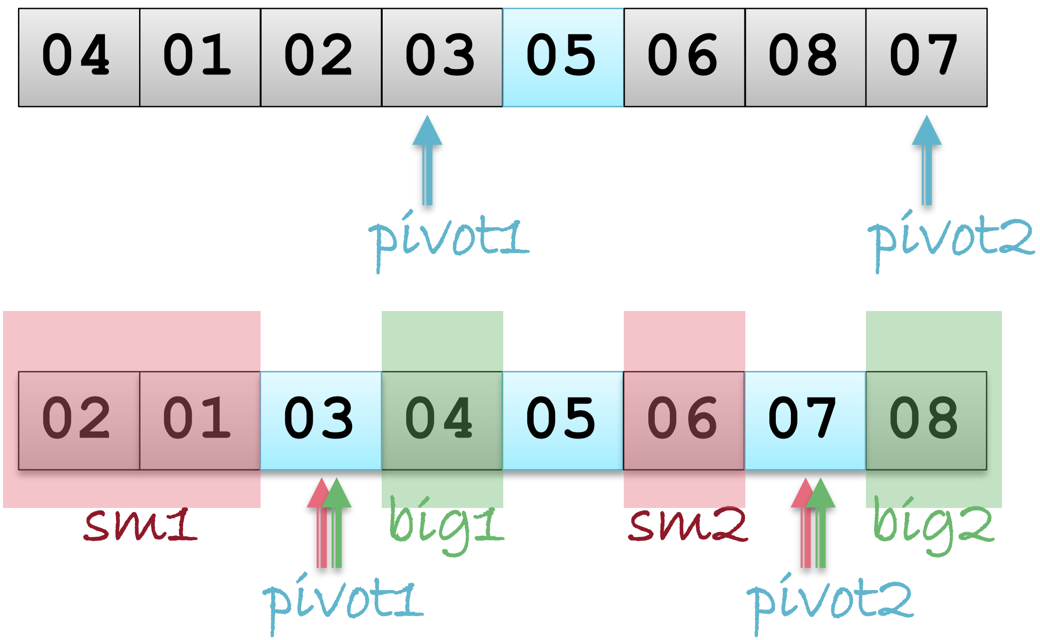
Quicksort Best Case



First partition

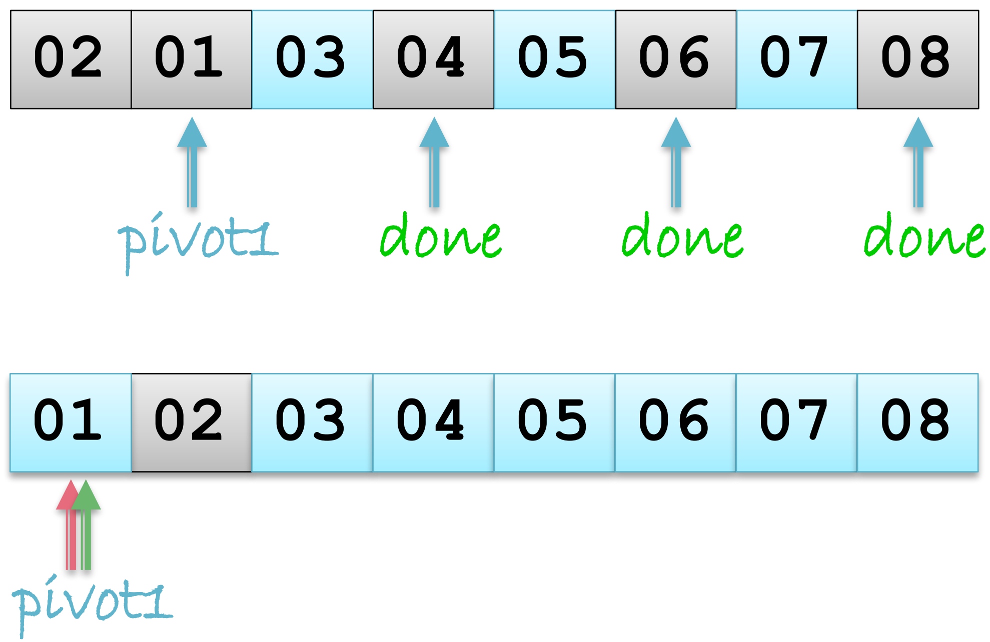
Quicksort Best Case

Second partition



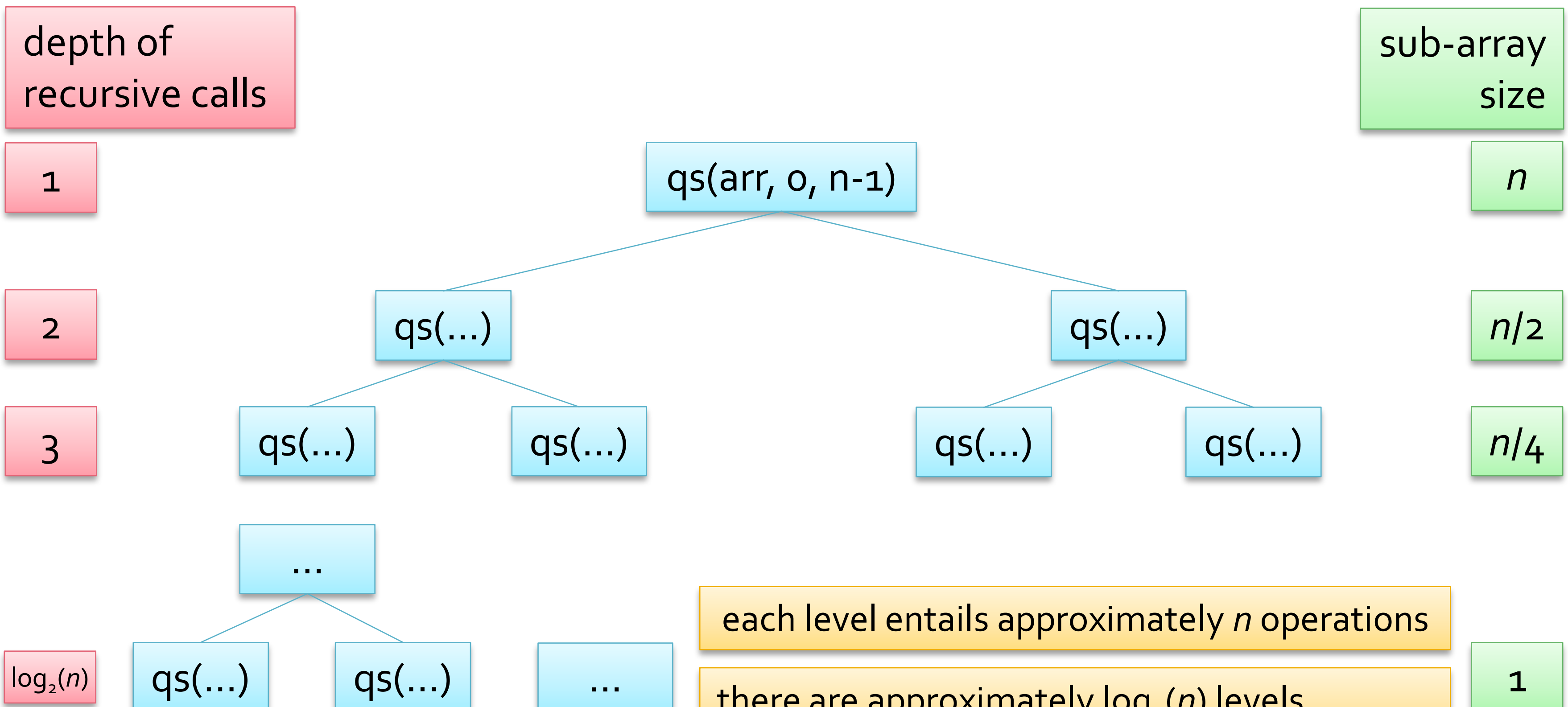
Quicksort Best Case

Third partition



Quicksort Recursion Tree

Assume the best case – each partition splits its sub-array in half



each level entails approximately n operations

there are approximately $\log_2(n)$ levels

approximately $\log_2(n) * n$ operations in total

Quicksort Best Case

- Each sub-array is divided in half in each partition
  - Each time a series of sub-arrays are partitioned *n* (approximately) comparisons are made
  - The process ends once all the sub-arrays left to be partitioned are of size 1
- How many times does *n* have to be divided in half before the result is 1?
  - $\log_2(n)$ times
  - Quicksort performs $n * \log_2 n$ operations in the best case

Quicksort Worst Case

09	08	07	06	05	04	02	01

First partition

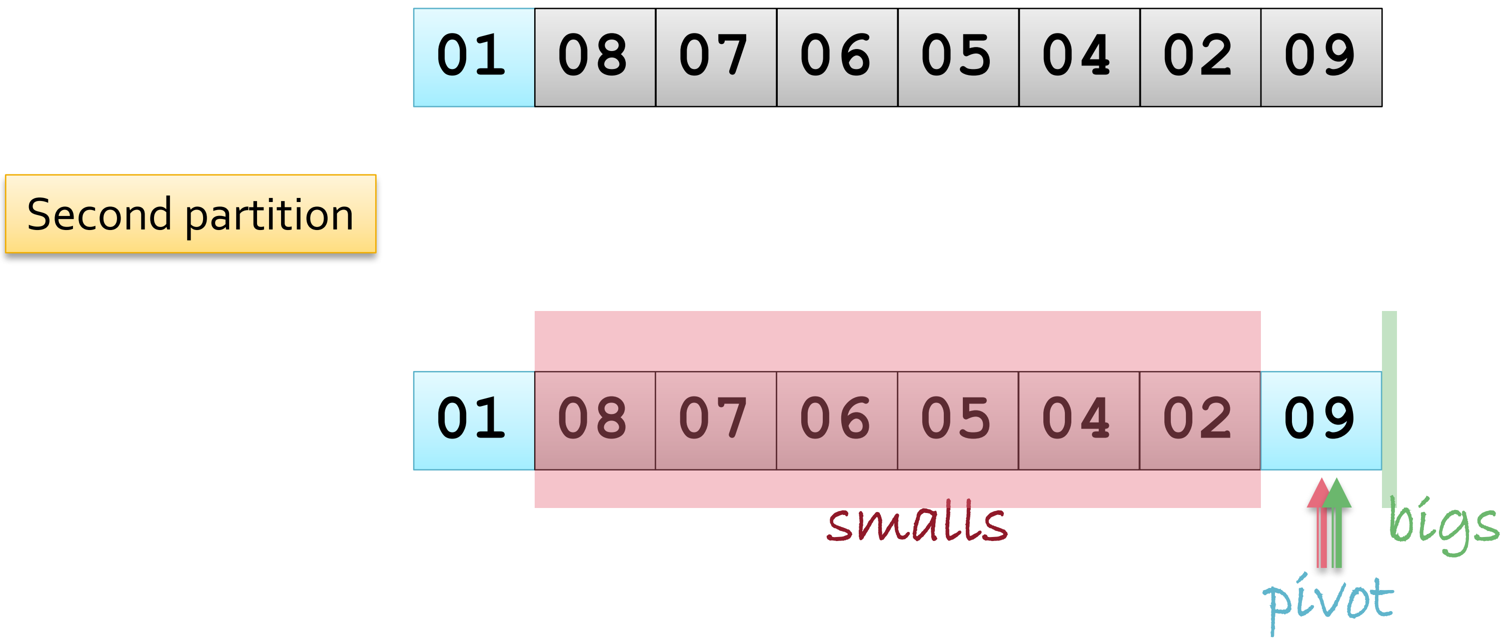
01	08	07	06	05	04	02	09

smalls

pivot

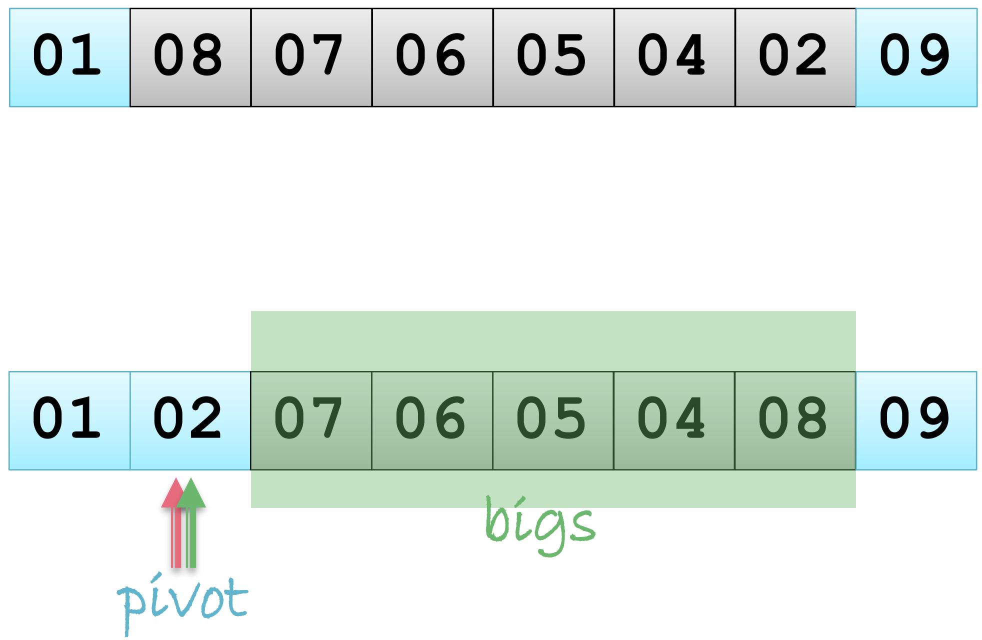
bigs

Quicksort Worst Case



Quicksort Worst Case

Third partition



Quicksort Worst Case

01	02	07	06	05	04	08	09

Fourth partition

01	02	07	06	05	04	08	09

smalls

pivot

Quicksort Worst Case

01	02	07	06	05	04	08	09

Fifth partition

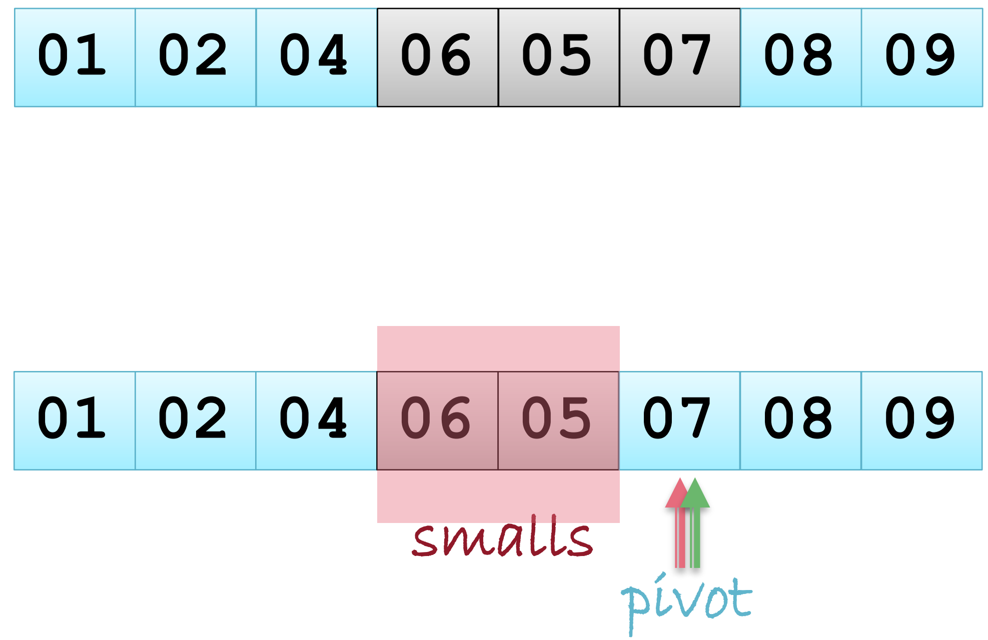
01	02	04	06	05	07	08	09

bigs

pivot

Quicksort Worst Case

Sixth partition



Quicksort Worst Case

01	02	04	06	05	07	08	09

Seventh partition!

01	02	04	05	06	07	08	09

pivot

Quicksort Recursion Tree

Assume the worst case – each partition step results in a single sub-array

depth of recursive calls		sub-array size
1	qs(arr, 0, n-1)	n
2	qs(...)	n-1
3	qs(...)	n-2
	...	
n	qs(...)	1

each level entails on average $n/2$ operations

there are approximately n levels

approximately $n^2/2$ operations in total

Quicksort Worst Case

- Every partition step ends with no values on one side of the pivot
  - The array has to be partitioned n times, not $\log_2(n)$ times
  - In the worst-case Quicksort performs around n^2 operations
- The worst case usually occurs when the array is nearly sorted (in either direction)

Quicksort Average Case

- With a large array we would have to be very, very unlucky to get the worst case
  - Unless there was some reason for the array to already be partially sorted
- The average case is much more like the best case than the worst case
- There is an easy way to fix a partially sorted arrays to that it is ready for quicksort
  - Randomize the positions of the array elements!