Faculty of Information Technology, Monash University

FIT2004: Algorithms and Data Structures

Week 3:
Quick Sort and its Analysis

These slides are prepared by M. A. Cheema and are based on the material developed by Arun Konagurthu and Lloyd Allison.

Things to note/remember

- Assignment 1 due 23-Aug 23:59:00
- Assignment 2 released soon
  - Requires dynamic programming (taught in week 4) – don’t miss the lecture
  - Deadline: 6-Sep-2019 23:59:00

Quick Sort and its Analysis

1. Algorithm and partitioning
2. Complexity Analysis
3. Improving Worst-case complexity
   A. Quick Select
   B. Quick Sort in O(N log N) worst-case

Quick Sort Idea

1. If list is length 1 or less, do nothing
2. Choose a pivot p
3. Put items <= p on the left, items >p on the right
4. Quicksort the left and right parts of the list

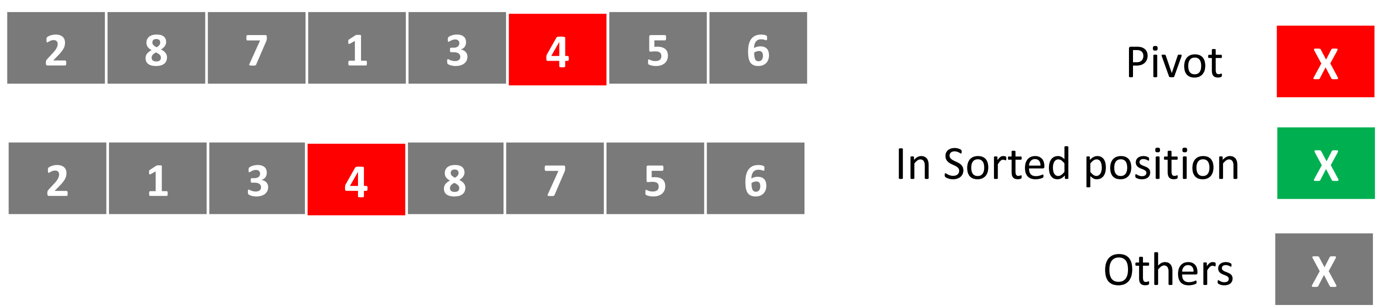
Quicksort

- Choose a pivot p

2	8	7	1	3	4	5	6

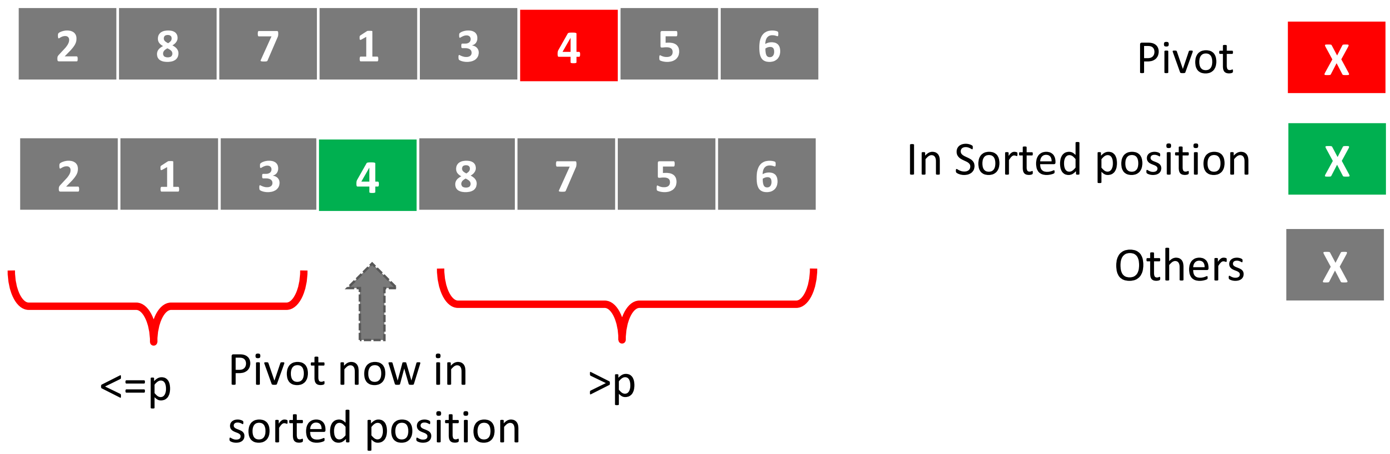
Quicksort

- Choose a pivot p
- Partition the array in two sub-arrays w.r.t. p
  - LEFT ← elements smaller than or equal to p
  - RIGHT ← elements greater than p



Quicksort

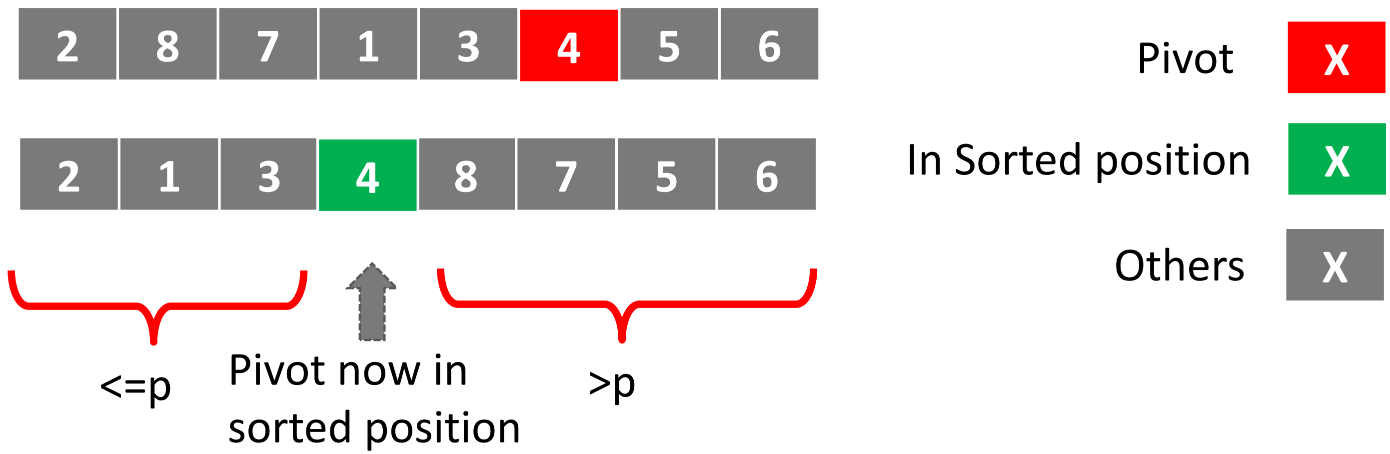
- Choose a pivot p
- Partition the array in two sub-arrays w.r.t. p
  - LEFT ← elements smaller than or equal to p
  - RIGHT ← elements greater than p



Quicksort

- Choose a pivot p
- Partition the array in two sub-arrays w.r.t. p
  - LEFT ← elements smaller than or equal to p
  - RIGHT ← elements greater than p

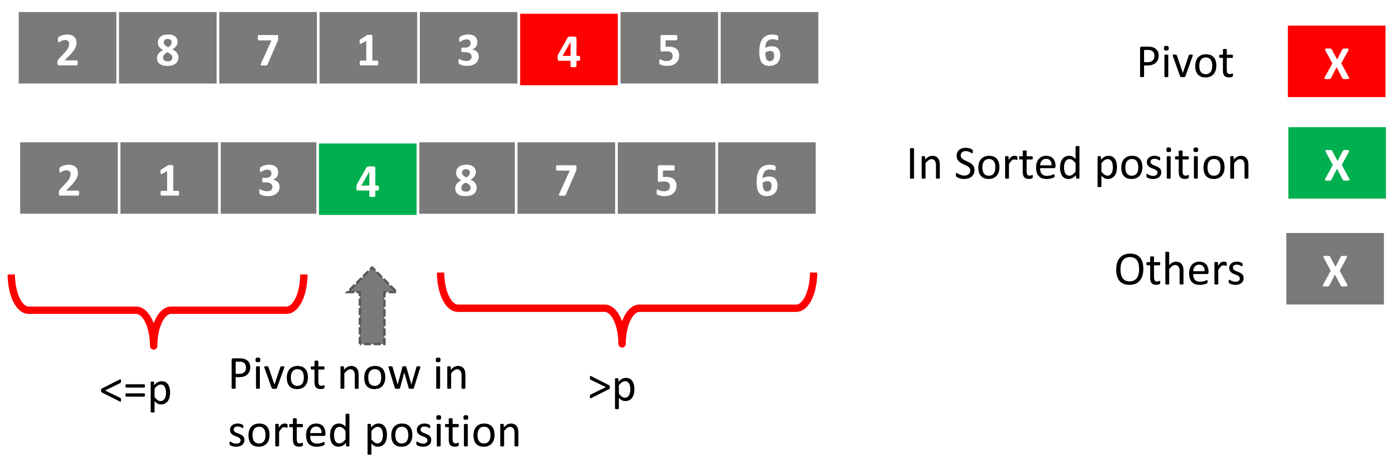
Partitioning



Quicksort

Partitioning

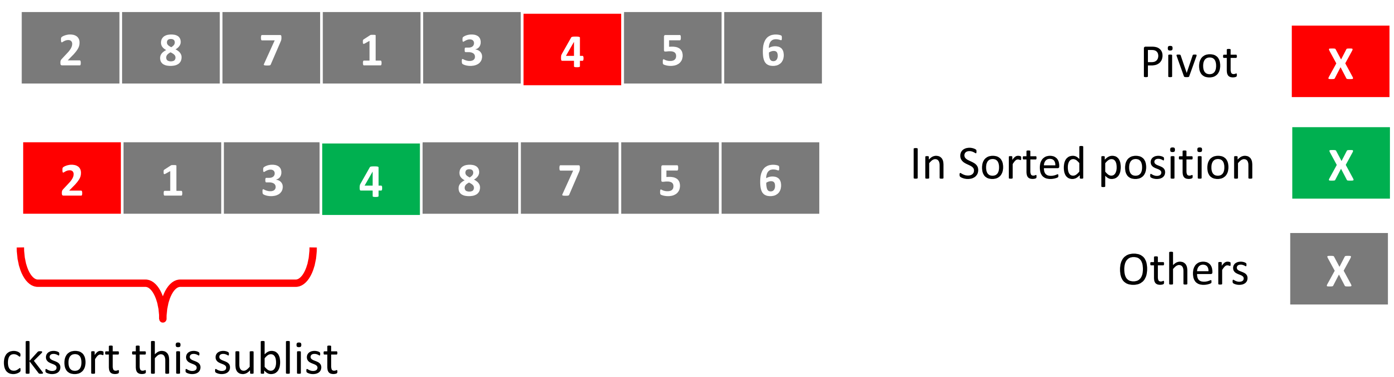
- Choose a pivot p
- Partition the array in two sub-arrays w.r.t. p
  - LEFT ← elements smaller than or equal to p
  - RIGHT ← elements greater than p
- Quicksort(LEFT)



Quicksort

- Choose a pivot p
- Partition the array in two sub-arrays w.r.t. p
  - LEFT ← elements smaller than or equal to p
  - RIGHT ← elements greater than p
- Quicksort(LEFT)

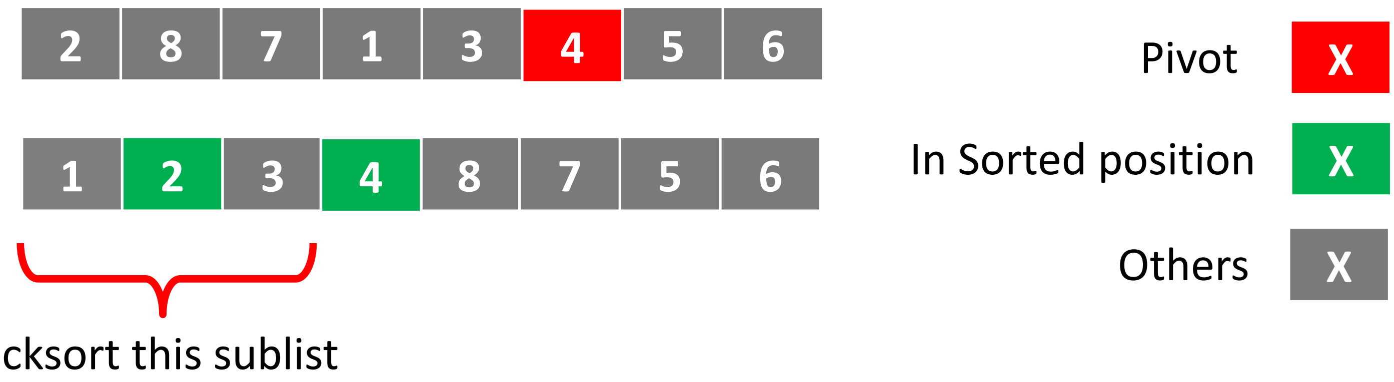
Partitioning



Quicksort

Partitioning

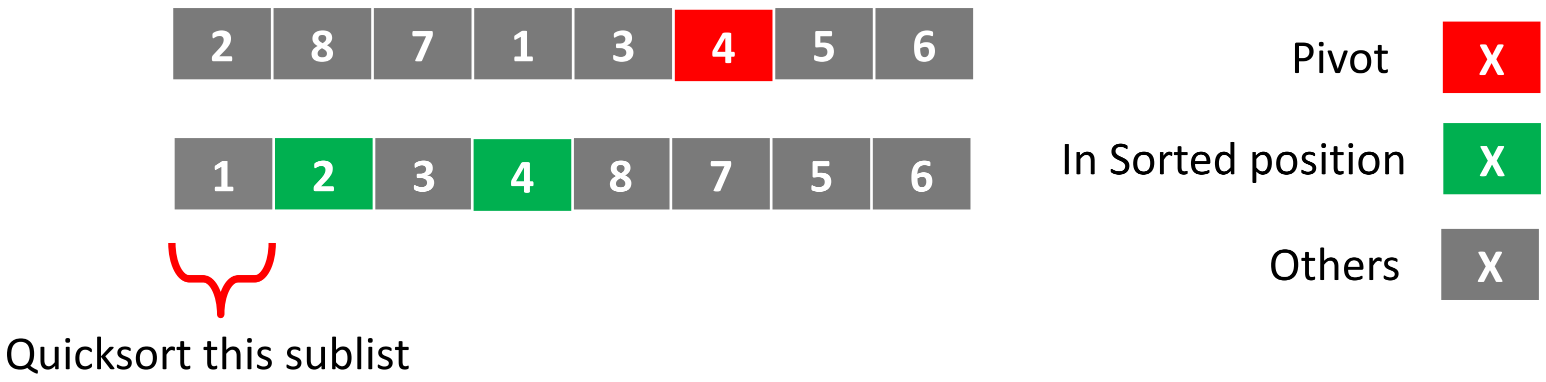
- Choose a pivot p
- Partition the array in two sub-arrays w.r.t. p
    - LEFT ← elements smaller than or equal to p
    - RIGHT ← elements greater than p
- Quicksort(LEFT)



Quicksort

Partitioning

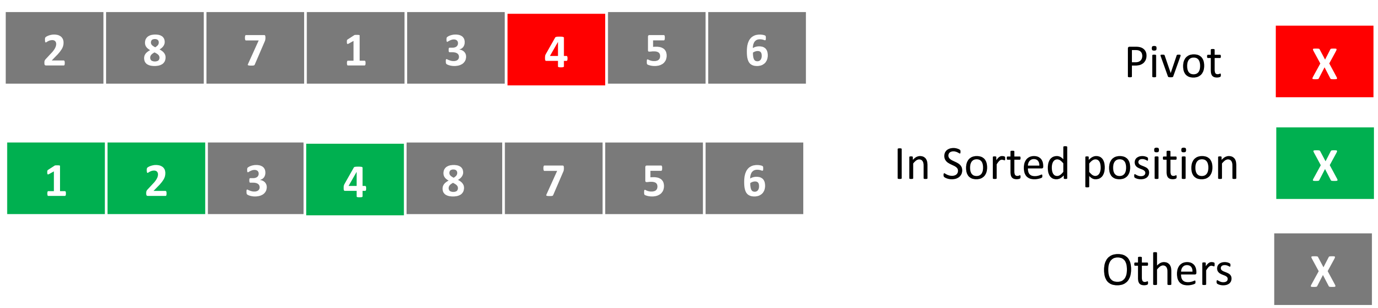
- Choose a pivot p
- Partition the array in two sub-arrays w.r.t. p
    - LEFT ← elements smaller than or equal to p
    - RIGHT ← elements greater than p
- Quicksort(LEFT)



Quicksort

- Choose a pivot p
- Partition the array in two sub-arrays w.r.t. p
  - LEFT ← elements smaller than or equal to p
  - RIGHT ← elements greater than p
- Quicksort(LEFT)

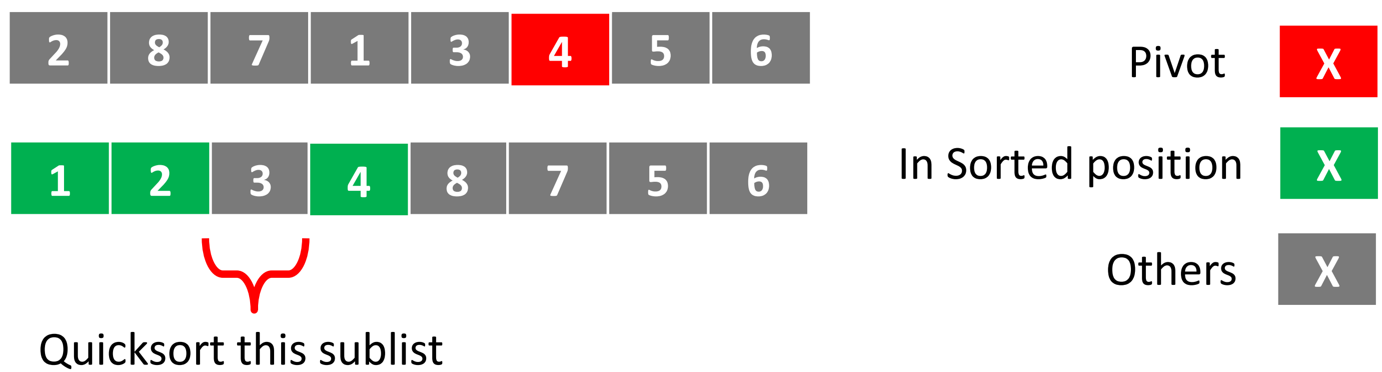
Partitioning



Quicksort

Partitioning

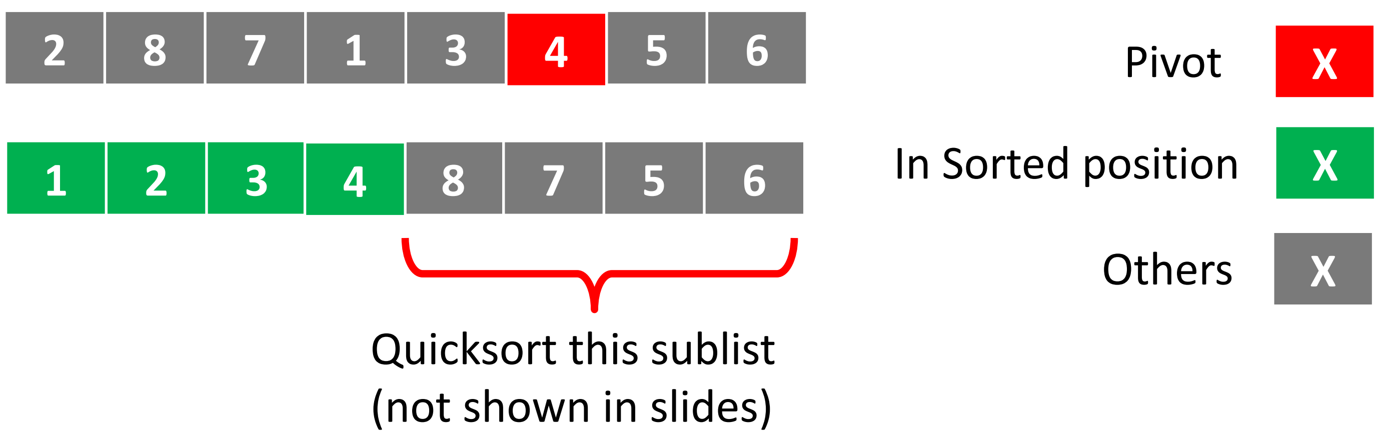
- Choose a pivot p
- Partition the array in two sub-arrays w.r.t. p
  - LEFT ← elements smaller than or equal to p
  - RIGHT ← elements greater than p
- Quicksort(LEFT)
- Quicksort(RIGHT)



Quicksort

- Choose a pivot p
- Partition the array in two sub-arrays w.r.t. p
  - LEFT ← elements smaller than or equal to p
  - RIGHT ← elements greater than p
- Quicksort(LEFT)
- Quicksort(RIGHT)

Partitioning



Partitioning: An out-of-place version

- Initialize two lists LEFT and RIGHT
- For each element e (except pivot)
  - If e ≤ pivot
    - Insert e in LEFT
  - If e > pivot
    - Insert e in RIGHT

2	8	7	1	3	4	5	6

Partitioning: An out-of-place version

- Initialize two lists LEFT and RIGHT
- For each element e (except pivot)
  - If e ≤ pivot
    - Insert e in LEFT
  - If e > pivot
    - Insert e in RIGHT

2	8	7	1	3	4	5	6

LEFT

RIGHT

Partitioning: An out-of-place version

- Initialize two lists LEFT and RIGHT
- For each element e (except pivot)
  - If e ≤ pivot
    - Insert e in LEFT
  - If e > pivot
    - Insert e in RIGHT

2	8	7	1	3	4	5	6

LEFT

RIGHT

Partitioning: An out-of-place version

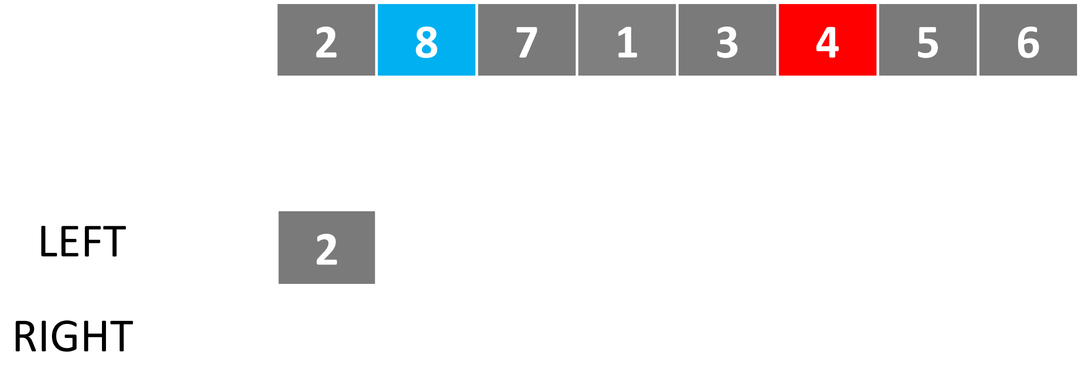
- Initialize two lists LEFT and RIGHT
- For each element e (except pivot)
  - If e ≤ pivot
    - Insert e in LEFT
  - If e > pivot
    - Insert e in RIGHT

2	8	7	1	3	4	5	6

LEFT	2
RIGHT	

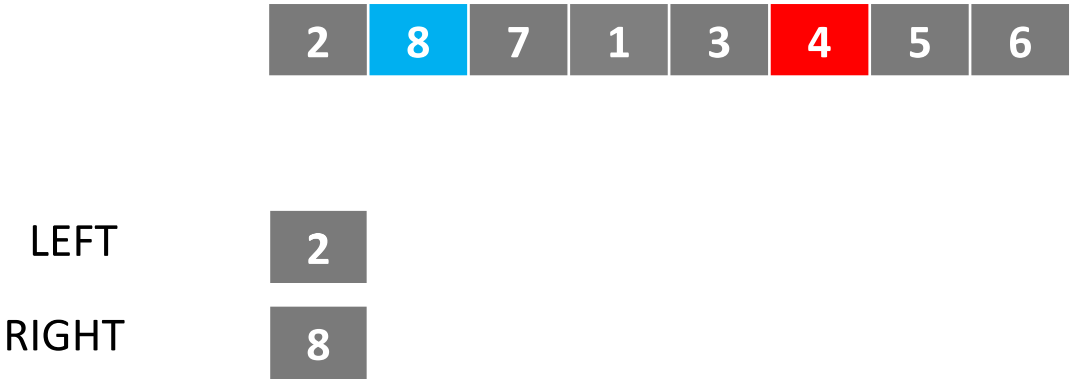
Partitioning: An out-of-place version

- Initialize two lists LEFT and RIGHT
- For each element e (except pivot)
  - If e ≤ pivot
    - Insert e in LEFT
  - If e > pivot
    - Insert e in RIGHT



Partitioning: An out-of-place version

- Initialize two lists LEFT and RIGHT
- For each element e (except pivot)
  - If e ≤ pivot
    - Insert e in LEFT
  - If e > pivot
    - Insert e in RIGHT



Partitioning: An out-of-place version

- Initialize two lists LEFT and RIGHT
- For each element e (except pivot)
  - If e ≤ pivot
    - Insert e in LEFT
  - If e > pivot
    - Insert e in RIGHT

2	8	7	1	3	4	5	6

LEFT	2
RIGHT	8

Partitioning: An out-of-place version

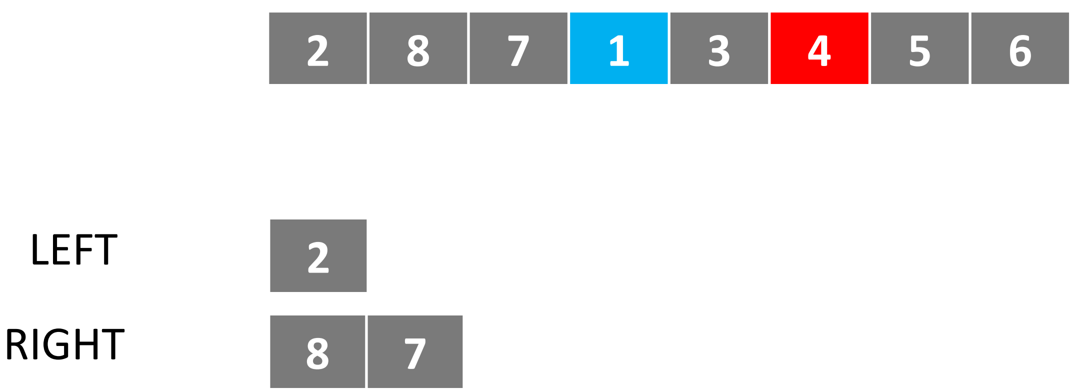
- Initialize two lists LEFT and RIGHT
- For each element e (except pivot)
  - If e ≤ pivot
    - Insert e in LEFT
  - If e > pivot
    - Insert e in RIGHT

2	8	7	1	3	4	5	6

LEFT	2	
RIGHT	8	7

Partitioning: An out-of-place version

- Initialize two lists LEFT and RIGHT
- For each element e (except pivot)
  - If e ≤ pivot
    - Insert e in LEFT
  - If e > pivot
    - Insert e in RIGHT



Partitioning: An out-of-place version

- Initialize two lists LEFT and RIGHT
- For each element e (except pivot)
  - If e ≤ pivot
    - Insert e in LEFT
  - If e > pivot
    - Insert e in RIGHT

2	8	7	1	3	4	5	6

LEFT	2	1
RIGHT	8	7

Partitioning: An out-of-place version

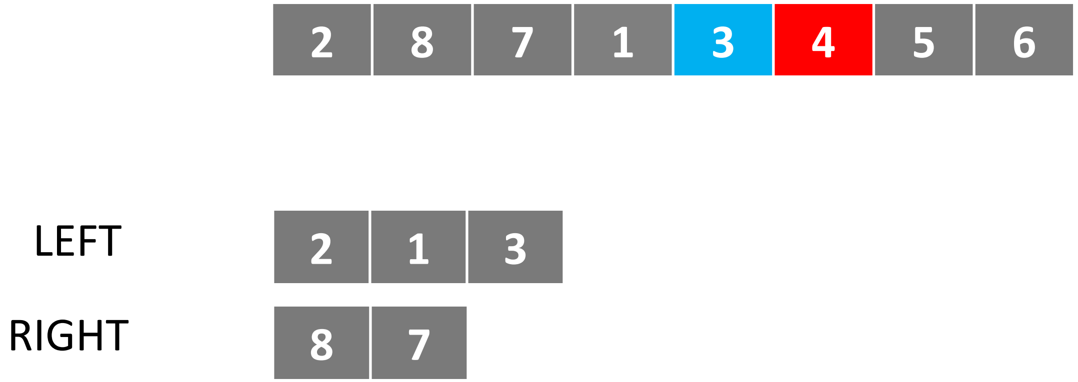
- Initialize two lists LEFT and RIGHT
- For each element e (except pivot)
  - If e ≤ pivot
    - Insert e in LEFT
  - If e > pivot
    - Insert e in RIGHT

2	8	7	1	3	4	5	6

LEFT	2	1
RIGHT	8	7

Partitioning: An out-of-place version

- Initialize two lists LEFT and RIGHT
- For each element e (except pivot)
  - If e ≤ pivot
    - Insert e in LEFT
  - If e > pivot
    - Insert e in RIGHT



Partitioning: An out-of-place version

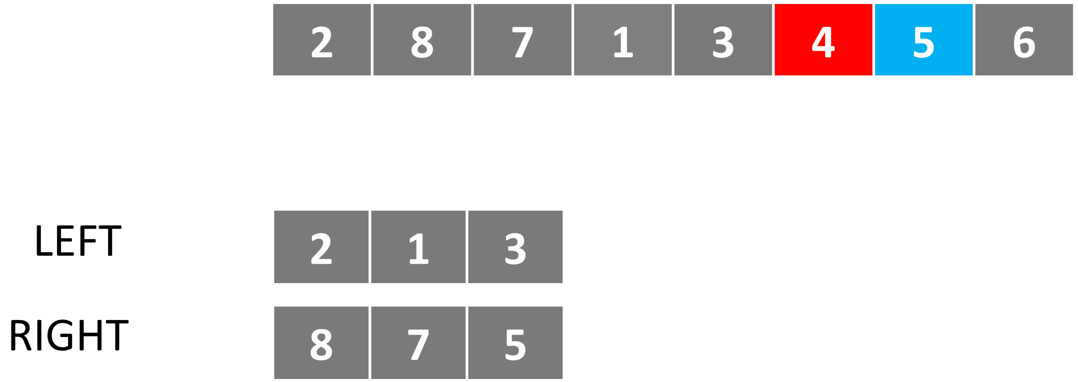
- Initialize two lists LEFT and RIGHT
- For each element e (except pivot)
  - If e ≤ pivot
    - Insert e in LEFT
  - If e > pivot
    - Insert e in RIGHT

2	8	7	1	3	4	5	6

LEFT	2	1	3
RIGHT	8	7	

Partitioning: An out-of-place version

- Initialize two lists LEFT and RIGHT
- For each element e (except pivot)
  - If e ≤ pivot
    - Insert e in LEFT
  - If e > pivot
    - Insert e in RIGHT



Partitioning: An out-of-place version

- Initialize two lists LEFT and RIGHT
- For each element e (except pivot)
  - If e ≤ pivot
    - Insert e in LEFT
  - If e > pivot
    - Insert e in RIGHT

2	8	7	1	3	4	5	6

LEFT	2	1	3
RIGHT	8	7	5

Partitioning: An out-of-place version

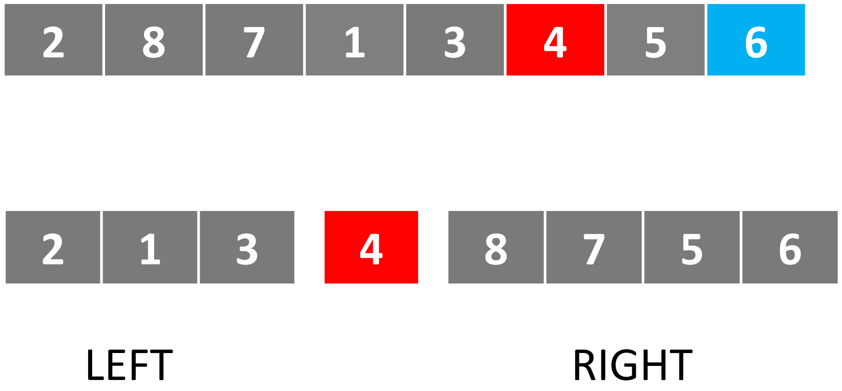
- Initialize two lists LEFT and RIGHT
- For each element e (except pivot)
  - If e ≤ pivot
    - Insert e in LEFT
  - If e > pivot
    - Insert e in RIGHT

2	8	7	1	3	4	5	6

LEFT	2	1	3	
RIGHT	8	7	5	6

Partitioning: An out-of-place version

- Initialize two lists LEFT and RIGHT
- For each element e (except pivot)
  - If e ≤ pivot
    - Insert e in LEFT
  - If e > pivot
    - Insert e in RIGHT
- Copy LEFT+ [pivot] + RIGHT over the input array



Partitioning: An out-of-place version

- Initialize two lists LEFT and RIGHT
- For each element e (except pivot)
  - If e ≤ pivot
    - Insert e in LEFT
  - If e > pivot
    - Insert e in RIGHT
- Copy LEFT+ [pivot] + RIGHT over the input array

2	**1**	**3**	**4**	**8**	**7**	**5**	**6**

LEFT RIGHT

Partitioning: An out-of-place version

- Initialize two lists LEFT and RIGHT
- For each element e (except pivot)
  - If e ≤ pivot
    - Insert e in LEFT
  - If e > pivot
    - Insert e in RIGHT
- Copy LEFT+ [pivot] + RIGHT over the input array

2	1	3	4	8	7	5	6

- Array is now correctly partitioned

Partitioning: An out-of-place version

- Initialize two lists LEFT and RIGHT
- For each element e (except pivot)
  - If e ≤ pivot
    - Insert e in LEFT
  - If e > pivot
    - Insert e in RIGHT
- Copy LEFT+ [pivot] + RIGHT over the input array

2	1	3	4	8	7	5	6

- Array is now correctly partitioned
- Algorithm is clearly not in place

Partitioning: An out-of-place version

- Initialize two lists LEFT and RIGHT
- For each element e (except pivot)
  - If e ≤ pivot
    - Insert e in LEFT
  - If e > pivot
    - Insert e in RIGHT
- Copy LEFT+ [pivot] + RIGHT over the input array

2	1	3	4	8	7	5	6

- Array is now correctly partitioned
- Algorithm is clearly not in place
- Is this algorithm stable?

Partitioning: An out-of-place version

- Initialize two lists LEFT and RIGHT
- For each element e (except pivot)
  - If e ≤ pivot
    - Insert e in LEFT
  - If e > pivot
    - Insert e in RIGHT
- Copy LEFT+ [pivot] + RIGHT over the input array

2	1	3	4	8	7	5	6

- Array is now correctly partitioned
- Algorithm is clearly not in place
- Is this algorithm stable? No. Elements which are equal to the pivot end up on the left regardless

Partitioning: In place (Hoare's)

Swap pivot to the front (position 1)

2	8	6	4	1	7	3	5

Partitioning: In place (Hoare's)

Swap pivot to the front (position 1)

L_bad = 2, R_bad = N

4	8	6	2	1	7	3	5

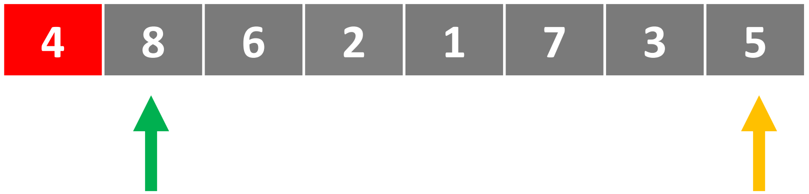
Partitioning: In place (Hoare's)

Swap pivot to the front (position 1)

L_bad = 2, R_bad = N

Repeat until L_bad and R_bad cross

- move L_bad right until we find a "bad" element, i.e. > pivot
- move R_bad left until we find a "bad" element, i.e. < pivot
- swap these elements



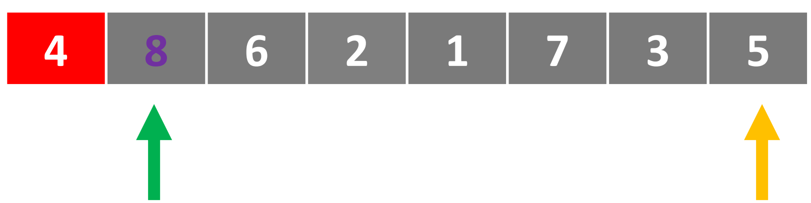
Partitioning: In place (Hoare's)

Swap pivot to the front (position 1)

L_bad = 2, R_bad = N

Repeat until L_bad and R_bad cross

- move L_bad right until we find a "bad" element, i.e. > pivot
- move R_bad left until we find a "bad" element, i.e. < pivot
- swap these elements



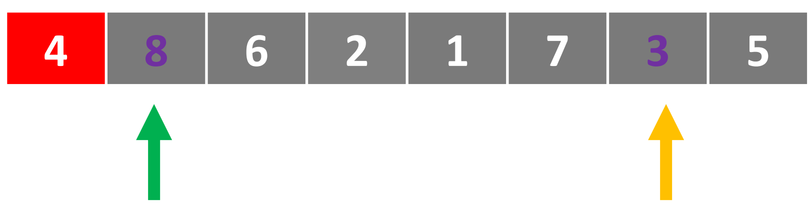
Partitioning: In place (Hoare's)

Swap pivot to the front (position 1)

L_bad = 2, R_bad = N

Repeat until L_bad and R_bad cross

- move L_bad right until we find a "bad" element, i.e. > pivot
- move R_bad left until we find a "bad" element, i.e. < pivot
- swap these elements



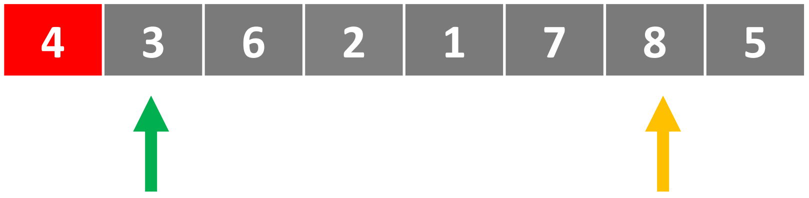
Partitioning: In place (Hoare's)

Swap pivot to the front (position 1)

L_bad = 2, R_bad = N

Repeat until L_bad and R_bad cross

- move L_bad right until we find a "bad" element, i.e. > pivot
- move R_bad left until we find a "bad" element, i.e. < pivot
- swap these elements



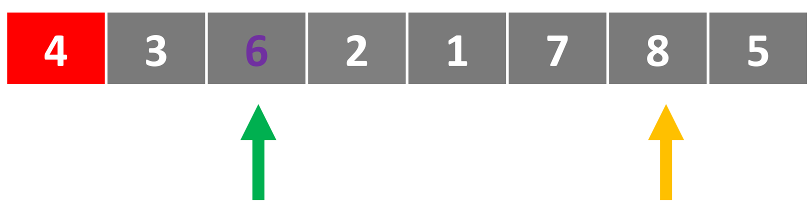
Partitioning: In place (Hoare's)

Swap pivot to the front (position 1)

L_bad = 2, R_bad = N

Repeat until L_bad and R_bad cross

- move L_bad right until we find a "bad" element, i.e. > pivot
- move R_bad left until we find a "bad" element, i.e. < pivot
- swap these elements



Partitioning: In place (Hoare's)

Swap pivot to the front (position 1)

L_bad = 2, R_bad = N

Repeat until L_bad and R_bad cross

- move L_bad right until we find a "bad" element, i.e. > pivot
- move R_bad left until we find a "bad" element, i.e. < pivot
- swap these elements

4	3	6	2	1	7	8	5

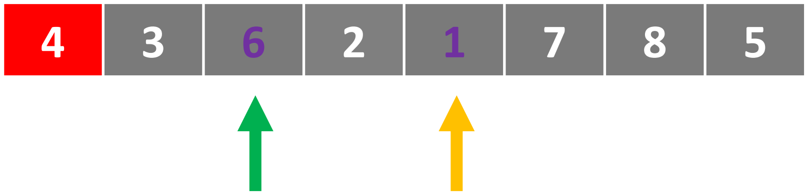
Partitioning: In place (Hoare's)

Swap pivot to the front (position 1)

L_bad = 2, R_bad = N

Repeat until L_bad and R_bad cross

- move L_bad right until we find a "bad" element, i.e. > pivot
- move R_bad left until we find a "bad" element, i.e. < pivot
- swap these elements



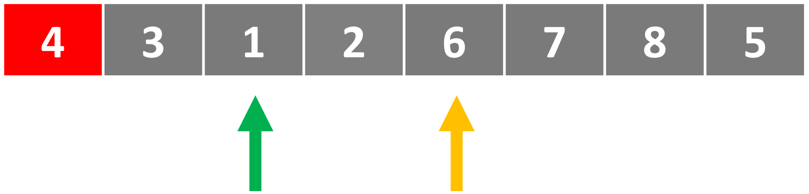
Partitioning: In place (Hoare’s)

Swap pivot to the front (position 1)

L_bad = 2, R_bad = N

Repeat until L_bad and R_bad cross

- move L_bad right until we find a “bad” element, i.e. > pivot
- move R_bad left until we find a “bad” element, i.e. < pivot
- swap these elements



Partitioning: In place (Hoare’s)

Swap pivot to the front (position 1)

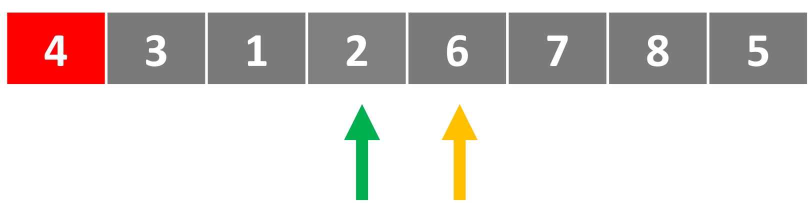
L_bad = 2, R_bad = N

Repeat until L_bad and R_bad cross

  move L_bad right until we find a “bad” element, i.e. > pivot

  move R_bad left until we find a “bad” element, i.e. < pivot

  swap these elements



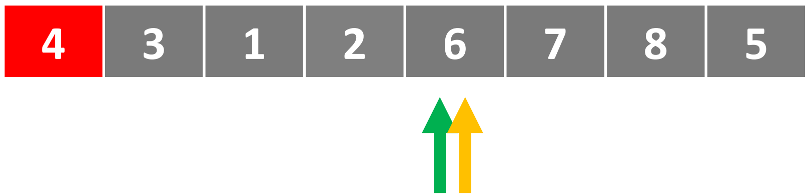
Partitioning: In place (Hoare's)

Swap pivot to the front (position 1)

L_bad = 2, R_bad = N

Repeat until L_bad and R_bad cross

- move L_bad right until we find a "bad" element, i.e. > pivot
- move R_bad left until we find a "bad" element, i.e. < pivot
- swap these elements



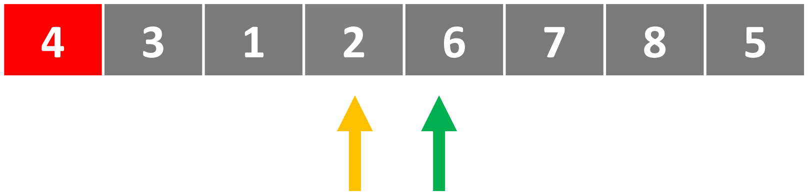
Partitioning: In place (Hoare's)

Swap pivot to the front (position 1)

L_bad = 2, R_bad = N

Repeat until L_bad and R_bad cross

- move L_bad right until we find a "bad" element, i.e. > pivot
- move R_bad left until we find a "bad" element, i.e. < pivot
- swap these elements



Partitioning: In place (Hoare's)

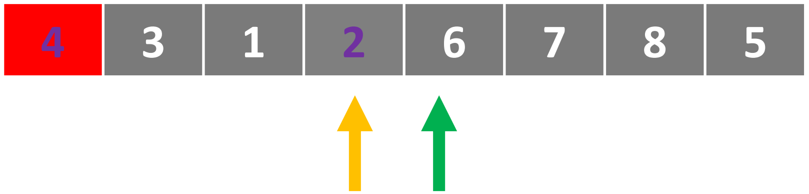
Swap pivot to the front (position 1)

L_bad = 2, R_bad = N

Repeat until L_bad and R_bad cross

- move L_bad right until we find a "bad" element, i.e. > pivot
- move R_bad left until we find a "bad" element, i.e. < pivot
- swap these elements

swap pivot to R_bad



Partitioning: In place (Hoare's)

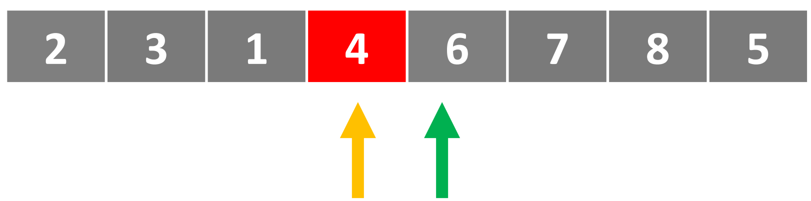
Swap pivot to the front (position 1)

L_bad = 2, R_bad = N

Repeat until L_bad and R_bad cross

- move L_bad right until we find a "bad" element, i.e. > pivot
- move R_bad left until we find a "bad" element, i.e. < pivot
- swap these elements

swap pivot to R_bad



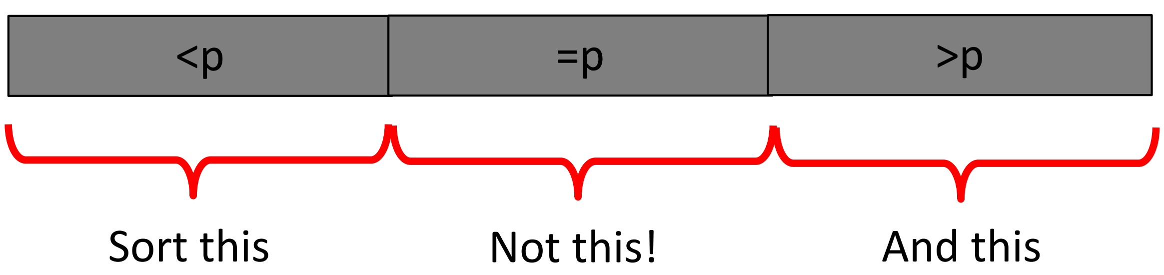
Partitioning: In place (Hoare's)

- Pros:
  - Each element only swapped once (except pivot)
  - Simple idea
  - Simple invariant (what is it?)

- Cons:
  - Very tricky to implement without bugs
    - Termination conditions
    - Edge cases
    - Off by one errors
  - Not stable
  - What about duplicates?

Partition and duplicates

- If the list has many duplicates, then sometimes…
- One will be chosen as the pivot
- All the others **should** go next to the pivot (and therefore not need to be moved any more)
- But the algorithms we have seen would require them to be sorted in the recursive calls!
- We want a partition method that does this:



Dutch National Flag Problem

- Given a list of elements and a function that maps them to red, white and blue
- Arrange the list to look like the dutch national flag

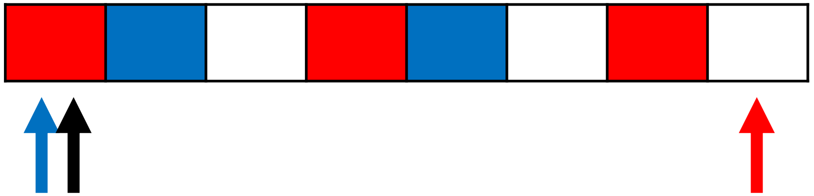
- This is equivalent to our problem
- Our function maps elements less than the pivot to blue, equal elements to white, and greater elements to red

Dutch National Flag Algoithm

boundary1=1,

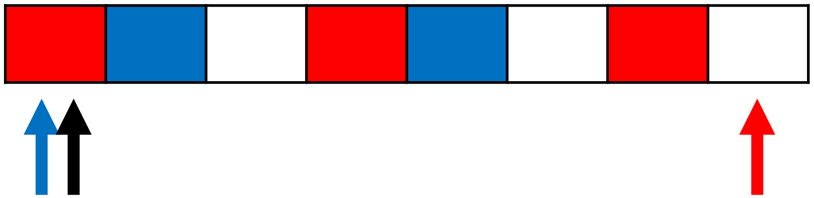
j=1

boundary2 = n



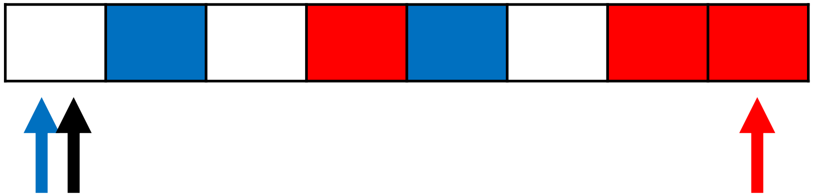
Dutch National Flag Algoithm

```
boundary1=1,
j=1
boundary2 = n
While j <=boundary2
    if array[j] is blue
        swap array[boundary1], array[j]
        boundary1 += 1
        j += 1
    elif array[j] is red
        swap array[j], array[boundary2]
        boundary2 -= 1
    else
        j += 1
```



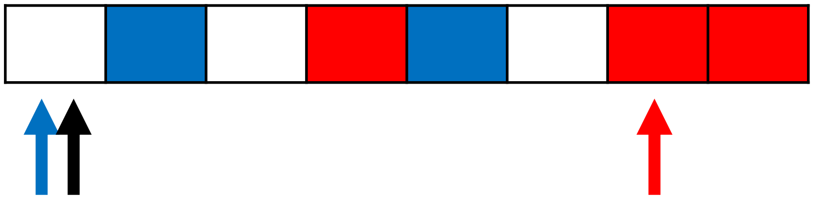
Dutch National Flag Algoithm

```
boundary1=1,
j=1
boundary2 = n
While j <=boundary2
    if array[j] is blue
        swap array[boundary1], array[j]
        boundary1 += 1
        j += 1
    elif array[j] is red
        swap array[j], array[boundary2]
        boundary2 -= 1
    else
        j += 1
```



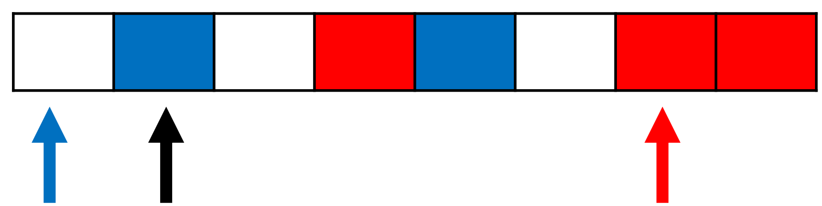
Dutch National Flag Algoithm

```
boundary1=1,
j=1
boundary2 = n
While j <=boundary2
    if array[j] is blue
        swap array[boundary1], array[j]
        boundary1 += 1
        j += 1
    elif array[j] is red
        swap array[j], array[boundary2]
        boundary2 -= 1
    else
        j += 1
```



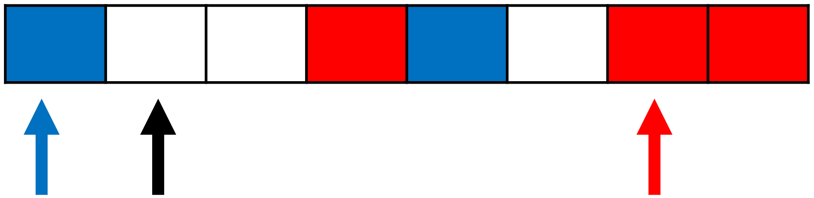
Dutch National Flag Algoithm

```
boundary1=1,
j=1
boundary2 = n
While j <=boundary2
    if array[j] is blue
        swap array[boundary1], array[j]
        boundary1 += 1
        j += 1
    elif array[j] is red
        swap array[j], array[boundary2]
        boundary2 -= 1
    else
        j += 1
```



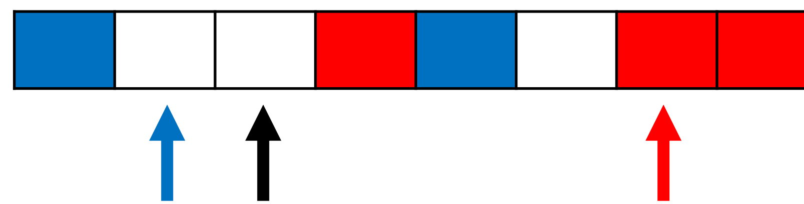
Dutch National Flag Algoithm

```
boundary1=1,
j=1
boundary2 = n
While j <=boundary2
    if array[j] is blue
        swap array[boundary1], array[j]
        boundary1 += 1
        j += 1
    elif array[j] is red
        swap array[j], array[boundary2]
        boundary2 -= 1
    else
        j += 1
```



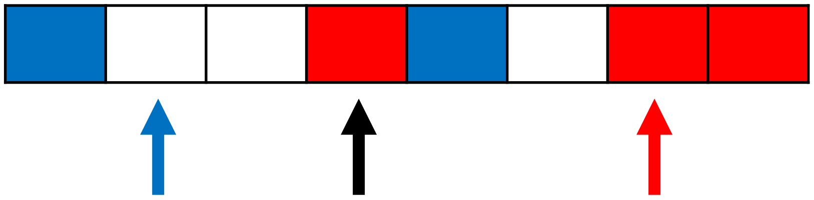
Dutch National Flag Algoithm

```
boundary1=1,
j=1
boundary2 = n
While j <=boundary2
    if array[j] is blue
        swap array[boundary1], array[j]
        boundary1 += 1
        j += 1
    elif array[j] is red
        swap array[j], array[boundary2]
        boundary2 -= 1
    else
        j += 1
```



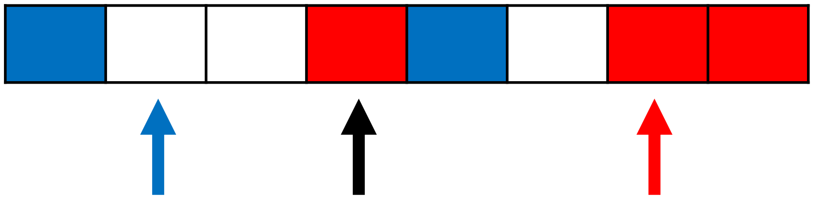
Dutch National Flag Algoithm

```
boundary1=1,
j=1
boundary2 = n
While j <=boundary2
    if array[j] is blue
        swap array[boundary1], array[j]
        boundary1 += 1
        j += 1
    elif array[j] is red
        swap array[j], array[boundary2]
        boundary2 -= 1
    else
        j += 1
```



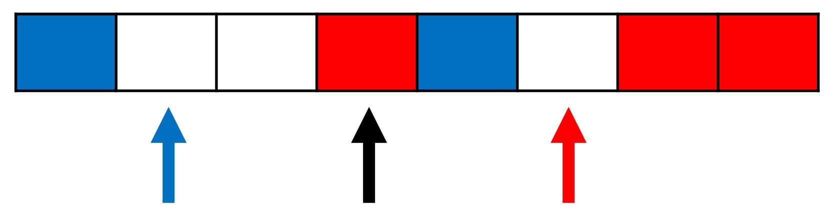
Dutch National Flag Algoithm

```
boundary1=1,
j=1
boundary2 = n
While j <=boundary2
  if array[j] is blue
    swap array[boundary1], array[j]
    boundary1 += 1
    j += 1
  elif array[j] is red
    swap array[j], array[boundary2]
    boundary2 -= 1
  else
    j += 1
```



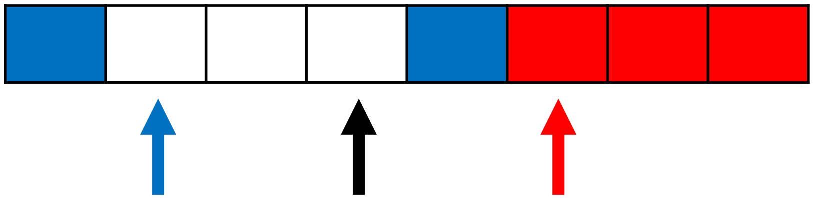
Dutch National Flag Algoithm

```
boundary1=1,
j=1
boundary2 = n
While j <=boundary2
   if array[j] is blue
      swap array[boundary1], array[j]
      boundary1 += 1
      j += 1
   elif array[j] is red
      swap array[j], array[boundary2]
      boundary2 -= 1
   else
      j += 1
```



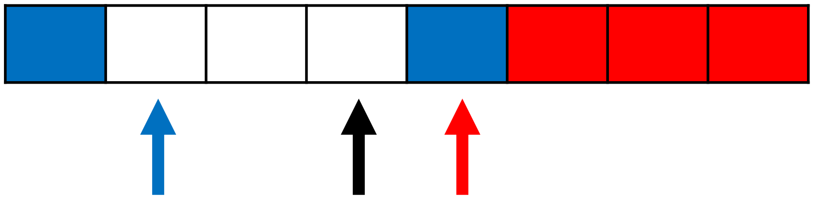
Dutch National Flag Algoithm

```
boundary1=1,
j=1
boundary2 = n
While j <=boundary2
    if array[j] is blue
        swap array[boundary1], array[j]
        boundary1 += 1
        j += 1
    elif array[j] is red
        swap array[j], array[boundary2]
        boundary2 -= 1
    else
        j += 1
```



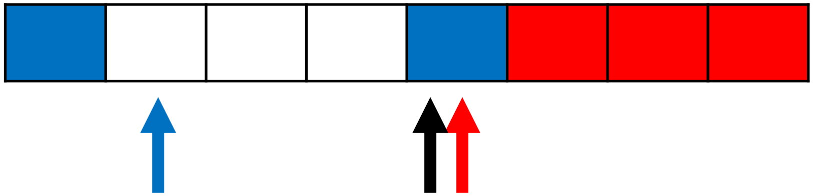
Dutch National Flag Algoithm

```
boundary1=1,
j=1
boundary2 = n
While j <=boundary2
    if array[j] is blue
        swap array[boundary1], array[j]
        boundary1 += 1
        j += 1
    elif array[j] is red
        swap array[j], array[boundary2]
        boundary2 -= 1
    else
        j += 1
```



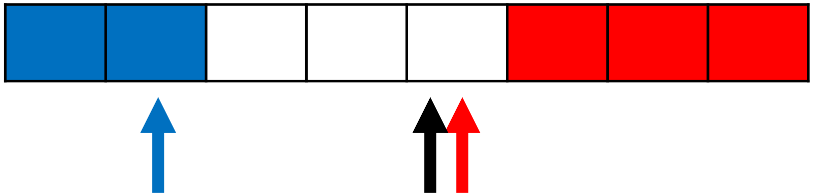
Dutch National Flag Algoithm

```
boundary1=1,
j=1
boundary2 = n
While j <=boundary2
    if array[j] is blue
        swap array[boundary1], array[j]
        boundary1 += 1
        j += 1
    elif array[j] is red
        swap array[j], array[boundary2]
        boundary2 -= 1
    else
        j += 1
```



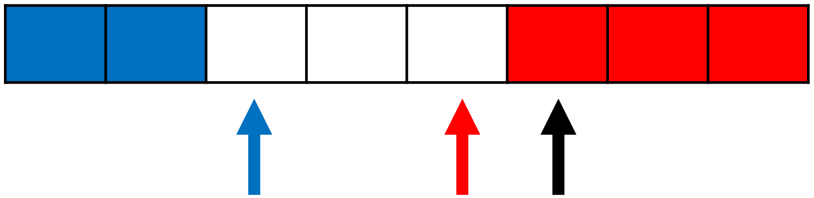
Dutch National Flag Algoithm

```
boundary1=1,
j=1
boundary2 = n
While j <=boundary2
    if array[j] is blue
        swap array[boundary1], array[j]
        boundary1 += 1
        j += 1
    elif array[j] is red
        swap array[j], array[boundary2]
        boundary2 -= 1
    else
        j += 1
```



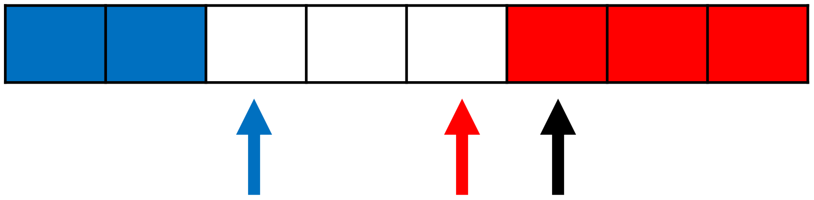
Dutch National Flag Algoithm

```
boundary1=1,
j=1
boundary2 = n
While j <=boundary2
    if array[j] is blue
        swap array[boundary1], array[j]
        boundary1 += 1
        j += 1
    elif array[j] is red
        swap array[j], array[boundary2]
        boundary2 -= 1
    else
        j += 1
```



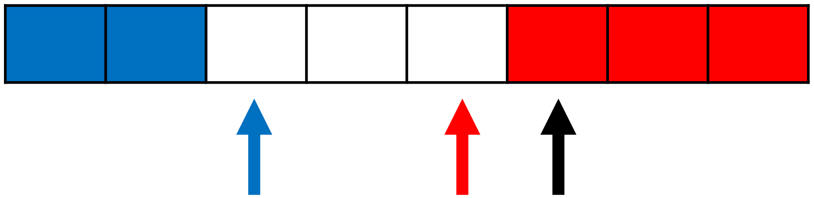
Dutch National Flag Algoithm

```
boundary1=1,
j=1
boundary2 = n
While j <=boundary2
    if array[j] is blue
        swap array[boundary1], array[j]
        boundary1 += 1
        j += 1
    elif array[j] is red
        swap array[j], array[boundary2]
        boundary2 -= 1
    else
        j += 1
Return boundary1, boundary2
```



Dutch National Flag Algoithm

```
boundary1=1,
j=1
boundary2 = n
While j <=boundary2
    if array[j] is blue
        swap array[boundary1], array[j]
        boundary1 += 1
        j += 1
    elif array[j] is red
        swap array[j], array[boundary2]
        boundary2 -= 1
    else
        j += 1
Return boundary1, boundary2
```



Now quicksort the red and blue parts

Partitioning summary

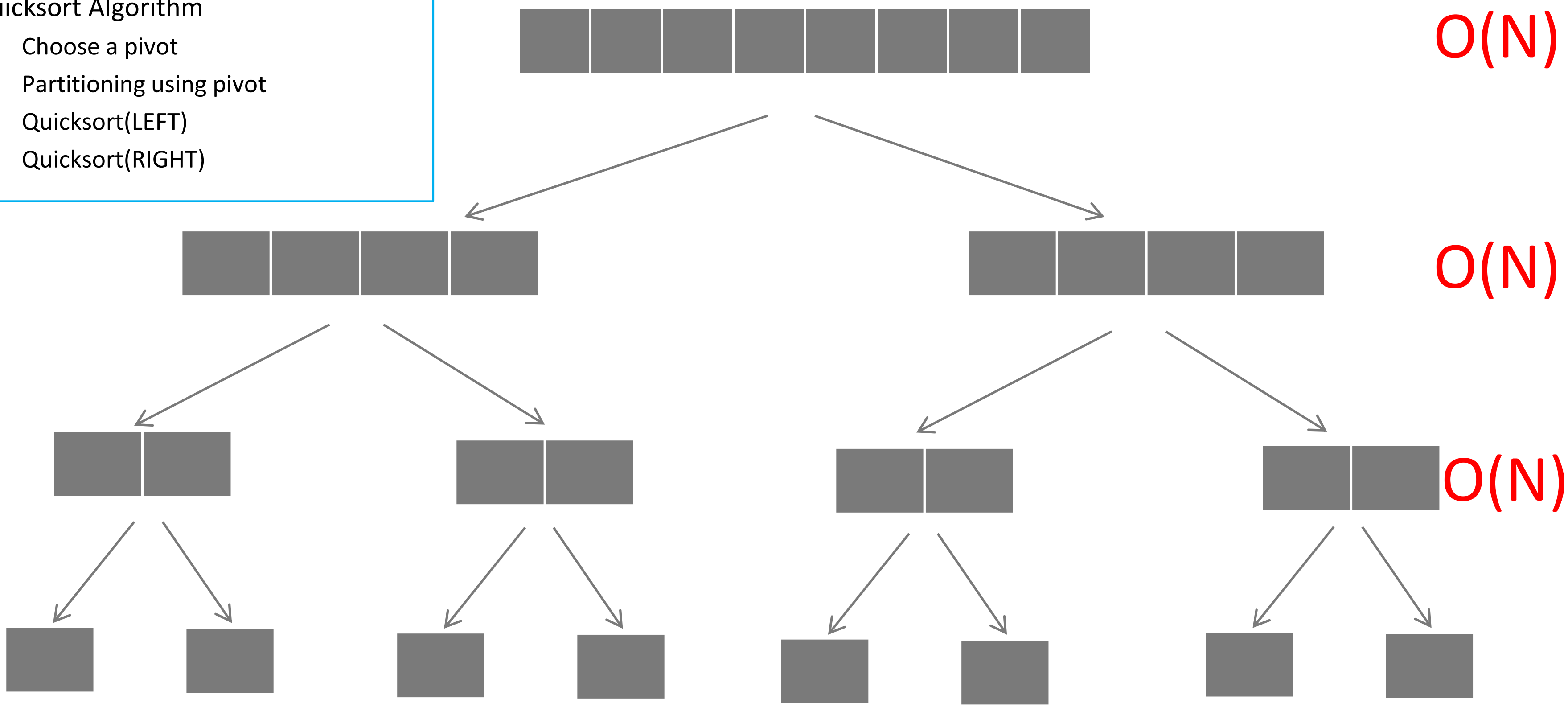
- Lots to consider
- State of the art is more complex
- Objectives
  - Minimise swaps
  - Minimise work in recursive calls
  - Be in place
- How to make these stable? A question for the tute…

Quick Sort and its Analysis

1. Algorithm and partitioning
2. Complexity Analysis
3. Improving Worst-case complexity
   A. Quick Select
   B. Quick Sort in O(N log N) worst-case

Best-case time complexity

Quicksort Algorithm
- Choose a pivot
- Partitioning using pivot
- Quicksort(LEFT)
- Quicksort(RIGHT)

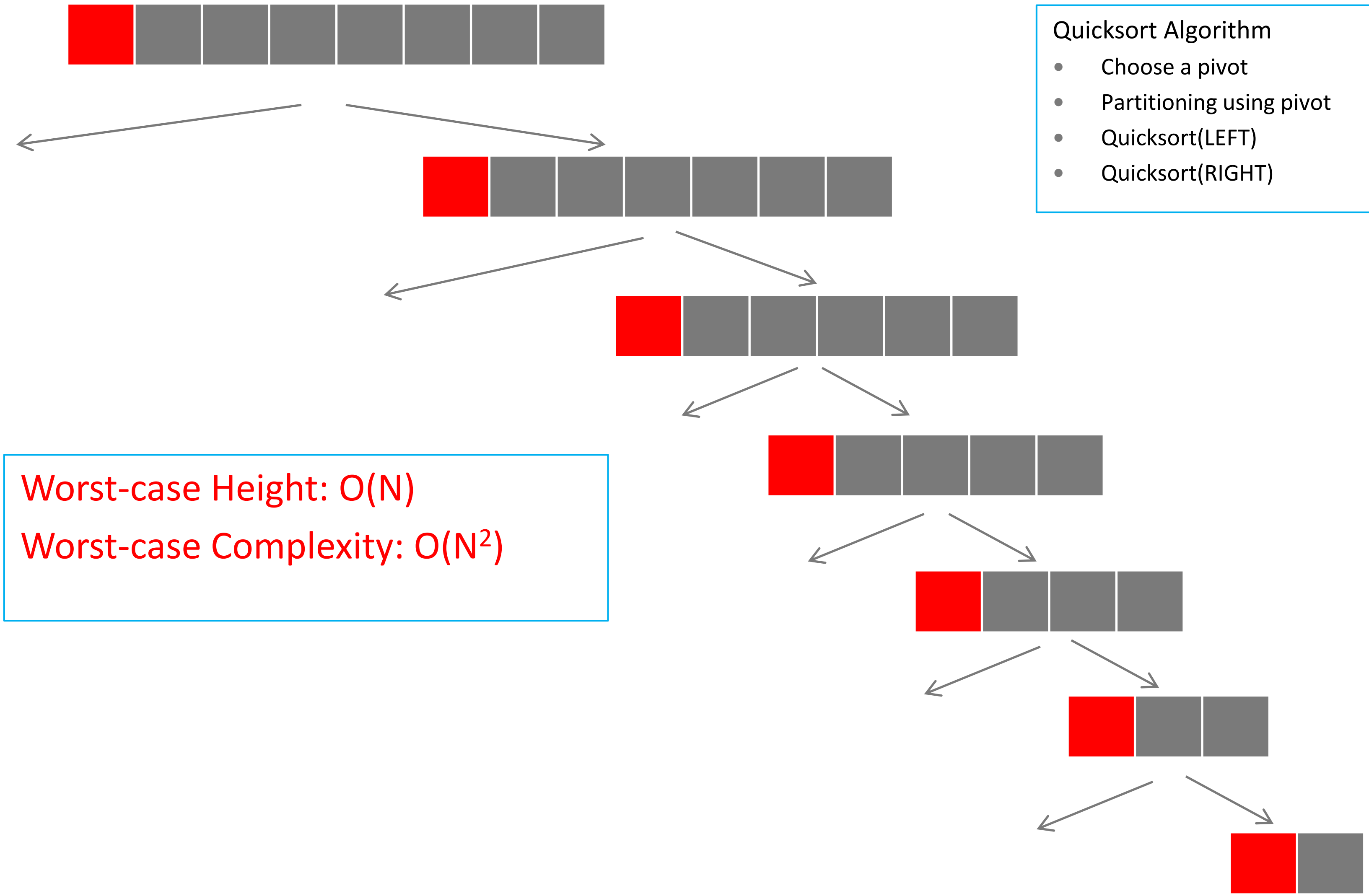


Best-case Height: O(log N)
Best-case complexity: O(N log N)

Important: Quicksort is not in-place even when in-place partitioning is used. Why?

Recursion depth is at least O(log N)

Worst-case Time Complexity



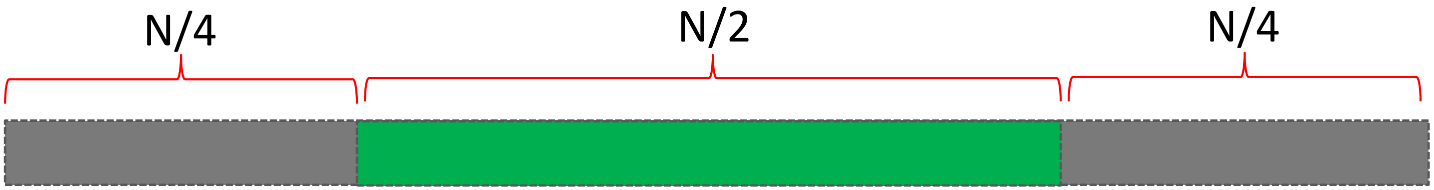
Quicksort Algorithm

- Choose a pivot
- Partitioning using pivot
- Quicksort(LEFT)
- Quicksort(RIGHT)

Worst-case Height: O(N)

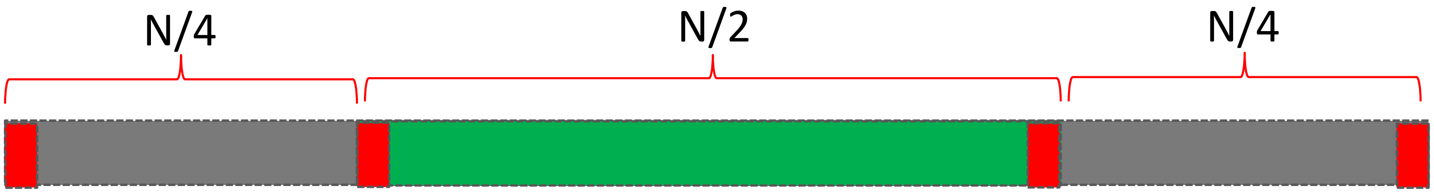
Worst-case Complexity: O(N^2)

Average-case Time complexity



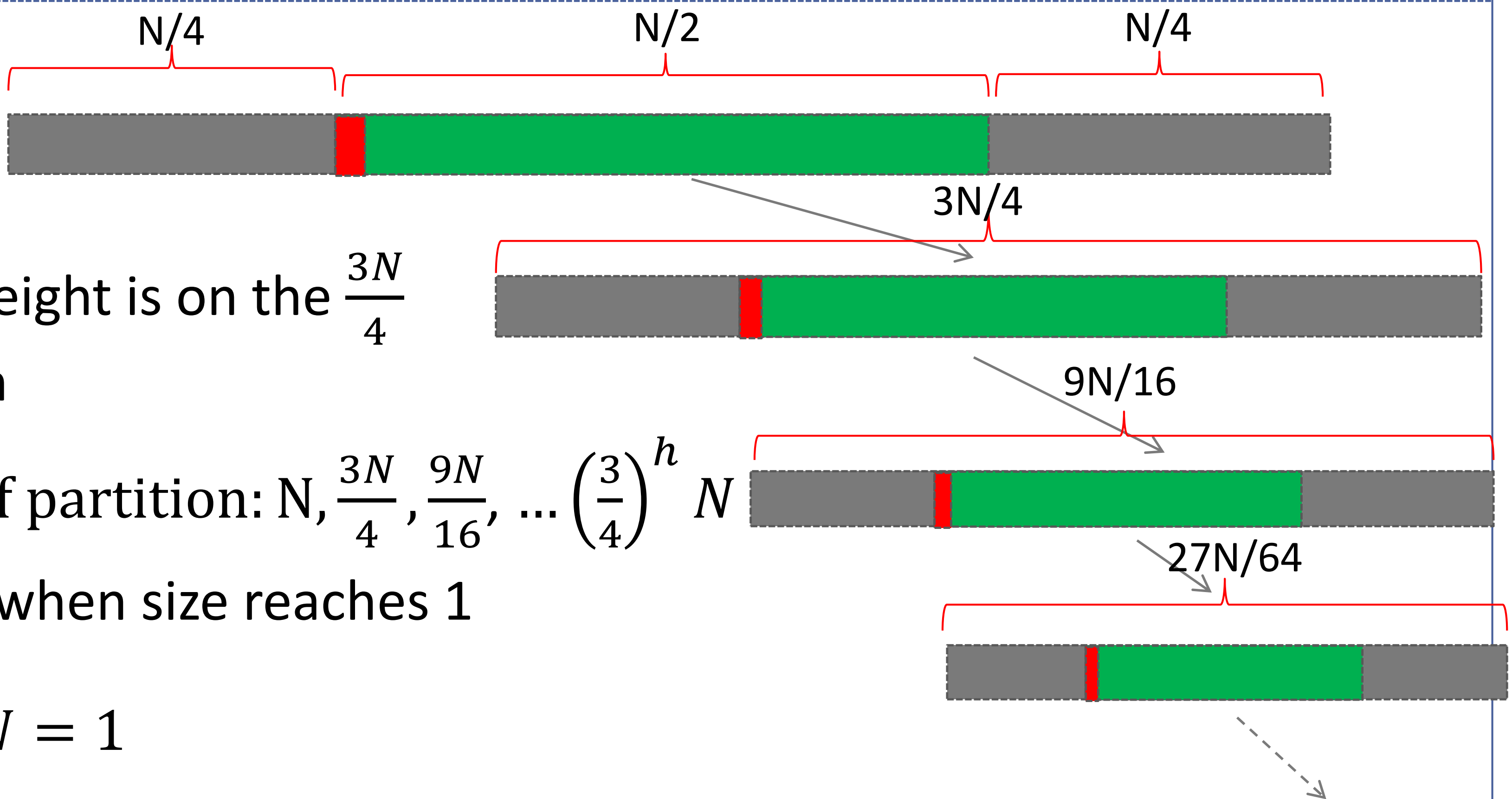
- After partitioning, pivot has 50% probability to be in the green sub-array and has 50% probability to be in one of the two grey sub-arrays.
  - i.e., on average, pivot will be in green half of the time and in grey half of the time

Average-case Time complexity



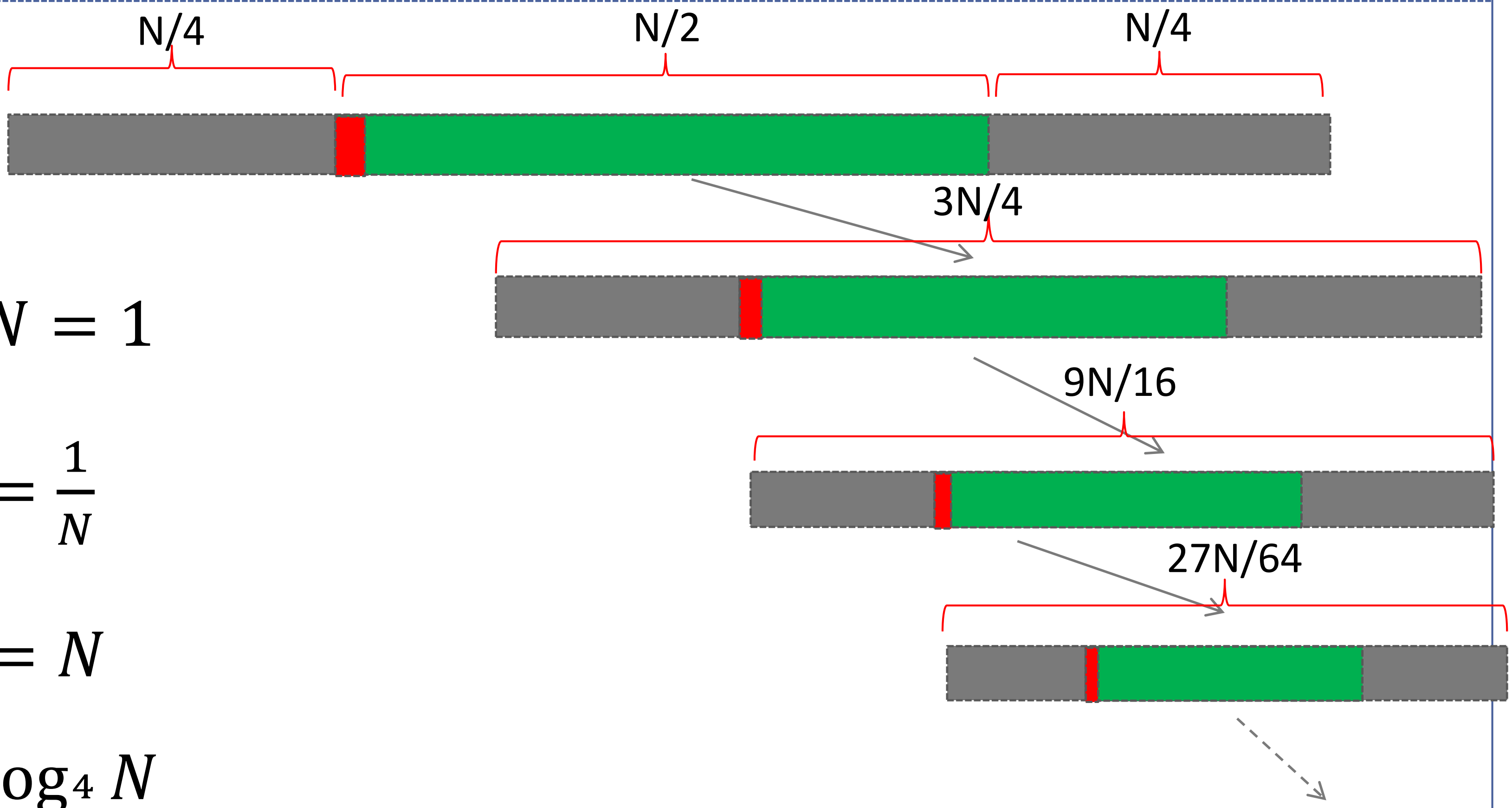
- If pivot is in grey sub-array
  - The worst-case (most unbalanced) partition sizes will be 1 and N-1
- If pivot is in green sub-array
  - The worst-case partition sizes will be N/4 and 3N/4
- For the purpose of the following argument, we assume these one of these worst case scenarios always happen
- The complexity we obtain will therefore be **at least as bad** as the true complexity
- Let h be the height when pivot is **always** in green.

Height when pivot always in green



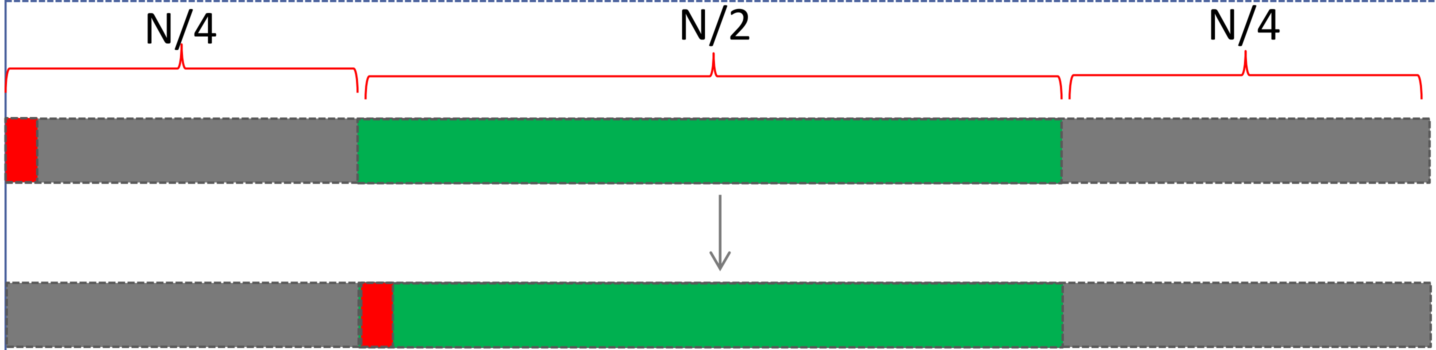
- Max height is on the $\frac{3N}{4}$ branch
- Size of partition: N, $\frac{3N}{4}, \frac{9N}{16}, \ldots \left(\frac{3}{4}\right)^h N$
- Stops when size reaches 1
- $\left(\frac{3}{4}\right)^h N = 1$

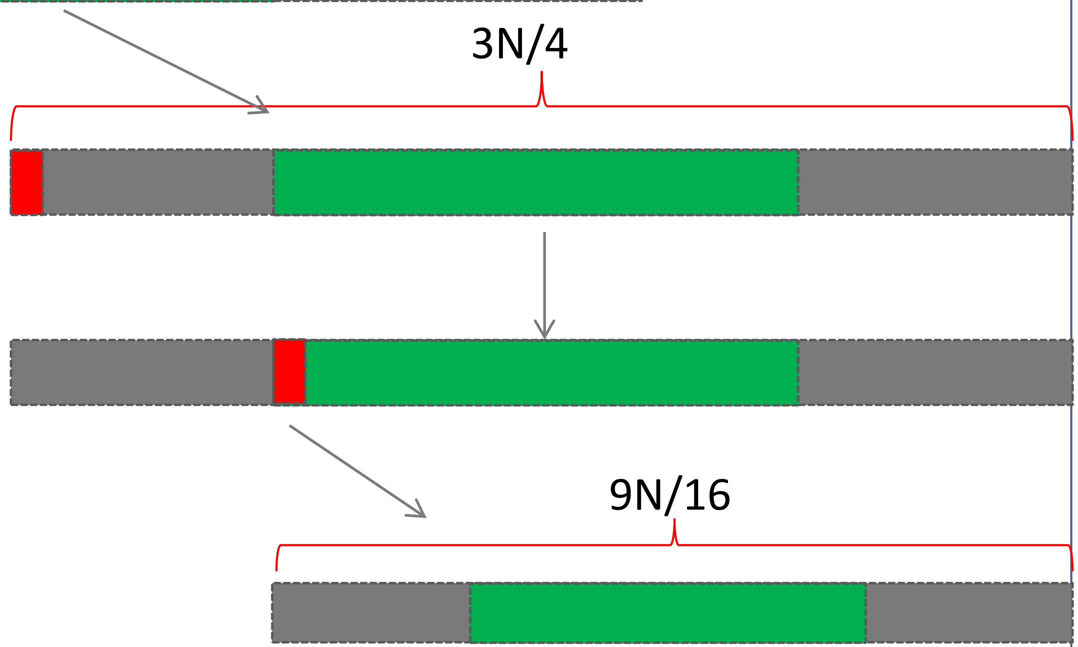
Height when pivot always in green



- $\left(\frac{3}{4}\right)^h N = 1$
- $\left(\frac{3}{4}\right)^h = \frac{1}{N}$
- $\left(\frac{4}{3}\right)^h = N$
- $h = \log_{\frac{4}{3}} N$

Average case





- In reality, pivot will be in green half the time
- Previously we had $h = \log_{\frac{4}{3}} N$
- Now this doubles
- $h = 2\log_{\frac{4}{3}} N$
- h is still $O(\log_{\frac{4}{3}} N)$

Average case Time complexity

- Therefore, height in average case is O(log N)
- Like before, the cost at each level is O(N)
- The average case complexity is thus O(N log N)

Does $O(\log_a N) = O(\log_b N)$ if a and b are constants?

Change of base rule: $$\log_a N = \frac{\log_b N}{\log_b a}$$

So the base of the log doesn't matter for complexity (though it does in practice)

Best-case time complexity using recurrence

Quicksort Algorithm
- Choose a pivot
- Partitioning using pivot
- Quicksort(LEFT)
- Quicksort(RIGHT)

Recurrence relation:

$$T(1) = b$$

$$T(N) = c*N + T(N/2) + T(N/2) = 2*T(N/2) + c*N$$

Solution (exercise in last week):

$$O(N \log N)$$

Worst-case complexity using recurrence

Quicksort Algorithm
- Choose a pivot
- Partitioning using pivot
- Quicksort(LEFT)
- Quicksort(RIGHT)

Recurrence relation:

$$T(1) = b$$

$$T(N) = T(N-1) + c*N$$

Solution:

$$O(N^2)$$

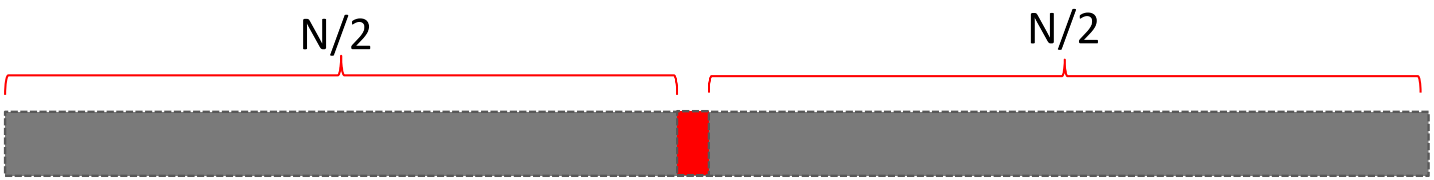
Break Time Problem (not examinable)

- There are 25 horses (who each run at some different fixed speed and never get tired)
- We want to find the 3 fastest
- We can race 5 horses at a time
- We cannot time the horses, only observe the order in which they finish
- How many races do we need?

Quick Sort and its Analysis

1. Algorithm
2. Complexity Analysis
3. Improving Worst-case complexity
   A. Quick Select
   B. Quick Sort in O(N log N) worst-case

Quicksort with O(N log N) in worst-case



Quicksort Algorithm
- Choose median as a pivot
- Partitioning using pivot
- Quicksort(LEFT)
- Quicksort(RIGHT)

Idea:

- Don't choose pivot randomly!
  - Instead, always choose median as the pivot.
  - If we can find median in O(N), the worst-case cost of quicksort would be?
    - O(N log N)
- How do we choose median in O(N)?
- First, we take a detour and see algorithms to answer k-th order statistics

K-th Order Statistics

- Problem: Given an **unsorted** array, find k-th smallest element in the array
  - If k=1 (i.e., find the smallest), we can easily do this in O(N) using the linear algorithm we saw in the last week.
- Median can be computed by setting k appropriately (e.g., k = len(array)/2)
- For general k, how can we solve this efficiently?
  - Sort the elements and return k-th element – takes O(N log N)
  - Can we do better?
    - Yes, Quick Select

Quick Sort and its Analysis

1. Algorithm and partitioning
2. Complexity Analysis
3. Improving Worst-case complexity
   A. Quick Select
   B. Quick Sort in O(N log N) worst-case

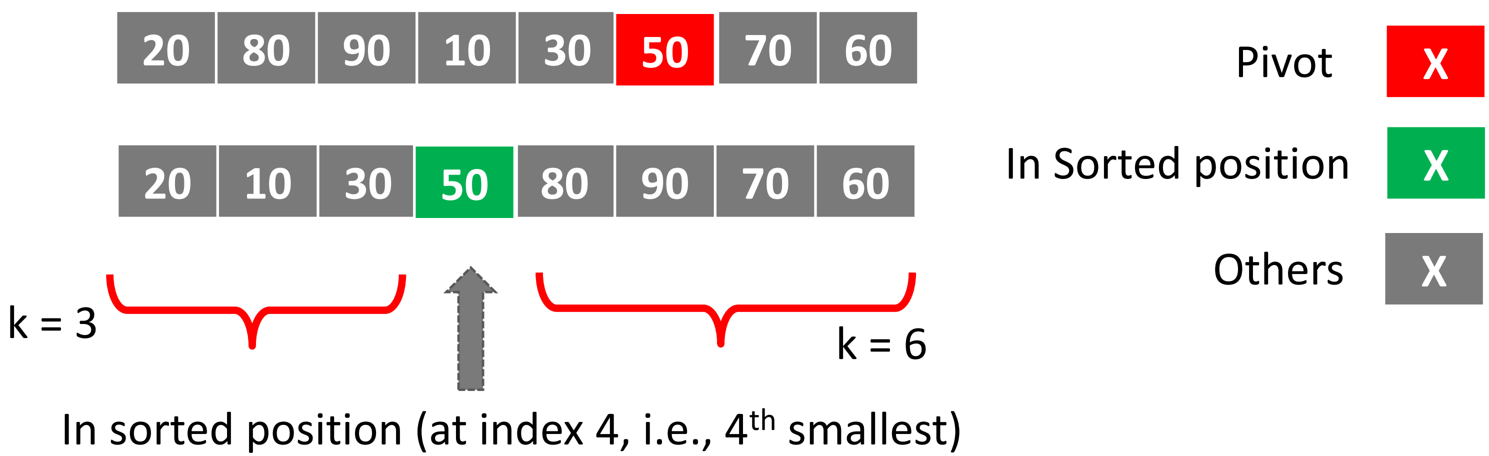
Quick Select

- Choose a pivot p
- Partition the array in two sub-arrays w.r.t. p (same partitioning as in quicksort)
  - LEFT ← elements smaller than or equal to p
  - RIGHT ← elements greater than p
- If index(pivot) == k:
  - Return pivot
- If k > index(pivot)
  - QuickSelect(RIGHT)
- Else:
  - QuickSelect(LEFT)

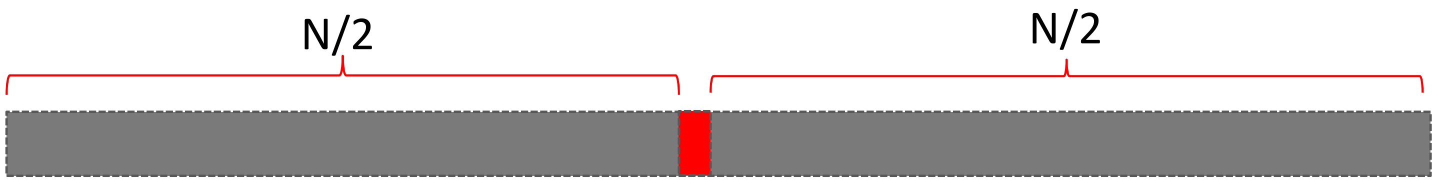
Best-case time complexity?
- $O(N)$

Worst-case time complexity?
- $O(N^2)$

Average-case time complexity?
- $O(N)$ – same arguments as for quicksort



Quicksort with O(N log N) in worst-case



Quicksort Algorithm

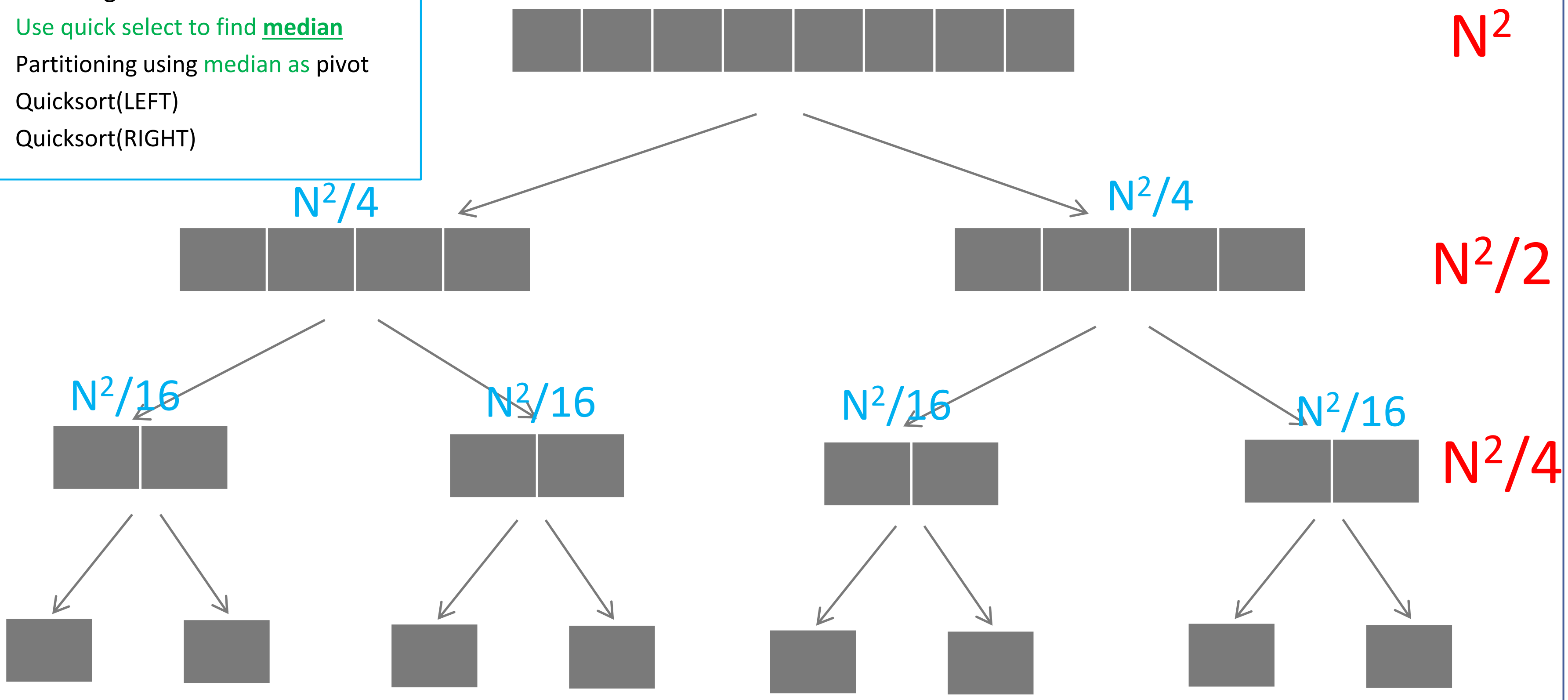
- Use quick select to find **median**
- Partitioning using median as pivot
- Quicksort(LEFT)
- Quicksort(RIGHT)

- Call **Quick Select** with k=len(array)/2?
- The value returned by Quick Select will be median.
- Choose this as the pivot.
- What will be the best-case cost of such quick sort?
  - O(N log N)
- What is the worst-case cost?

Quick Sort Worst-case when using Quick Select to choose pivot

Quicksort Algorithm

- Use quick select to find **median**
- Partitioning using median as pivot
- Quicksort(LEFT)
- Quicksort(RIGHT)



Worst-case cost at level k: $N^2/2^k$

Total cost: $N^2 + N^2/2 + N^2/4 + \ldots + 1 = N^2(1 + ½ + ¼ + \ldots)$

$= O(N^2)$

Where are we?

- Trying to make quicksort Nlog(N) in the worst case
- Need to find median in O(N)
- We have an algorithm (quickselect) which finds median in O(N) in the best case (and average case)…
- But it is $O(N^2)$ in the worst case (which would make quicksort **slower**)… sigh
- We want to make quickselect always take O(N)
- What do we need? A median pivot for **quickselect!**

Where are we?

- What do we need? A median pivot for **quickselect!**
- **But that is what quickselect is meant to do…**
- Sounds impossible – in order for quickselect to run in O(N) we need to find a good (i.e. median) pivot in O(N), but that was exactly the problem quickselect was meant to solve!
- The trick – relax definition of a "good pivot"
- A good pivot is anything which cuts the list into fixed fractions
- E.g. it would be enough to always cut it 70:30
- Even 99:1 would be ok for NlogN, but slower in practice, so the closer to 50:50 the better

Quick Sort and its Analysis

1. Algorithm and partitioning
2. Complexity Analysis
3. Improving Worst-case complexity
   A. Quick Select
   B. Quick Sort in O(N log N) worst-case

Median of medians (not examinable)

1	15	10	10	7	20	8	19	11	2	12	16	12
12	20	5	8	2	6	19	1	15	4	13	20	2
15	17	10	14	13	7	15	7	11	10	16	18	10
7	2	15	4	20	16	18	1	8	17	16	6	17
7	8	16	18	19	20	12	10	11	1	19	13	5

Median of medians (not examinable)

- Sort groups of size five

Bigger

15	20	16	18	20	20	19	19	15	17	19	20	17
12	17	15	14	19	20	18	10	11	10	16	18	12
7	15	10	10	13	16	15	7	11	4	16	16	10
7	8	10	8	7	7	12	1	11	2	13	13	5
1	2	5	4	2	6	8	1	8	1	12	6	2

Smaller

Median of medians (not examinable)

Sort groups of size five

Find the medians

Bigger

15	20	16	18	20	20	19	19	15	17	19	20	17
12	17	15	14	19	20	18	10	11	10	16	18	12
7	15	10	10	13	16	15	7	11	4	16	16	10
7	8	10	8	7	7	12	1	11	2	13	13	5
1	2	5	4	2	6	8	1	8	1	12	6	2

Smaller

Median of medians (not examinable)

- Sort groups of size five
- Find the medians
- Find the median of those!
- (Note that the columnd **do not** actually get sorted, just shown here in sorted order for clarity)

Bigger

17	15	19	16	18	17	15	20	20	19	20	19	20
10	12	10	15	14	12	11	19	17	18	20	16	18
4	7	7	10	10	10	11	13	15	15	16	16	16
2	7	1	10	8	5	11	7	8	12	7	13	13
1	1	1	5	4	2	8	2	2	8	6	12	6

Smaller

Smaller

Bigger

Median of medians (not examinable)

- Median of medians is bigger than half the medians

Bigger

Smaller

17	15	19	16	18	17	15	20	20	19	20	19	20
10	12	10	15	14	12	11	19	17	18	20	16	18
4	7	7	10	10	10	11	13	15	15	16	16	16
2	7	1	10	8	5	11	7	8	12	7	13	13
1	1	1	5	4	2	8	2	2	8	6	12	6

Bigger

Smaller

Median of medians (not examinable)

- Median of medians is bigger than half the medians
- So it is bigger than all the red values as well

Bigger

Smaller

17	15	19	16	18	17	15	20	20	19	20	19	20
10	12	10	15	14	12	11	19	17	18	20	16	18
4	7	7	10	10	10	11	13	15	15	16	16	16
2	7	1	10	8	5	11	7	8	12	7	13	13
1	1	1	5	4	2	8	2	2	8	6	12	6

Bigger

Smaller

Median of medians (not examinable)

- Median of medians is smaller than half the medians

Bigger

Smaller

17	15	19	16	18	17	15	20	20	19	20	19	20
10	12	10	15	14	12	11	19	17	18	20	16	18
4	7	7	10	10	10	11	13	15	15	16	16	16
2	7	1	10	8	5	11	7	8	12	7	13	13
1	1	1	5	4	2	8	2	2	8	6	12	6

Bigger

Smaller

Median of medians (not examinable)

- Median of medians is smaller than half the medians
- So it is smaller than the green values as well

Bigger

17	15	19	16	18	17	15	20	20	19	20	19	20
10	12	10	15	14	12	11	19	17	18	20	16	18
4	7	7	10	10	10	11	13	15	15	16	16	16
2	7	1	10	8	5	11	7	8	12	7	13	13
1	1	1	5	4	2	8	2	2	8	6	12	6

Smaller

Smaller

Bigger

Median of medians (not examinable)

- Median of medians is greater than 30% and also less than 30%, so its in the middle 40%
- The worst split we can get using the MoM is 70:30!
- However, we did need to find the exact median of n/5 items… how?

Bigger

Smaller

17	15	19	16	18	17	15	20	20	19	20	19	20
10	12	10	15	14	12	11	19	17	18	20	16	18
4	7	7	10	10	10	11	13	15	15	16	16	16
2	7	1	10	8	5	11	7	8	12	7	13	13
1	1	1	5	4	2	8	2	2	8	6	12	6

Bigger

Smaller

Quicksort with O(N log N) in worst-case (not examinable)

Median_of_medians(list[1..n])

divide into sublists of size 5

medians = [median of each sublist]

use quickselect to find the median of **medians**

Quicksort with O(N log N) in worst-case (not examinable)

```
Median_of_medians(list[1..n])
    if n <= 5
        use insertion sort to find the median, and return it
    divide into sublists of size 5
    medians = [median of each sublist]
    use quickselect to find the median of medians
```

Quicksort with O(N log N) in worst-case (not examinable)

```
Median_of_medians(list[1..n])
    if n <= 5
        use insertion sort to find the median, and return it
    divide into sublists of size 5
    medians = [median of each sublist]
    return quickselect(medians, (len(medians)+1)/2)
```

Quicksort with O(N log N) in worst-case (not examinable)

This call uses quickselect!
But with a weaker pivot

```
Quickselect(list, lo, hi, k)
    if lo > hi
        return array[k]
    pivot = median_of_medians(list, lo, hi, k)
    mid = partition(array, lo, hi, pivot)
    if mid > k
        return quickselect(array, lo, mid-1, k)
    elif k > mid
        return quickselect(array, mid+1, hi, k)
    else
        return array[k]
```

Quicksort with O(N log N) in worst-case (not examinable)

This call uses quickselect!
But with a weaker pivot

Quickselect(list, lo, hi, k)
    if lo > hi
        return array[k]
    pivot = **median_of_medians**(list, lo, hi, k) (
    mid = *partition*(array, lo, hi, pivot) **(70:30 pivot in worst)**
    if mid > k
        return *quickselect*(array, lo, mid-1, k) **(n/7 in worst)**
    elif k > mid
        return *quickselect*(array, mid+1, hi, k) **(n/7 in worst)**
    else
        return array[k]

Quicksort with O(N log N) in worst-case (not examinable)

Quickselect time complexity recurrence

$$T(n) = T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + an$$

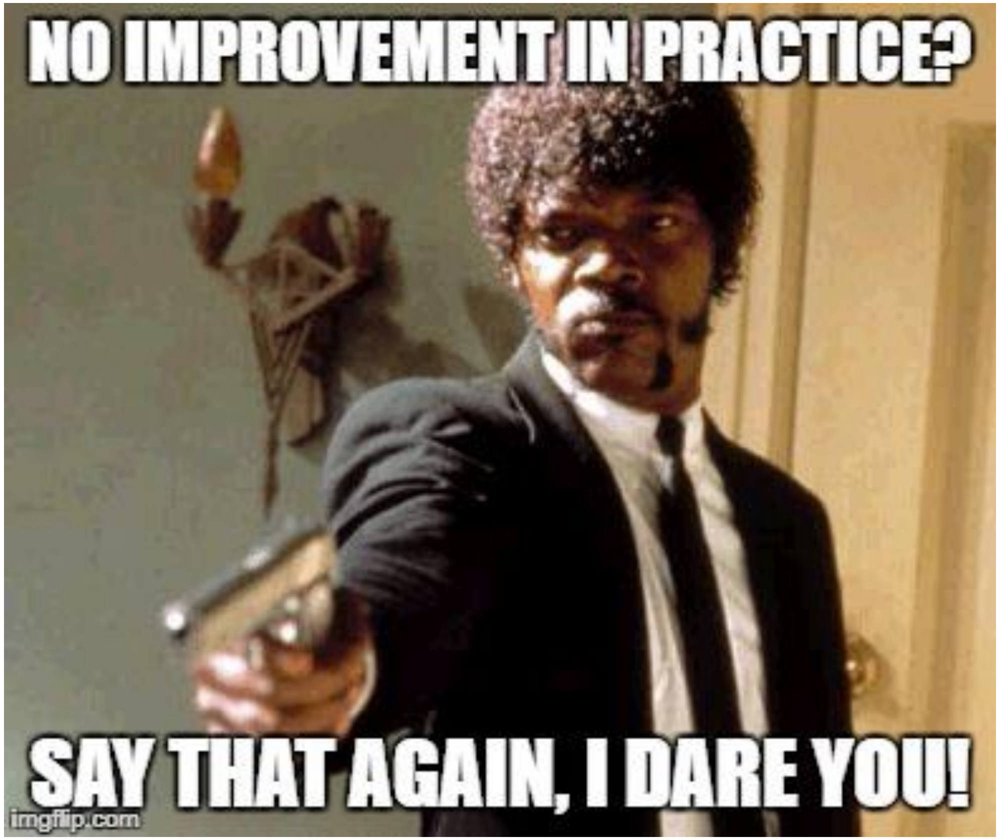
- $T\left(\frac{n}{5}\right)$ for recursing on the list of the medians of groups of 5 (inside the call to median of medians)
- $T\left(\frac{7n}{10}\right)$ for the main recursive call, which is guaranteed to have split the list at least 30:70 (because the pivot was selected by MoM)
- an for the linear time partition algorithm + time to find medians of groups of five

Solving this give linear time!

So armed with a linear time quickselect, we can now quicksort in NlogN worst case…

Anticlimax (examinable)

- Although using “median of medians” reduces worst-case complexity to O(N log N), in practice choosing random pivots works better.
  - However, theoretical improvement in worst-case is quite satisfying.



Concluding Remarks

Summary

- Quicksort and its analysis. Quicksort can be made O(N log N) in worst-case which is mostly of theoretical interest but does not usually improve performance in practice.
- It is better to do a simple pivot selection which takes less time (like random selection)

Coming Up Next

- Dynamic Programming – (super important and powerful tool, **assignment 2 is all about dynamic programming)**

Things to do before next lecture

- Make sure you understand this lecture completely especially the (examinable) average-case complexity analysis of quicksort

Average-case complexity using recurrence
(NOT EXAMINABLE)

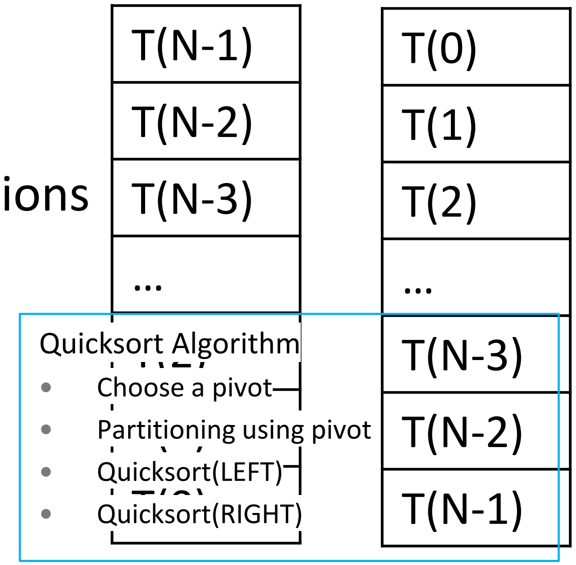
Recurrence relation:

T(N) = ???

- For simplicity, assume partitioning costs (N+1) operations
- Assume pivot is at index k

$T_k(N) = (N+1) + T(N-k) + T(k-1)$

- Average cost is the average for k being from 1 to N

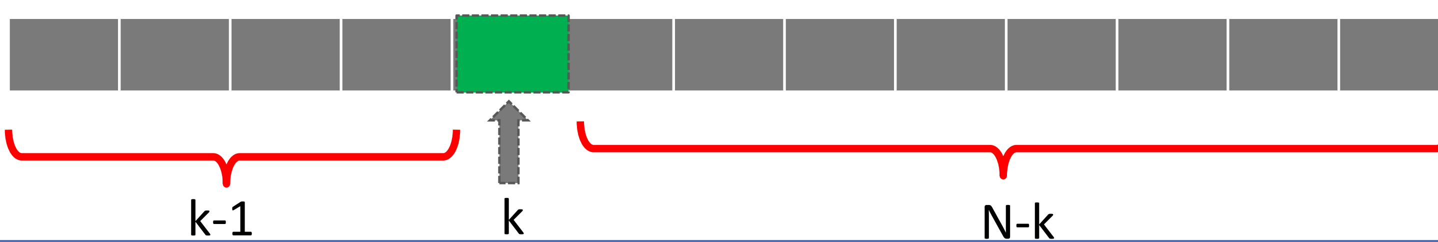


$$T(N) = \frac{\sum_{k=1}^{N} T_k(N)}{N}$$

$$\sum_{k=1}^{N} T(N-k) = \sum_{k=1}^{N} T(k-1)$$

$$T(N) = (N+1) + \frac{\sum_{k=1}^{N} T(N-k) + T(k-1)}{N}$$

$$T(N) = (N+1) + \frac{2}{N}\sum_{k=1}^{N} T(k-1)$$



Average-case complexity using recurrence
(NOT EXAMINABLE)

Recurrence relation:

T(1) = b

Multiplying N on both sides

$$T(N) = (N+1) + \frac{2}{N}\sum_{k=1}^{N} T(k-1)$$

$$N.T(N) = N(N+1) + 2\sum_{k=1}^{N} T(k-1) \longrightarrow \text{(A)}$$

$$(N-1).T(N-1) = N(N-1) + 2\sum_{k=1}^{N-1} T(k-1) \longrightarrow \text{(B)}$$

$$N.T(N) - (N-1).T(N-1) = 2N + 2T(N-1)$$

(A) – (B)

$$N.T(N) = 2N + 2T(N-1) + (N-1).T(N-1) = 2N + (N+1).T(N-1)$$

Simplify

$$T(N) = 2 + \frac{N+1}{N} T(N-1)$$

Divide both sides by N

Average-case complexity using recurrence
(NOT EXAMINABLE)

Recurrence relation:

T(1) = b $\qquad T(N) = 2 + \frac{N+1}{N}T(N-1) \longrightarrow$ (A)

Let's solve it:

$T(N-1) = 2 + \frac{N}{N-1}T(N-2)$ Cost for T(N-1)

Replace T(N-1) in (A)

$$T(N) = 2 + \frac{N+1}{N}(2 + \frac{N}{N-1}T(N-2)) = 2 + \frac{2(N+1)}{N} + \frac{N+1}{N-1}T(N-2) \longrightarrow \text{(B)}$$

$T(N-2) = 2 + \frac{N-1}{N-2}T(N-3)$ Cost for T(N-2)

Replace T(N-2) in (B)

$$T(N) = 2 + \frac{2(N+1)}{N} + \frac{2(N+1)}{N-1} + \frac{2(N+1)}{N-2}T(N-3)$$

$$T(N) = 2 + \frac{2(N+1)}{N} + \frac{2(N+1)}{N-1} + \frac{2(N+1)}{N-2} + \ldots + \frac{2(N+1)}{N-k+2} + \frac{2(N+1)}{N-k+1}T(N-k)$$

See the pattern for k?

Average-case complexity using recurrence
(NOT EXAMINABLE)

Recurrence relation:

T(1) = b $\qquad T(N) = 2 + \frac{N+1}{N} T(N-1)$

Let's solve it:

$$T(N) = 2 + \frac{2(N+1)}{N} + \frac{2(N+1)}{N-1} + \frac{2(N+1)}{N-2} + \ldots + \frac{2(N+1)}{N-k+2} + \frac{2(N+1)}{N-k+1} T(N-k)$$

N-k =1 → k = N-1

$$T(N) = 2 + \frac{2(N+1)}{N} + \frac{2(N+1)}{N-1} + \frac{2(N+1)}{N-2} + \ldots + \frac{2(N+1)}{3} + \frac{2(N+1)}{2} T(1)$$

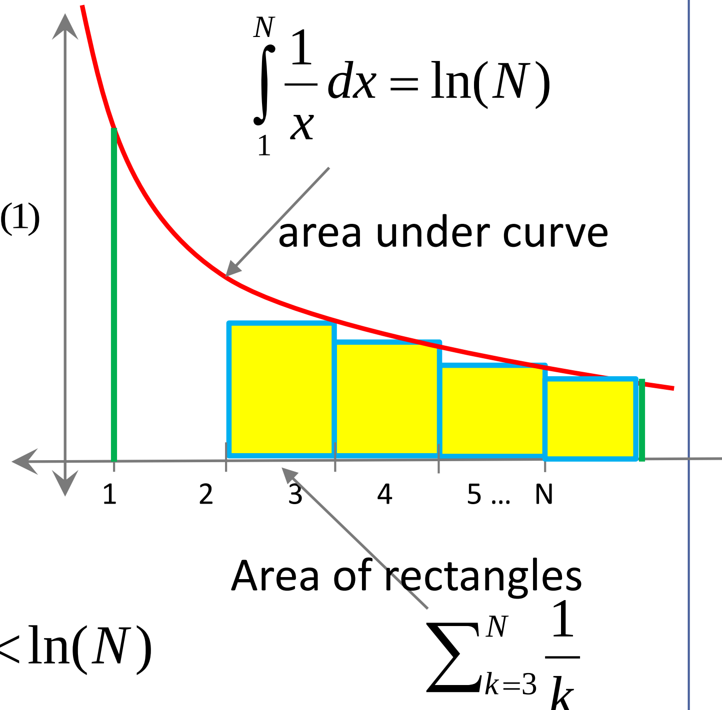
Simplify

$$T(N) = 2 + 2(N+1)\left(\frac{1}{N} + \frac{1}{N-1} + \frac{1}{N-2} + \ldots + \frac{1}{3}\right) + b(N+1)$$

$$T(N) = 2 + b(N+1) + 2(N+1)\sum_{k=3}^{N} \frac{1}{k}$$

$$T(N) < 2 + b(N+1) + 2(N+1)\ln(N)$$

T(N) = O (N log N)



$$\sum_{k=3}^{N} \frac{1}{k} < \ln(N)$$