Faculty of Information Technology, Monash University

FIT2004: Algorithms and Data Structures

Week 3: Quick Sort and Quick Select

Overview

Divide and conquer (W 1-3)	Greedy algorithms (W 4-5)	Dynamic programming (W 6-7)	Network flow (W 8-9)	Data structures (W 10-11)

- What we covered so far?
  - Divide and conquer algorithm design paradigm
  - Complexity analysis for recursive algorithms (recurrence relations)
  - Correctness of algorithms
  - Non-comparison based algorithms: counting and radix sort
- Today's lecture
  - Quick sort and Quick select

Quicksort and its Analysis

1. Algorithm and partitioning
2. Complexity analysis
3. Improving worst-case complexity
   A. Quick select
   B. Quicksort in O(N log N) worst-case

Quicksort 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

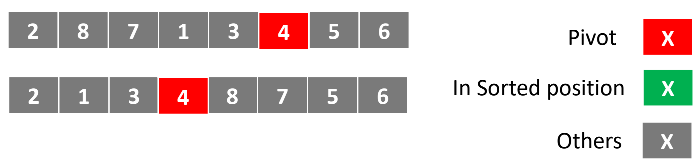
Quicksort Example

- Choose a pivot p

2	8	7	1	3	4	5	6

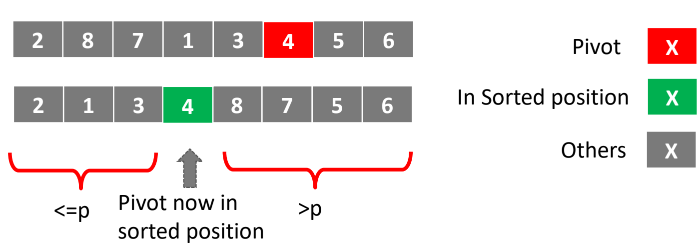
Quicksort Example

- 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 Example

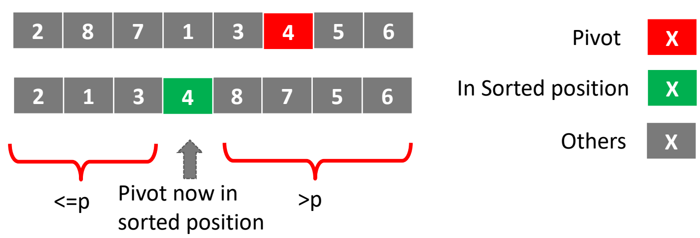
- Choose a pivot p
- Partition the array in two sub-arrays w.r.t. p
  - LEFT ← elements smaller than or equal to p
  - RIGHT ← elements greater than p



Quicksort Example

- 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

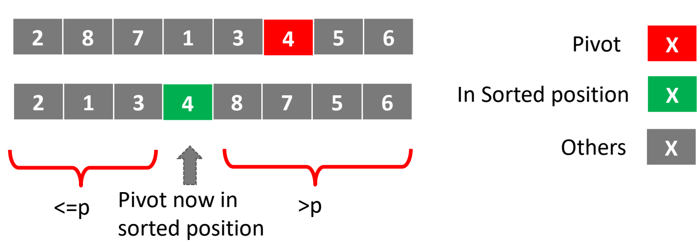
Partitioning



Quicksort Example

- 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)

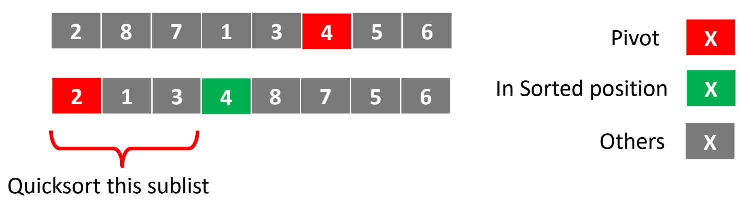
Partitioning



Quicksort Example

- 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)

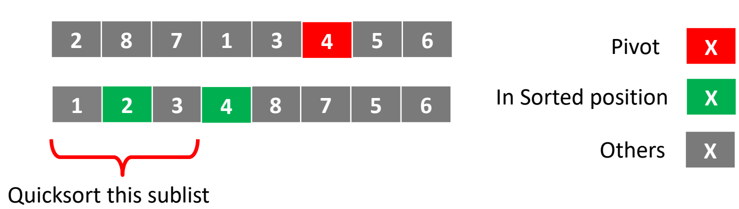
Partitioning



Quicksort Example

- 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 Example

- 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

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

Quicksort this sublist

Pivot	X
In Sorted position	X
Others	X

Quicksort Example

- 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

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

Pivot	X
In Sorted position	X
Others	X

Quicksort Example

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)

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

Quicksort this sublist

Pivot X

In Sorted position X

Others X

Quicksort Example

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)

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

Quicksort this sublist
(not shown in slides)

Pivot X

In Sorted position X

Others X

Quicksort Algorithm

```
Quicksort(A, p, r)
    if p < r
        then q ← Partition(A, p, r)
                Quicksort(A, p, q-1)
                Quicksort(A, q+1, r)
```

Initial call is `Quicksort(A, 1, len(A) )`

Naïve Partitioning Algorithm: Out-of-place

- 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

Naïve Partitioning Algorithm: Out-of-place

- 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

Naïve Partitioning Algorithm: Out-of-place

- 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

Naïve Partitioning Algorithm: Out-of-place

- 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	

Naïve Partitioning Algorithm: Out-of-place

- 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	

Naïve Partitioning Algorithm: Out-of-place

- 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

Naïve Partitioning Algorithm: Out-of-place

- 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

Naïve Partitioning Algorithm: Out-of-place

- 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

Naïve Partitioning Algorithm: Out-of-place

- 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

Naïve Partitioning Algorithm: Out-of-place

- 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

Naïve Partitioning Algorithm: Out-of-place

- 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

Naïve Partitioning Algorithm: Out-of-place

- 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	

Naïve Partitioning Algorithm: Out-of-place

- 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	

Naïve Partitioning Algorithm: Out-of-place

- 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

Naïve Partitioning Algorithm: Out-of-place

- 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

Naïve Partitioning Algorithm: Out-of-place

- 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

Naïve Partitioning Algorithm: Out-of-place

- 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	8	7	1	3	4	5	6

2	1	3		4		8	7	5	6

LEFT RIGHT

Naïve Partitioning Algorithm: Out-of-place

- 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?

Naïve Partitioning Algorithm: Out-of-place

- 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
- Can we make it stable? Yes, how? See lecture notes Algorithm 15, page 27

Naïve Partitioning Algorithm: Out-of-place

Activity 1:

Find out how to make the algorithm stable.

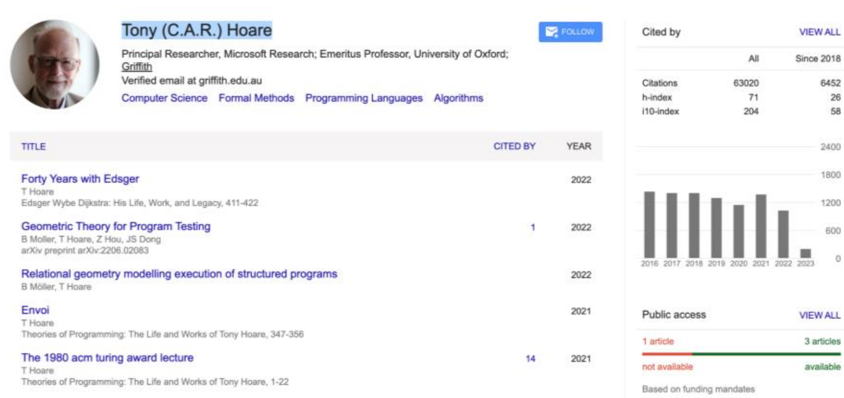
RECAP:

In-place algorithm: An algorithm that has O(1) auxiliary space complexity. i.e., it only requires constant space in addition to the space taken by the input

Stable algorithm: An algorithm is called stable if it maintains the relative ordering of elements that have equal keys. This applies mostly in sorting algorithms.

Algorithm's Heroes

- Tony Hoare is a British computer scientist, and winner of the 1980 Turing Award.
- He is best known for his fundamental contributions to the definition and design of programming languages, and for the development of Quicksort



Hoare Partitioning Algorithm: In-place

Swap pivot to the front (position 1)

2	8	6	4	1	7	3	5

Hoare Partitioning Algorithm: In-place

Swap pivot to the front (position 1)

L_bad = 2, R_bad = N

4	8	6	2	1	7	3	5

Hoare Partitioning Algorithm: In-place

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	8	6	2	1	7	3	5

Hoare Partitioning Algorithm: In-place

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	8	6	2	1	7	3	5

Hoare Partitioning Algorithm: In-place

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	8	6	2	1	7	3	5

Hoare Partitioning Algorithm: In-place

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

Hoare Partitioning Algorithm: In-place

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

Hoare Partitioning Algorithm: In-place

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

Hoare Partitioning Algorithm: In-place

Swap pivot to the front (position 1)

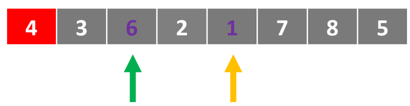
L_bad = 2, R_bad = N

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



Hoare Partitioning Algorithm: In-place

Swap pivot to the front (position 1)

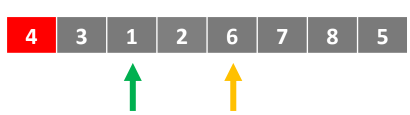
L_bad = 2, R_bad = N

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



Hoare Partitioning Algorithm: In-place

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	1	2	6	7	8	5

Hoare Partitioning Algorithm: In-place

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	1	2	6	7	8	5

Hoare Partitioning Algorithm: In-place

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. $\leq$ pivot

  swap these elements

4	3	1	2	6	7	8	5

Hoare Partitioning Algorithm: In-place

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. $\leq$ pivot

  swap these elements

swap pivot to R_bad

4	3	1	2	6	7	8	5

Hoare Partitioning Algorithm: In-place

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

2	3	1	4	6	7	8	5

Hoare Partitioning Algorithm: In-place

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

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 > pivot
  move R_bad left until ≤ pivot
  swap these elements
swap pivot to R_bad

At the start of each iteration of the loop:

  All elements on left side of L_bad are less than or equal to the pivot.

  All elements on right side of R_bad are greater than or equal to the pivot.

  The elements in between L_bad and R_bad are yet to be partitioned.

Hoare Partitioning Algorithm: In-place

- Pros:
  - Each element only swapped once (except pivot)
  - Simple idea
  - Simple invariant (what is it?)
- Cons:
  - Very tricky to implement without bugs
    - **Termination conditions** (It only ensures proper partitioning but not necessarily the correct final position of the pivot.)
    - **Edge cases** (When the array contains only 2 elements or all elements are identical, special handling is needed to ensure correct behavior.)
    - **Off by one errors** (as it uses two moving pointers that traverse towards each other, the algorithm may fail if they are not updated correctly)
  - Not stable
  - What would be the performance if all elements in the array have the same value?

Hoare Partitioning Algorithm: In-place

- Pros:
  - Each element only swapped once (except pivot)
  - Simple idea
  - Simple invariant (what is it?)
- Cons:
  - Very tricky to implement without bugs
    - **Termination conditions** (It only ensures proper partitioning but not necessarily the correct final position of the pivot.)
    - **Edge cases** (When the array contains only 2 elements or all elements are identical, special handling is needed to ensure correct behavior.)
    - **Off by one errors** (as it uses two moving pointers that traverse towards each other, the algorithm may fail if they are not updated correctly)
  - Not stable
  - What would be the performance if all elements in the array have the same value? **It still works correctly as the loop terminates when $i \geq j$.**

Hoare Partitioning Algorithm: In-place

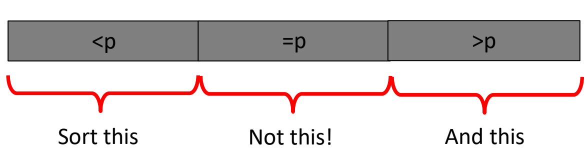
- Pros:
  - Each element only swapped once (except pivot)
  - Simple idea
  - Simple invariant (what is it?)

- Cons:
  - Very tricky to implement without bugs
    - Termination conditions
    - Edge cases
    - Off by one errors
  - Not stable
  - How about duplicates? Or what would be the performance if all elements in the array have the same value?
    - **It performs very badly when there are many elements that are equal to the pivot as it keeps on checking all possible pairs in each iteration.**

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:



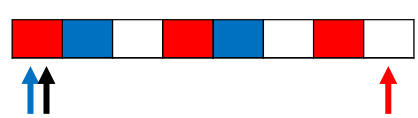
Dutch National Flag Partition Problem

- 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

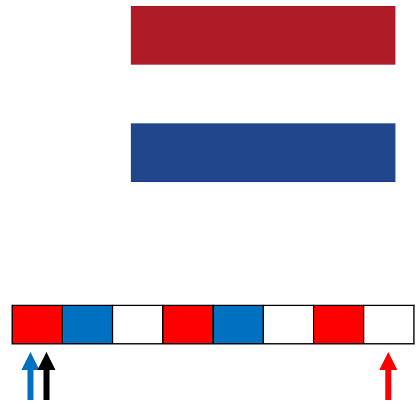
Dutch National Flag Partition Algorithm

boundary1=1,
j=1
boundary2 = n



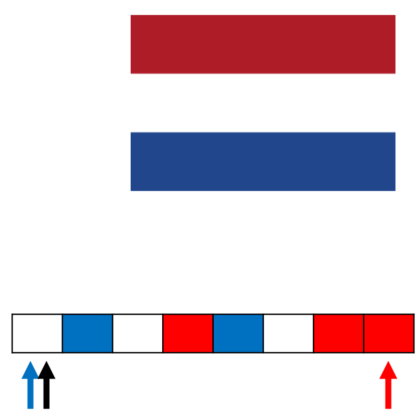
Dutch National Flag Algorithm

```
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
```



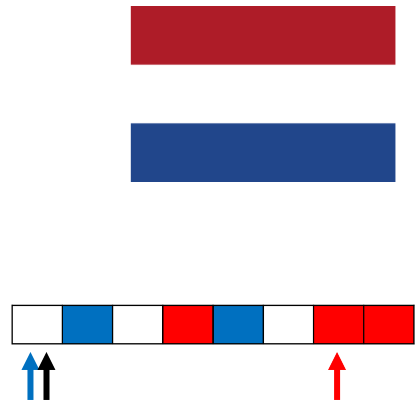
Dutch National Flag Algorithm

```
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
```



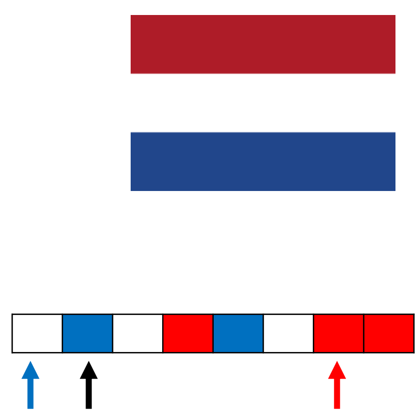
Dutch National Flag Algorithm

```
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
```



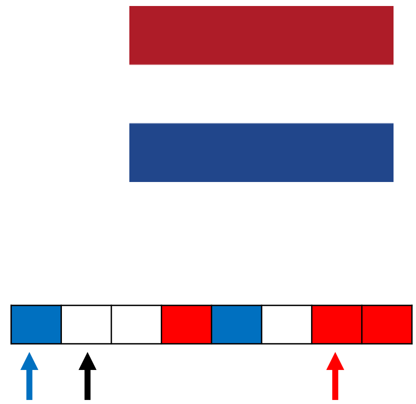
Dutch National Flag Algorithm

```
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
```



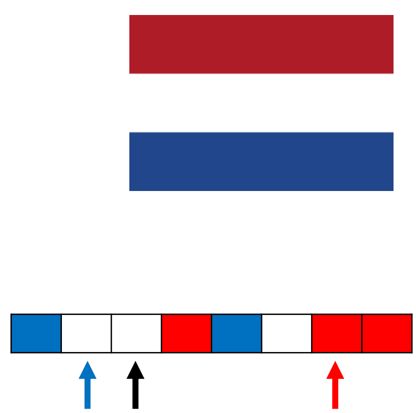
Dutch National Flag Algorithm

```
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
```



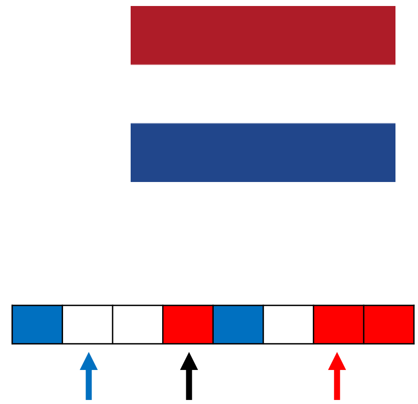
Dutch National Flag Algorithm

```
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
```



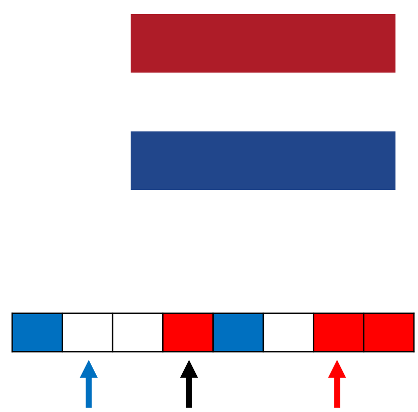
Dutch National Flag Algorithm

```
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
```



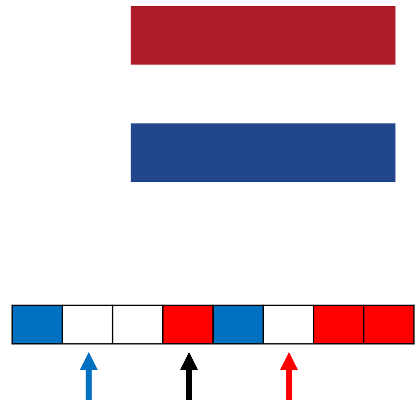
Dutch National Flag Algorithm

```
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
```



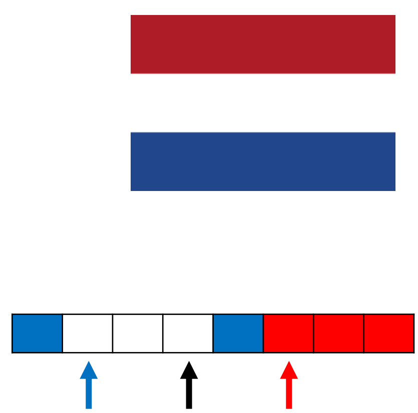
Dutch National Flag Algorithm

```
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
```



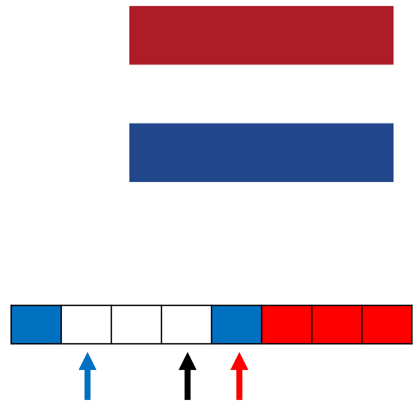
Dutch National Flag Algorithm

```
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
```



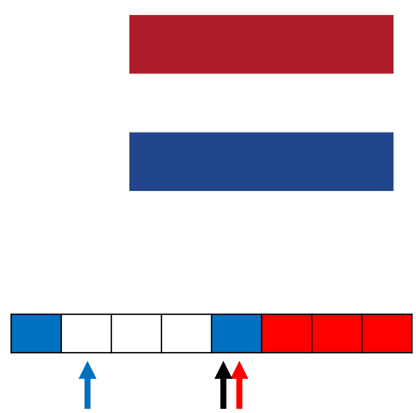
Dutch National Flag Algorithm

```
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
```



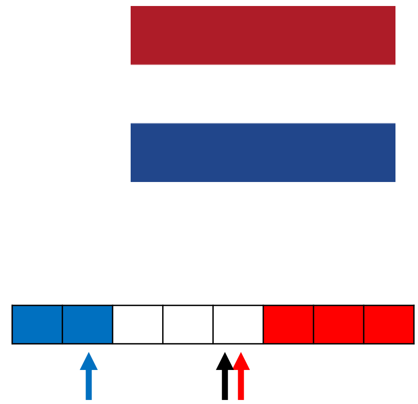
Dutch National Flag Algorithm

```
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
```



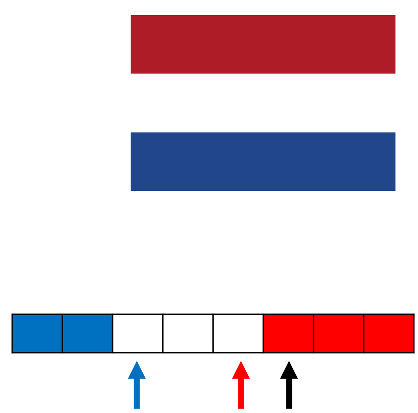
Dutch National Flag Algorithm

```
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
```



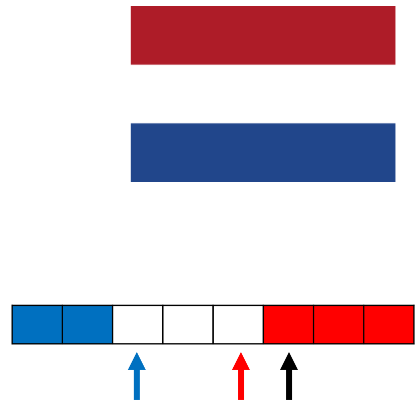
Dutch National Flag Algorithm

```
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
```



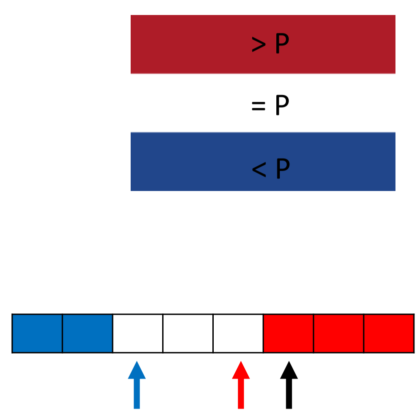
Dutch National Flag Algorithm

```
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 Algorithm

```
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

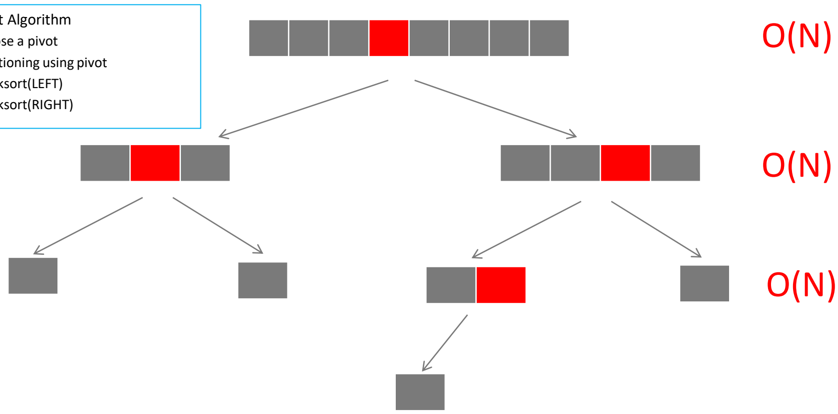
- Lots to consider
- State of the art is more complex
- Objectives
  - Minimise swaps
  - Minimise work in recursive calls
  - Be in place
- Both Hoare and DNF partition schemes are not stable
  - How to make these stable? We have seen some methods in the tutorial!

Quicksort and its Analysis

1. Algorithm and partitioning
2. Complexity analysis
3. Improving worst-case complexity
   A. Quick select
   B. Quicksort in O(N log N) worst-case

Best-case Time Complexity

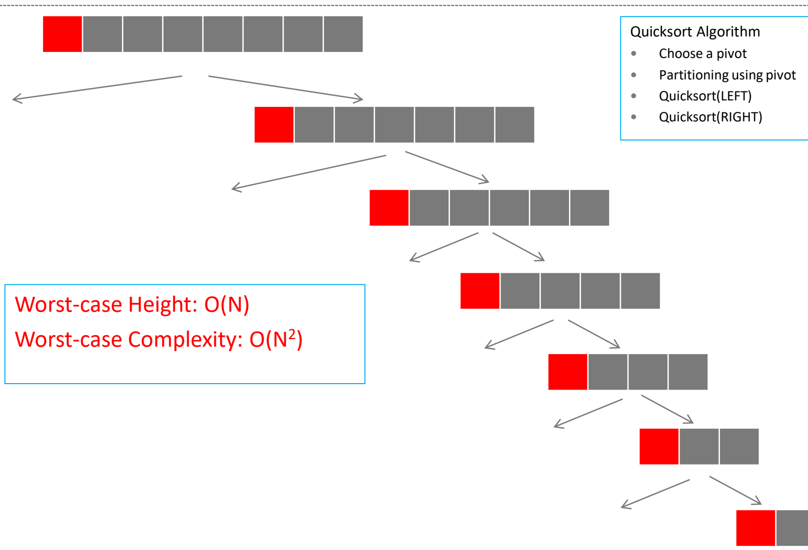
Quicksort Algorithm
- Choose a pivot
- Partitioning using pivot
- Quicksort(LEFT)
- Quicksort(RIGHT)



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)

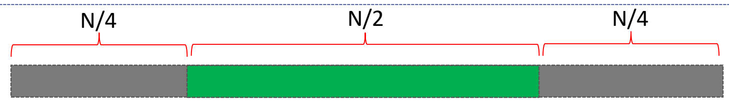
Worst-case Time Complexity



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

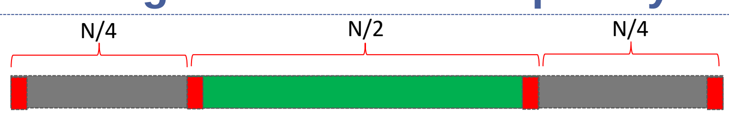
Worst-case Height: O(N)
Worst-case Complexity: $O(N^2)$

Average-case Time Complexity



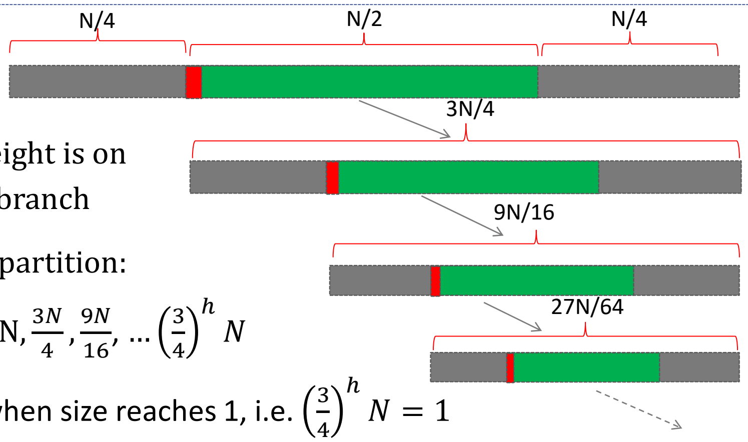
- After partitioning, pivot has 50% probability to be in the green sub-array and has 50% probability to be in one of the two grey sub-arrays.
  - i.e., on average, pivot will be in green half of the time and in grey half of the time

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 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

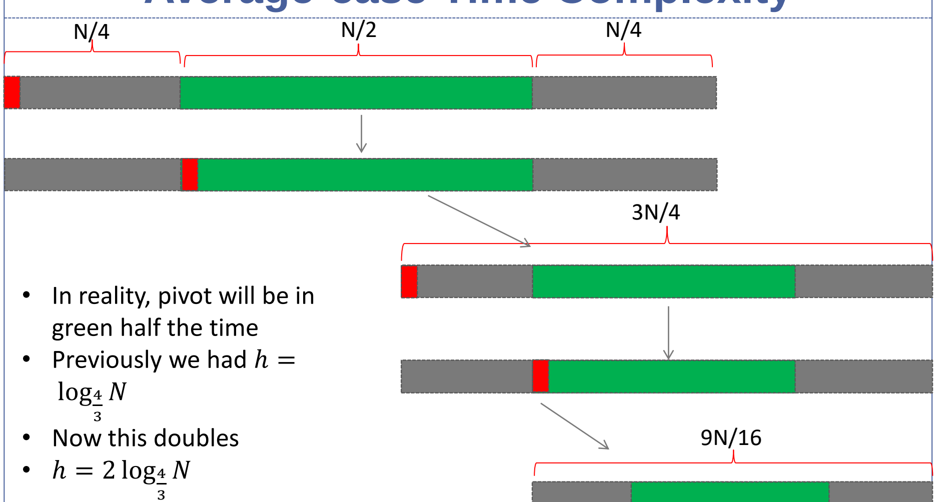
Size of partition:

$$\mathrm{N}, \frac{3N}{4}, \frac{9N}{16}, \ldots \left(\frac{3}{4}\right)^h N$$

Stops when size reaches 1, i.e. $\left(\frac{3}{4}\right)^h N = 1$

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

Average-case Time Complexity



- 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

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)

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

Worst-case Time Complexity using Recurrence

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

Recurrence relation:

$T(1) = b$

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

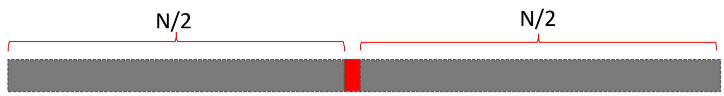
Solution:

$O(N^2)$

Quicksort and its Analysis

1. Algorithm
2. Complexity analysis
3. Improving worst-case complexity
   A. Quick select
   B. Quicksort 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

Quicksort and its Analysis

1. Algorithm and partitioning
2. Complexity analysis
3. Improving worst-case complexity
   A. Quick select
   B. Quicksort in O(N log N) worst-case

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 discussed 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 Select

- Quick select is **not** a sorting algorithm
- Quick select(L, k) returns the k^{th} smallest element in L (or the index of that element)

50	80	90	10	30	20	70	60

- In the above list
  - Quickselect(L, 1) = 10
  - Quickselect(L, 2) = 20
  - Quickselect(L, 5) = 60
- Quick select **does not** find a particular number

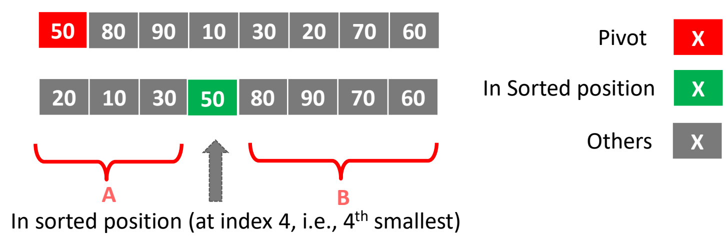
Quick Select Algorithm

```
QuickSelect(array[lo..hi], k)
    if hi > lo then
        pivot = array[lo]
        mid = Partition(array[lo..hi], pivot)
         if k < mid then
              return QuickSelect(array[lo..mid-1], k)   --- A
         else if k > mid then
              return QuickSelect(array[mid+1..hi], k)   --- B
         else
              return array[k]
    else
         return array[k]
```

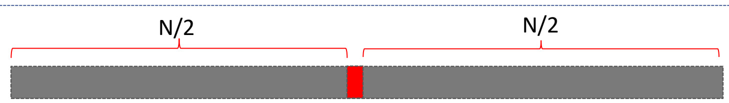
Best-case time complexity?
- O(N)

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

Average-case time complexity?
- O(N) – same arguments as for quicksort (check week 4 tutorial and lecture notes)



Quicksort Best-case when using Quick Select to choose pivot

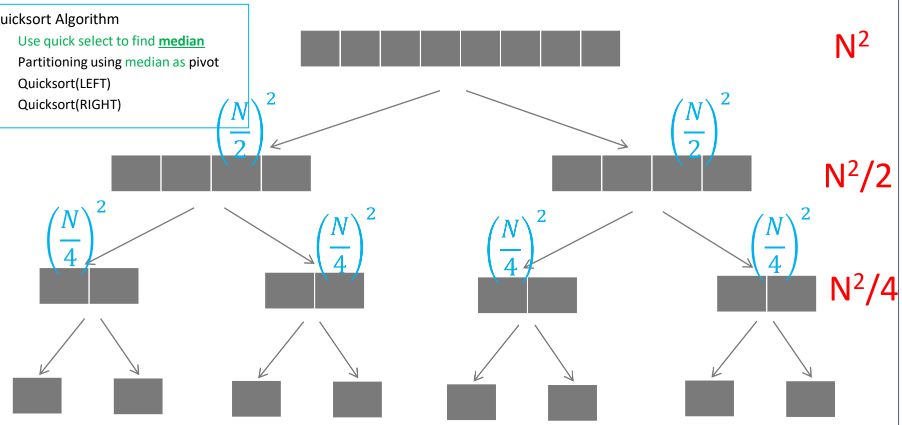


- Call **Quick Select** with k=len(array)/2?
- The value returned by Quick Select will be median.
- Choose this as the pivot.
- What will be the best-case cost of such quick sort?
  - O(N log N)
- What is the worst-case cost?

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

Quicksort 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)



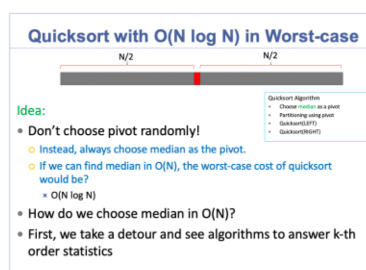
Worst-case cost at level k: $N^2/2^k$

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 O(N log N) in the worst case
- Need to find median in O(N)
- We have an algorithm (quick select) 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**)
- We want to make quick select always take O(N)
- What do we need? A median pivot for **quick select!**



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

N/2 N/2

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

FIT2004: Lecture 3 - Quick Sort and Select

Selecting pivot using median of medians results in linear time in the worst case

Where are we?

- What do we need? A median pivot for **quick select!**

- **But that is what quick select is meant to do…**

- Sounds impossible – in order for quick select to run in O(N) we need to find a good (i.e. median) pivot in O(N), but that was exactly the problem quick select 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 O(N log N), but slower in practice, so the closer to 50:50 the better (See Tutorial Week 4)

Quicksort and its Analysis

1. Algorithm and partitioning
2. Complexity analysis
3. Improving worst-case complexity
   A. Quick select
   B. Quicksort in O(N log N) worst-case

Algorithm's Heroes

- **Median of Medians:** Very clever algorithm designed by Manuel Blum, Robert Floyd, Vaughan Pratt, Ronald Rivest and Robert Tarjan (1971)
  - Manuel Blum - Turing Award 1995: complexity theory, cryptography, program checking
  - Robert Floyd - Turing Award 1978: theory of parsing, the semantics of programming languages, automatic program verification, automatic program synthesis, analysis of algorithms
  - Ronald Rivest - Turing Award 2002: cryptography, the R in RSA cryptosystem that used for online banking, e-commerce etc
  - Robert Tarjan - Turing Award 1986: for fundamental achievements in the design and analysis of algorithms and data structures
  - Vaughan Pratt is the only author of the algorithm without a Turing award (so far). He was one of the key people in Sun Microsystems

Manuel Blum

Blum explains the tuition behind his contributions (video)

Why Median of Medians?

- Now we know that the performance of Quicksort depends on a good pivot.
- If we choose a bad pivot (e.g. smallest or largest element), QuickSort ends up at $O(n^2)$ in the worst case.
- If we pick a good pivot (close to the median), QuickSort will balance partitions and runs in $O(n \log n)$ time.
- The Median of Medians method guarantees a pivot that is at least reasonably close to the median, preventing QuickSort ending up at $O(n^2)$ in the worst case.

Median of Medians

- 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

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

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

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

- 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

- 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

- 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

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

- 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 wanted 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

Median of Medians Algorithm

```
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
```

- We first divide the n elements into groups of 5.
- There are n/5 groups (approximately).
- Each group is sorted, and the median of each group is chosen.

Median of Medians Algorithm

```
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
```

- The medians of the groups form a smaller array of size n/5.
- We recursively find the median of these n/5 medians, which becomes the pivot

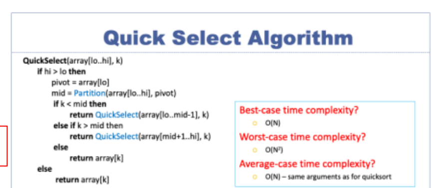
Median of Medians Algorithm

```
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)
```

- After choosing the median of medians as the pivot, at least 30% of elements are less than the pivot, at least 30% of elements are greater than the pivot and the pivot itself is in the middle 40% of elements.
- Thus, the partition is at worst 70:30 ensuring that no recursive call gets more than 7n/10 elements, preventing worst-case $O(n^2)$

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

```
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)
  else if k > mid
    return QuickSelect(array, mid+1, hi, k)
  else
    return array[k]
```



Quick Select Algorithm

```
QuickSelect(array[lo..hi], k)
    if hi > lo then
        pivot = array[lo]
        mid = Partition(array[lo..hi], pivot)
        if k < mid then
            return QuickSelect(array[lo..mid-1], k)
        else if k > mid then
            return QuickSelect(array[mid+1..hi], k)
        else
            return array[k]
    else
        return array[k]
```

Best-case time complexity?
- O(N)

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

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

This call uses QuickSelect!
But with a weaker pivot

Recall: the worst split we can get using the MoM is 70:30!

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

```
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 split in worst)
  if mid > k
    return QuickSelect(array, lo, mid-1, k)    (7n/10 in worst)
  else if k > mid
    return QuickSelect(array, mid+1, hi, k)    (7n/10 in worst)
  else
    return array[k]
```


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

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 (inside the quick select), 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 gives linear time!

So armed with a linear time quickselect, we can now quicksort in O(N log N) worst case…

Anticlimax

- 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.
- Also, quick **select** is an extremely useful algorithm in general

Reading

- Course Notes: Section 3.2, Chapter 4
- You can also check algorithms textbooks for contents related to this lecture, e.g.:
  - CLRS: Chapters 7 and 9
  - KT: Section 13.5
  - Rou: Chapters 5 and 6

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.
- In practice, it is better to do a simple pivot selection which takes less time (like random selection)

Coming Up Next

- Introduction to Graphs

Things to do before next lecture

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