MONASH University

MONASH
INFORMATION
TECHNOLOGY

FIT2004
Algorithms and Data Structures

Ian Wern Han Lim
lim.wern.han@monash.edu

Referencing materials by
Nathan Companez, Aamir Cheema, Arun Konagurthu and Lloyd Allison

GROUP OF EIGHT AUSTRALIA

Faculty of Information Technology, Monash University

Ready?

Agenda

- Quick Sort

Agenda

- Quick Sort
  - Analysis of time
  - Analysis of space

Agenda

- Quick Sort
  - Analysis of time
  - Analysis of space
  - But in detail!

Agenda

- Quick Sort
  - Analysis of time
  - Analysis of space
  - But in detail!
    - Partitioning strategy etc…

Let us begin…

Quicksort
Brief description

- How would you describe quick sort?

Quicksort
Brief description

- How would you describe quick sort?
  – Divide and conquer

Quicksort
Brief description

- How would you describe quick sort?
  - Divide and conquer

pivot

Quicksort
Brief description

- How would you describe quick sort?
  - Divide and conquer

Quicksort
Brief description

- How would you describe quick sort?
  - Divide and conquer

- Partition-based on the pivot
  - Smaller to the left of pivot
  - Bigger to the right of pivot

Quicksort
Brief description

- How would you describe quick sort?
  - Divide and conquer

- Partition-based on the pivot
  - Smaller or equal to the left of pivot
  - Bigger to the right of pivot

Quicksort
Brief description

- How would you describe quick sort?
  - Divide and conquer
- Partition-based on the pivot
  - Smaller or equal to the left of pivot
  - Bigger to the right of pivot
- Divide based on the partition

Quicksort
Brief description

- How would you describe quick sort?
  - Divide and conquer
- Partition-based on the pivot
  - Smaller or equal to the left of pivot
  - Bigger to the right of pivot
- Divide based on the partition

Repeat

- How would you describe quick sort?
  - Divide and conquer
- Partition-based on the pivot
  - Smaller or equal to the left of pivot
  - Bigger to the right of pivot
- Divide based on the partition

Repeat
Till partition size 1...

Quicksort
Brief description

- How would you describe quick sort?
  - Divide and conquer

- Partition-based on the pivot
  - Smaller or equal to the left of pivot
  - Bigger to the right of pivot

- Divide based on the partition

Repeat
Till partition size 1…

Quicksort
Brief description

- How would you describe quick sort?
  - Divide and conquer
- Partition-based on the pivot
  - Smaller or equal to the left of pivot
  - Bigger to the right of pivot
- Divide based on the partition

Repeat
Till partition size 1…

Questions?

- Given a list

2	8	7	1	3	4	5	6

Quicksort
Example

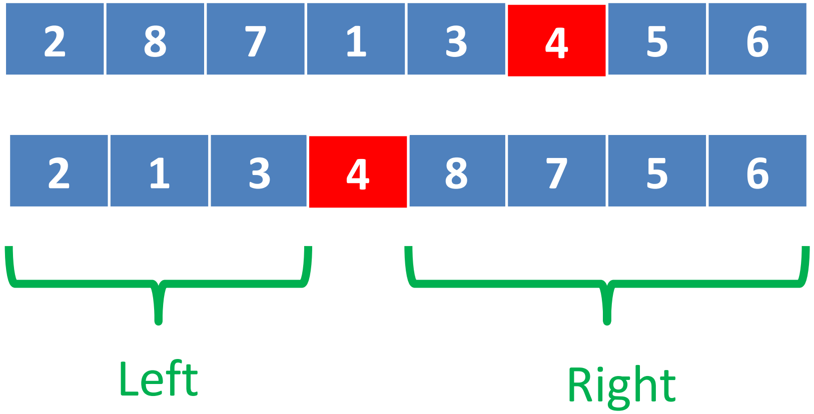
- Given a list
- Choose a pivot (doesn't matter which)

2	8	7	1	3	4	5	6

Quicksort

Example

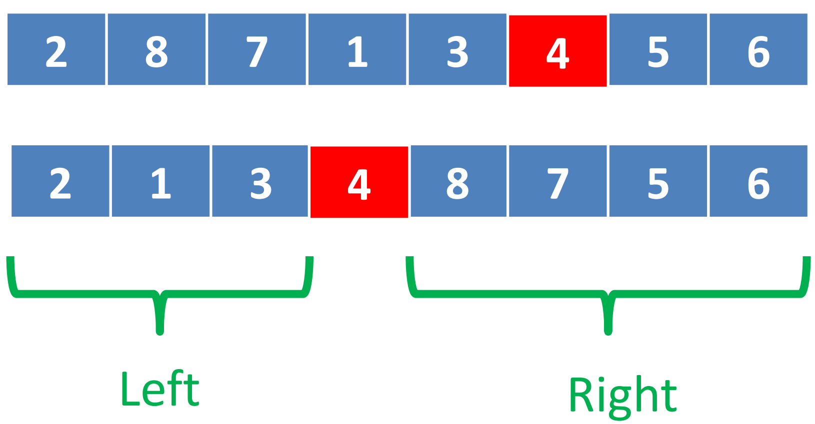
- Given a list
- Choose a pivot (doesn't matter which)
- Ensure invariant
  - Left <= pivot
  - Right > pivot



Quicksort
Example

- Given a list
- Choose a pivot (doesn't matter which)
- Ensure invariant
  – Left <= pivot
  – Right > pivot

Partitioning

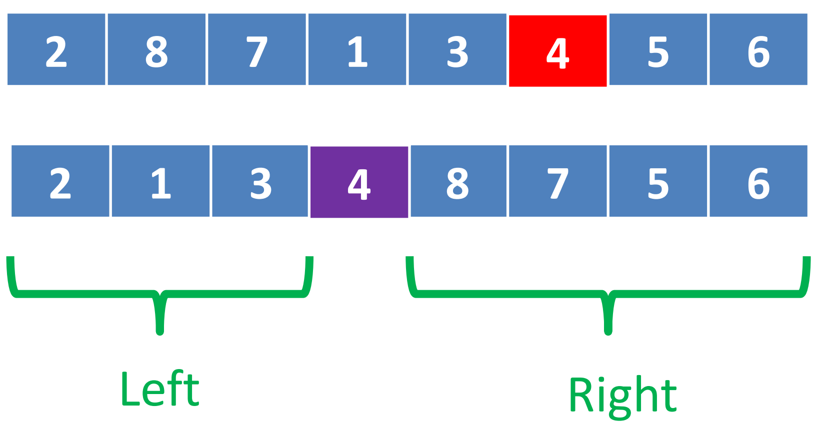


MONASH University

- Given a list
- Choose a pivot (doesn't matter which)
- Ensure invariant
  - Left <= pivot
  - Right > pivot

Partitioning

- Pivot is now in sorted position (or in order)

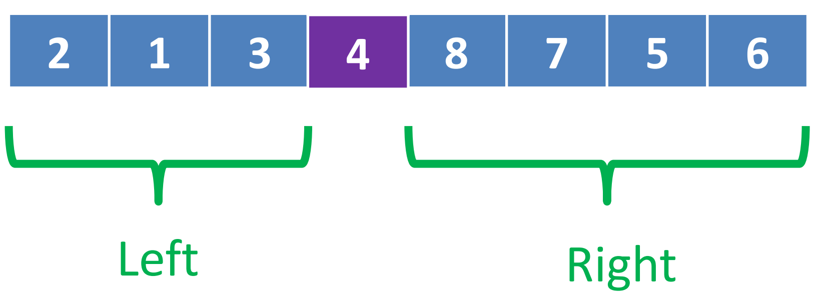


Quicksort
Example

- Given a list
- Choose a pivot (doesn't matter which)
- Ensure invariant
  - Left <= pivot
  - Right > pivot

Partitioning

- Pivot is now in sorted position (or in order)
- Then we repeat for left and right



Quicksort
Example

- Given a list
- Choose a pivot (doesn't matter which)
- Ensure invariant
  - Left <= pivot
  - Right > pivot

Partitioning

- Pivot is now in sorted position (or in order)
- Then we repeat for left and right

2	1	3	4	8	7	5	6

MONASH University

- Given a list
- Choose a pivot (doesn't matter which)
- Ensure invariant
  - Left <= pivot
  - Right > pivot

Partitioning

- Pivot is now in sorted position (or in order)
- Then we repeat for left and right

1	2	3	4	8	7	5	6

MONASH University

- Given a list
- Choose a pivot (doesn't matter which)
- Ensure invariant
  – Left <= pivot
  – Right > pivot

Partitioning

- Pivot is now in sorted position (or in order)
- Then we repeat for left and right

1	2	3	4	8	7	5	6

Quicksort

Example

- Given a list
- Choose a pivot (doesn't matter which)
- Ensure invariant
  - Left <= pivot
  - Right > pivot

Partitioning

- Pivot is now in sorted position (or in order)
- Then we repeat for left and right

1	2	3	4	8	7	5	6

Quicksort

Example

MONASH University

- Given a list
- Choose a pivot (doesn't matter which)
- Ensure invariant
  - Left <= pivot
  - Right > pivot

Partitioning

- Pivot is now in sorted position (or in order)
- Then we repeat for left and right

1	2	3	4	8	7	5	6

Quicksort

Example

- Given a list
- Choose a pivot (doesn't matter which)
- Ensure invariant
  - Left <= pivot
  - Right > pivot

Partitioning

- Pivot is now in sorted position (or in order)
- Then we repeat for left and right

1	2	3	4	8	7	5	6

Quicksort
Example

- Given a list
- Choose a pivot (doesn't matter which)
- Ensure invariant
  - Left <= pivot
  - Right > pivot

Partitioning

- Pivot is now in sorted position (or in order)
- Then we repeat for left and right

1	2	3	4	8	7	5	6

- Given a list
- Choose a pivot (doesn't matter which)
- Ensure invariant
  - Left <= pivot
  - Right > pivot

  Partitioning
- Pivot is now in sorted position (or in order)
- Then we repeat for left and right
- Till sorted

1	2	3	4	5	6	7	8

Questions?

- What is partitioning?

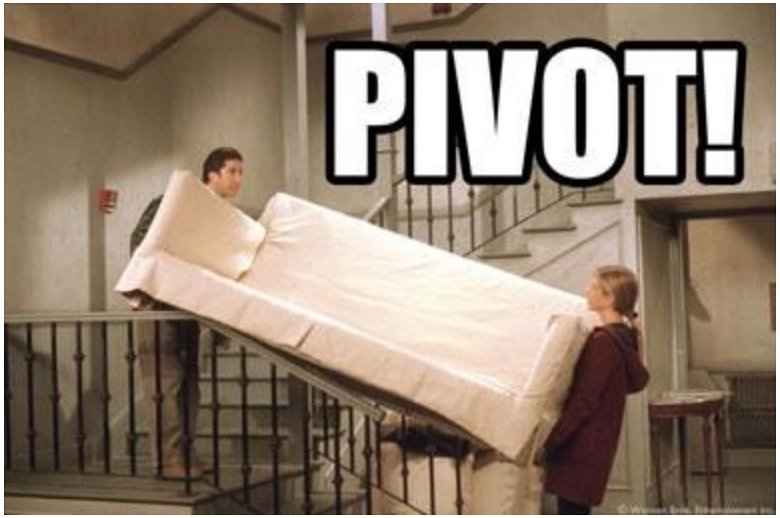
- What is partitioning?
  – Separate the list into parts

Partitioning

- What is partitioning?
  - Separate the list into parts
  - Here, mainly the left and the right

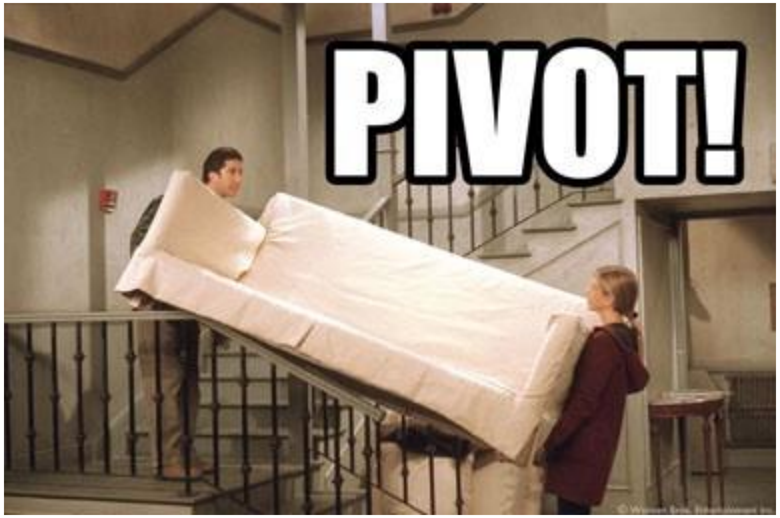
Quicksort
Partitioning

- What is partitioning?
  - Separate the list into parts
  - Here, mainly the left and the right
- Partition is based-on pivot



- What is partitioning?
  - Separate the list into parts
  - Here, mainly the left and the right

- Partition is based-on pivot
  - Out-of-place
  - Hoare’s
  - Lomuto’s



Questions?

Quicksort
Out-of-place Partitioning

- Not in-place

Quicksort
Out-of-place Partitioning

- Not in-place

not O(1) auxiliary space

2	8	7	1	3	4	5	6

LEFT

RIGHT

Quicksort
Out-of-place Partitioning

- Not in-place

2	8	7	1	3	4	5	6

LEFT

O(2n)
O(n)

RIGHT

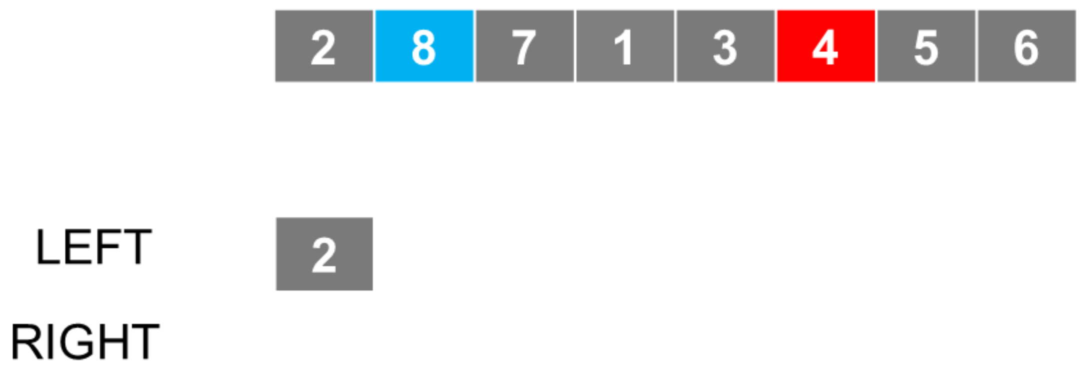
Quicksort
Out-of-place Partitioning

- Not in-place

2	8	7	1	3	4	5	6

LEFT	2
RIGHT	

- Not in-place



Quicksort
Out-of-place Partitioning

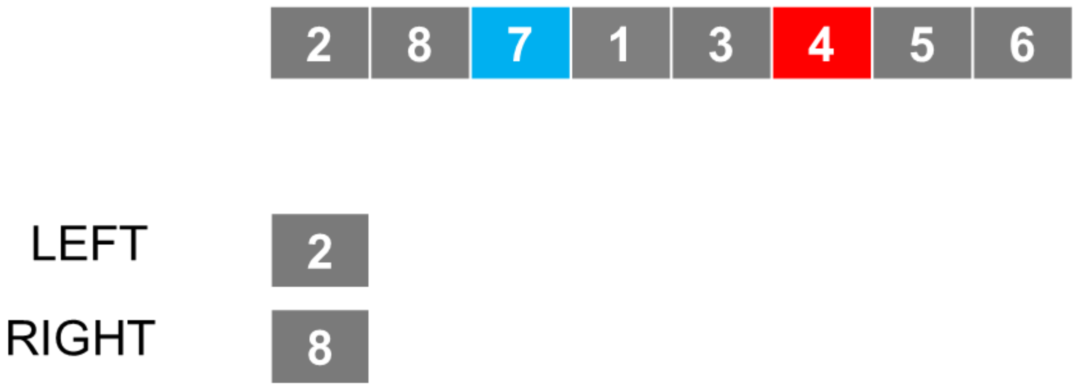
- Not in-place

2	8	7	1	3	4	5	6

LEFT	2
RIGHT	8

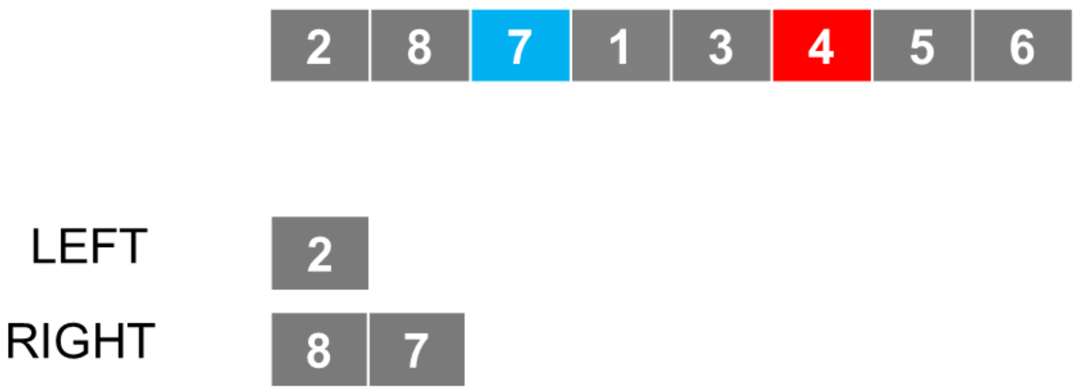
Quicksort
Out-of-place Partitioning

- Not in-place

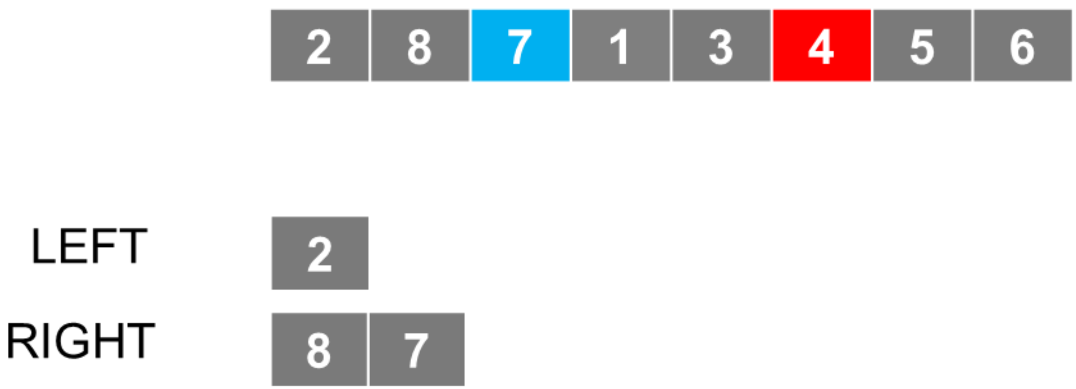


Quicksort
Out-of-place Partitioning

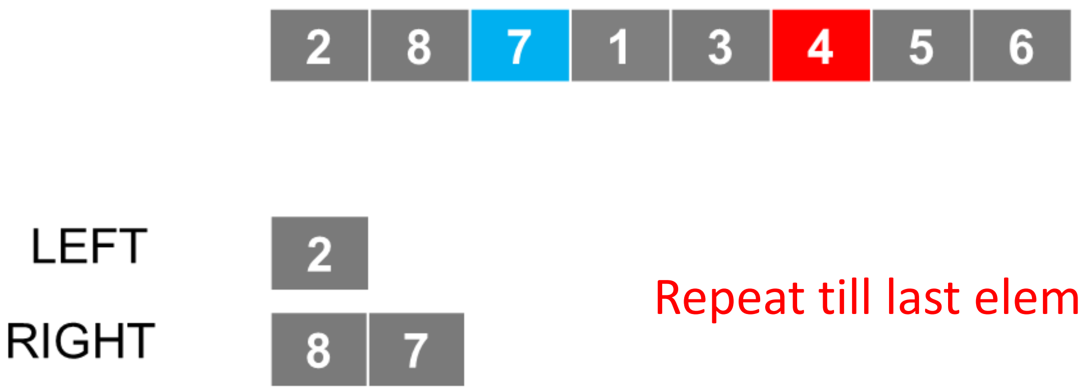
- Not in-place



- Not in-place



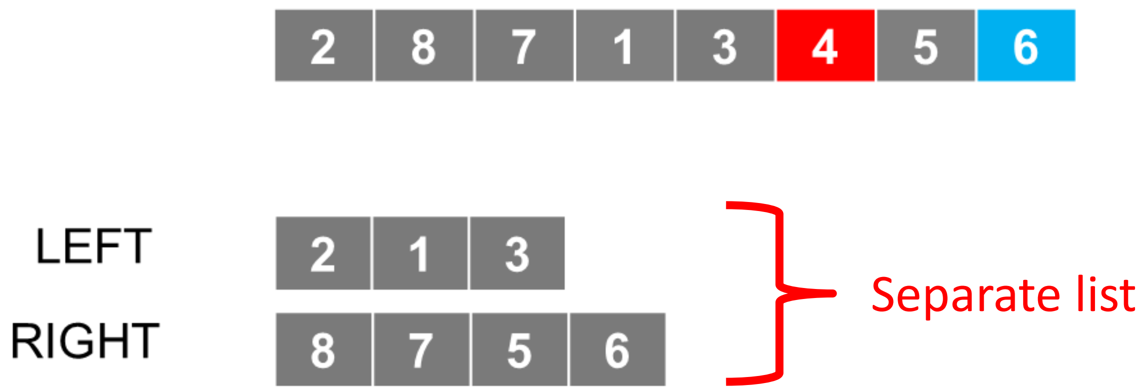
- Not in-place



Repeat till last element

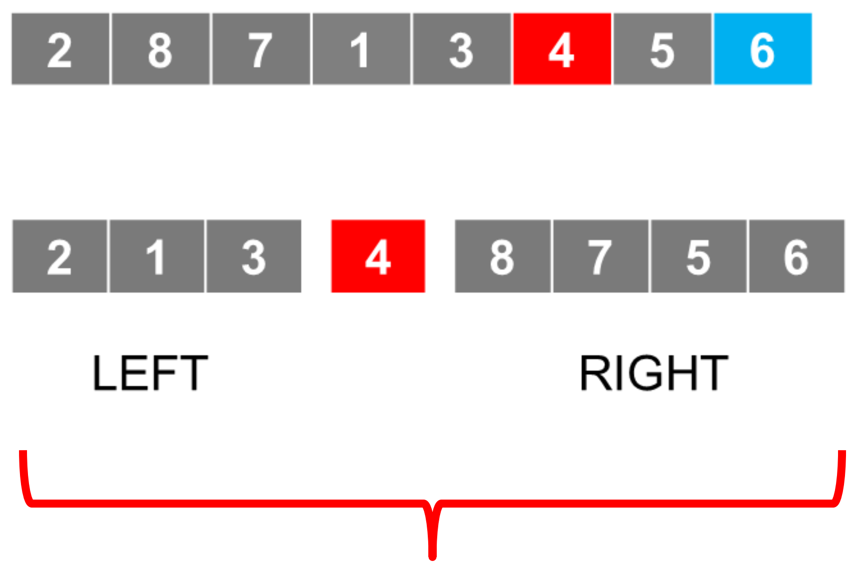
Quicksort
Out-of-place Partitioning

- Not in-place



Quicksort
Out-of-place Partitioning

- Not in-place



Separate list/ onto original list

Quicksort
Out-of-place Partitioning

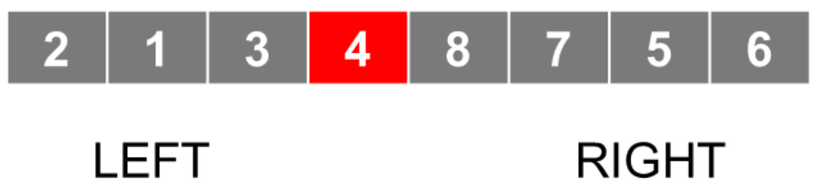
- Not in-place

2	1	3	4	8	7	5	6

LEFT RIGHT

Quicksort
Out-of-place Partitioning

- Not in-place
  - Need left temporary list
  - Need right temporary list



Quicksort
Out-of-place Partitioning

- Not in-place
  - Need left temporary list
  - Need right temporary list
  - Combined back to the original list

2	1	3	4	8	7	5	6

LEFT RIGHT

Quicksort
Out-of-place Partitioning

- Not in-place
  - Need left temporary list
  - Need right temporary list
  - Combined back to the original list

2	1	3	4	8	7	5	6

LEFT RIGHT

  - Is the algorithm stable?

Quicksort
Out-of-place Partitioning

- Not in-place
  - Need left temporary list
  - Need right temporary list
  - Combined back to the original list

2	1	3	4	8	7	5	6

LEFT RIGHT

  - Is the algorithm stable?
    - <= pivot to the left
    - > pivot to the right

Quicksort
Out-of-place Partitioning

- Not in-place
  - Need left temporary list
  - Need right temporary list
  - Combined back to the original list

2	1	3	4	8	7	5	6

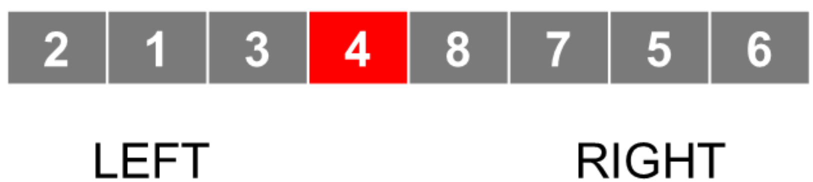
LEFT RIGHT

  - Is the algorithm stable?
    - <= pivot to the left: everything == pivot to the left of the pivot!
    - > pivot to the right

Quicksort
Out-of-place Partitioning

MONASH University

- Not in-place
  - Need left temporary list
  - Need right temporary list
  - Combined back to the original list



  - Is the algorithm stable? NO
    - <= pivot to the left: everything == pivot to the left of the pivot!
    - > pivot to the right

Quicksort
Out-of-place Partitioning

- Not in-place
  - Need left temporary list
  - Need right temporary list
  - Combined back to the original list

2	1	3	4	8	7	5	6

LEFT RIGHT

  - Is the algorithm stable? NO
    - <= pivot to the left: everything == pivot to the left of the pivot!
    - > pivot to the right
    - But we can make it stable by having 2 separate list for == pivot
      - Anything can be stable with more memory!

Quicksort
Out-of-place Partitioning

- Not in-place
  - Need left temporary list
  - Need right temporary list
  - Combined back to the original list

O(N) additional space beside recursive stack

2	1	3	4	8	7	5	6

LEFT RIGHT

  - Is the algorithm stable? NO with enough memory can make all sorting stable remain stable by maintain relative order of the same values in the list
    - <= pivot to the left: everything == pivot to the left of the pivot!
    - > pivot to the right
    - But we can make it stable by having 2 separate list for == pivot
      - Anything can be stable with more memory!

Questions?

In-place Partitioning (Hoare's)

- We want to make it in-place
- We want to make it fast
- We want to make it stable

Quicksort
In-place Partitioning (Hoare's)

- We want to make it in-place
  - Save memory
- We want to make it fast
- We want to make it stable

Quicksort
In-place Partitioning (Hoare's)

- We want to make it in-place
  - Save memory
- We want to make it fast
  - Avoid copying many items
- We want to make it stable

- We want to make it in-place
  - Save memory
- We want to make it fast
  - Avoid copying many items
  - Avoid swapping many times
- We want to make it stable

Quicksort
In-place Partitioning (Hoare's)

- We want to make it in-place
  - Save memory
- We want to make it fast
  - Avoid copying many items
  - Avoid swapping many times
- We want to make it stable
  - Can we?

Quicksort
In-place Partitioning (Hoare's)

- We want to make it in-place
  - Save memory
- We want to make it fast
  - Avoid copying many items
  - Avoid swapping many times
    - Main focus here: Swap every item only once except pivot
- We want to make it stable
  - Can we?

In-place Partitioning (Hoare's)

- We want to make it in-place
  - Save memory
- We want to make it fast
  - Avoid copying many items
  - Avoid swapping many times
    - Main focus here: Swap every item only once except pivot
- We want to make it stable
  - Can we?

- I will use Nathan's slide here for consistency
  - But I will add in notes on top

Partitioning: In place (Hoare's)

Swap pivot to the front (position 1)

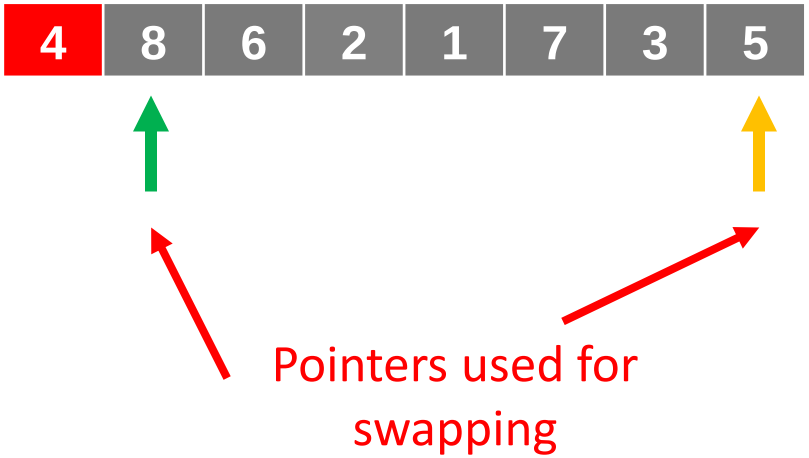
Index starts from 1

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



Partitioning: In place (Hoare's)

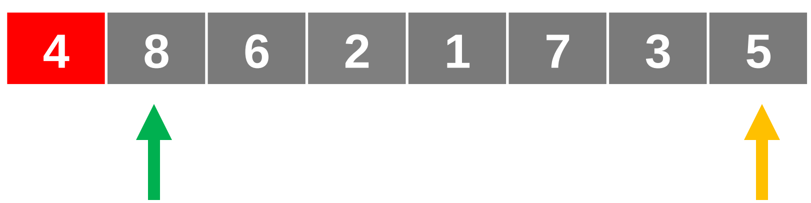
Swap pivot to the front (position 1)

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)

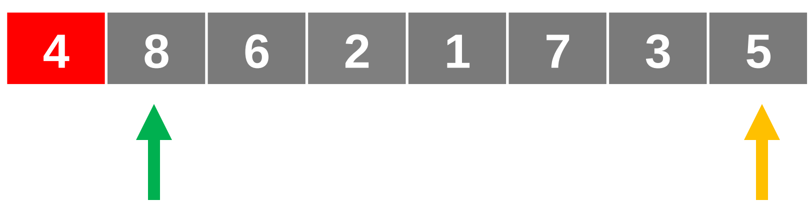
Terminating condition

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



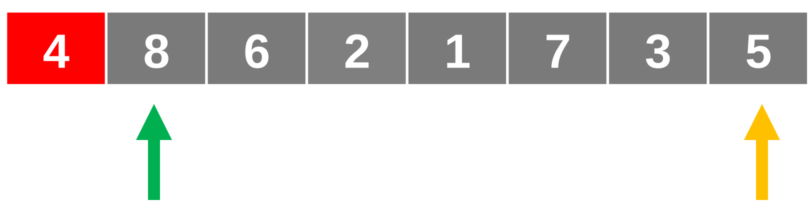
Partitioning: In place (Hoare's)

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



Recall
left <= pivot
right > pivot

Partitioning: In place (Hoare's)

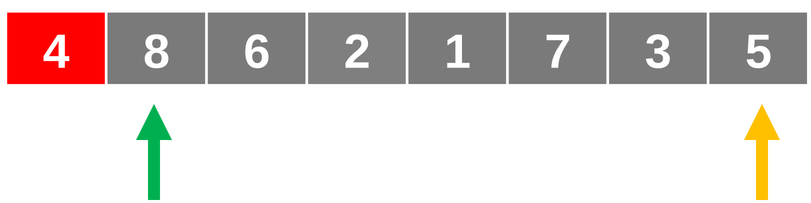
Swap pivot to the front (position 1)

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 once
only per element



Partitioning: In place (Hoare's)

Swap pivot to the front (position 1)

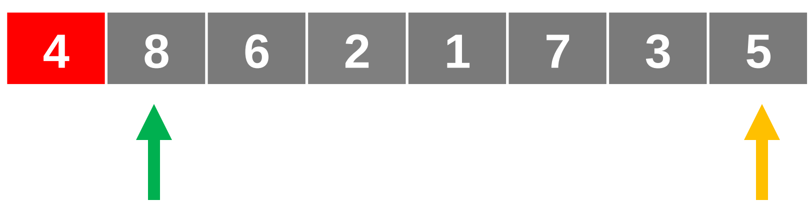
L_bad = 2, R_bad = N

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



Let's start!

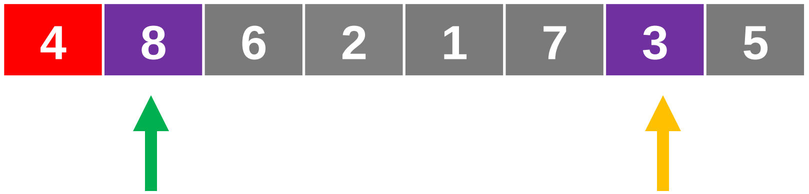
Partitioning: In place (Hoare's)

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)

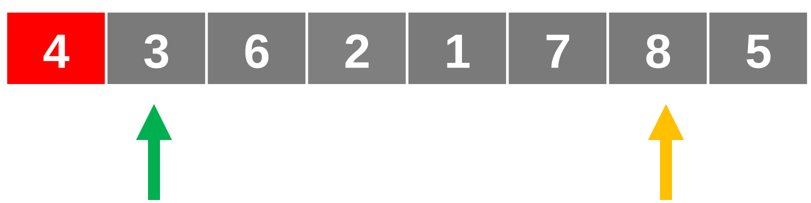
L_bad = 2, R_bad = N

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



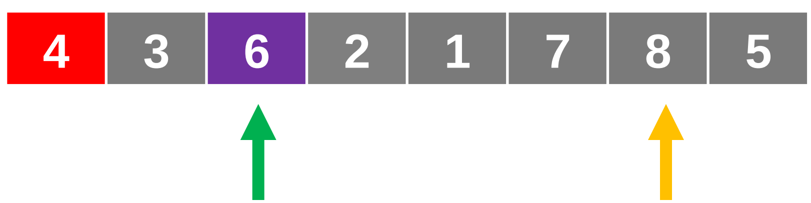
Partitioning: In place (Hoare's)

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



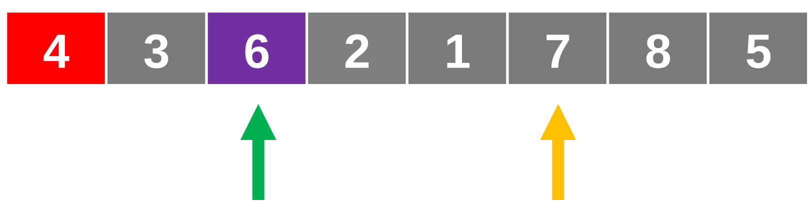
Partitioning: In place (Hoare's)

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)

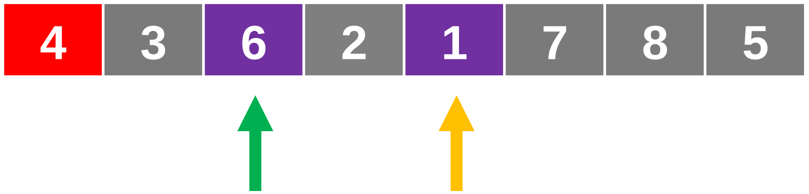
L_bad = 2, R_bad = N

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



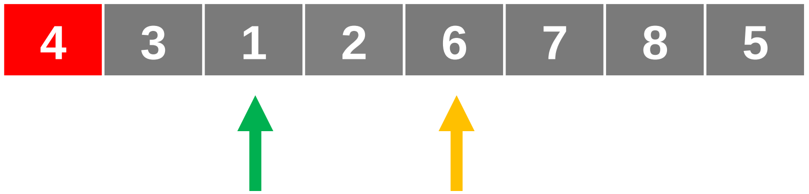
Partitioning: In place (Hoare's)

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



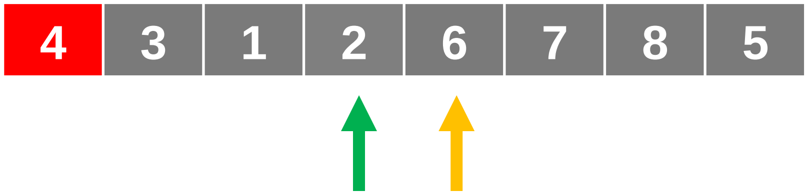
Partitioning: In place (Hoare's)

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



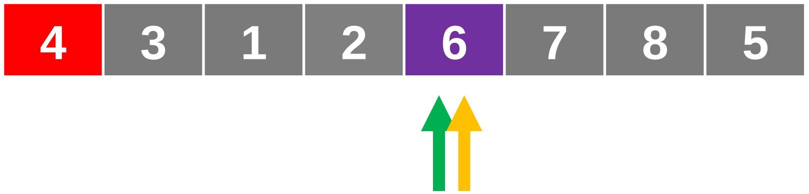
Partitioning: In place (Hoare's)

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



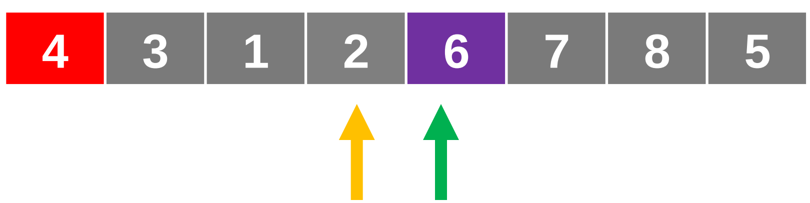
Partitioning: In place (Hoare's)

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)

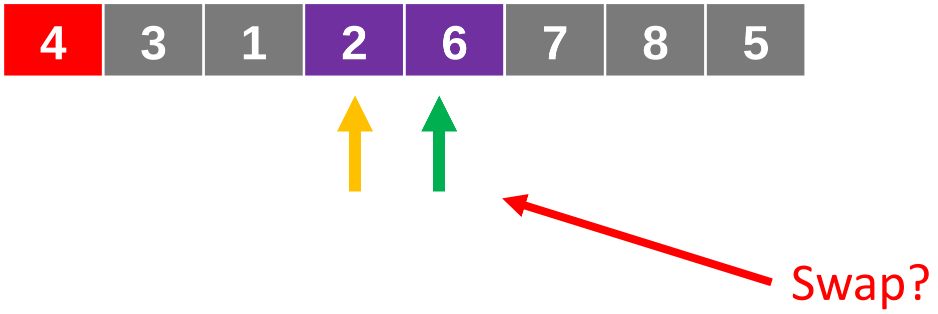
Swap pivot to the front (position 1)

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)

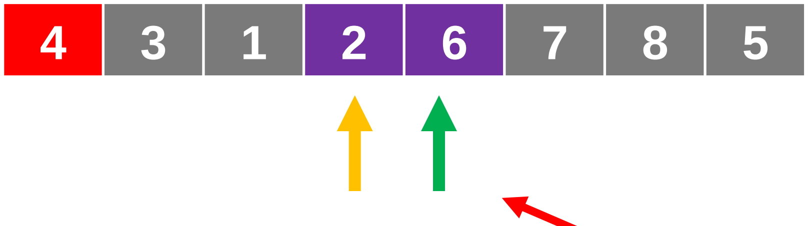
Swap pivot to the front (position 1)

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



Swap?

NO! It is a bug in the algo

Partitioning: In place (Hoare's)

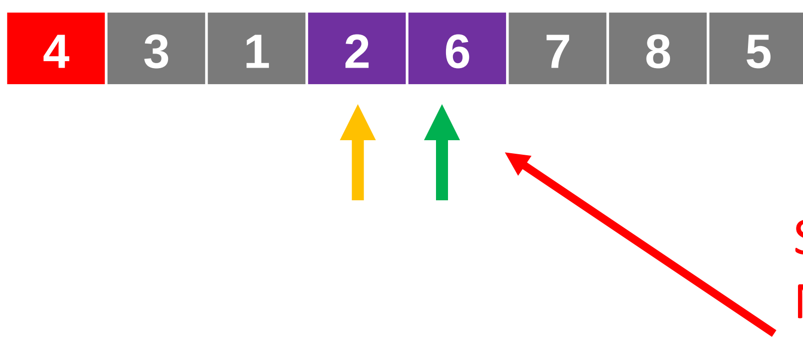
Swap pivot to the front (position 1)

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



Swap?
NO! It is a bug in the algo
So you know
what to change?

Partitioning: In place (Hoare's)

Swap pivot to the front (position 1)

L_bad = 2, R_bad = N

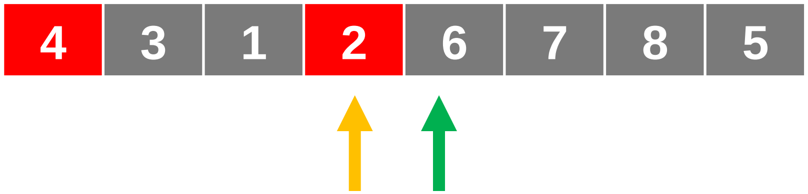
Repeat until L_bad and R_bad cross

    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

2	3	1	4	6	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

swap pivot to R_bad

4 is now in sorted position!

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

2	3	1	4	6	7	8	5

Repeat LEFT and RIGHT

Questions?

Quicksort
In-place Partitioning (Hoare's)

- Invariant?

- Invariant?
  - L_bad?
  - R_bad?

Quicksort
In-place Partitioning (Hoare's)

- Invariant?
  - L_bad? Everything to left to L_bad is less/ same than pivot
  - R_bad? Everything to right of R_bad is great than pivot

- Invariant?
  - L_bad? Everything to left to L_bad is less/ same than pivot
  - R_bad? Everything to right of R_bad is great than pivot
  - Between? Not processed yet…

- Invariant?
  - L_bad? Everything to left to L_bad is less/ same than pivot
  - R_bad? Everything to right of R_bad is great than pivot
  - Between? Not processed yet…

  - What about the pivot? Think about it…

Questions?

Quicksort
In-place Partitioning (Hoare's)

- We want to make it in-place
  - Save memory
- We want to make it fast
  - Avoid copying many items
  - Avoid swapping many times
    - Main focus here: Swap every item only once except pivot
- We want to make it stable
  - Can we?

MONASH University

- YES: We want to make it in-place
  - Save memory
- We want to make it fast
  - Avoid copying many items
  - Avoid swapping many times
    - Main focus here: Swap every item only once except pivot
- We want to make it stable
  - Can we?

Quicksort
In-place Partitioning (Hoare's)

- YES: We want to make it in-place
  - Save memory
- YES: We want to make it fast
  - Avoid copying many items
  - Avoid swapping many times
    - Main focus here: Swap every item only once except pivot
- We want to make it stable
  - Can we?

Quicksort

In-place Partitioning (Hoare's)

- YES: We want to make it in-place
  - Save memory
- YES: We want to make it fast
  - Avoid copying many items
  - Avoid swapping many times
    - Main focus here: Swap every item only once except pivot
- YES: We want to make it stable
  - Can we?
  - We can add a condition for L_bad and R_bad to help it be stable by using the original index of the pivot with some math…

- YES: We want to make it in-place
  - Save memory
- YES: We want to make it fast
  - Avoid copying many items
  - Avoid swapping many times
    - Main focus here: Swap every item only once except pivot
- ~~YES~~ NO: We want to make it stable
  - Can we?
  - We can add a condition for L_bad and R_bad to help it be stable by using the original index of the pivot with some math…
  - The final swapping of the pivot from 1st position to R_bad would mess up the stability

Quicksort
In-place Partitioning (Hoare's)

- YES: We want to make it in-place
  - Save memory
- YES: We want to make it fast
  - Avoid copying many items
  - Avoid swapping many times
    - Main focus here: Swap every item only once except pivot
- ~~YES~~ NO: We want to make it stable
  - Can we?
  - We can add a condition for L_bad and R_bad to help it be stable by using the original index of the pivot with some math…
  - The final swapping of the pivot from 1^{st} position to R_bad would mess up the stability
  - But we know from Tutorial 03 we can make anything stable with memory…

- YES: We want to make it in-place
  - Save memory
- YES: We want to make it fast
  - Avoid copying many items
  - Avoid swapping many times
    - Main focus here: Swap every item only once except pivot
- ~~YES~~ NO: We want to make it stable
  - Can we?
  - We can add a condition for L_bad and R_bad to help it be stable by using the original index of the pivot with some math…
  - The final swapping of the pivot from 1st position to R_bad would mess up the stability
  - But we know from Tutorial 03 we can make anything stable with memory… but won't be in place…

Questions?

- Code it out and see…

- Code it out and see…
  - There is another special edge case that cause this algorithm to fail…

- Code it out and see…
  - There is another special edge case that cause this algorithm to fail…
  - Unless you add in a special check =)

Quicksort
In-place Partitioning (Hoare's)

- Code it out and see…
  - There is another special edge case that cause this algorithm to fail…
  - Unless you add in a special check =)

  - I might answer this on Slack/ MS Teams later since no interaction for online =(

choose the pivot as either biggest or smallest

Questions?

Quicksort
In-place Partitioning (Lomuto's)

- This is the one you are familiar with
  - From FIT1008

Quicksort
In-place Partitioning (Lomuto’s)

- This is the one you are familiar with
  - From FIT1008
- In place

Quicksort

In-place Partitioning (Lomuto's)

- This is the one you are familiar with
  - From FIT1008

- In place
- Swap each element multiple times

Quicksort
In-place Partitioning (Lomuto’s)

- This is the one you are familiar with
  - From FIT1008

- In place
- Swap each element multiple times
- … and still unstable

In-place Partitioning (Lomuto's)

- This is the one you are familiar with
  - From FIT1008

- In place
- Swap each element multiple times
  - Worse than Hoare's
- … and still unstable

Quicksort
In-place Partitioning (Lomuto's)

- This is the one you are familiar with
  - From FIT1008
- In place
- Swap each element multiple times
  - Worse than Hoare's
- … and still unstable
- Easier to understand
- Easier to implement
  - Recall some of the bugs and fail cases
    I mentioned in Hoare's algorithm shown…

Questions?

- So you now learnt all 3
  - Out-of-place
  - Hoare’s
  - Lomuto’s

MONASH University

- So you now learnt all 3
  - Out-of-place
  - Hoare's
  - Lomuto's
- And the entire partitioning gave us an idea to improve it more…
  - Due to the stability concern…

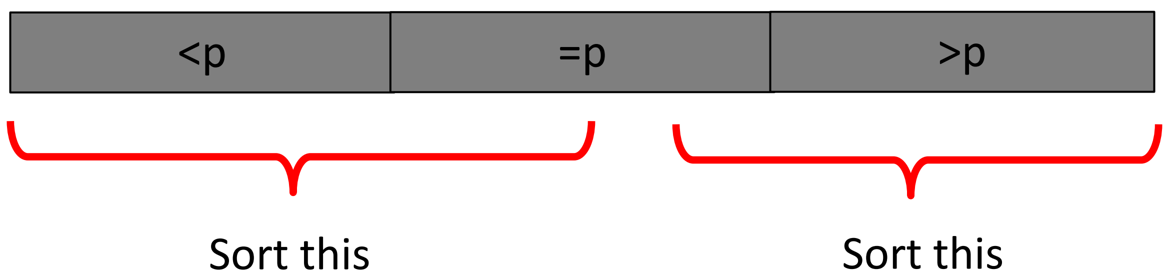
- So you now learnt all 3
  - Out-of-place
  - Hoare's
  - Lomuto's

- And the entire partitioning gave us an idea to improve it more…
  - Due to the stability concern…

- The stability issue however gives us an idea…

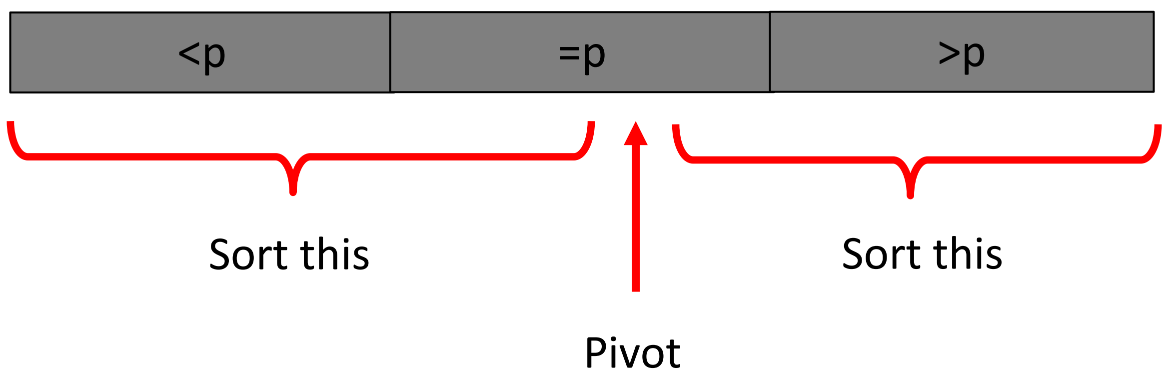
Quicksort
Partitioning…

- The stability issue however gives us an idea…

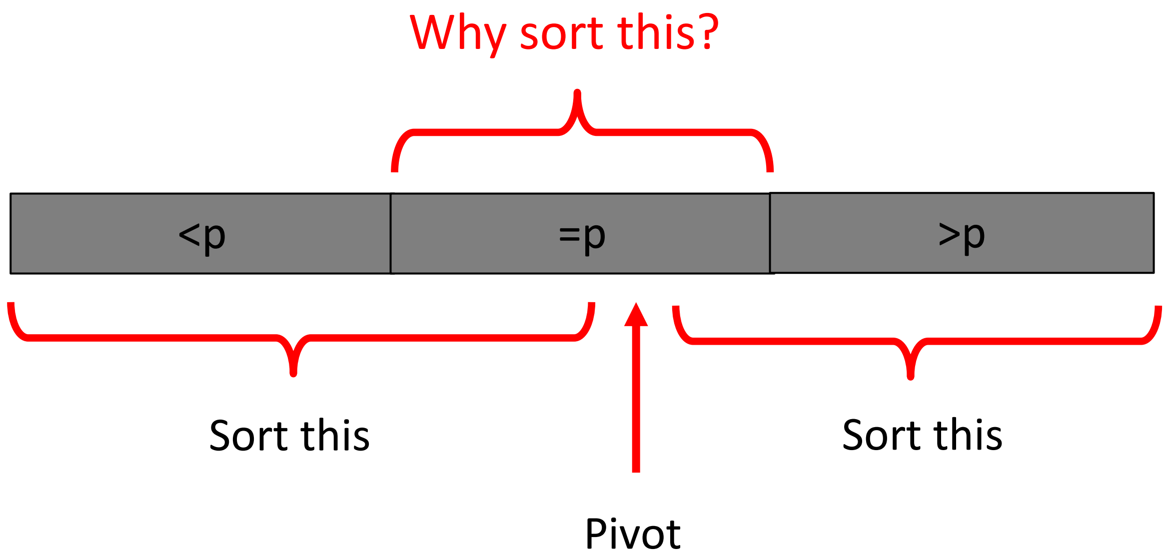


Quicksort
Partitioning…

- The stability issue however gives us an idea…

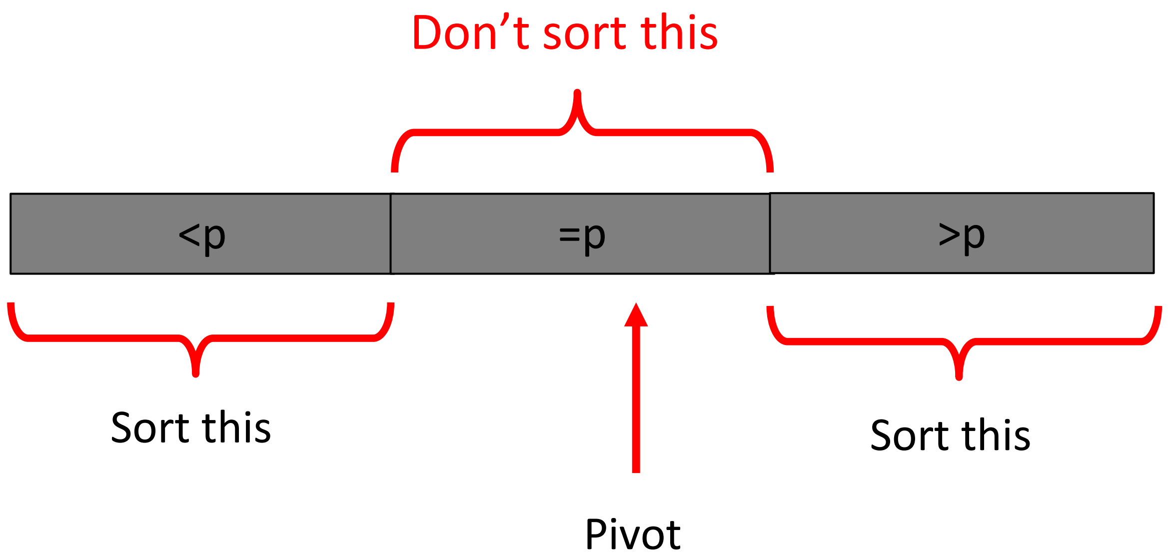


- The stability issue however gives us an idea…



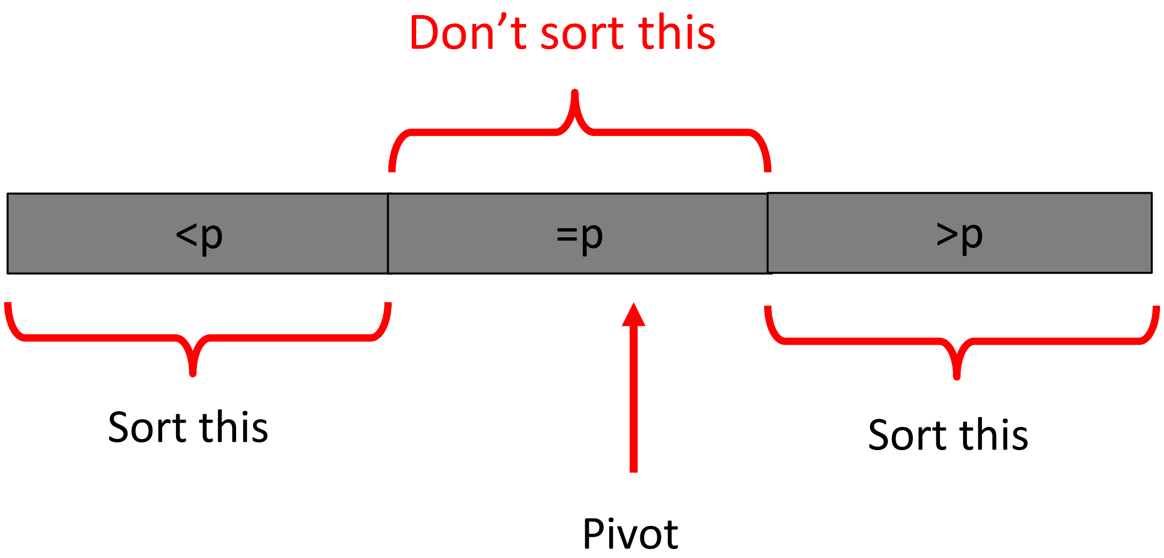
Quicksort
Partitioning…

- The stability issue however gives us an idea…



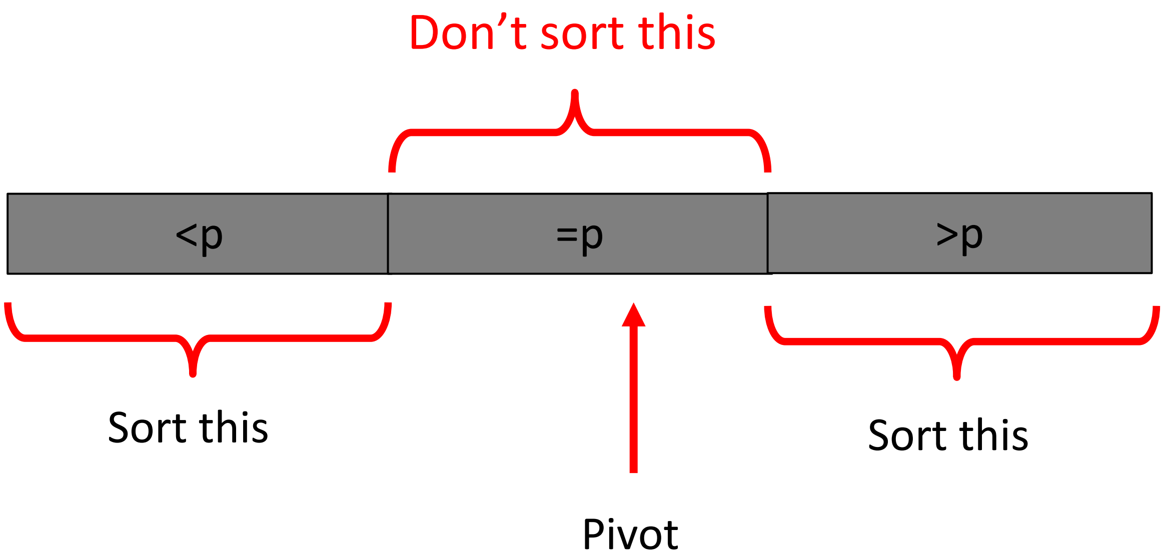
Quicksort
Partitioning…

- The stability issue however gives us an idea…



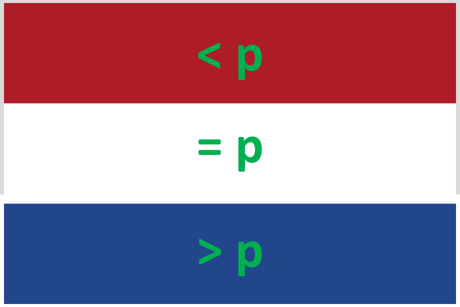
Quicksort
Partitioning…

- The stability issue however gives us an idea…

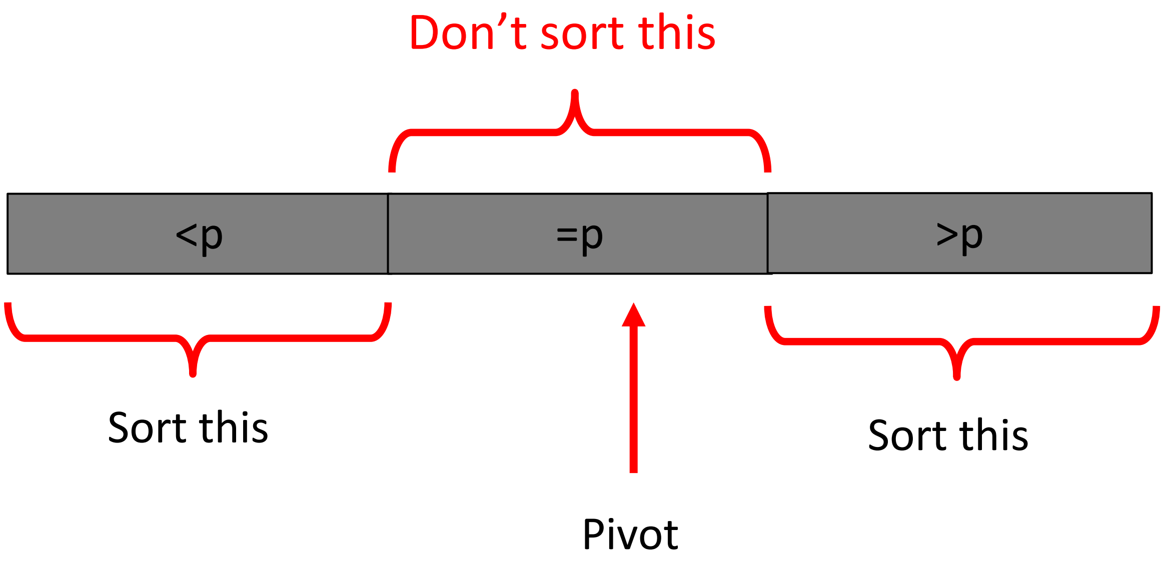


- This lead us to the Dutch national flag problem

Quicksort
Partitioning…

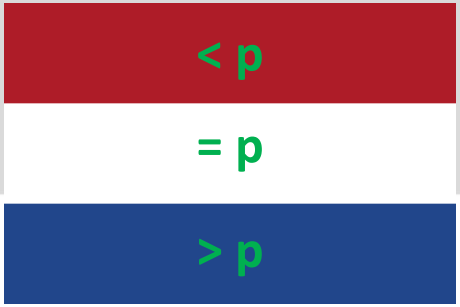


- The stability issue however gives us an idea…

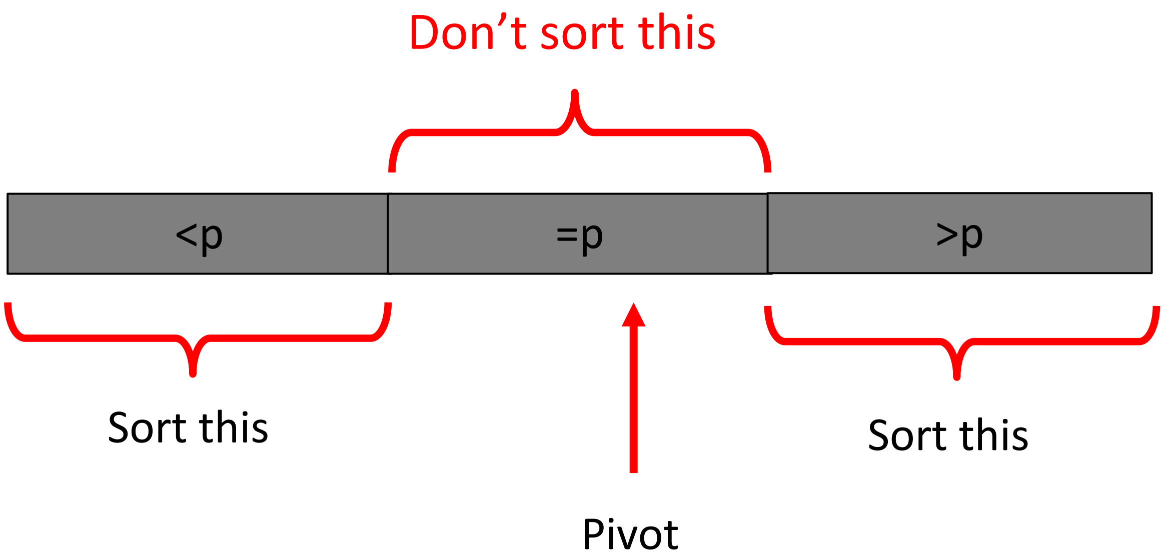


- This lead us to the Dutch national flag problem

Quicksort
Partitioning…



- The stability issue however gives us an idea…



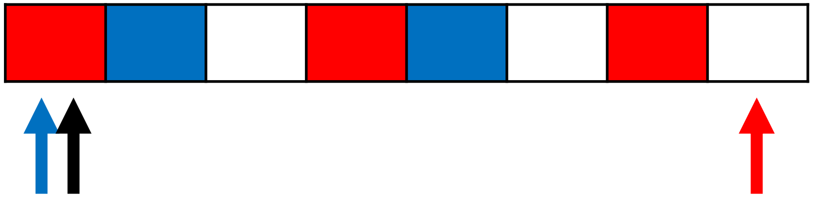
- This lead us to the Dutch national flag problem
  - Let us look at Nathan’s illustration

Dutch National Flag Algoithm

boundary1=1,

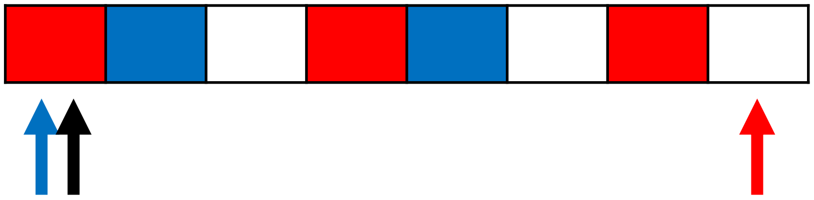
j=1

boundary2 = n



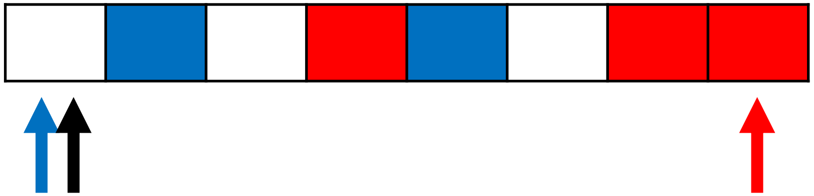
Dutch National Flag Algoithm

```
boundary1=1,
j=1 j to go through array
boundary2 = n
While j <=boundary2
    if array[j] is blue if it is blue swap with boundary 1
        swap array[boundary1], array[j]
        boundary1 += 1
        j += 1
    elif array[j] is red  if it is red swap with boundary 2
        swap array[j], array[boundary2]
        boundary2 -= 1
    else
        j += 1
```



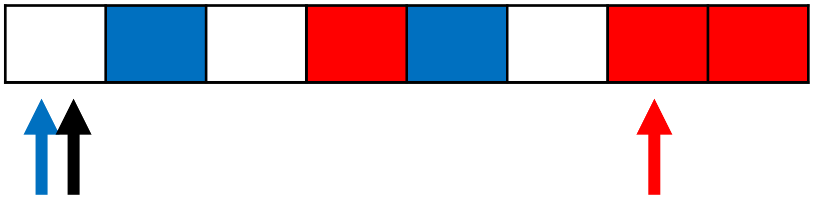
Dutch National Flag Algoithm

```
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
```



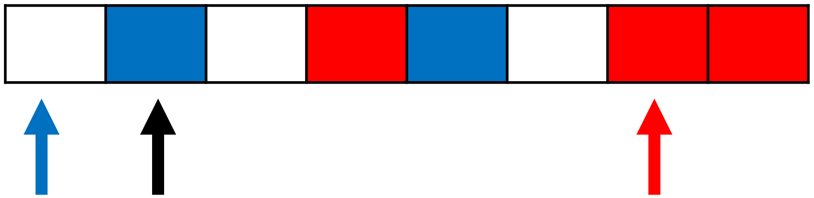
Dutch National Flag Algoithm

```
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
```



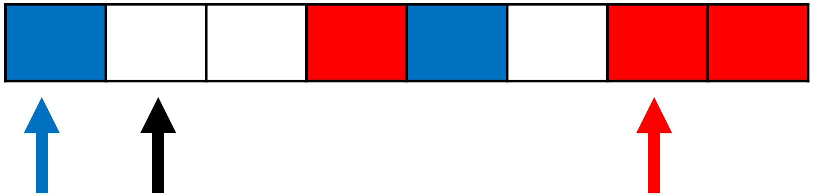
Dutch National Flag Algoithm

```
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
```



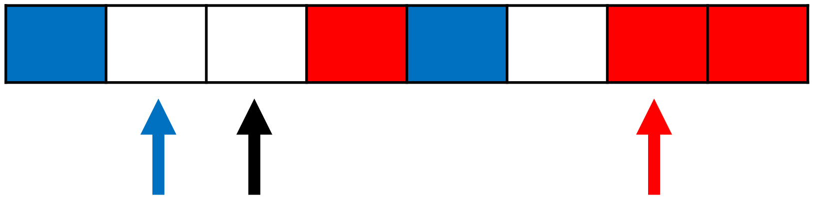
Dutch National Flag Algoithm

```
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
```



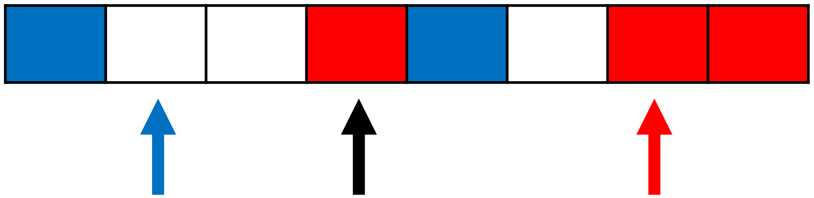
Dutch National Flag Algoithm

```
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        array[j[ and blue_ boundary = white = +=1
        j += 1
```



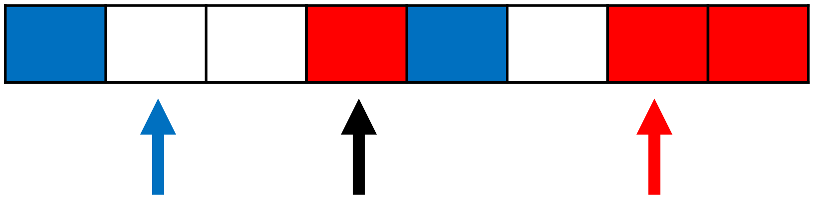
Dutch National Flag Algoithm

```
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
```



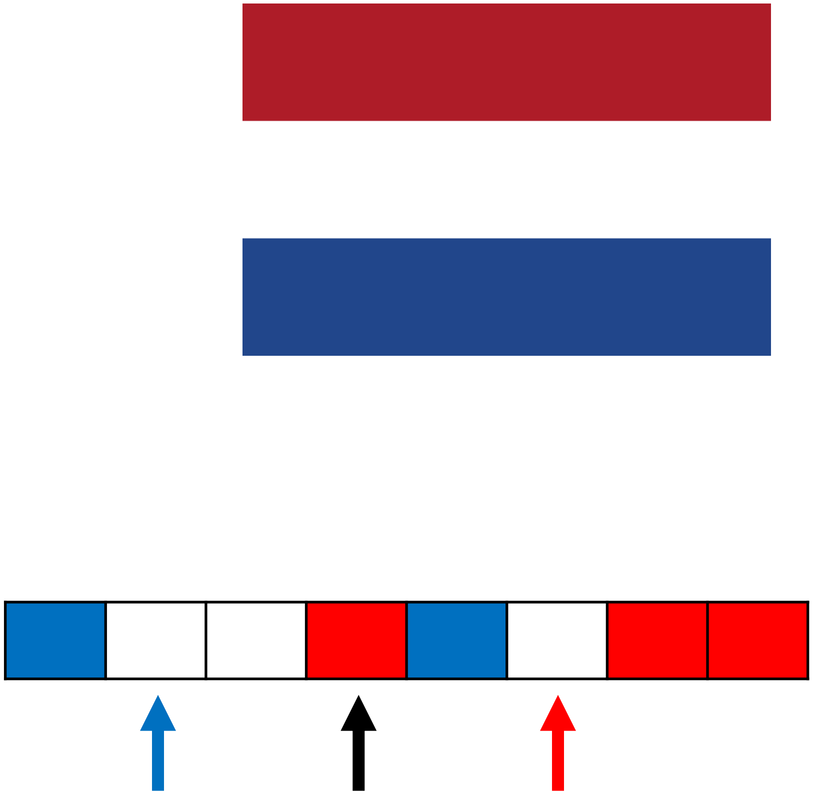
Dutch National Flag Algoithm

```
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
```



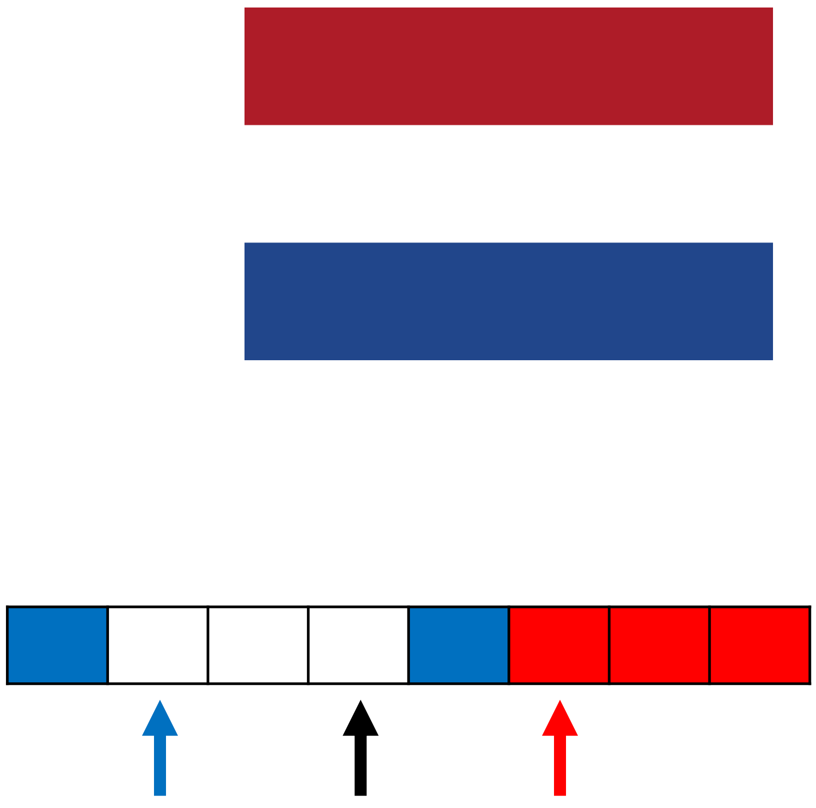
Dutch National Flag Algoithm

```
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
```



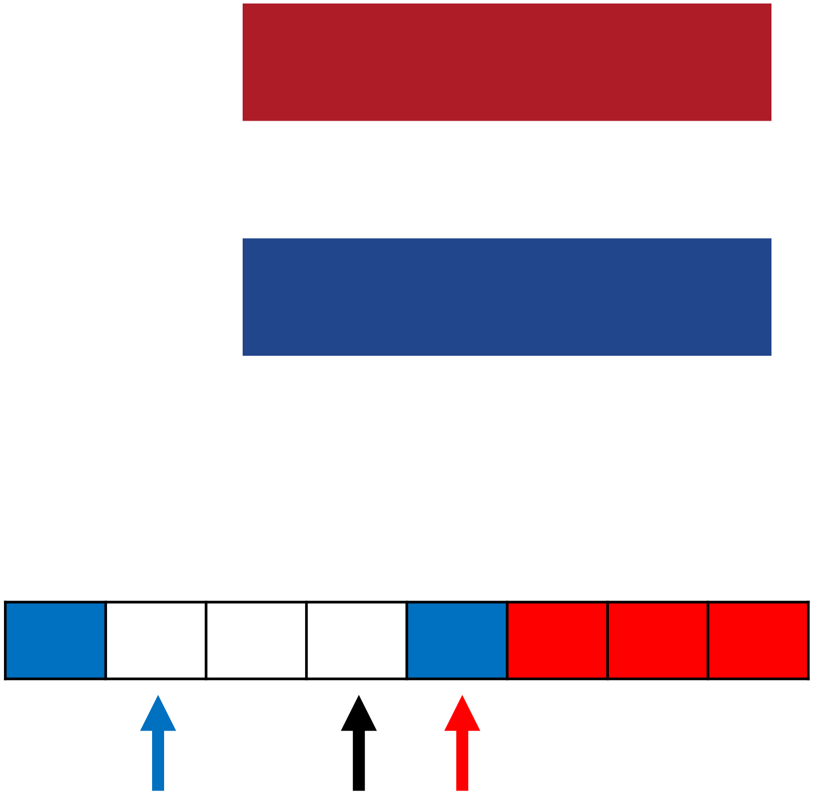
Dutch National Flag Algoithm

```
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
```



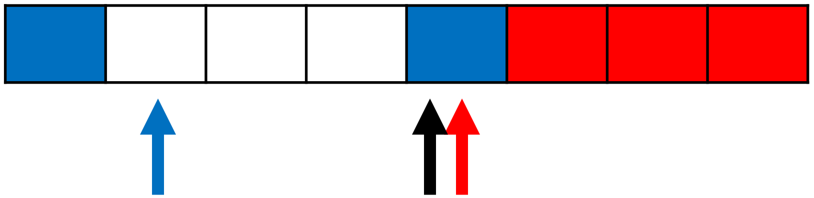
Dutch National Flag Algoithm

```
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
```



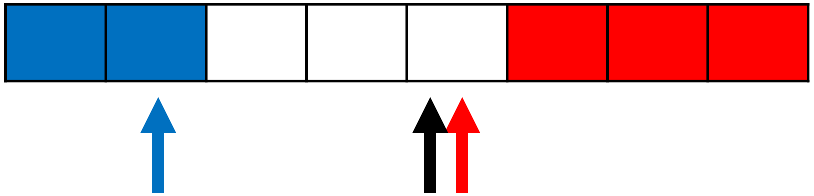
Dutch National Flag Algoithm

```
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
```



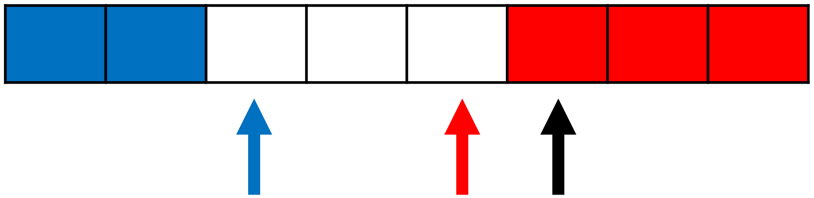
Dutch National Flag Algoithm

```
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
```



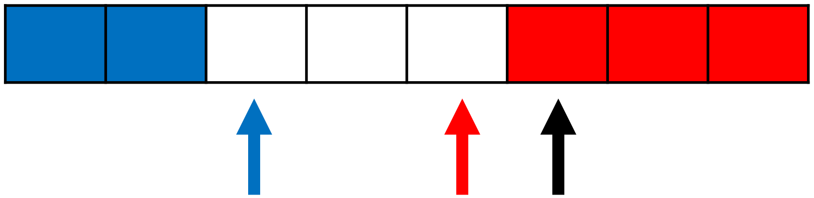
Dutch National Flag Algoithm

```
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
```



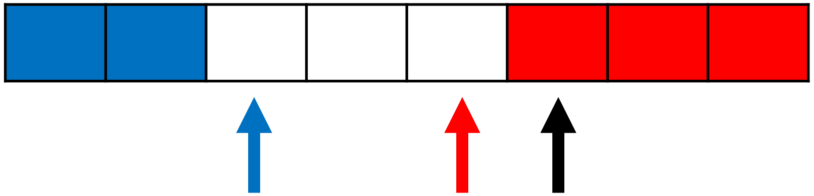
Dutch National Flag Algoithm

```
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   j to go through array un til boundary 2
    if array[j] is blue   if j = blue swap with boundary1 (value >= pivot)
        swap array[boundary1], array[j]
        boundary1 += 1
        j += 1            if j = red swap with boundary2 (value <= pivot)
    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

Questions?

- What are the invariants?
  - List[1…boundary1-1] is blue
  - List[boundary2+1...N] is red
  - List[boundary1…j-1] is white
  - List[j….boundary2] is unprocessed
    - This will be empty when I exit loop at j>boundary2

- What are the invariants?
  - List[1…boundary1-1] is blue
  - List[boundary2+1…N] is red
  - List[boundary1…j-1] is white
  - List[j….boundary2] is unprocessed only exit when partition is not done yet
    - This will be empty when I exit loop at j>boundary2

  - Note it depends if you define boundary1 and boundary2 to be inclusive or exclusive when coding…

  - Code it yourself, there is a specific case which this algorithm still fails
    - You'll need extra if-else in the loop itself

Questions?

Quicksort
Partitioning…

- Minimize swaps
- Minimize work in recursive sort

- Be in-place

Quicksort
Partitioning…

- Minimize swaps
  – We saw this with Hoare’s
- Minimize work in recursive sort

- Be in-place

if j = red swap with boundary2 (value <= pivot)

Partitioning…

- Minimize swaps
  - We saw this with Hoare’s
- Minimize work in recursive sort
  - We saw this with Dutch national flag

- Be in-place

- Minimize swaps
  - We saw this with Hoare’s
- Minimize work in recursive sort
  - We saw this with Dutch national flag
  - Left partition and right partition is smaller now
- Be in-place

- Minimize swaps
  – We saw this with Hoare's
- Minimize work in recursive sort
  – We saw this with Dutch national flag
  – Left partition and right partition is smaller now
- Be in-place
  – Save memory

Partitioning…

- Minimize swaps
  - We saw this with Hoare's
- Minimize work in recursive sort
  - We saw this with Dutch national flag
  - Left partition and right partition is smaller now
- Be in-place
  - Save memory
- Stable?

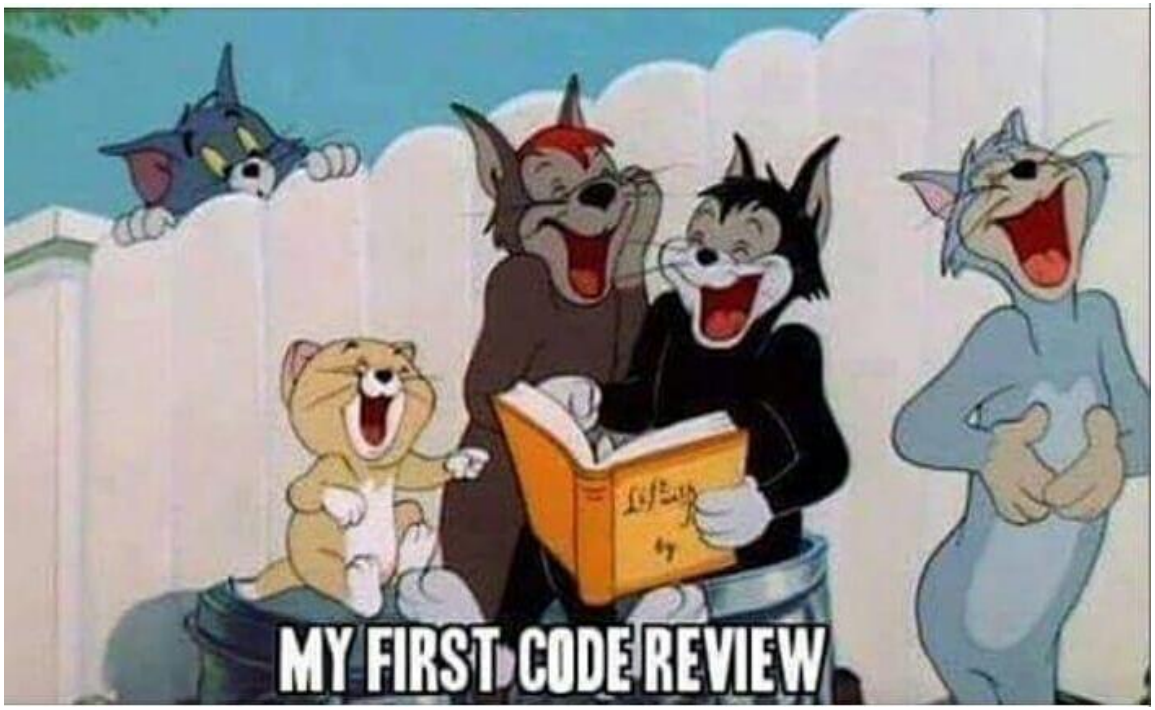
Quicksort
Partitioning…

- Minimize swaps
  - We saw this with Hoare's
- Minimize work in recursive sort
  - We saw this with Dutch national flag
  - Left partition and right partition is smaller now

- Be in-place
  - Save memory

- Stable?
  - We discuss more in the tutorials…

- Activity, why not we search online together and judge people's quick sort! #CodeReview



MONASH University

Questions?

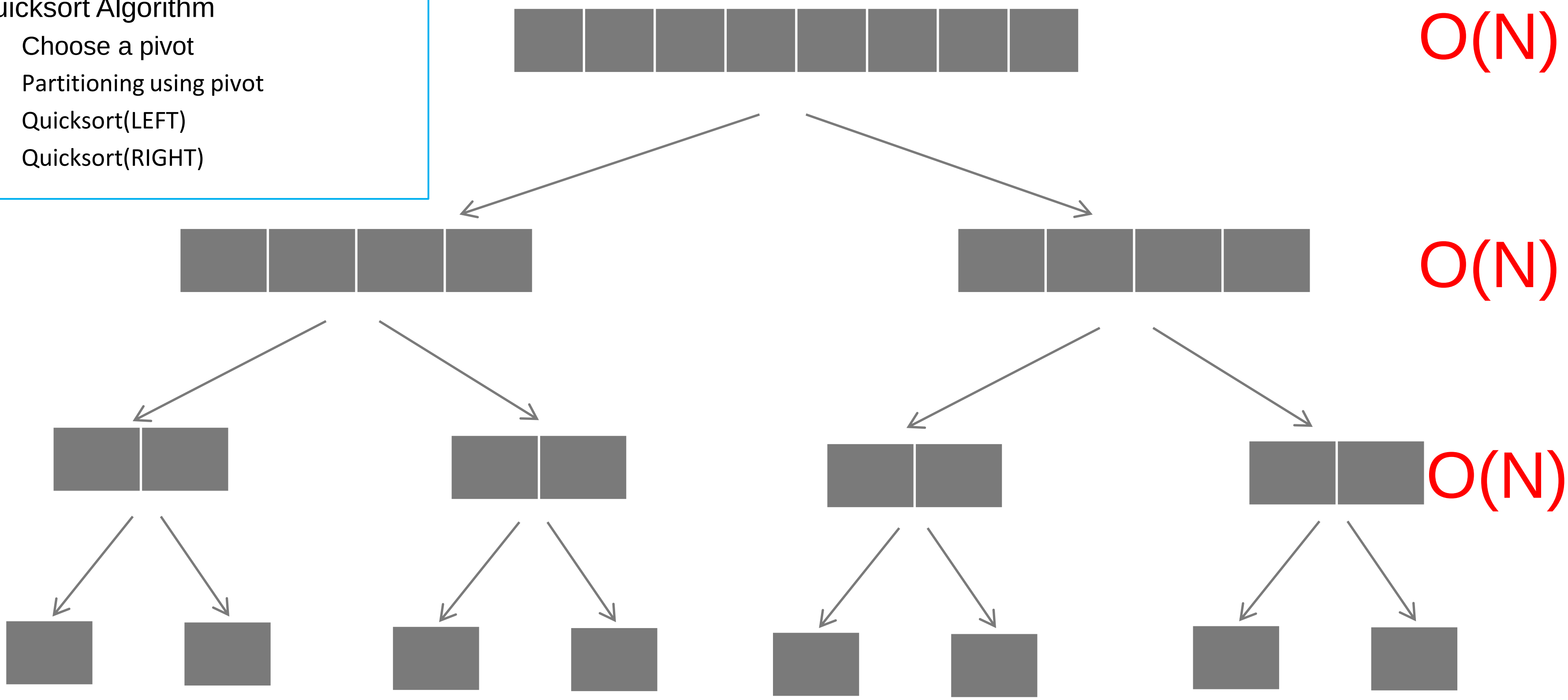
- Time complexity

- Time complexity
  - Best
  - Worst

Best-case time complexity

Quicksort Algorithm

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

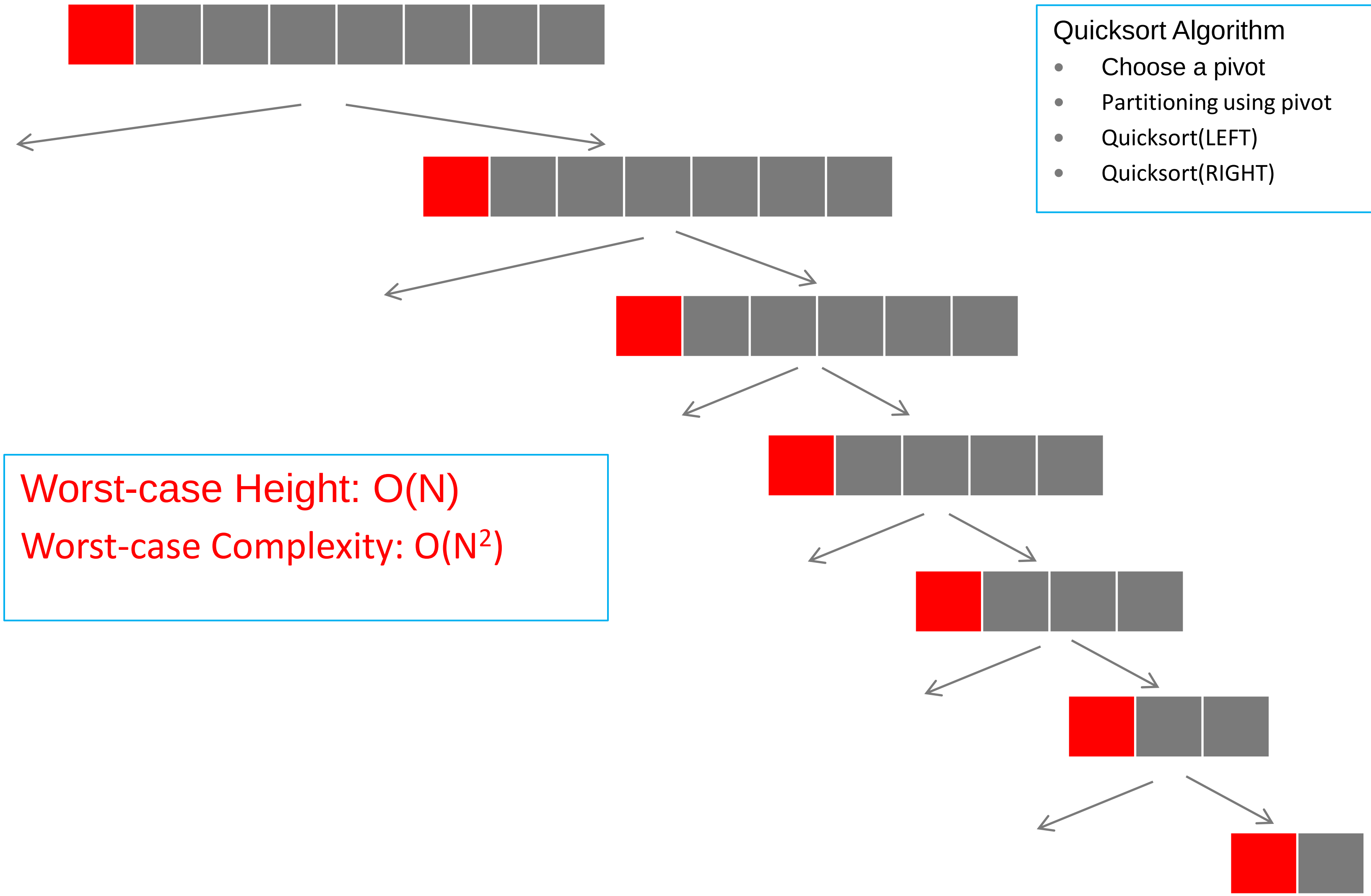


Best-case Height: O(log N)
Best-case complexity: O(N log N)

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

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: $O(N^2)$

Quicksort
Complexity Analysis

- Time complexity
  - Best
  - Worst
  - We all know this from so many discussions in FIT1008

- Time complexity
  - Best
    - When pivot split left-right evenly
  - Worst
    - When pivot split 1 side (left or right) is empty
  - We all know this from so many discussions in FIT1008

Quicksort
Complexity Analysis

- Time complexity
  - Best
    - When pivot split left-right evenly
    - O(N log N)
  - Worst
    - When pivot split 1 side (left or right) is empty
    - O(N^2)
  - We all know this from so many discussions in FIT1008

Questions?

Quicksort
Complexity Analysis

- Or we can use math
  - Like in tutorial

Quicksort
Complexity Analysis

- Or we can use math
  - Like in tutorial
  - Write the recurrence relation for the best case and worst case
    In class activity!

Quicksort
Complexity Analysis

- Or we can use math
  - Like in tutorial
  - Best case

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
Complexity Analysis

- Or we can use math
  - Like in tutorial
  - Worst case

Recurrence relation:

$$T(1) = b$$

$$T(N) = T(N\text{-}1) + c*N$$

Solution:

$O(N^2)$ either far left or far right

Questions?

MONASH University

- Time complexity
  - Best
    - When pivot split left-right evenly
    - O(N log N)
  - Worst
    - When pivot split 1 side (left or right) is empty
    - O(N^2)
  - We all know this from so many discussions in FIT1008
  - Something new is average complexity…
    - This is something I usually prefer to explain by hand in class, let me try here…

Quicksort

Complexity Analysis

- Time complexity
  - Best
    - When pivot split left-right evenly
    - O(N log N)
  - Worst
    - When pivot split 1 side (left or right) is empty
    - O(N^2)
  - We all know this from so many discussions in FIT1008
  - Something new is average complexity…
    - This is something I usually prefer to explain by hand in class, let me try here…
    - Why?

MONASH University

- Time complexity
  - Best
    - When pivot split left-right evenly
    - O(N log N)
  - Worst
    - When pivot split 1 side (left or right) is empty
    - O(N^2)
    - Probability of this to occur is very very very low
  - We all know this from so many discussions in FIT1008
  - Something new is average complexity…
    - This is something I usually prefer to explain by hand in class, let me try here…
    - Why?

quicksort does nor include pivot
better than merger sort less work on every recursion
on average complexity

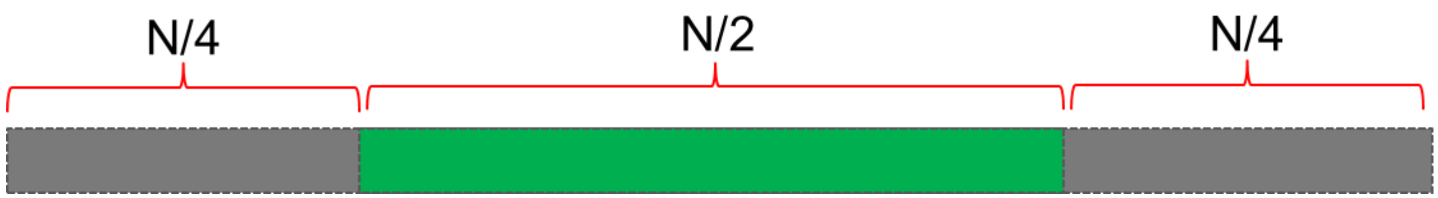
Questions?

- Consider a list with N elements

Quicksort
Complexity Analysis

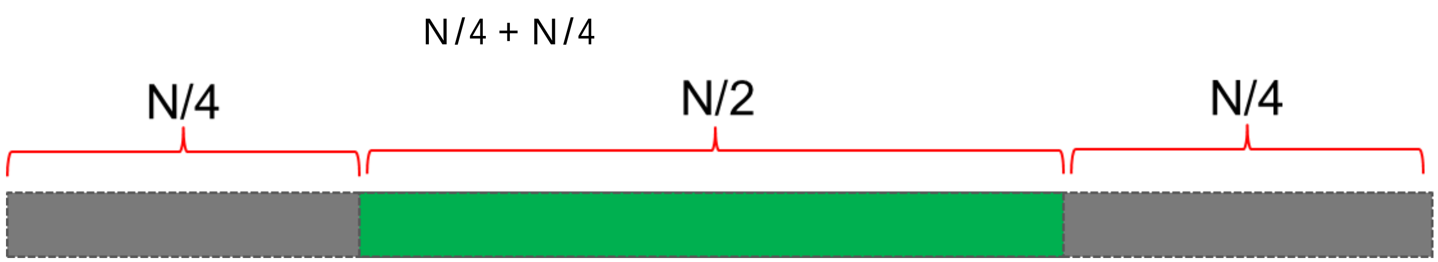
- Consider a list with N elements
  - And then we partition it

- Consider a list with N elements
  - And then we partition it



Quicksort
Complexity Analysis

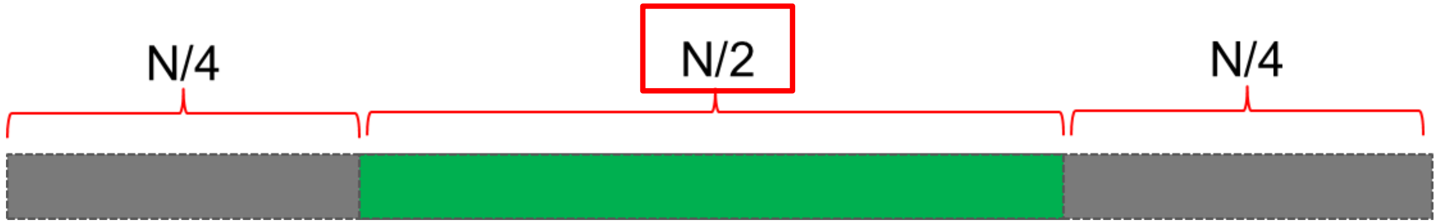
- Consider a list with N elements
  - And then we partition it



  - What is the probability we would land on the green area?

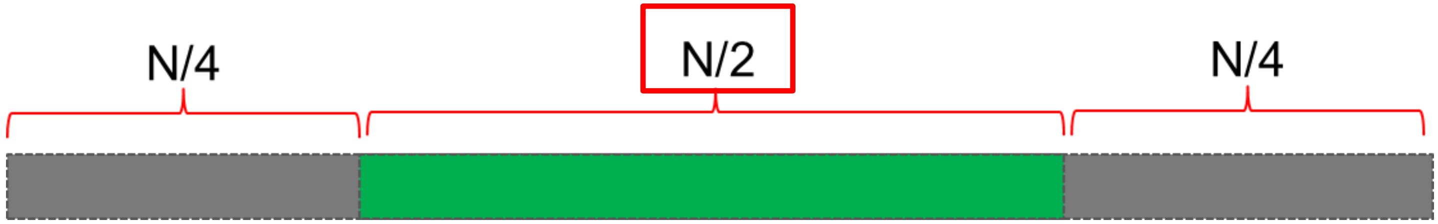
Quicksort
Complexity Analysis

- Consider a list with N elements
  - And then we partition it



  - What is the probability we would land on the green area?
    - So 50% probability

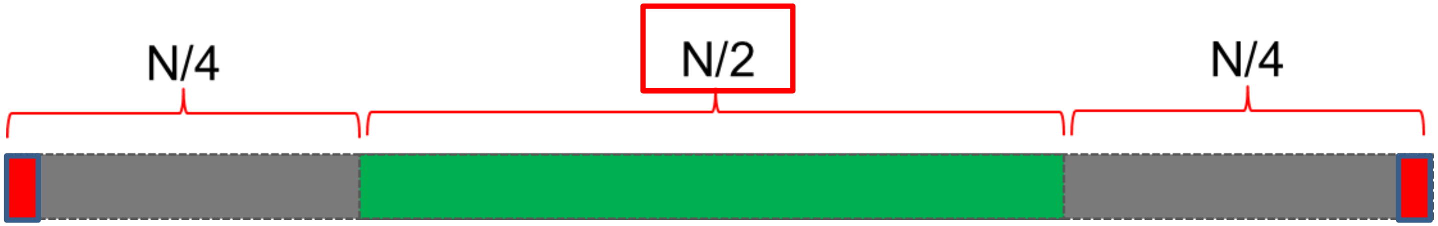
- Consider a list with N elements
  - And then we partition it



  - What is the probability we would land on the green area?
    - So 50% probability
  - Worst case in grey area?

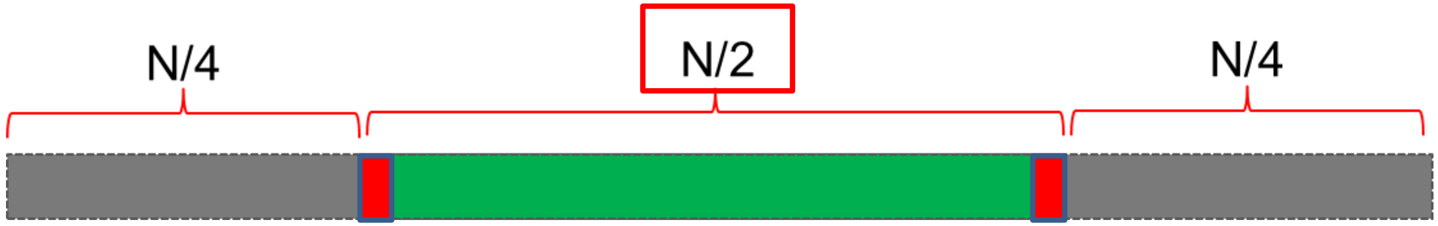
MONASH University

- Consider a list with N elements
  - And then we partition it



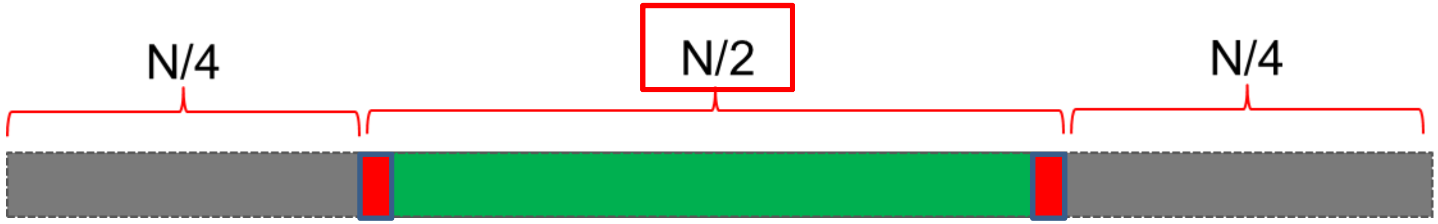
  - What is the probability we would land on the green area?
    - So 50% probability
  - Worst case in grey area? Pivot 1 or N-1

- Consider a list with N elements
  - And then we partition it



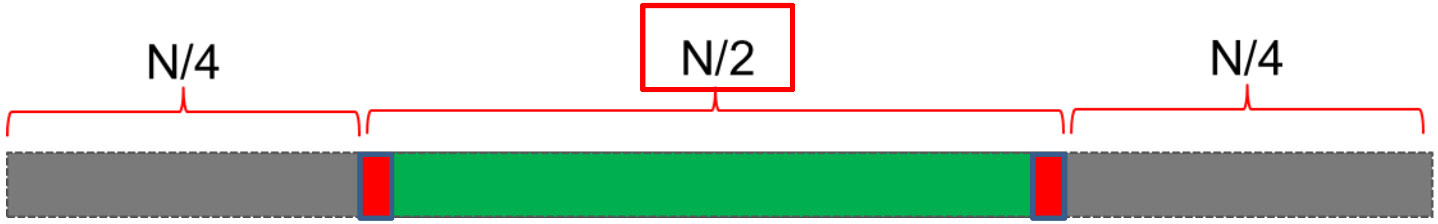
  - What is the probability we would land on the green area?
    - So 50% probability
  - Worst case in grey area? Pivot 1 or N-1
  - Worst case in green area? Pivot at N/4 or 3N/4

- Consider a list with N elements
  - And then we partition it



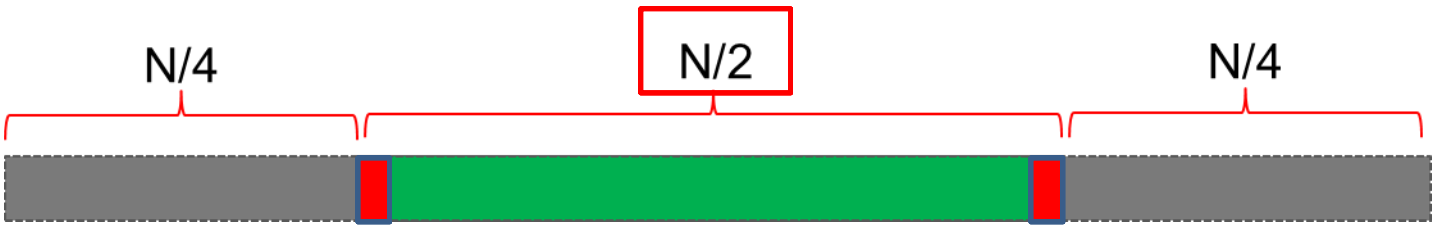
  - What is the probability we would land on the green area?
    - So 50% probability
  - Worst case in grey area? Pivot 1 or N-1
  - Worst case in green area? Pivot at N/4 or 3N/4
  - If we always hit the green area, we will get a maximum recursion height of h…

- Consider a list with N elements
  – And then we partition it



  – What is the probability we would land on the green area?
    - So 50% probability
  – Worst case in grey area? Pivot 1 or N-1
  – Worst case in green area? Pivot at N/4 or 3N/4
  – If we always hit the green area, we will get a maximum recursion height of h… So what is the upper bound for the height if we land on green 50%?

- Consider a list with N elements
  - And then we partition it



  - What is the probability we would land on the green area?
    - So 50% probability
  - Worst case in grey area? Pivot 1 or N-1
  - Worst case in green area? Pivot at N/4 or 3N/4
  - If we always hit the green area, we will get a maximum recursion height of h… So what is the upper bound for the height if we land on green 50%? 2h

N * (1/2)^h = 1
N = 1 / (1/2)^h
N = 2^h
h = log2 N 2h = 2log2 N = log2 (N^2) / log2 1 = log (N^2)/ 0 (assum to be 1) = N^2
N * h = Nlog(N) , if h = 2h, N * 2h = N^2

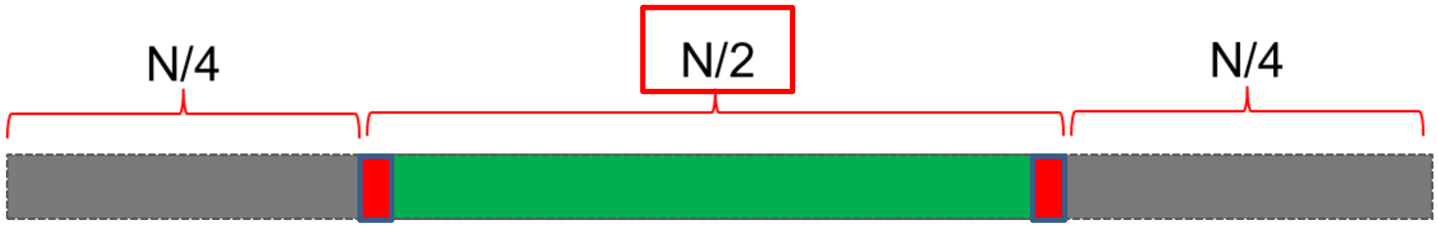
after h=1, N -> N/2

if always choose correct pivot (median)
or orginal array and the following shirked array
for recursion, N is halved very time

Quicksort

Complexity Analysis

- Consider a list with N elements
  – And then we partition it

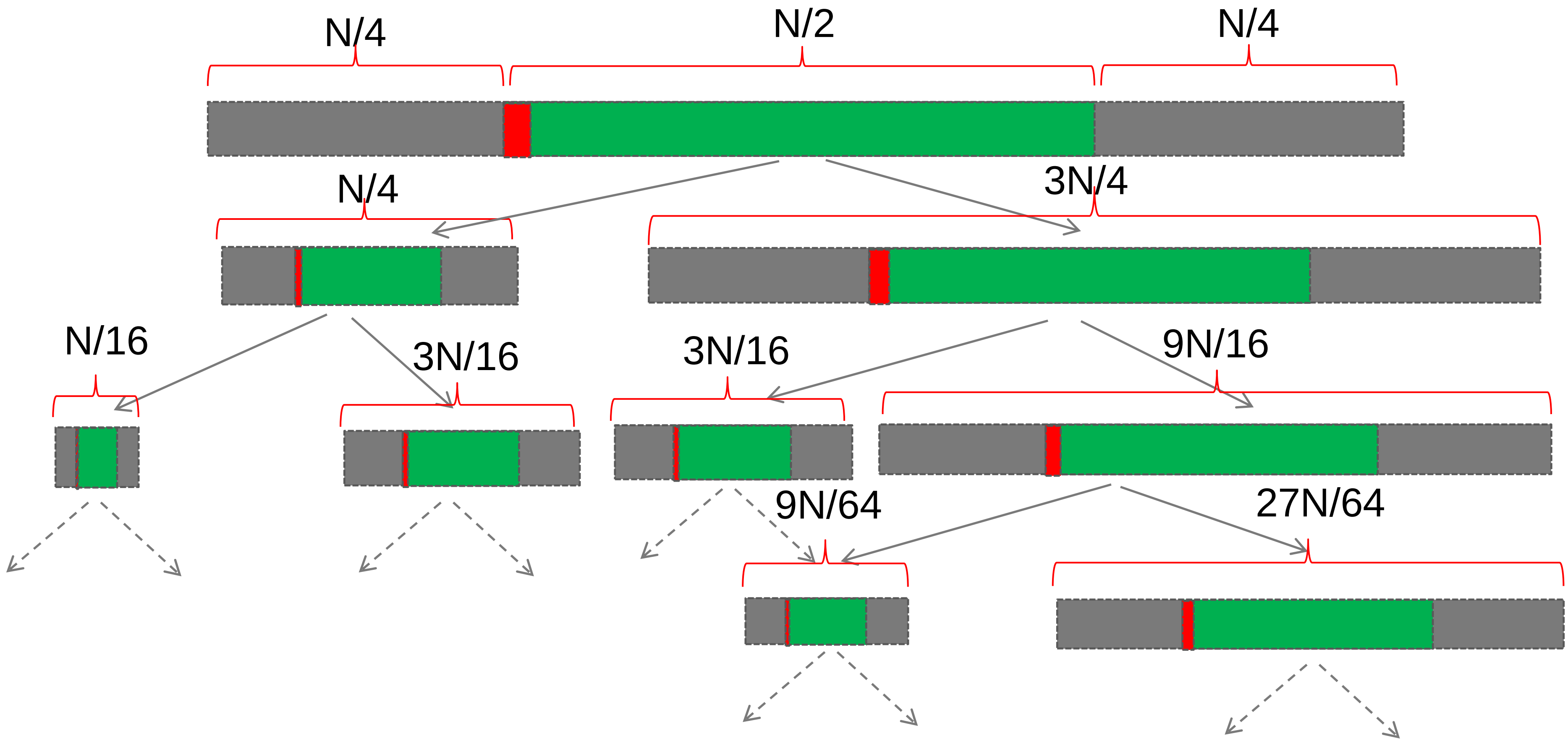


  – What is the probability we would land on the green area?
    - So 50% probability
  – Worst case in grey area? Pivot 1 or N-1
  – Worst case in green area? Pivot at N/4 or 3N/4
  – If we always hit the green area, we will get a maximum recursion height of h… So what is the upper bound for the height if we land on green 50%? 2h

log2 N = h if pivot in the middle
upper bound = 2h (if pivot fall at the end) worse case

- So we calculate what is h

- Let us just do the normal drawing



Quicksort
Complexity Analysis

- Let us just do the normal drawing
  - From your recursive knowledge, T(N) -> T(3N/4)

Quicksort

Complexity Analysis

- Let us just do the normal drawing
  - From your recursive knowledge, T(N) -> T(3N/4)
  - Reach base case when size is 1, thus base when T(1)

- Let us just do the normal drawing
  - From your recursive knowledge, T(N) -> T(3N/4)
  - Reach base case when size is 1, thus base when T(1)
  - Thus…

- Let us just do the normal drawing
  - From your recursive knowledge, T(N) -> T(3N/4)
  - Reach base case when size is 1, thus base when T(1)

  - Thus… N(3/4)^h = 1

Quicksort

Complexity Analysis

- Let us just do the normal drawing higher complexity partition one
  - From your recursive knowledge, T(N) -> T(3N/4) biggest complexity
  - Reach base case when size is 1, thus base when T(1) 3/4 part would become just 1 element where it is base case

we can get the max height (h = how many time of 3/4 would need to shrink N into size 1 (base case)

  - Thus… N(3/4)^h = 1
  - Which gives us $(3/4)^h$ N = 1 → N = $(4/3)^h$ → h = $\log_{4/3}$ N

N /1 = 1/(3/4)^h = (3/4)^(-h) = ((3/4)^-1)^h = (4/3)^h
N = (4/3)^h log 4/3 N = log 4/3 (4/3)^h = h

- Let us just do the normal drawing
  - From your recursive knowledge, T(N) -> T(3N/4)
  - Reach base case when size is 1, thus base when T(1)

  - Thus… N(3/4)^h = 1
  - Which gives us $(3/4)^h$ N = 1 → N = $(4/3)^h$ → h = $\log_{4/3}$ N

  - Our maximum depth for average case is 2h

- Let us just do the normal drawing
  - From your recursive knowledge, T(N) -> T(3N/4)
  - Reach base case when size is 1, thus base when T(1)

  - Thus… N(3/4)^h = 1
  - Which gives us $(3/4)^h N = 1 \rightarrow N = (4/3)^h \rightarrow h = \log_{4/3} N$

  - Our maximum depth for average case is 2h
    - If within green always it is h
    - But it isn't always green, so it can be more than h (by some factor)
    - But since we have a 50% chance at each level
    - We can just average it out to 2h

MONASH University

- Let us just do the normal drawing
  - From your recursive knowledge, T(N) -> T(3N/4)
  - Reach base case when size is 1, thus base when T(1)

  - Thus… N(3/4)^h = 1
  - Which gives us $(3/4)^h$ N = 1 → N = $(4/3)^h$ → h = $\log_{4/3}$ N

  - Our maximum depth for average case is 2h
  - Which give us 2 $\log_{4/3}$ N

- Let us just do the normal drawing
  - From your recursive knowledge, T(N) -> T(3N/4)
  - Reach base case when size is 1, thus base when T(1)

  - Thus… N(3/4)^h = 1
  - Which gives us $(3/4)^h N = 1 \rightarrow N = (4/3)^h \rightarrow h = \log_{4/3} N$

  - Our maximum depth for average case is 2h
  - Which give us $2 \log_{4/3} N$

  - Therefore, average case height is O(log N)

MONASH University

- Let us just do the normal drawing
  - From your recursive knowledge, T(N) -> T(3N/4)
  - Reach base case when size is 1, thus base when T(1)
  - Thus… N(3/4)^h = 1
  - Which gives us $(3/4)^h$ N = 1 → N = $(4/3)^h$ → h = $\log_{4/3}$ N
  - Our maximum depth for average case is 2h
  - Which give us 2 $\log_{4/3}$ N
  - Therefore, average case height is O(log N)
  - Each level have a partition cost of O(N)

- Let us just do the normal drawing
  - From your recursive knowledge, T(N) -> T(3N/4)
  - Reach base case when size is 1, thus base when T(1)

  - Thus… N(3/4)^h = 1
  - Which gives us $(3/4)^h$ N = 1 → N = $(4/3)^h$ → h = $\log_{4/3}$ N

  - Our maximum depth for average case is 2h
  - Which give us 2 $\log_{4/3}$ N

  - Therefore, average case height is O(log N)
  - Each level have a partition cost of O(N)
  - Total average cost is O(N log N)

MONASH University

- Let us just do the normal drawing
  - From your recursive knowledge, T(N) -> T(3N/4)
  - Reach base case when size is 1, thus base when T(1)

  - Thus… N(3/4)^h = 1
  - Which gives us $(3/4)^h$ N = 1 → N = $(4/3)^h$ → h = $\log_{4/3} N$

  - Our maximum depth for average case is 2h
  - Which give us $2 \log_{4/3} N$

  - Therefore, average case height is O(log N)
  - Each level have a partition cost of O(N)
  - Total average cost is O(N log N)

MONASH University

- Let us just do the normal drawing
  - From your recursive knowledge, T(N) -> T(3N/4)
  - Reach base case when size is 1, thus base when T(1)

    3/4 part would become just 1 element where it is base case

    we can get the max height (h = how many time of 3/4 would need to shrink N into size 1 (base case)
  - Thus… N(3/4)^h = 1
  - Which gives us $(3/4)^h$ N = 1 → N = $(4/3)^h$ → h = $\log_{4/3} N$

    N /1 = 1/(3/4)^h = (3/4)^(-h) = ((3/4)^-1)^h = (4/3)^h
    N = (4/3)^h log 4/3 N = log 4/3 (4/3)^h = h
  - Our maximum depth for average case is 2h
  - Which give us $2 \log_{4/3} N$
  - Therefore, average case height is O(log N)
  - Each level have a partition cost of O(N)
  - Total average cost is O(N log N) ← But it isn't base 2 for log?

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: $$\log_a N = \frac{\log_b N}{\underbrace{\log_b a}_{\text{constant}}}$$

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

Questions?

Quicksort

Complexity Analysis

- Can be done with math as well
  - Just like best case and worst case

- Can be done with math as well
  - Just like best case and worst case
  - Just that it is really painful to do… and thus not examinable

after h=1, N -> N/2

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

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

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

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

T(N-1)	T(0)
T(N-2)	T(1)
T(N-3)	T(2)
…	…
	T(N-3)
	T(N-2)
	T(N-1)

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

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

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

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

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)

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

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

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:

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

N-k =1 → k = N-1

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

Simplify

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

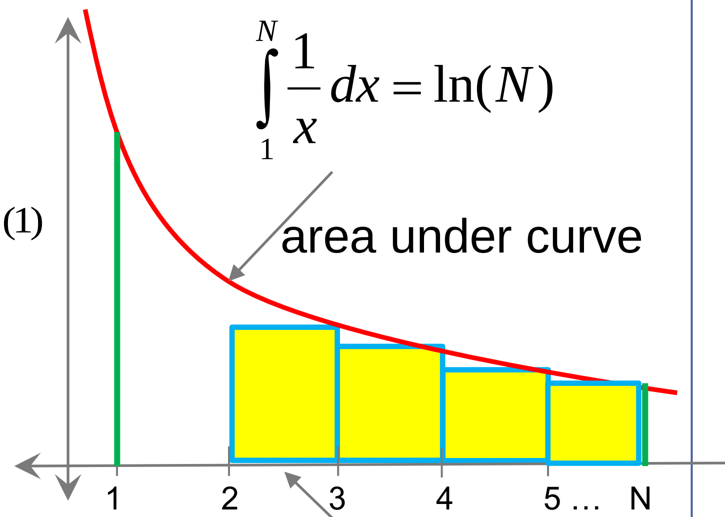
$$T(N) = 2 + b(N+1) + 2(N+1)\sum_{k=3}^{N} \frac{1}{k}$$

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

T(N) = O (N log N)

$$\int_1^N \frac{1}{x} dx = \ln(N)$$

area under curve



Area of rectangles

$$\sum_{k=3}^{N} \frac{1}{k}$$

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

Questions?

- Good sorting algorithm
  - Using divide and conquer

Quick Sort
Summary

- Good sorting algorithm
  - Using divide and conquer
  - Pivot will always be at sorted position after each iteration

Quick Sort
Summary

- Good sorting algorithm
  - Using divide and conquer
  - Pivot will always be at sorted position after each iteration
    - We can ignore it at future iteration of sorting
    - Unlike merge sort

Summary

- Good sorting algorithm
  - Using divide and conquer
    - 3 partitioning strategy
    - … and we compared them all
  - Pivot will always be at sorted position after each iteration
    - We can ignore it at future iteration of sorting
    - Unlike merge sort

Quick Sort
Summary

- Good sorting algorithm
  - Using divide and conquer
    - 3 partitioning strategy
    - … and we compared them all
  - Pivot will always be at sorted position after each iteration
    - We can ignore it at future iteration of sorting
    - Unlike merge sort

- And we looked at the complexity
  - Best case
  - Worst case
  - Average case

- Good sorting algorithm
  - Using divide and conquer
    - 3 partitioning strategy
    - … and we compared them all
  - Pivot will always be at sorted position after each iteration
    - We can ignore it at future iteration of sorting
    - Unlike merge sort

- And we looked at the complexity
  - Best case
  - Worst case
  - Average case
  - Everything depends on pivot!

- Good sorting algorithm
  - Using divide and conquer
    - 3 partitioning strategy
    - … and we compared them all
  - Pivot will always be at sorted position after each iteration
    - We can ignore it at future iteration of sorting
    - Unlike merge sort

- And we looked at the complexity
  - Best case
  - Worst case
  - Average case
  - Everything depends on pivot! Next lecture on how to select pivot

Questions?

Thank You