

FIT2004

Algorithms and Data Structures

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

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



Faculty of Information Technology, Monash University

COMMONWEALTH OF AUSTRALIA

Copyright Regulations 1969

This material has been reproduced and communicated to you by or on behalf of Monash University pursuant to Part VB of the Copyright Act 1968 (the Act). The material in this communication may be subject to copyright under the Act. Any further reproduction or communication of this material by you may be the subject of copyright protection under the Act. Do not remove this notice

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

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...

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?

Quicksort

Example

- Given a list

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

Quicksort

Example

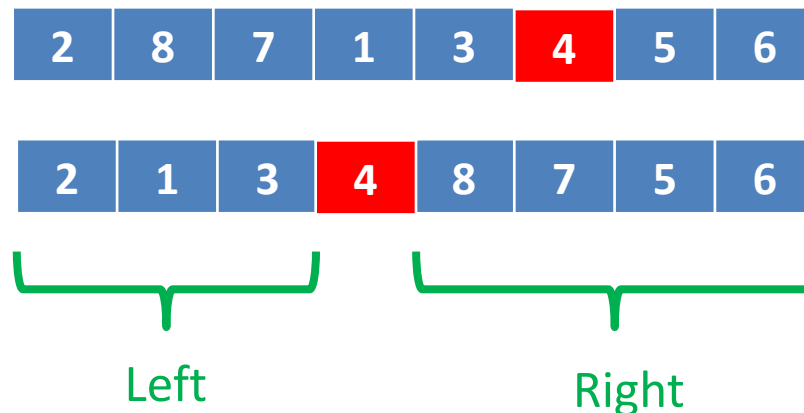
- Given a list
- Choose a **pivot** (doesn't matter which)

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

Quicksort

Example

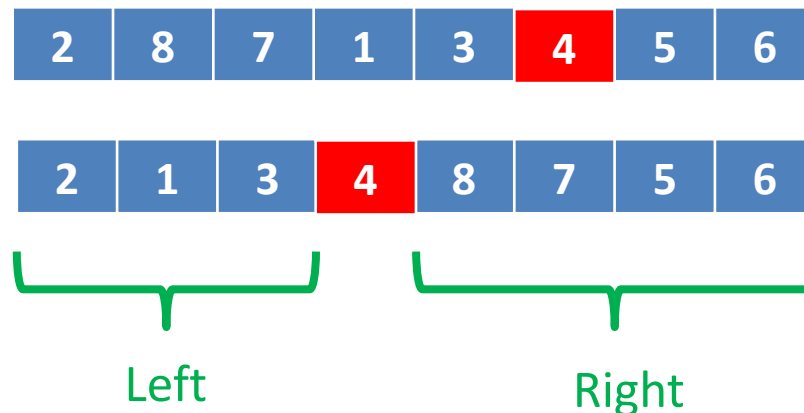
- Given a list
- Choose a **pivot** (doesn't matter which)
- Ensure invariant
 - Left \leq pivot
 - Right $>$ pivot



Quicksort

Example

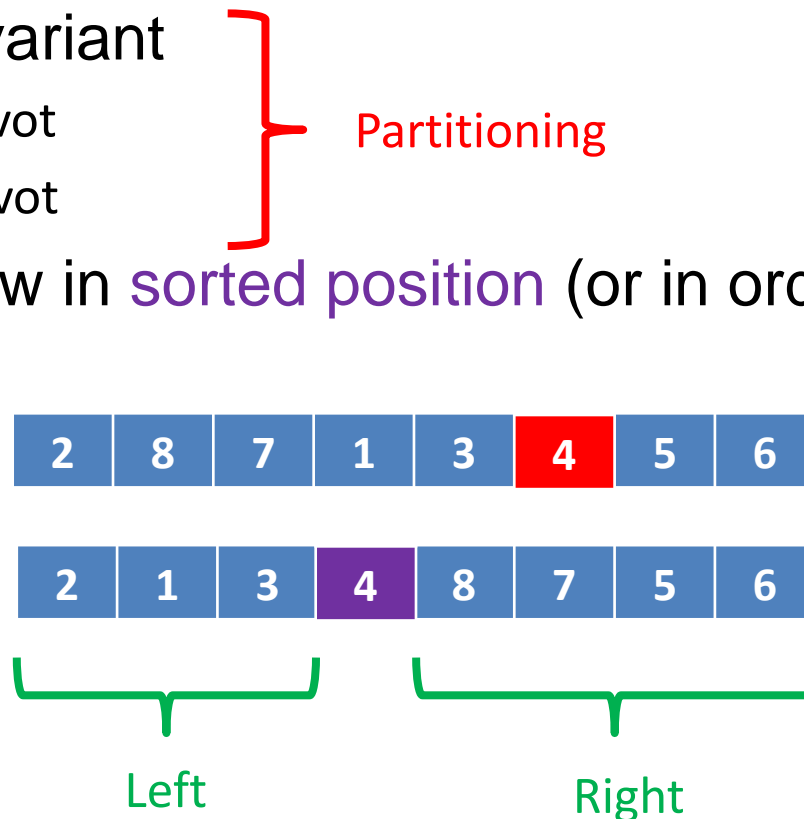
- Given a list
 - Choose a **pivot** (doesn't matter which)
 - Ensure invariant
 - Left \leq pivot
 - Right $>$ pivot
- } Partitioning



Quicksort

Example

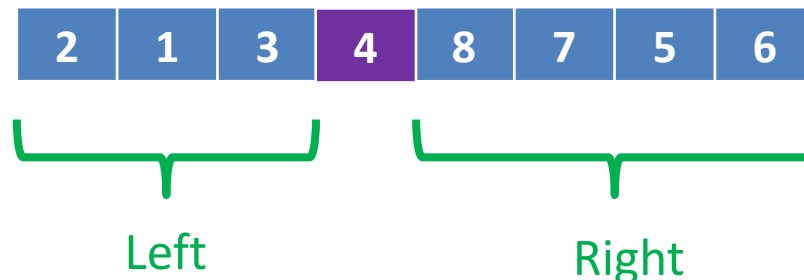
- Given a list
- Choose a **pivot** (doesn't matter which)
- Ensure invariant
 - Left \leq pivot
 - Right $>$ pivot
- Pivot is now in **sorted position** (or in order)



Quicksort

Example

- Given a list
 - Choose a **pivot** (doesn't matter which)
 - Ensure invariant
 - Left \leq 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 \leq 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 \leq 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 \leq 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 \leq 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 \leq 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 \leq 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 \leq 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 \leq pivot
 - Right $>$ pivot
- Pivot is now in **sorted position** (or in order)
- Then we repeat for **left** and **right**
- Till **sorted**

} Partitioning

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

Questions?

Quicksort

Partitioning

- What is partitioning?

Quicksort

Partitioning

- What is partitioning?
 - Separate the list into parts

- 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



Quicksort

Partitioning

- 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



LEFT

RIGHT

Quicksort

Out-of-place Partitioning

- Not in-place



LEFT

RIGHT

Quicksort

Out-of-place Partitioning

- Not in-place



LEFT
RIGHT



Quicksort

Out-of-place Partitioning

- Not in-place



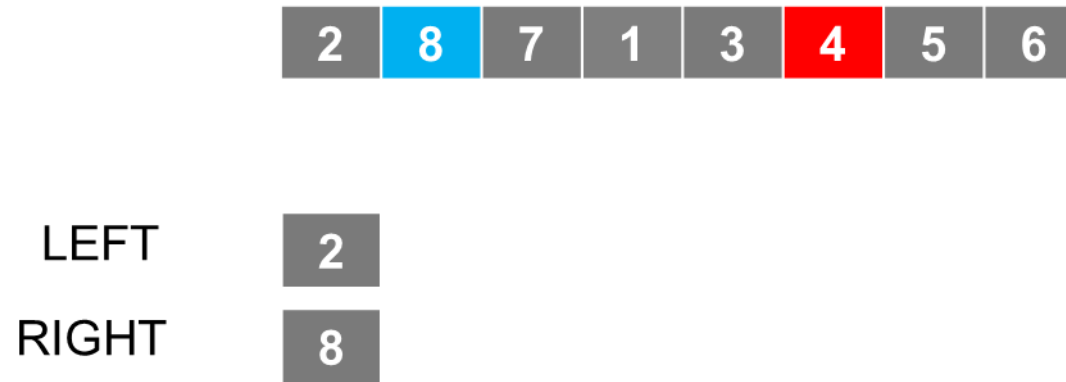
LEFT
RIGHT



Quicksort

Out-of-place Partitioning

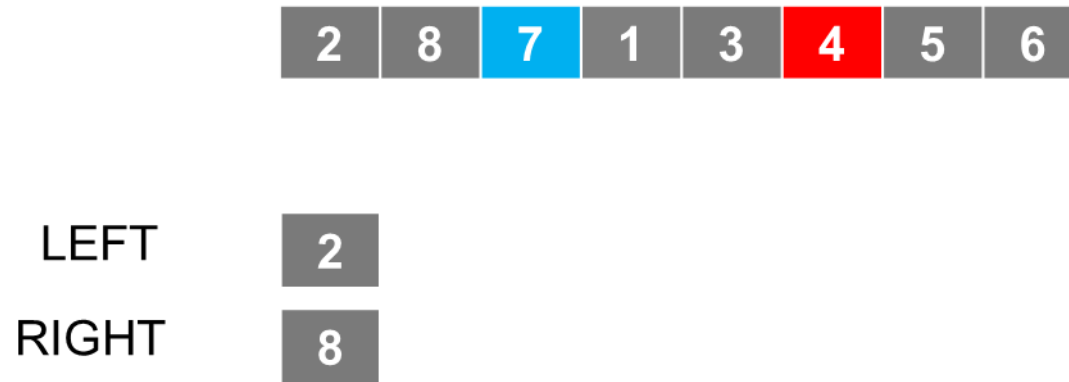
- Not in-place



Quicksort

Out-of-place Partitioning

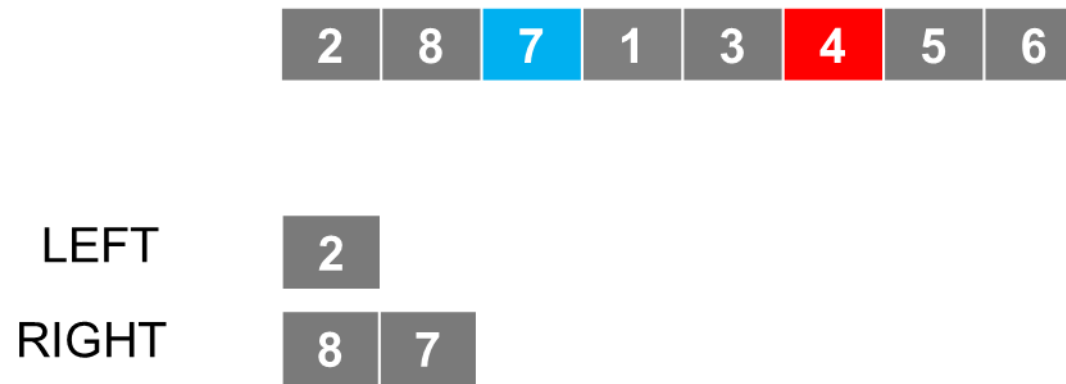
- Not in-place



Quicksort

Out-of-place Partitioning

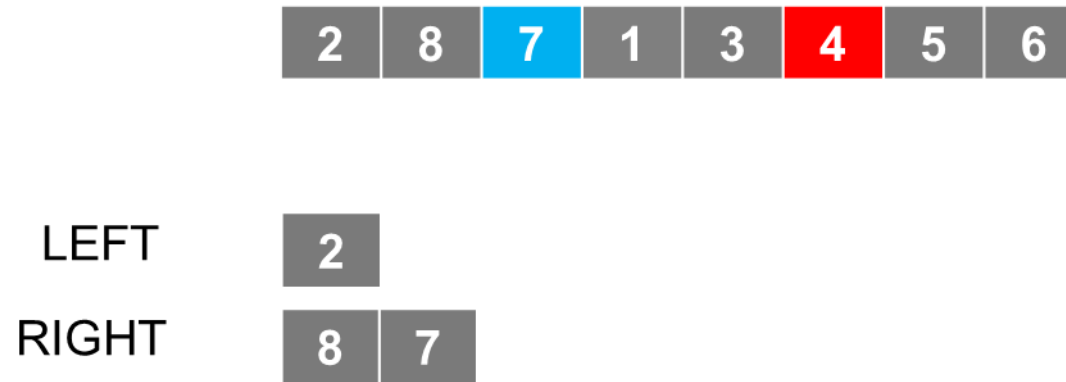
- Not in-place



Quicksort

Out-of-place Partitioning

- Not in-place



Quicksort

Out-of-place Partitioning

- Not in-place

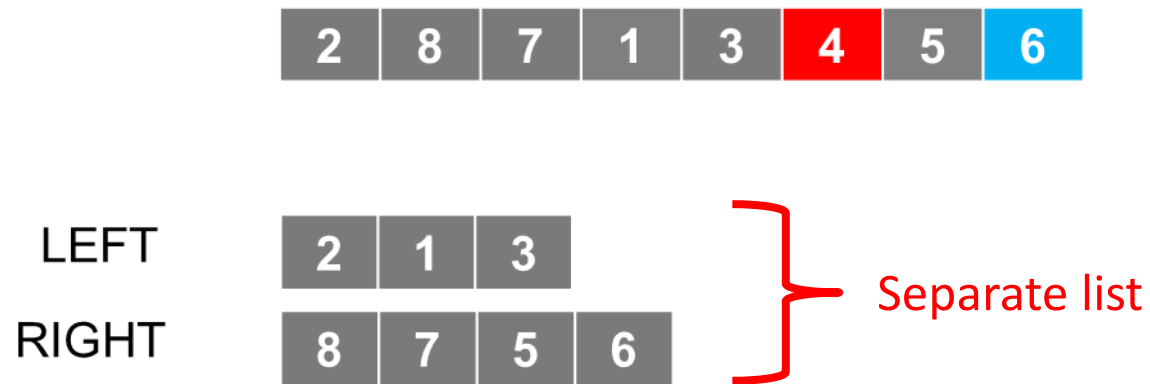


Repeat till last element

Quicksort

Out-of-place Partitioning

- Not in-place



Quicksort

Out-of-place Partitioning

- Not in-place



LEFT

RIGHT

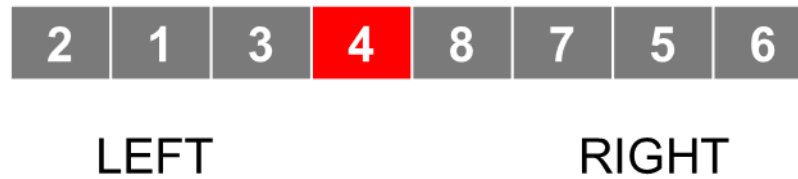


Separate list/ onto original list

Quicksort

Out-of-place Partitioning

- Not in-place



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



Quicksort

Out-of-place Partitioning

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



- 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



- Is the algorithm stable?
 - \leq 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



- Is the algorithm stable?
 - \leq pivot to the left: **everything \leq 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



- Is the algorithm stable? **NO**
 - \leq 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



- Is the algorithm stable? **NO**
 - \leq pivot to the left: **everything \leq 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



LEFT

RIGHT

- Is the algorithm stable? **NO**
 - \leq pivot to the left: **everything \leq 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?

Quicksort

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

- 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

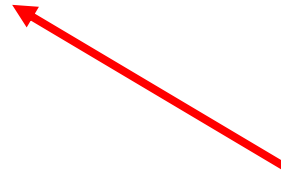
- We want to make it in-place
 - Save memory
- We want to make it fast
 - Avoid copying many items
 - Avoid swapping many times
- We want to make it stable
 - Can we?

- 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?

- 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



Pointers used for
swapping

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)

$L_bad = 2$, $R_bad = N$

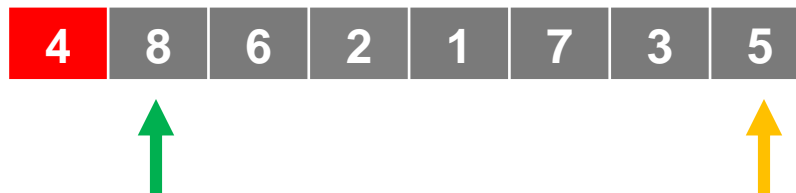
Repeat until L_bad and R_bad cross

Terminating condition

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)

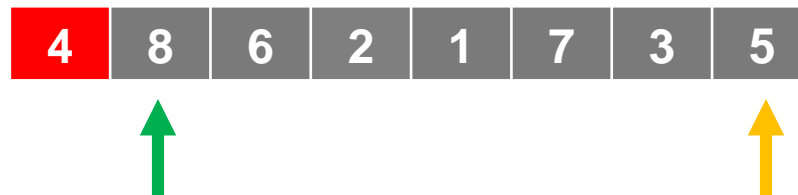
$L_bad = 2$, $R_bad = N$

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 \leq 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)

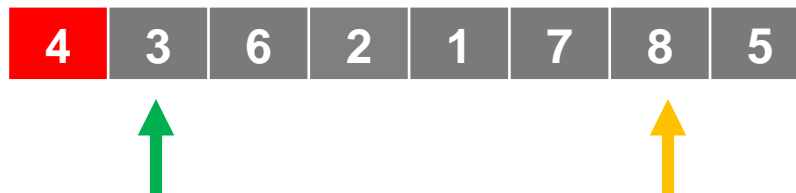
$L_bad = 2$, $R_bad = N$

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)

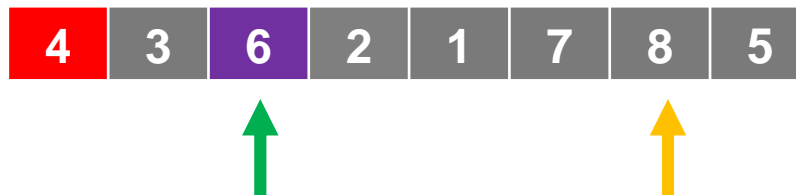
$L_bad = 2$, $R_bad = N$

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)

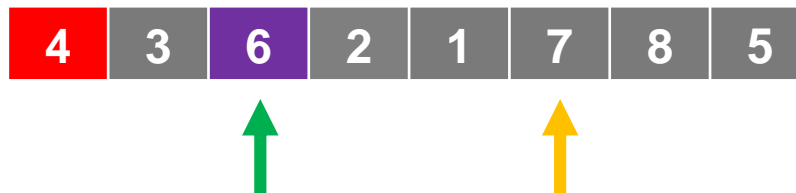
$L_bad = 2$, $R_bad = N$

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)

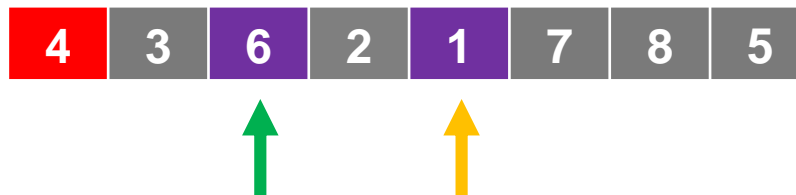
$L_bad = 2$, $R_bad = N$

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)

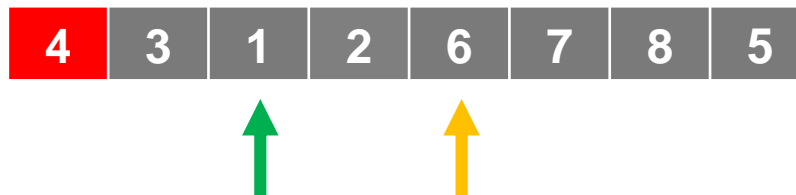
$L_bad = 2$, $R_bad = N$

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)

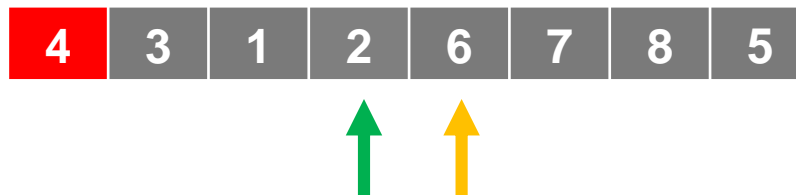
$L_bad = 2$, $R_bad = N$

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)

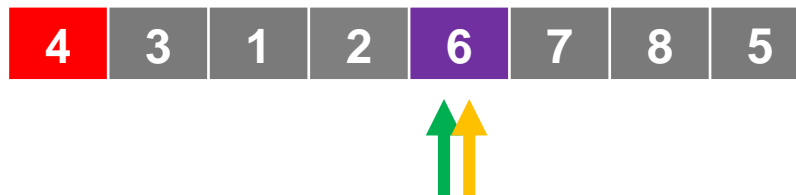
$L_bad = 2$, $R_bad = N$

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)

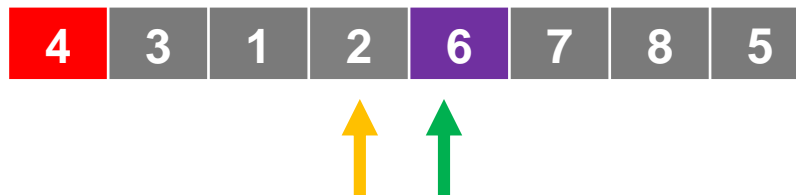
$L_bad = 2$, $R_bad = N$

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)

$L_bad = 2$, $R_bad = N$

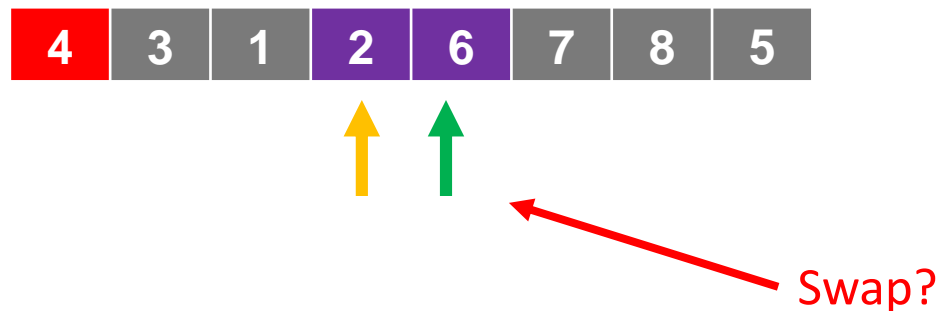
Repeat until L_bad and R_bad cross

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)

$L_bad = 2$, $R_bad = N$

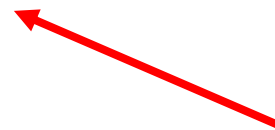
Repeat until L_bad and R_bad cross

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)

$L_bad = 2$, $R_bad = N$

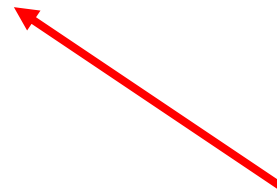
Repeat until L_bad and R_bad cross

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)

$L_bad = 2$, $R_bad = N$

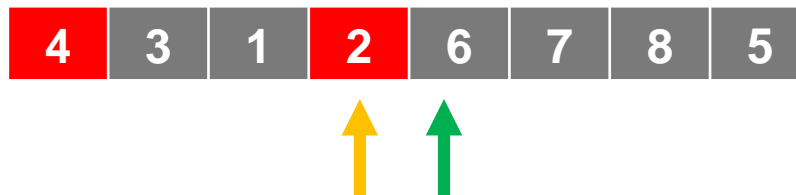
Repeat until L_bad and R_bad cross

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)

$L_bad = 2$, $R_bad = N$

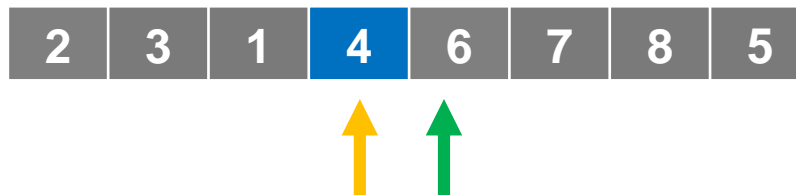
Repeat until L_bad and R_bad cross

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



Partitioning: In place (Hoare's)

Swap pivot to the front (position 1)

$L_bad = 2$, $R_bad = N$

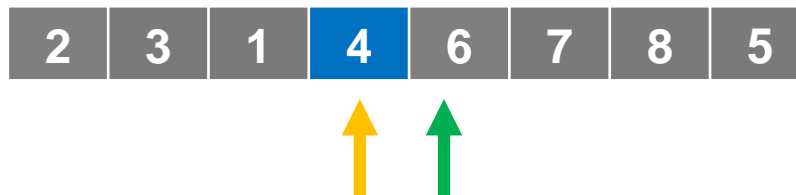
Repeat until L_bad and R_bad cross

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

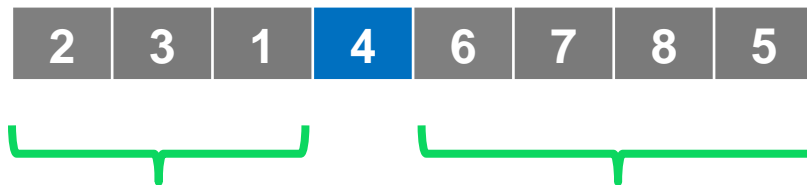
Repeat until L_bad and R_bad cross

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



Repeat LEFT and RIGHT

Questions?

Quicksort

In-place Partitioning (Hoare's)

- Invariant?

Quicksort

In-place Partitioning (Hoare's)

- Invariant?
 - L_bad?
 - R_bad?

- Invariant?
 - L_bad? Everything to left of 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 of 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 of 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?

- 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?

- **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?

- **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?

- **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 1st 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?

Quicksort

In-place Partitioning (Hoare's)

- 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 =)

- 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 later since no interaction for online =(

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

- This is the one you are familiar with
 - From FIT1008
- In place
- Swap each element multiple times
- ... 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

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

- 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...

Quicksort

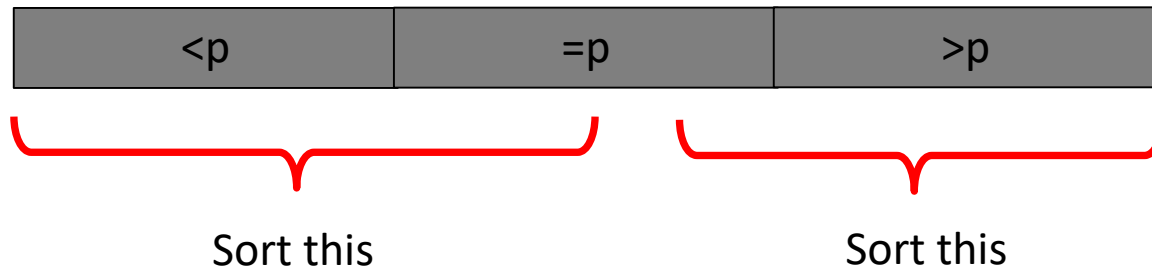
Partitioning...

- The stability issue however gives us an idea...

Quicksort

Partitioning...

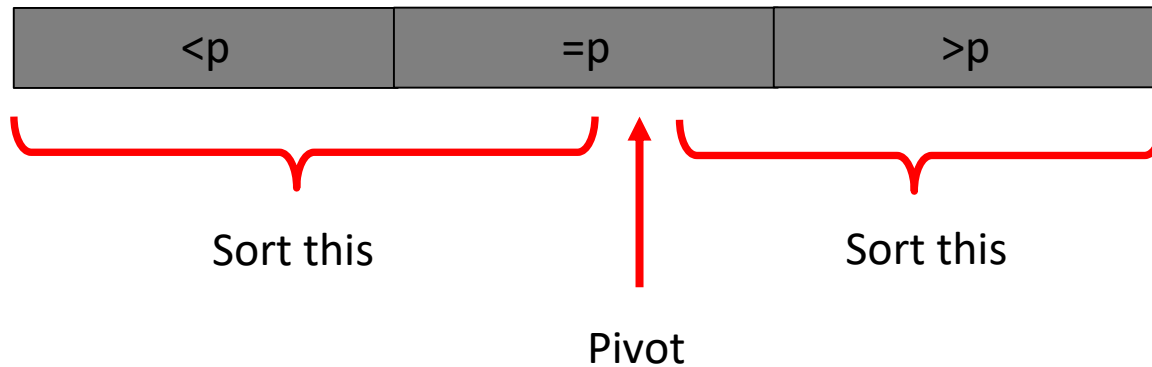
- The stability issue however gives us an idea...



Quicksort

Partitioning...

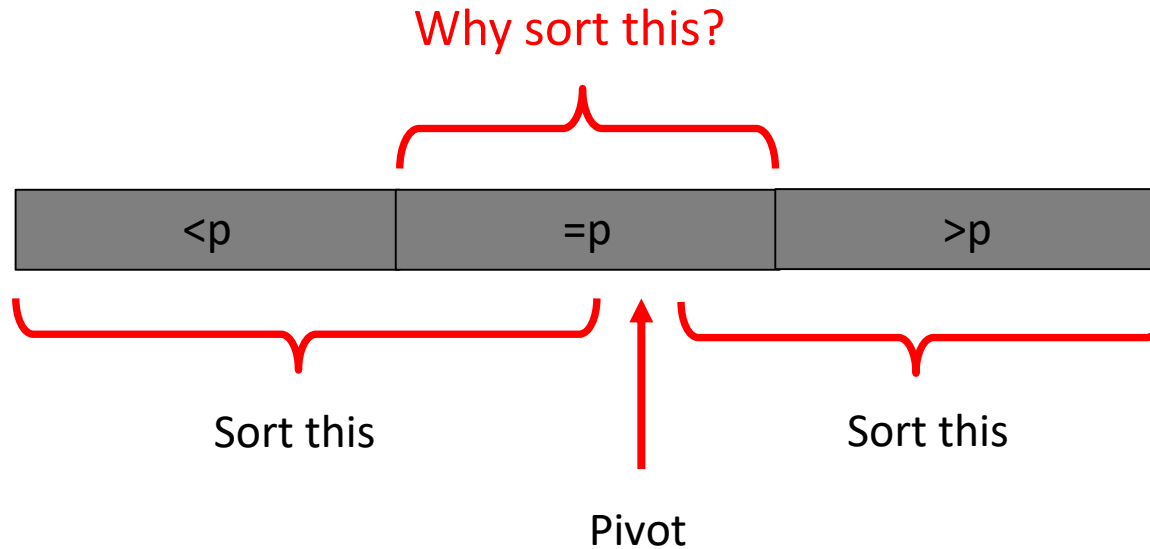
- The stability issue however gives us an idea...



Quicksort

Partitioning...

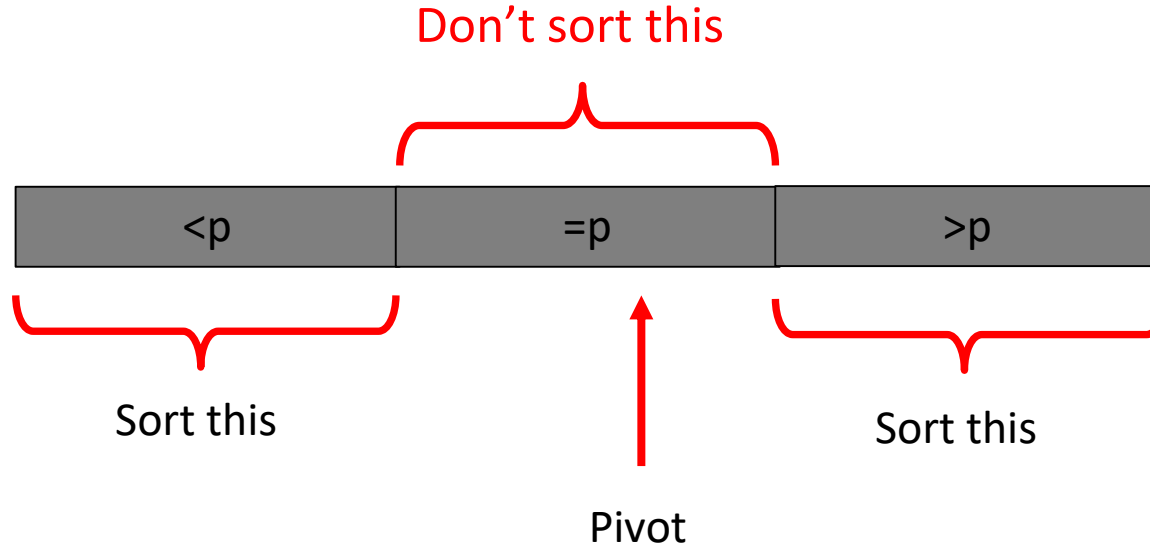
- The stability issue however gives us an idea...



Quicksort

Partitioning...

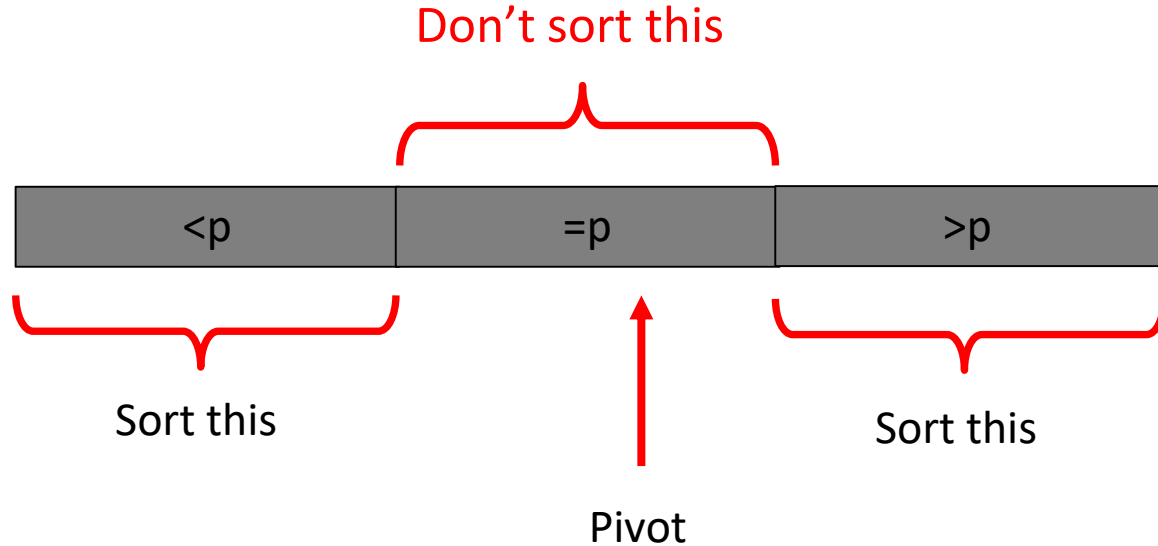
- The stability issue however gives us an idea...



Quicksort

Partitioning...

- The stability issue however gives us an idea...

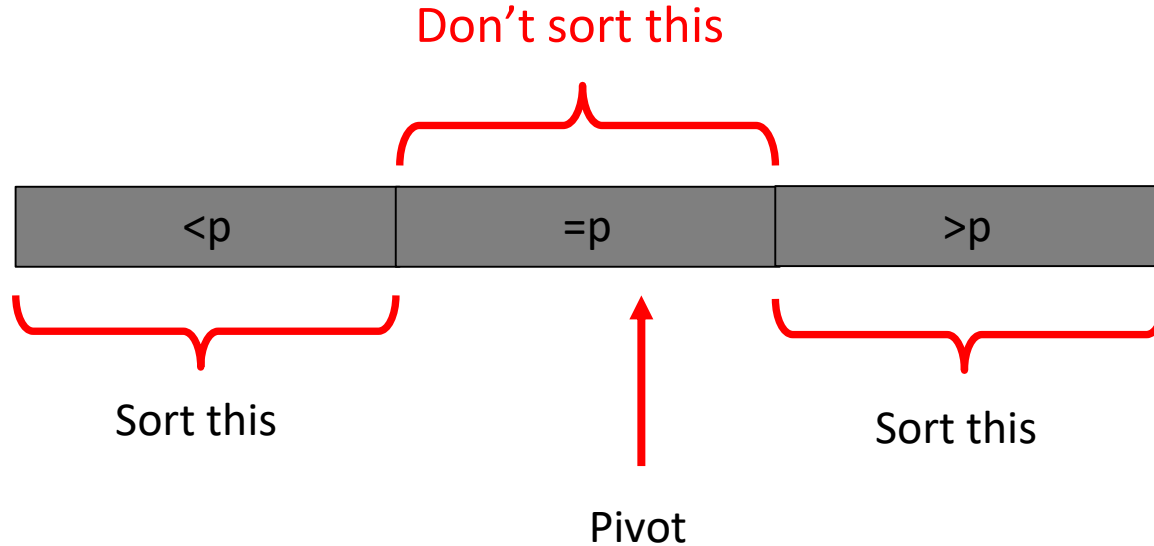


Quicksort

Partitioning...



- The stability issue however gives us an idea...



- This lead us to the Dutch national flag problem

Quicksort

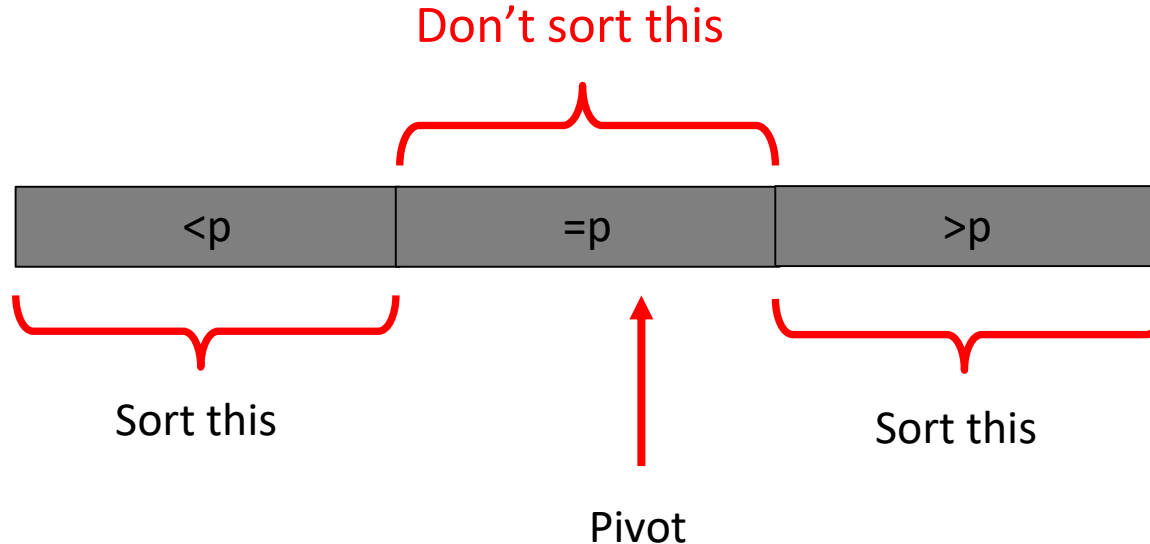
Partitioning...

< p

= p

> p

- The stability issue however gives us an idea...



- This lead us to the Dutch national flag problem

Quicksort

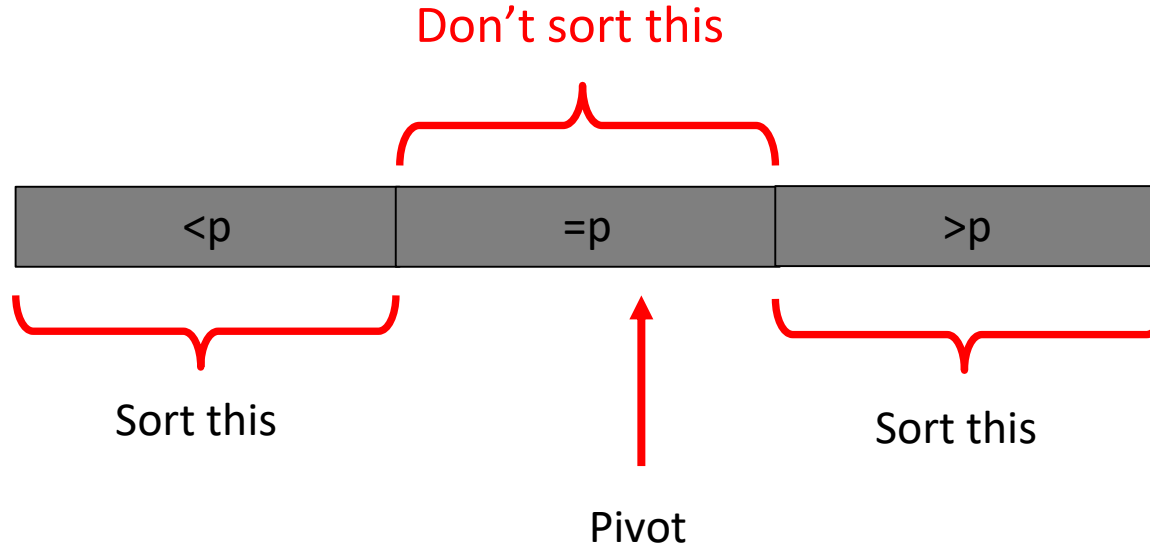
Partitioning...

< p

= p

> p

- The stability issue however gives us an idea...

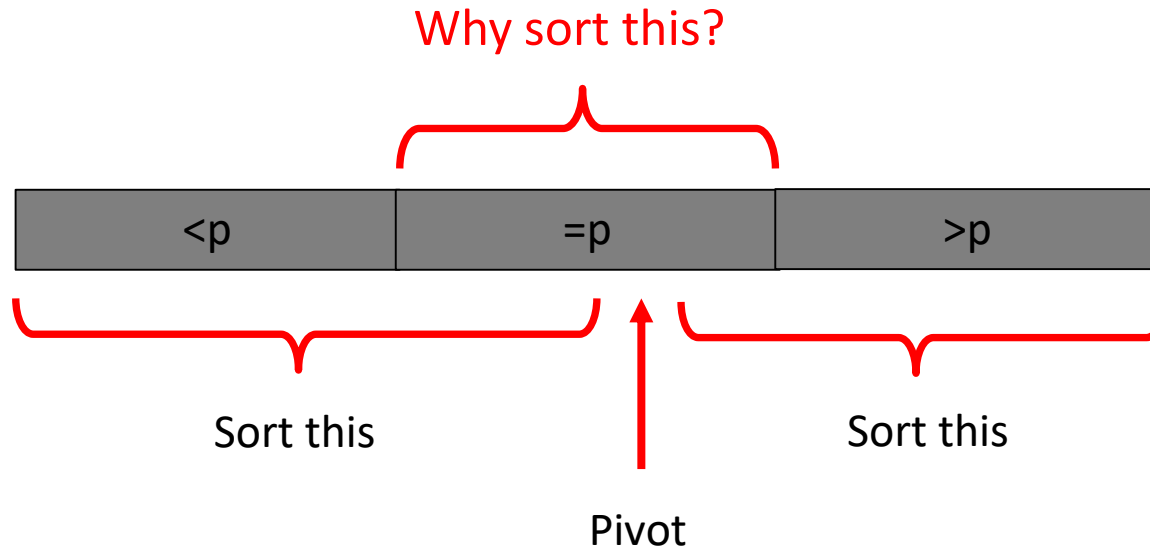


- This lead us to the Dutch national flag problem
 - Let us look at Nathan's illustration

Quicksort

Partitioning...

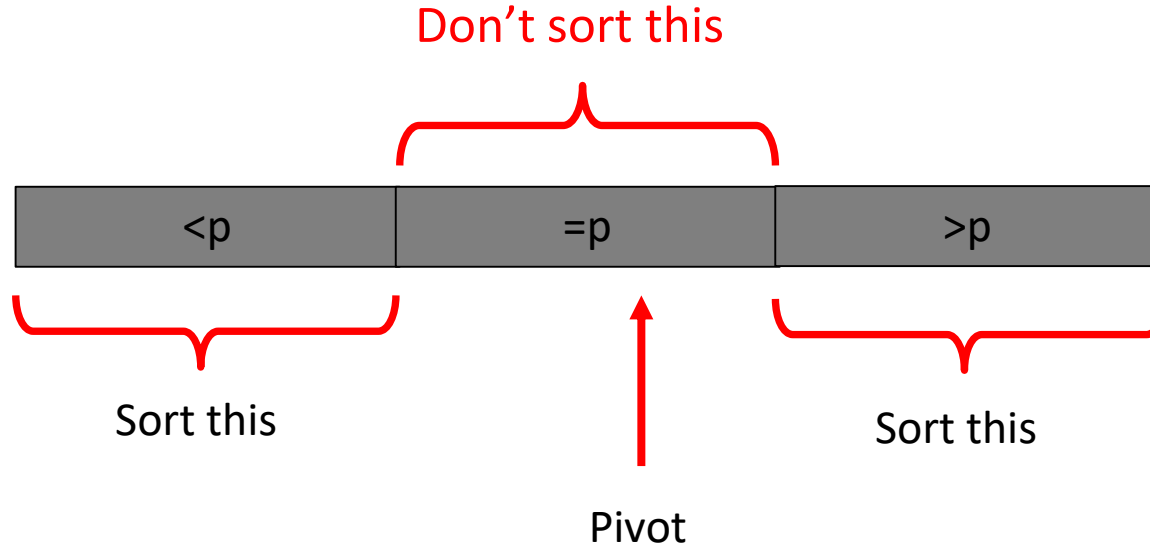
- The stability issue however gives us an idea...



Quicksort

Partitioning...

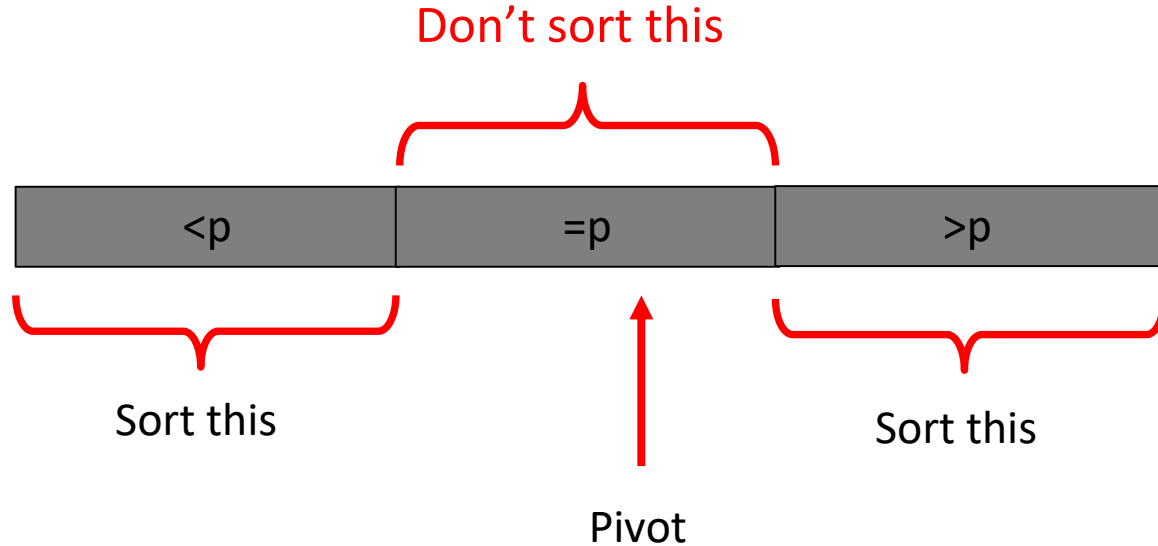
- The stability issue however gives us an idea...



Quicksort

Partitioning...

- The stability issue however gives us an idea...

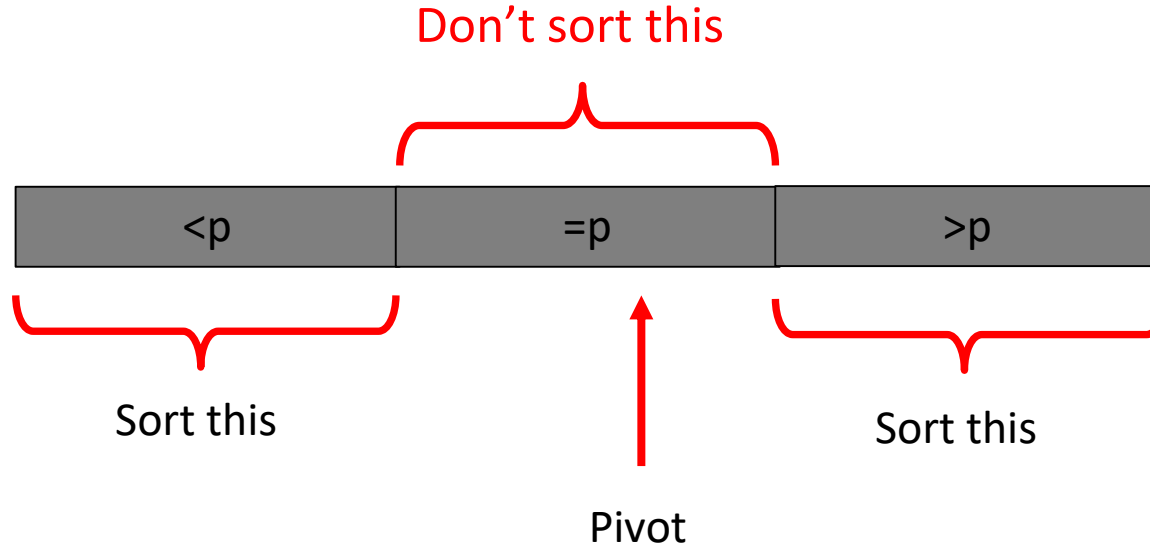


Quicksort

Partitioning...



- The stability issue however gives us an idea...



- This lead us to the Dutch national flag problem

Quicksort

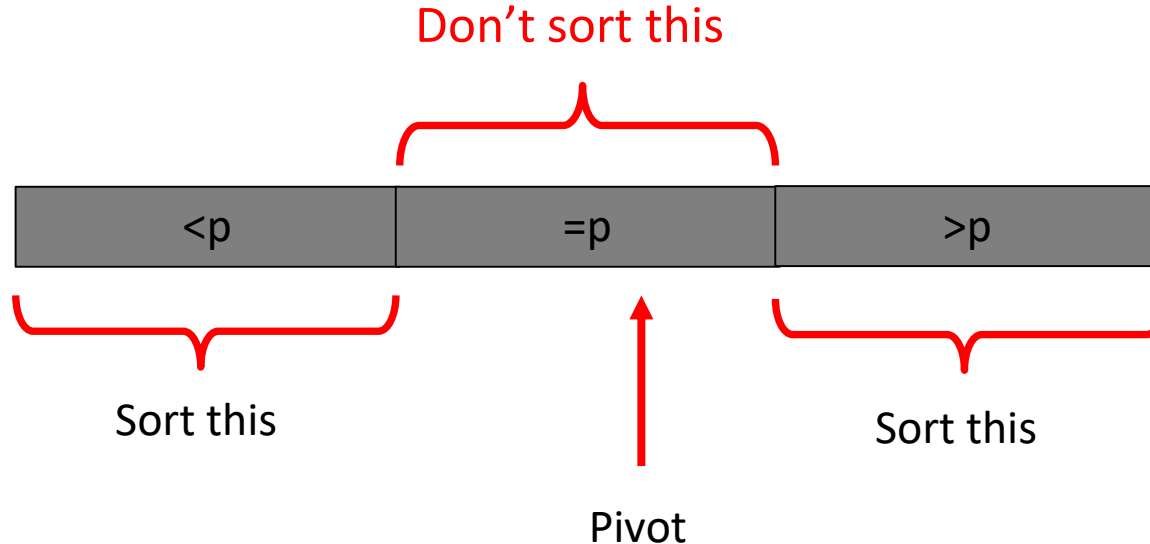
Partitioning...

< p

= p

> p

- The stability issue however gives us an idea...



- This lead us to the Dutch national flag problem

Quicksort

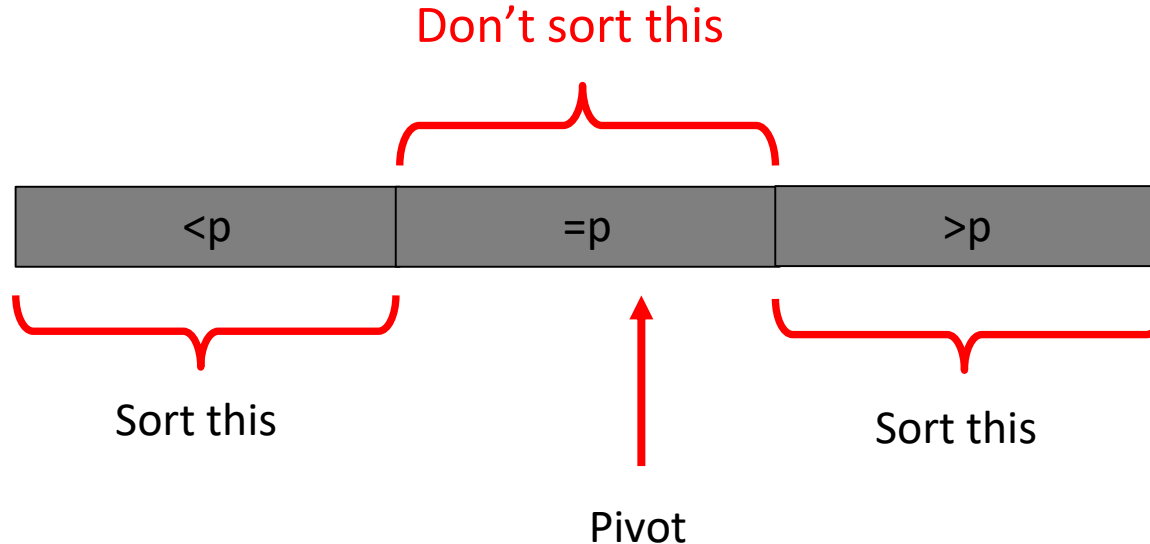
Partitioning...

< p

= p

> p

- The stability issue however gives us an idea...



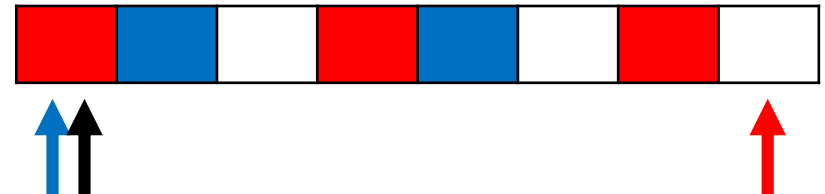
- This lead us to the Dutch national flag problem
 - Let us look at Nathan's illustration

Dutch National Flag 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

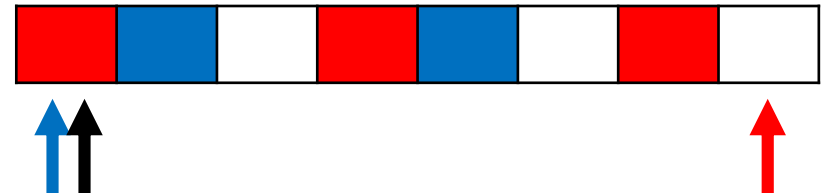
elif array[j] is red

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

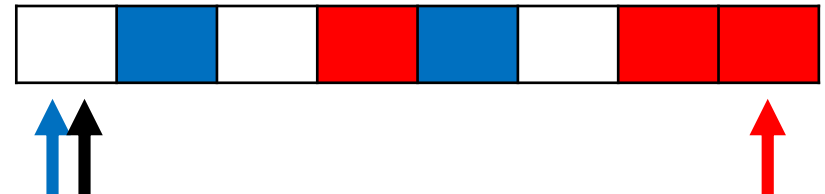
elif array[j] is red

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

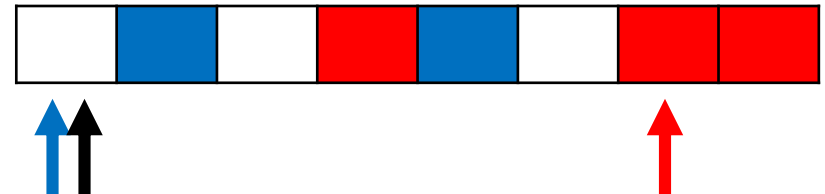
elif array[j] is red

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

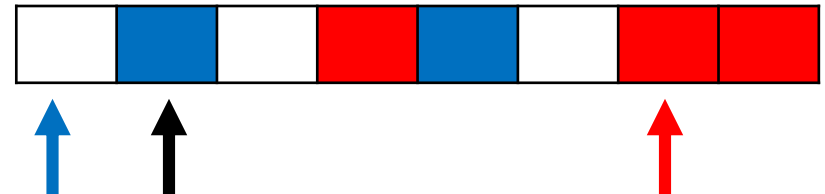
elif array[j] is red

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

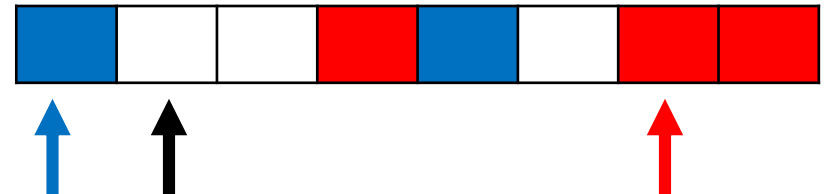
elif array[j] is red

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

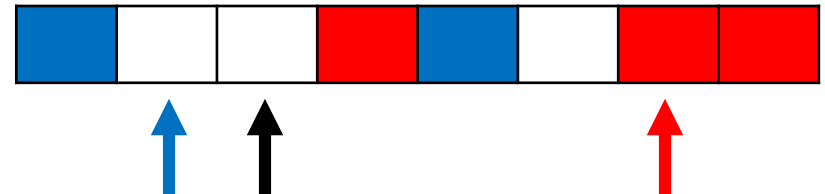
elif array[j] is red

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

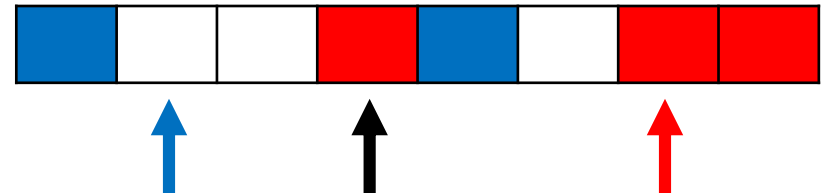
elif array[j] is red

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

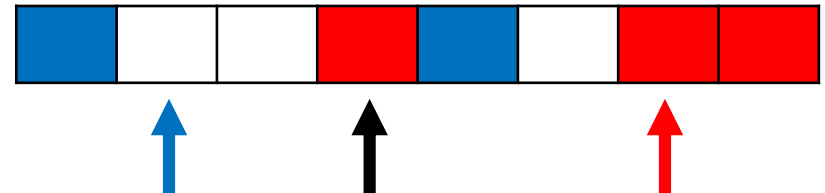
elif array[j] is red

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

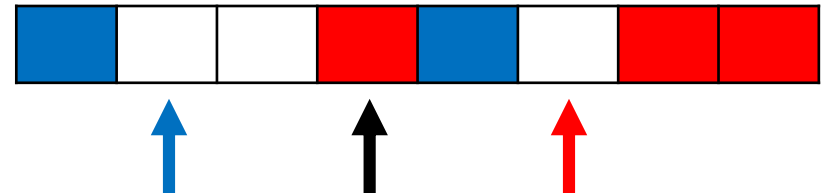
elif array[j] is red

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

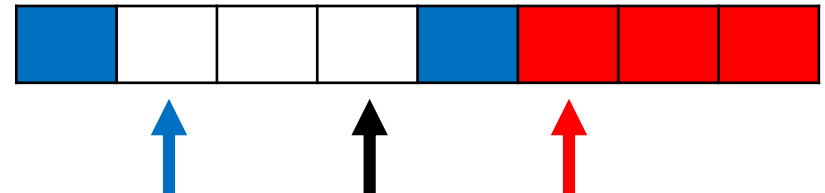
elif array[j] is red

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

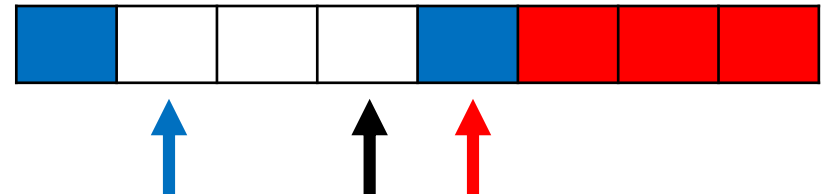
elif array[j] is red

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

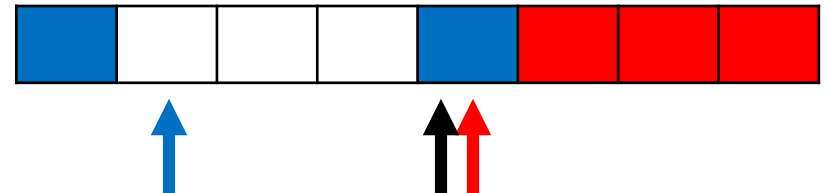
elif array[j] is red

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

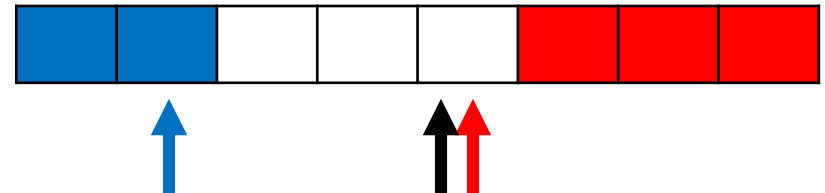
elif array[j] is red

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

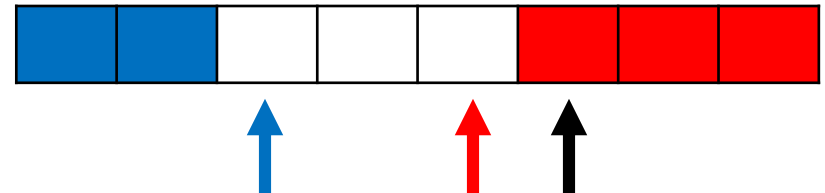
elif array[j] is red

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

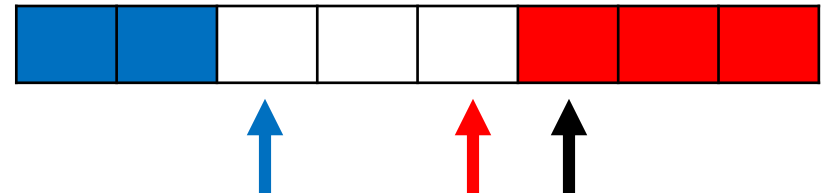
swap array[j], array[boundary2]

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

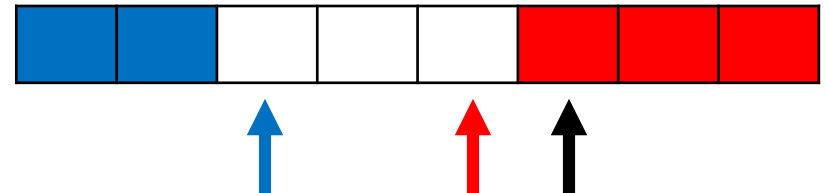
swap array[j], array[boundary2]

boundary2 -= 1

else

j += 1

Return boundary1, boundary2



Now quicksort the red and blue parts

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 $j > \text{boundary2}$

- 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
 - This will be empty when I exit loop at $j > \text{boundary2}$

- 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

Quicksort

Partitioning...

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

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

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

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?

Quicksort

Partitioning...

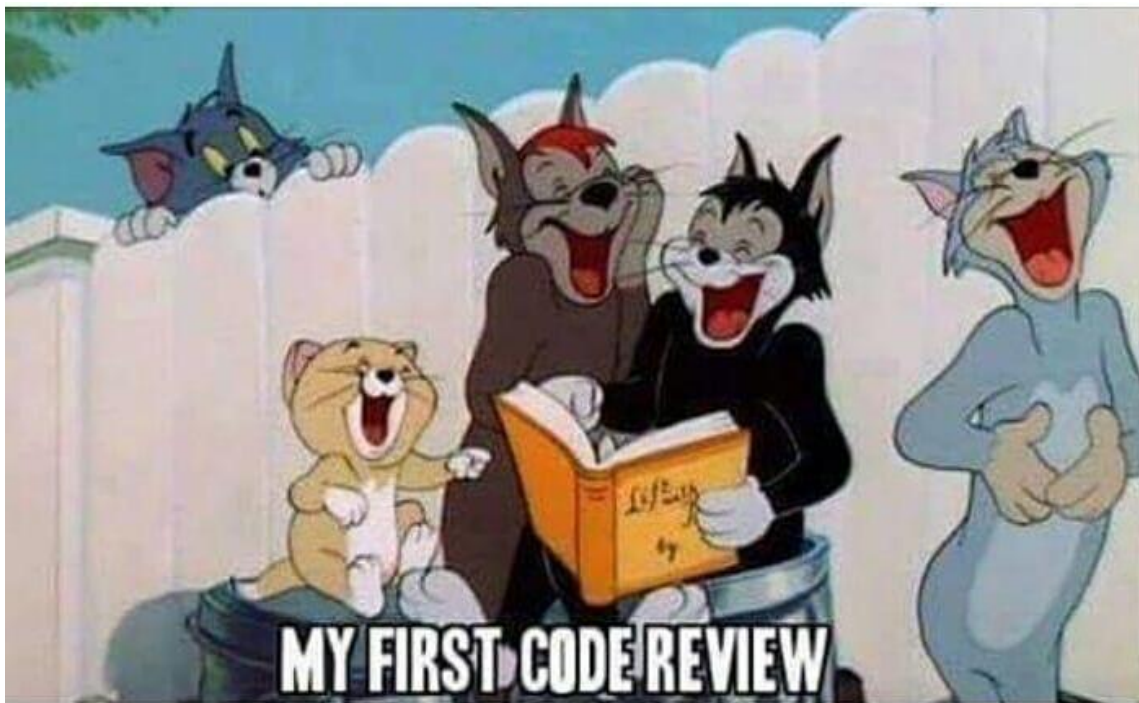
- Minimize swaps
 - We saw this with Hoare's
- Minimize work in recursive sort
 - We saw this with Dutch national flag
 - Left partition and right partition is smaller now
- Be in-place
 - Save memory
- Stable?
 - We discuss more in the tutorials...



Quicksort

Partitioning...

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



Questions?

Quicksort

Complexity Analysis

- Time complexity

Quicksort

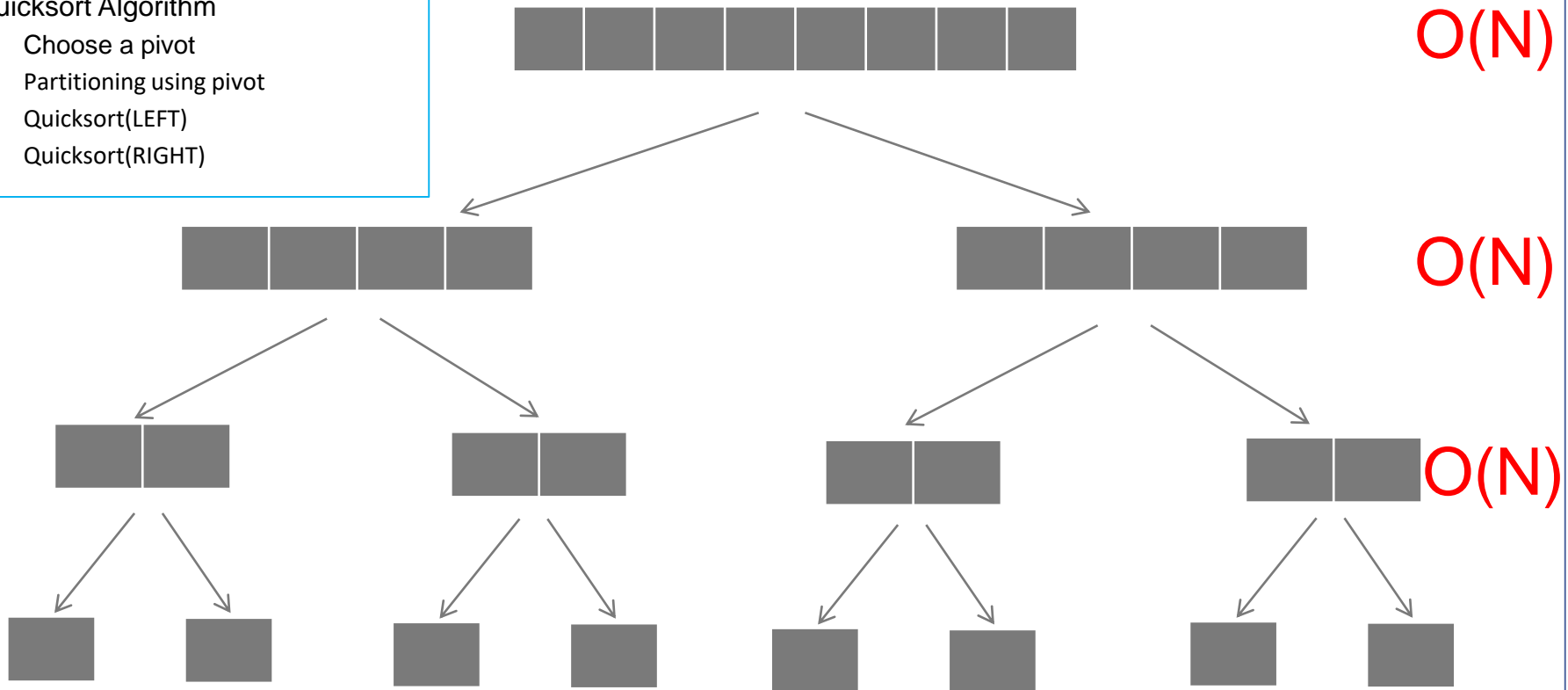
Complexity Analysis

- Time complexity
 - Best
 - Worst

Best-case time complexity

Quicksort Algorithm

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



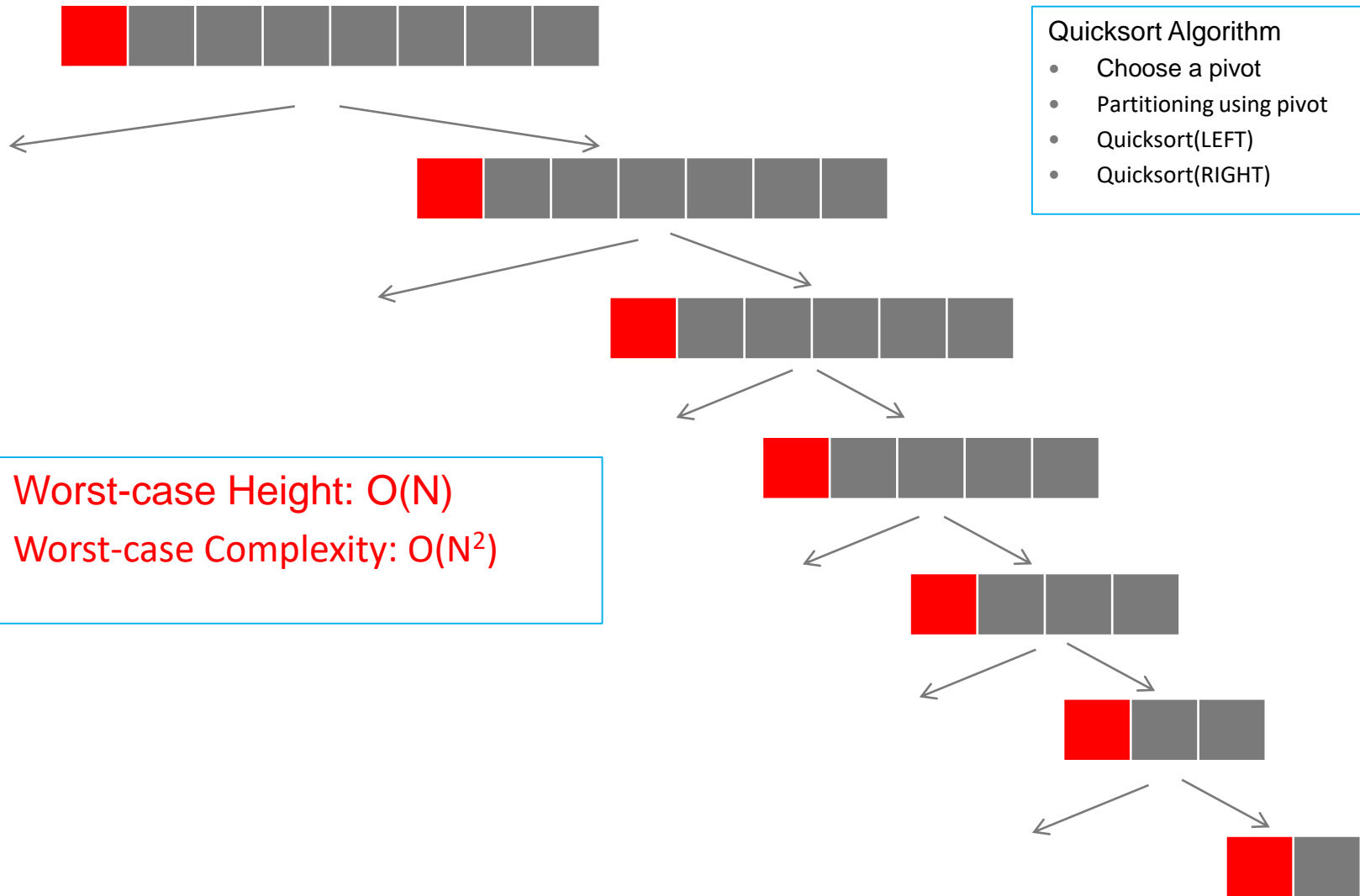
Best-case Height: $O(\log N)$

Best-case complexity: $O(N \log N)$

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

Recursion depth is at least $O(\log N)$

Worst-case Time Complexity



Quicksort

Complexity Analysis

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

Quicksort

Complexity Analysis

- 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:

$$T(1) = b$$

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

Solution:

$$O(N^2)$$



Questions?

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...

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?

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
 - $O(N^2)$
 - 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?

Questions?

Quicksort

Complexity Analysis

- Consider a list with N elements

Quicksort

Complexity Analysis

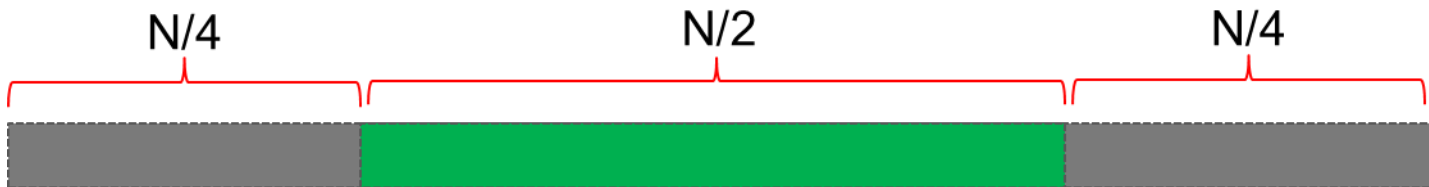
- Consider a list with N elements
 - And then we partition it



Quicksort

Complexity Analysis

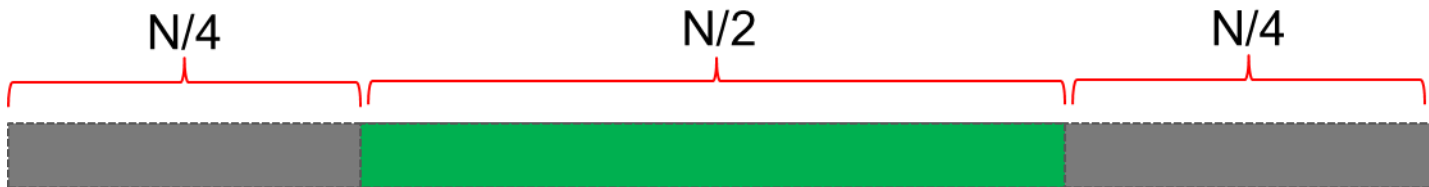
- Consider a list with N elements
 - And then we partition it



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

Quicksort

Complexity Analysis

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

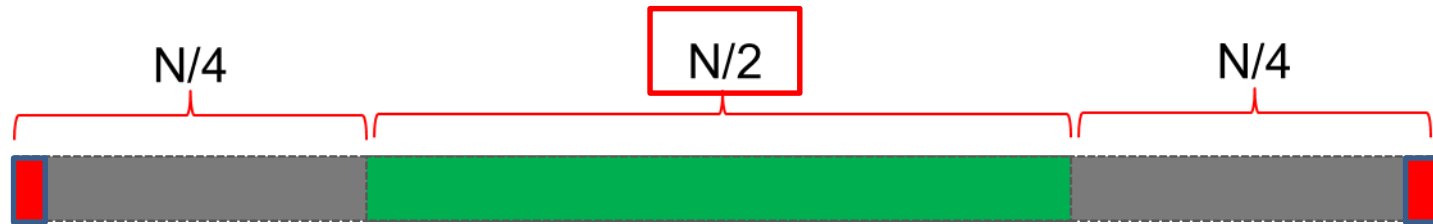


- What is the probability we would land on the green area?
 - So 50% probability
- Worst case in grey area?

Quicksort

Complexity Analysis

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

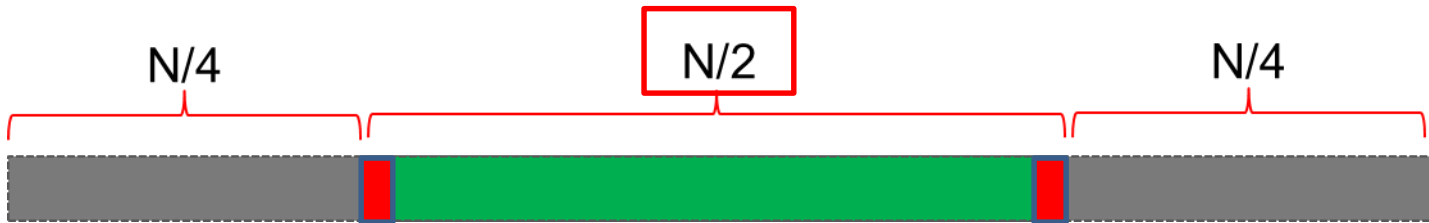


- What is the probability we would land on the green area?
 - So 50% probability
- Worst case in grey area? Pivot 1 or $N-1$

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 $3N/4$

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 ...

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%?

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%? $2h$

Quicksort

Complexity Analysis

- Consider a list with N elements

- And then we partition it



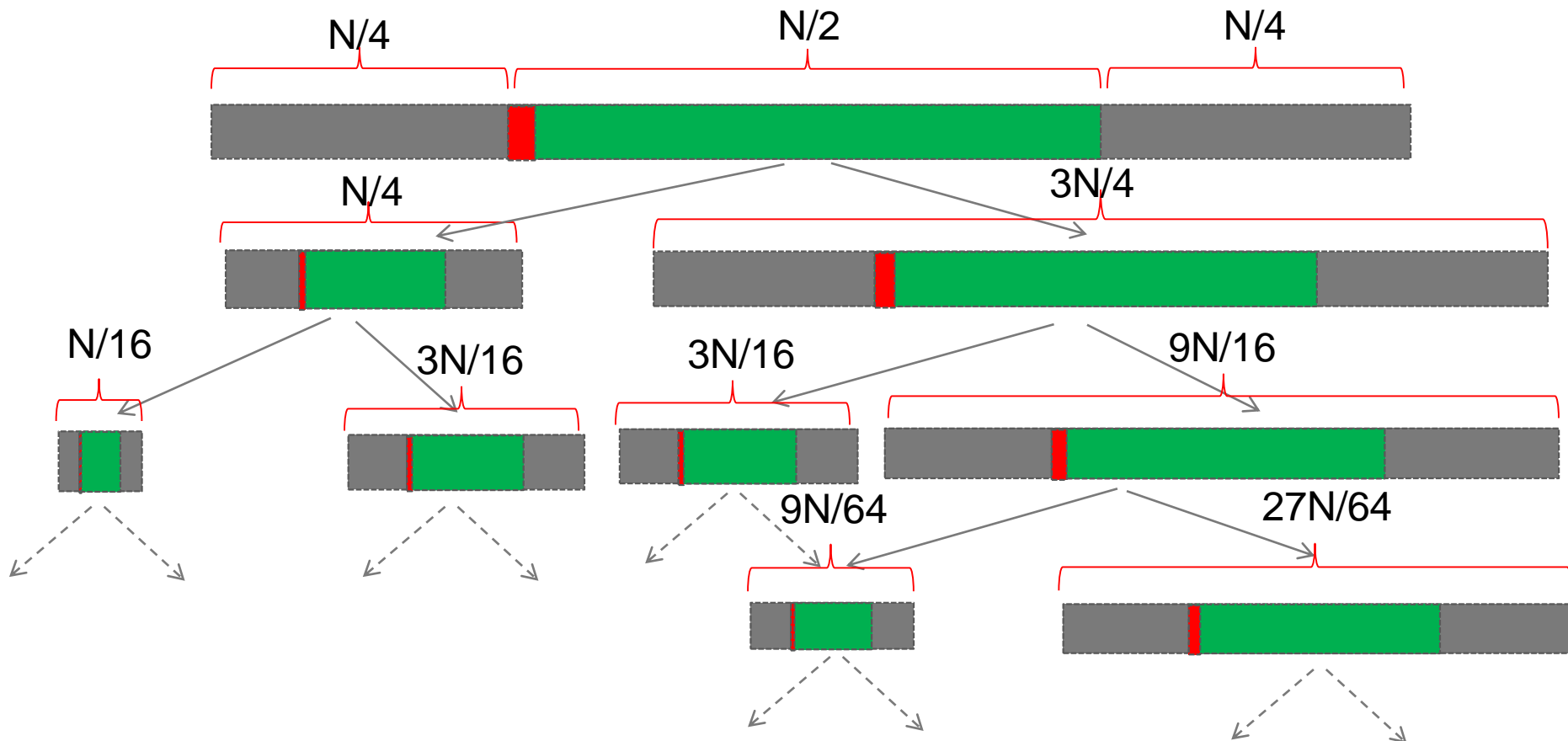
- What is the probability we would land on the green area?
 - So 50% probability
- Worst case in grey area? Pivot 1 or $N-1$
- Worst case in green area? Pivot at $N/4$ or $3N/4$
- If we always hit the green area, we will get a maximum recursion height of h ... So what is the upper bound for the height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

- So we calculate what is h

Quicksort

Complexity Analysis

- Let us just do the normal drawing



Quicksort

Complexity Analysis

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

Quicksort

Complexity Analysis

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

Quicksort

Complexity Analysis

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

Quicksort

Complexity Analysis

- Let us just do the normal drawing
 - From your recursive knowledge, $T(N) \rightarrow T(3N/4)$
 - 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
 - From your recursive knowledge, $T(N) \rightarrow T(3N/4)$
 - Reach base case when size is 1, thus base when $T(1)$
 - Thus... $N(3/4)^h = 1$
 - Which gives us $(3/4)^h N = 1 \rightarrow N = (4/3)^h \rightarrow h = \log_{4/3} N$

Quicksort

Complexity Analysis

- Let us just do the normal drawing
 - From your recursive knowledge, $T(N) \rightarrow T(3N/4)$
 - 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 $2h$

Quicksort

Complexity Analysis

- Let us just do the normal drawing
 - From your recursive knowledge, $T(N) \rightarrow T(3N/4)$
 - 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 $2h$

Quicksort

Complexity Analysis

- Let us just do the normal drawing
 - From your recursive knowledge, $T(N) \rightarrow T(3N/4)$
 - 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 $2 \log_{4/3} N$

Quicksort

Complexity Analysis

- Let us just do the normal drawing
 - From your recursive knowledge, $T(N) \rightarrow T(3N/4)$
 - 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 $O(\log N)$

Quicksort

Complexity Analysis

- Let us just do the normal drawing
 - From your recursive knowledge, $T(N) \rightarrow T(3N/4)$
 - 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 $O(\log N)$
 - Each level have a partition cost of $O(N)$

Quicksort

Complexity Analysis

- Let us just do the normal drawing
 - From your recursive knowledge, $T(N) \rightarrow T(3N/4)$
 - 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 $O(\log N)$
 - Each level have a partition cost of $O(N)$
 - Total average cost is $O(N \log N)$

Quicksort

Complexity Analysis

- Let us just do the normal drawing
 - From your recursive knowledge, $T(N) \rightarrow T(3N/4)$
 - 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 $O(\log N)$
 - Each level have a partition cost of $O(N)$
 - Total average cost is $O(N \log N)$

Quicksort

Complexity Analysis

- Let us just do the normal drawing
 - From your recursive knowledge, $T(N) \rightarrow T(3N/4)$
 - 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 $O(\log N)$
 - 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: $\log_a N = \frac{\log_b N}{\log_b a}$

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

Quicksort

Complexity Analysis

- 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**

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)
...	...
<div> <div>Quicksort Algorithm</div> <ul style="list-style-type: none"> Choose a pivot Partitioning using pivot Quicksort(LEFT) Quicksort(RIGHT) </div>	
	T(N-3)
	T(N-2)
	T(N-1)

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

Average-case complexity using recurrence (NOT EXAMINABLE)

Recurrence relation:

$$T(1) = b$$

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

Multiplying N on both sides

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

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

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

(A) - (B)

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

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

Simplify

Divide both sides by N

Average-case complexity using recurrence (NOT EXAMINABLE)

Recurrence relation:

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

Let's solve it:

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

Replace $T(N-1)$ in (A)

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

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

Replace $T(N-2)$ in (B)

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

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

See the pattern for k ?

Average-case complexity using recurrence (NOT EXAMINABLE)

Recurrence relation:

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

Let's solve it:

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

$$N-k=1 \rightarrow k=N-1$$

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

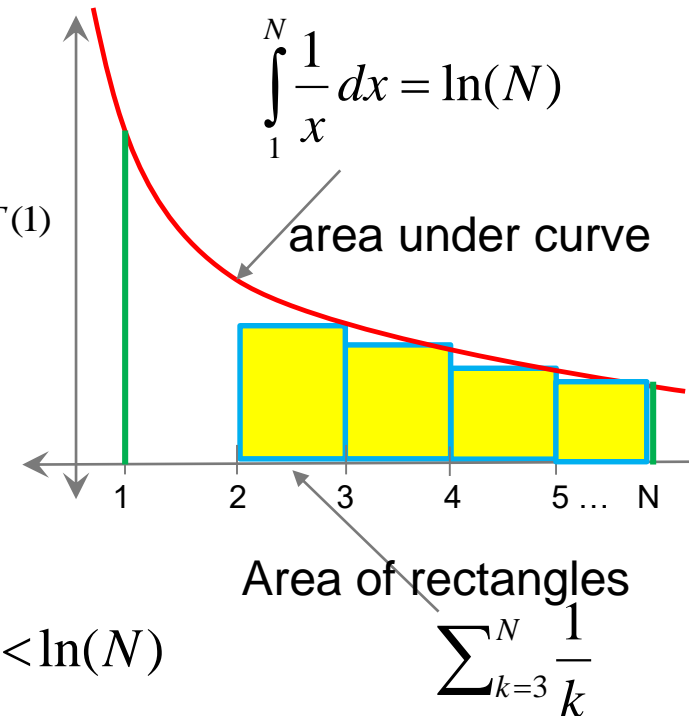
Simplify

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

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

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

$$T(N) = O(N \log N)$$



Questions?

Quick Sort

Summary

- Good sorting algorithm
 - Using divide and conquer

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

- 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

- 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

- 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

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
 - 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