CS2040 Tutorial 2

Week 4, AY 19/20 Sem 2

Wang Zhi Jian
wzhijian@u.nus.edu

Welcome to CS2040 Tutorials!

Hello!

Wang Zhi Jian wzhijian@u.nus.edu

Year 2, Computer Science / Mathematics

CS2040	Student	AY 18/19 (Sem 1)
	TA	AY 18/19 (Sem 2, Sem 4), AY 19/20 (Sem 1, Sem 2)
CS2040S	TA	AY 19/20 (Sem 1, Sem 2)

Interests:

Competitive Programming | Manga and Anime | Rhythm Games

Tutorials and Labs

Tutorial

Discuss concepts covered during lecture, and go through the problems in the weekly tutorial sheet.

Do try the problems in the tutorial sheets before the tutorial!

Lab

Work on a lab assignment (Kattis).

Need help with CS2040?

Post in the CS2040 Facebook Group

Answers to your questions may benefit your peers too.

Email Me

If you are shy :) I'll respond as soon as I check my mail.

Ask me for Consultation

Drop me text on telegram (faster) or send an email if you'd like to meet.

Need help with CS2040?

Telegram Group

Invitation link has been sent via email.

You are encouraged to join!

For discussions, dissemination of information, etc.

Feel free to ask for help in the group too.

Tutorial Attendance/Participation (3%)

Tutorial Attendance

Attendance will be taken for all tutorials.

If you cannot attend a tutorial or miss a tutorial, do let me know, and in advance if possible.

Tutorial Participation

You are encouraged to ask questions, share your thoughts and present your work.

Extra / Helpful Content

Slides with a light blue background contain extra / helpful content such as

- additional exercises
- additional titbits of knowledge to supplement content covered during the lecture
- glimpses into higher level algorithm modules.

I usually won’t talk about them during the tutorials, but you’re encouraged to take a look. Feel free to discuss the contents with me :)

Materials

All materials (tutorial slides, etc) will be uploaded to https://tinyurl.com/cs2040-materials. In case you lose them in your email.

Will be uploading extra problemsets nearer to the midterms and finals as well. Do look out for them!

Hints for Tutorials

I will be releasing a hint sheet after the release of each tutorial question sheet on LumiNUS.

These hints point you in the right direction to solve the problems, and contain sub questions to help you think about the tutorial problems.

The hints are to encourage you to attempt as much of the tutorial as you can, including the relatively more difficult questions.

Hints for Tutorials

My Suggestion

- Spend some time thinking about the problem first without the hints. Try to generate some of your own ideas.
- After a while, if you are stuck or if you’re not sure of your idea, take a look at the hints to get you going / see if you’re heading in the right direction.

Hints for Tutorials

My Suggestion

- Spend some time thinking about the problem first without the hints. Try to generate some of your own ideas.
- After a while, if you are stuck or if you're not sure of your idea, take a look at the hints to get you going / see if you're heading in the right direction.

Note that there is **no** hint sheet during midterms/finals, so look at the hints only after you have done some thinking of your own, instead of simply relying on the hint sheet for ideas.

How to do well in CS2040?

Understand Material Well

Make sure you know **exactly how and why** everything works. There are no magic algorithms in CS2040.

Attempt Tutorials

Tutorial questions are hand-picked by Enzio and I. We believe they expose you to a range of techniques that are applicable to a wide range of problems.

Do Extra Practice Questions

Exposure to more problems helps with problem solving. Ask me for more questions if you need.

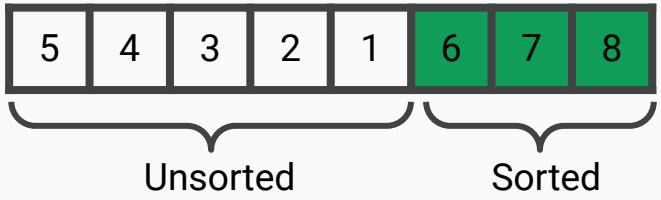
Two Basic Problems in Computer Science

Searching

Sorting

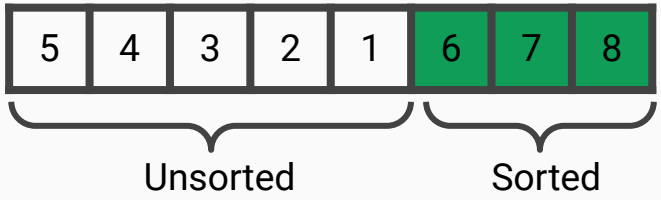
Bubble Sort

Bubble Sort

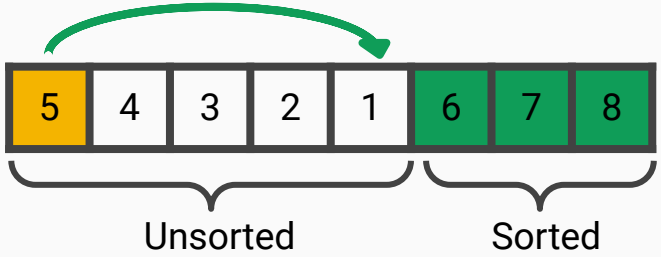


Maintains two regions:
sorted and **unsorted**
Elements in sorted region are in
correct final sorted positions

Bubble Sort

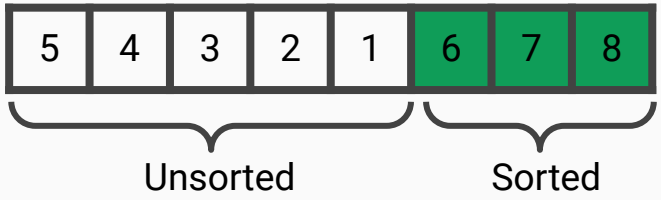


Maintains two regions:
sorted and **unsorted**
Elements in sorted region are in
correct final sorted positions

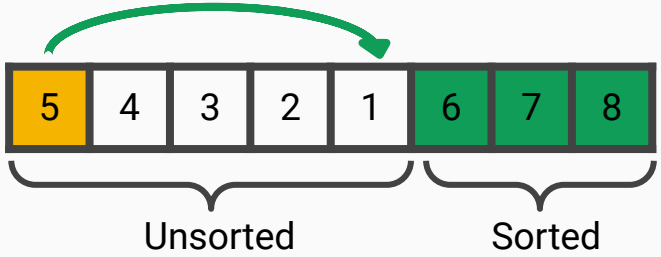


Each iteration of bubble sort:
biggest element in unsorted region is swapped along the array to correct final sorted position

Bubble Sort

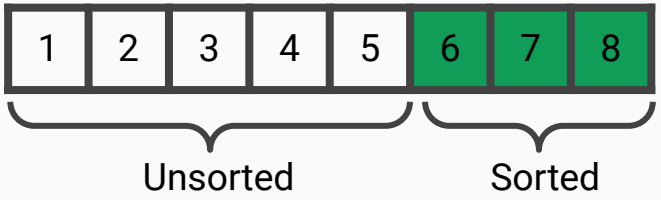


Maintains two regions:
sorted and **unsorted**
Elements in sorted region are in
correct final sorted positions



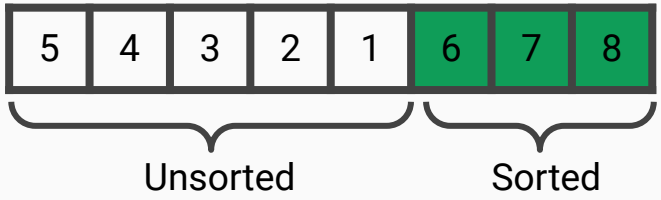
Each iteration of bubble sort:
biggest element in unsorted region is swapped along the array to correct final sorted position

has_swaps = false

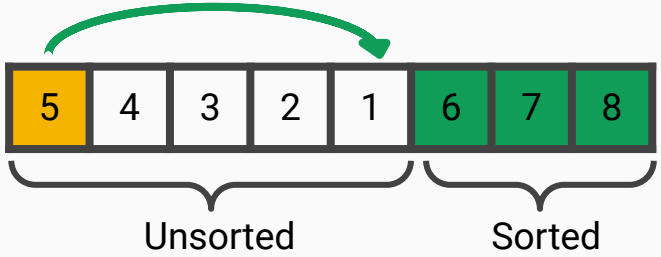


If no swaps made during an iteration, can terminate bubble sort early
Known as **bubble sort with early termination**

Bubble Sort



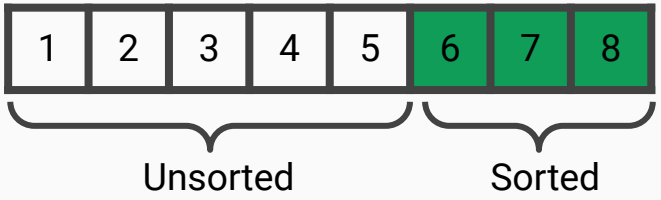
Maintains two regions:
sorted and **unsorted**
Elements in sorted region are in
correct final sorted positions



Each iteration of bubble sort:
biggest element in unsorted region is swapped along the array to correct final sorted position

Running Time: O(n^2)

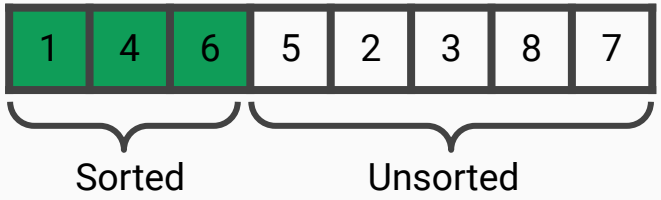
has_swaps = false



If no swaps made during an iteration, can terminate bubble sort early
Known as **bubble sort with early termination**

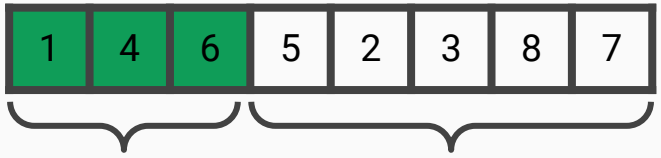
Insertion Sort

Insertion Sort

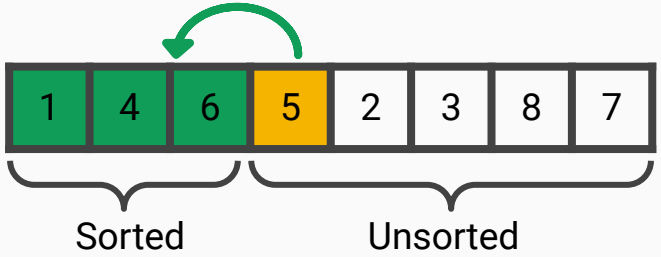


Maintains two regions:
sorted and **unsorted**
Elements in sorted region are **may not be in correct final sorted positions**

Insertion Sort

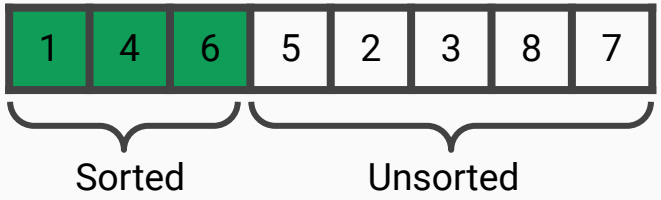


Maintains two regions:
sorted and **unsorted**
Elements in sorted region are **may not be in correct final sorted positions**

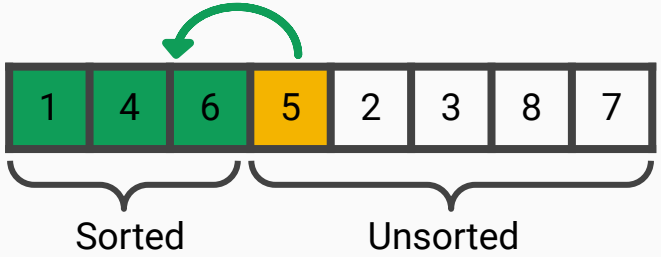


Each iteration of insertion sort:
first element in unsorted region is swapped along the array and inserted in correct position in sorted region

Insertion Sort

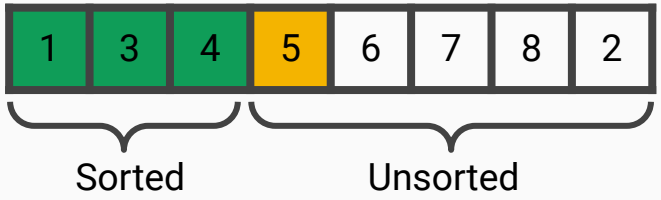


Maintains two regions:
sorted and **unsorted**
Elements in sorted region are **may not be in correct final sorted positions**



Each iteration of insertion sort:
first element in unsorted region is swapped along the array and inserted in correct position in sorted region

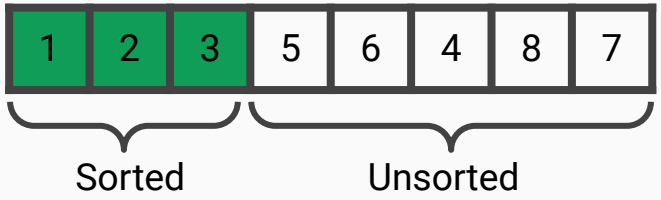
Running Time: O(n^2)
On (nearly) sorted array: O(n)



Good for **almost sorted arrays**
If first element in unsorted region is already bigger than all elements in sorted region, **no insertion is needed**

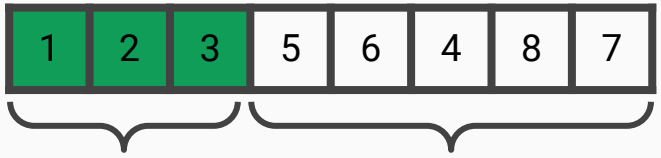
Selection Sort

Selection Sort

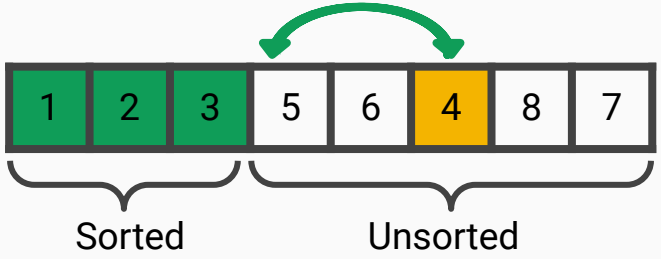


Maintains two regions:
sorted and **unsorted**
Elements in sorted region are in
correct final sorted positions

Selection Sort

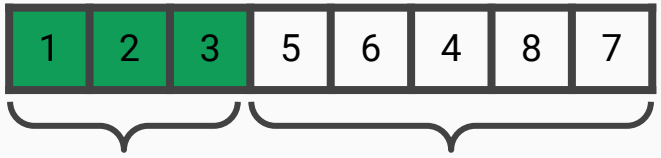


Maintains two regions:
sorted and **unsorted**
Elements in sorted region are in
correct final sorted positions

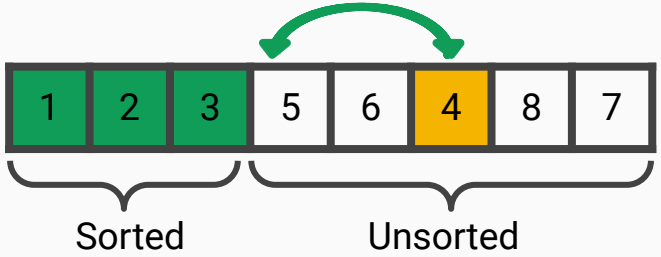


Each iteration of selection sort:
smallest element in unsorted region is swapped with first element in unsorted region to its **correct final sorted position**

Selection Sort



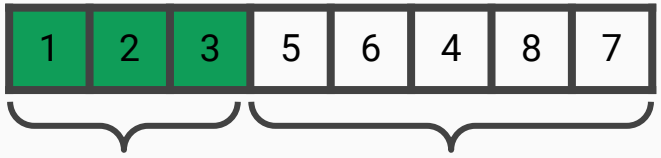
Maintains two regions:
sorted and **unsorted**
Elements in sorted region are in
correct final sorted positions



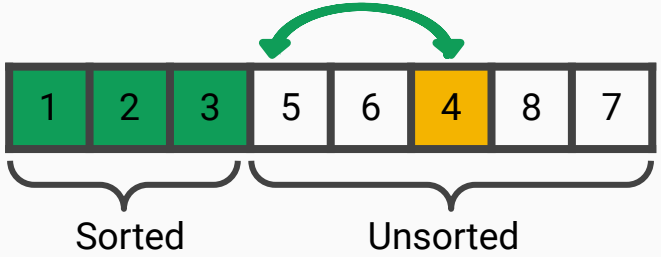
Each iteration of selection sort:
smallest element in unsorted region is swapped with first element in unsorted region to its **correct final sorted position**

Regardless of the initial order of the elements in the array: **will always perform the same number of operations**
Must scan through entire unsorted region to find minimum

Selection Sort



Maintains two regions:
sorted and **unsorted**
Elements in sorted region are in
correct final sorted positions

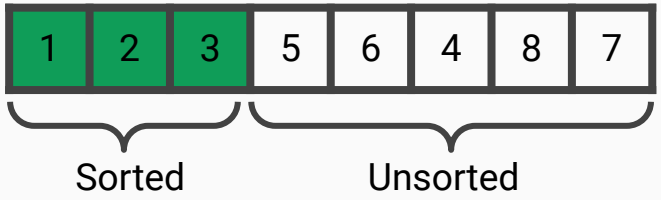


Each iteration of selection sort:
smallest element in unsorted region is swapped with first element in unsorted region to its **correct final sorted position**

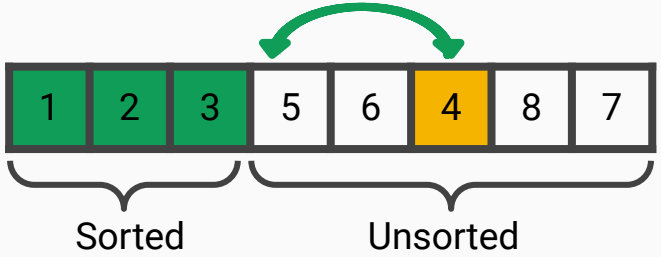
Regardless of the initial order of the elements in the array: **will always perform the same number of operations**
Must scan through entire unsorted region to find minimum

Minimizes swaps, therefore:
Good algorithm if **swaps are expensive**
1 swap / element compared to other algorithms

Selection Sort



Maintains two regions:
sorted and **unsorted**
Elements in sorted region are in
correct final sorted positions



Each iteration of selection sort:
smallest element in unsorted region is swapped with first element in unsorted region to its **correct final sorted position**

Running Time: O(n^2)

Regardless of the initial order of the elements in the array: **will always perform the same number of operations**
Must scan through entire unsorted region to find minimum

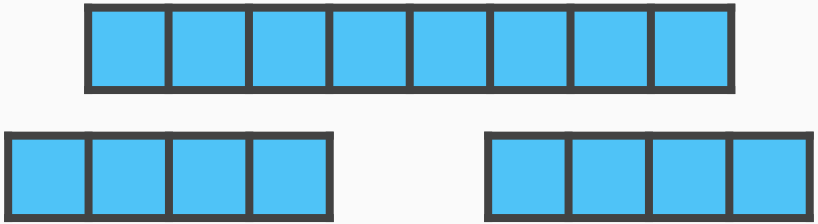
Minimizes swaps, therefore:
Good algorithm if **swaps are expensive**
1 swap / element compared to other algorithms

Merge Sort

Merge Sort

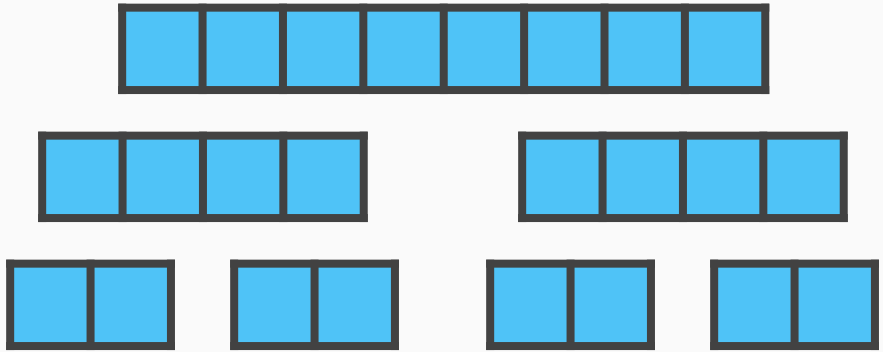
Divides arrays **in half recursively**

Merge Sort



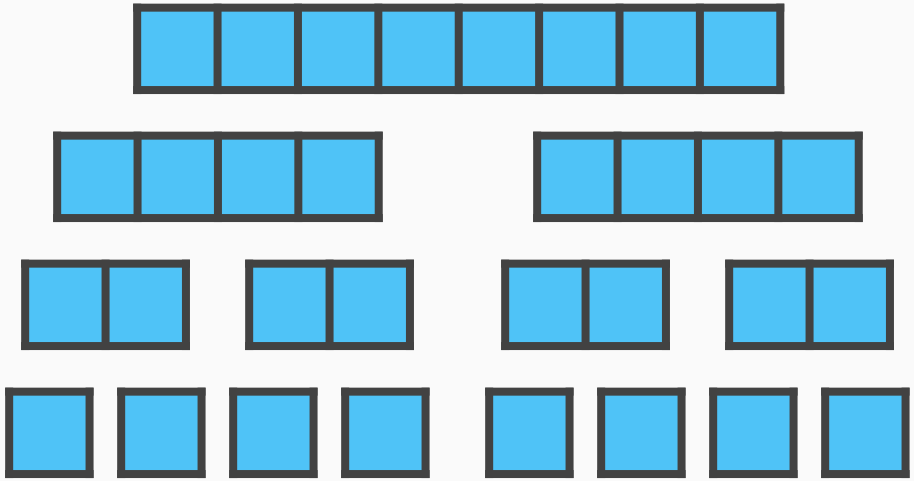
Divides arrays **in half recursively**

Merge Sort



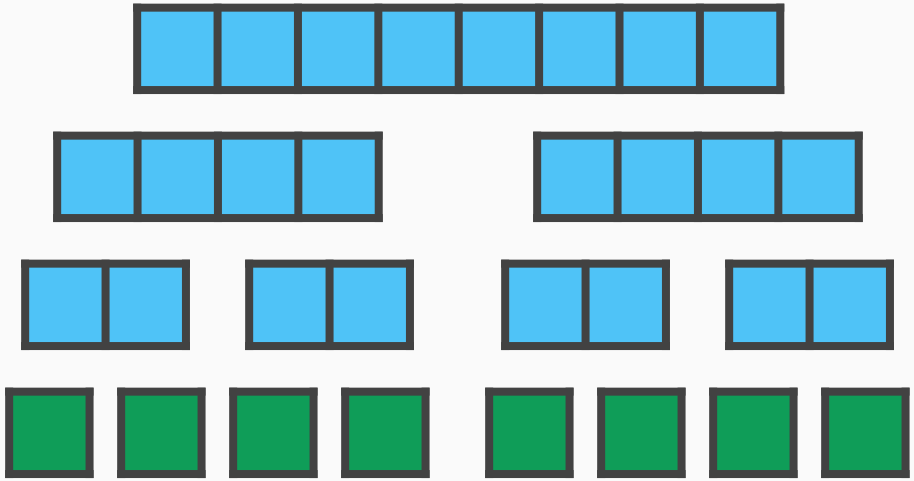
Divides arrays **in half recursively**

Merge Sort



Divides arrays **in half recursively**

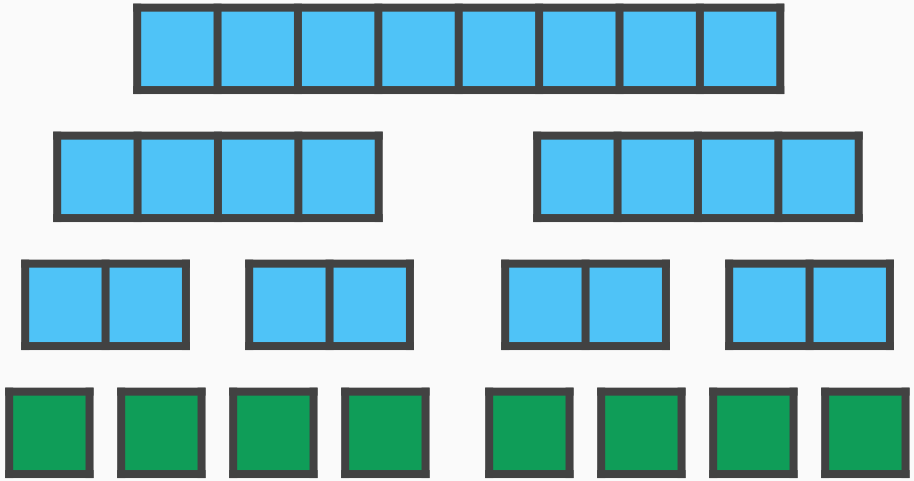
Merge Sort



Divides arrays **in half recursively**

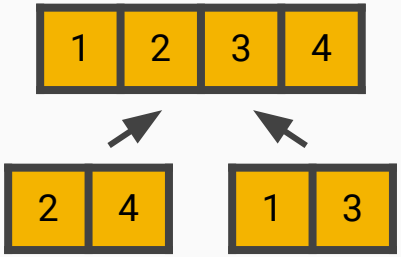
Base case: Arrays of size 1 are already sorted

Merge Sort



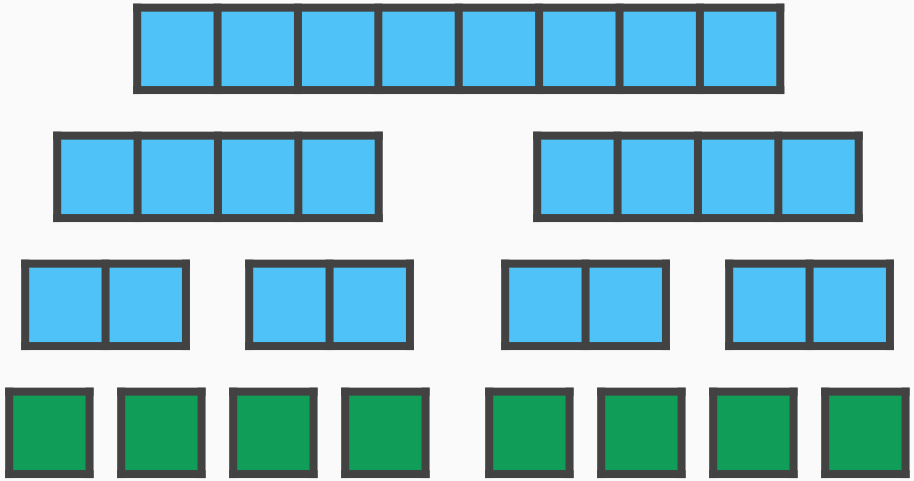
Divides arrays **in half recursively**

Base case: Arrays of size 1 are already sorted



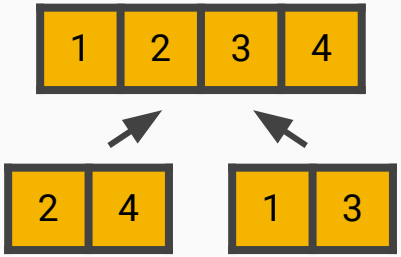
Merge subroutine: Given two sorted arrays, merge them into one sorted array in O(*n*) time

Merge Sort



Divides arrays **in half recursively**

Base case: Arrays of size 1 are already sorted



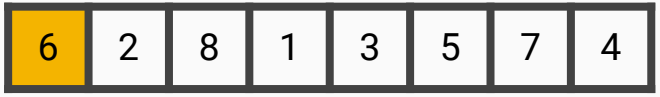
Merge subroutine: Given two sorted arrays, merge them into one sorted array in O(*n*) time

Running Time: O(*n* log *n*)

Quicksort

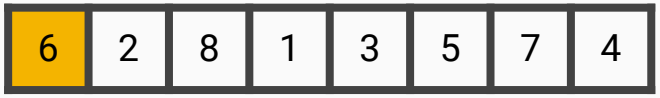
Quicksort

Quicksort

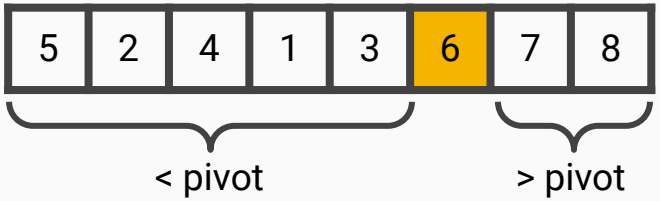


Pick pivot

Quicksort

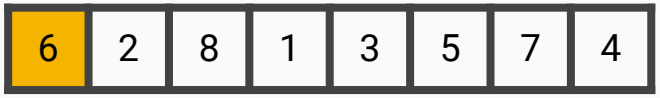


Pick pivot

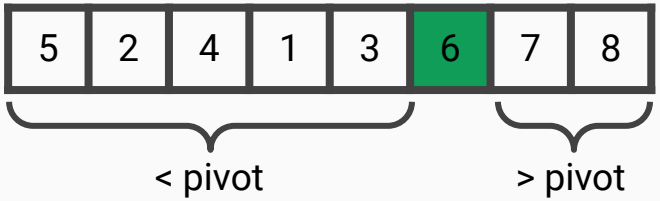


Partition elements using pivot

Quicksort

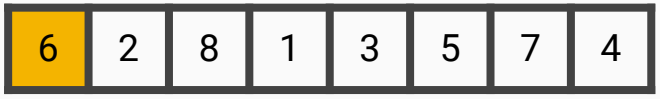


Pick pivot

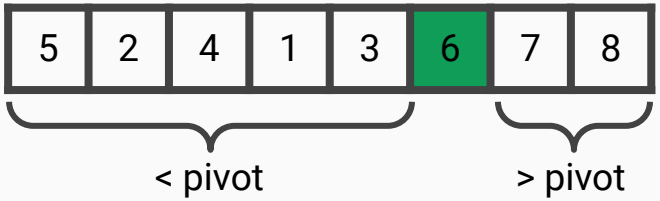


Partition elements using pivot
Pivot is in **correct final sorted position**

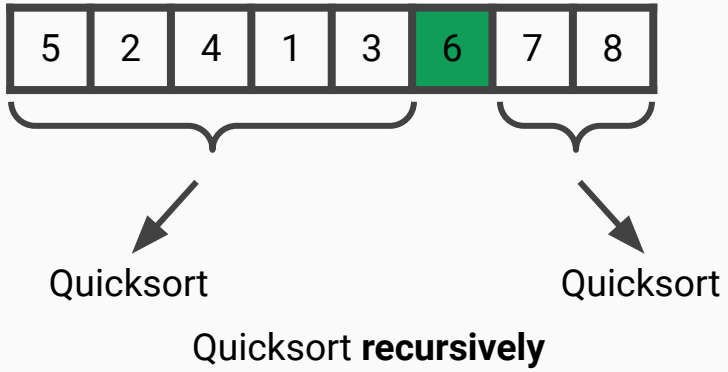
Quicksort



Pick pivot

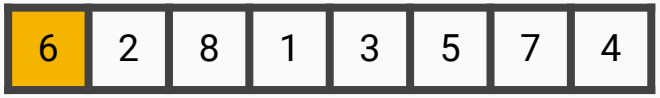


Partition elements using pivot
Pivot is in **correct final sorted position**

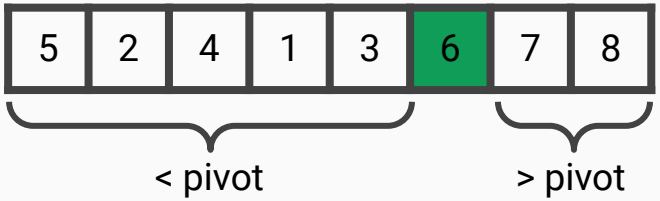


Quicksort **recursively**

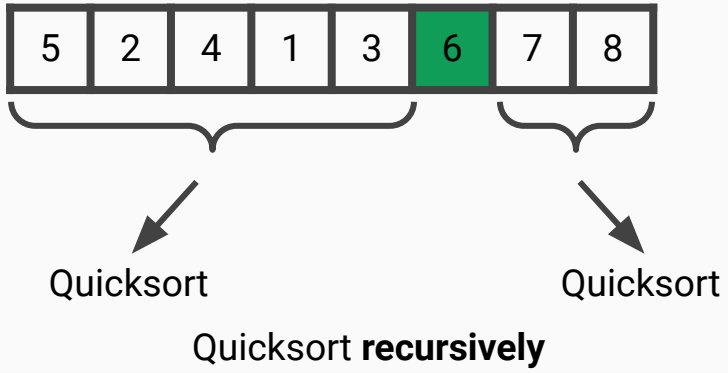
Quicksort



Pick pivot



Partition elements using pivot
Pivot is in **correct final sorted position**



Quicksort **recursively**

Running Time: Expected O(n log n)
Worst Case: O(n^2) for bad pivots

Sorting

In-place

No additional data structures are used to perform the sorting, other than a small number of additional variables.

Stable

Relative order of equal elements is preserved after the sorting is performed.

7 Summary of Sorting Algorithms

	Worst Case	Best Case	In-place?	Stable?
Selection Sort	$O(n^2)$	$O(n^2)$	Yes	No
Insertion Sort	$O(n^2)$	$O(n)$	Yes	Yes
Bubble Sort	$O(n^2)$	$O(n^2)$	Yes	Yes
Bubble Sort 2 (improved with flag)	$O(n^2)$	$O(n)$	Yes	Yes
Merge Sort	$O(n \log n)$	$O(n \log n)$	No	Yes
Radix Sort (non-comparison based)	$O(n)$ (see notes 1)	$O(n)$	No	Yes
Quick Sort	$O(n^2)$	$O(n \log n)$	Yes	No

Notes: 1. **O(n)** for Radix Sort is due to non-comparison based sorting.
2. **O(n log n)** is the best possible for comparison based sorting.

Question 1

You are compiling a list of students (ID, weight) in Singapore, for your CCA. However, due to budget cut, you are facing a problem in the amount of memory available for your computer. After loading all students in memory, the extra memory available can only hold up to 20% of the total students you have! Which sorting method should be used to sort all students based on weight (no fixed precision)?

Question 1

You are compiling a list of students (ID, weight) in Singapore, for your CCA. However, due to budget cut, you are facing a problem in the amount of memory available for your computer. After loading all students in memory, the extra memory available can only hold up to 20% of the total students you have! Which sorting method should be used to sort all students based on weight (no fixed precision)?

Need in-place sorting algorithm that works for floating point.

Quicksort

Question 1

After your success in creating the list for your CCA, you are hired as an intern in NUS to manage a student database. There are student records, already sorted by name. However, we want a list of students first ordered by age. For all students with the same age, we want them to be ordered by name. In other words, we need to preserve the ordering by name as we sort the data by age.

Question 1

After your success in creating the list for your CCA, you are hired as an intern in NUS to manage a student database. There are student records, already sorted by name. However, we want a list of students first ordered by age. For all students with the same age, we want them to be ordered by name. In other words, we need to preserve the ordering by name as we sort the data by age.

We need a stable sort.

For n students, Merge sort $\rightarrow O(n \log n)$. Radix sort $\rightarrow O(n)$.

Question 1

After finishing internship in NUS, you are invited to be an instructor for CS1010E. You have just finished marking the final exam papers randomly. You want to determine your students' grades, so you need to sort the students in order of marks. As there are many CA components, the marks have no fixed precision.

Question 1

After finishing internship in NUS, you are invited to be an instructor for CS1010E. You have just finished marking the final exam papers randomly. You want to determine your students' grades, so you need to sort the students in order of marks. As there are many CA components, the marks have no fixed precision.

Merge sort or Quicksort $\rightarrow$ O(n log n).

Question 1

Before you used the sorting method in (c), you realize the marks are already in sorted order. However, just to be very sure that you did not cut and paste a student record in the wrong order, you still want to sort the result.

Question 1

Before you used the sorting method in (c), you realize the marks are already in sorted order. However, just to be very sure that you did not cut and paste a student record in the wrong order, you still want to sort the result.

Insertion sort, as marks are almost sorted.

Bubble sort with early termination?

Question 1

Before you used the sorting method in (c), you realize the marks are already in sorted order. However, just to be very sure that you did not cut and paste a student record in the wrong order, you still want to sort the result.

Insertion sort, as marks are almost sorted.

Bubble sort with early termination?

No, consider the input 2 3 4 5 6 1.

Question 2: Finding the *k*th smallest element in an unsorted array

Given an array $A[1...n]$ containing n integers, find the kth smallest element in the array in $O(n \log n)$ time.

Question 2: Finding the *k*th smallest element in an unsorted array

Given an array $A[1...n]$ containing n integers, find the kth smallest element in the array in $O(n \log n)$ time.

Sort the array (mergesort/quicksort) and output $A[k]$.

Question 2: Finding the *k*th smallest element in an unsorted array

Given an array $A[1...n]$ containing n integers, find the kth smallest element in the array in expected $O(n)$ time.

Question 2: Finding the *k*th smallest element in an unsorted array

Quickselect Algorithm

Quickselect is similar to quicksort, except that after partitioning, instead of recursing down both halves of the array, we will only explore the part of the array that the *k*th element is in.

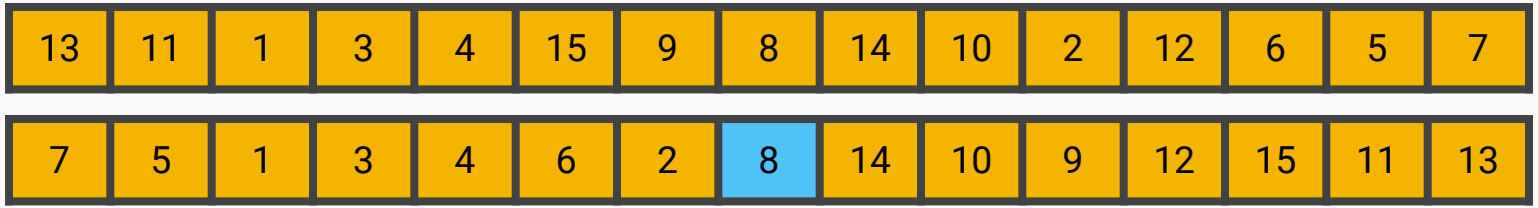
Quickselect

$k = 9$

Quickselect

$k = 9$

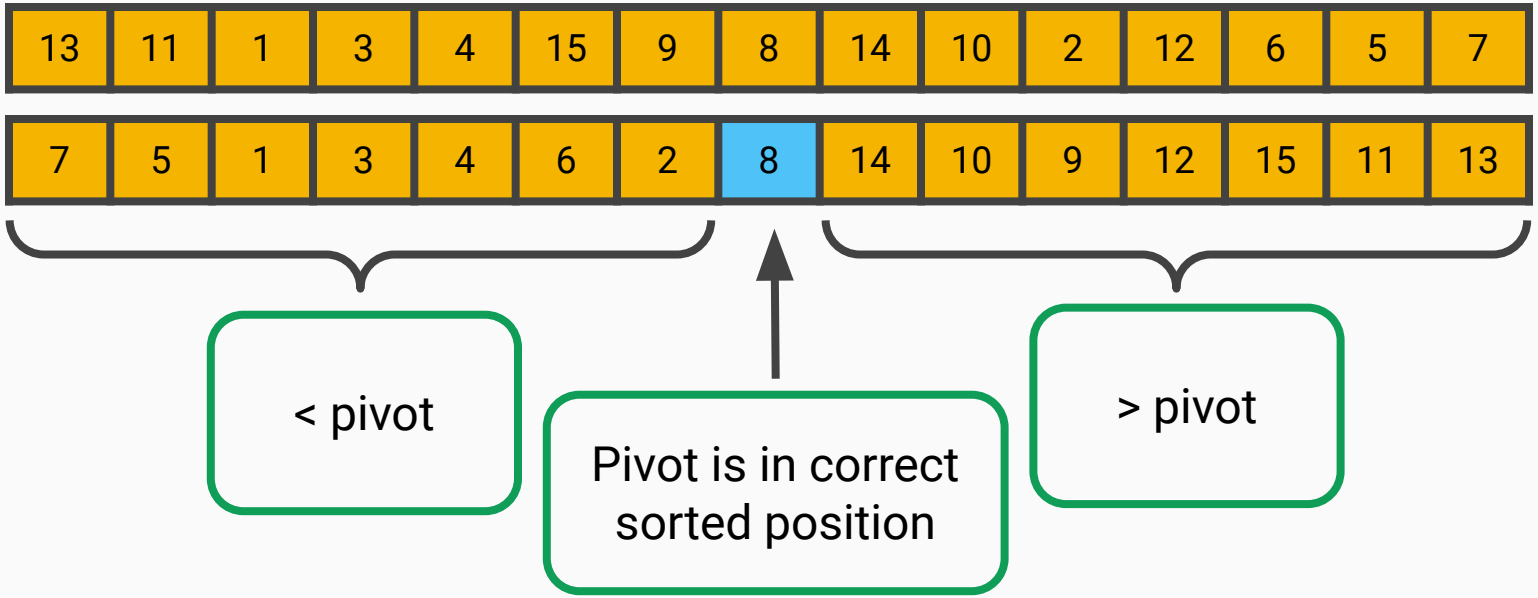
Partition



Quickselect

$k = 9$

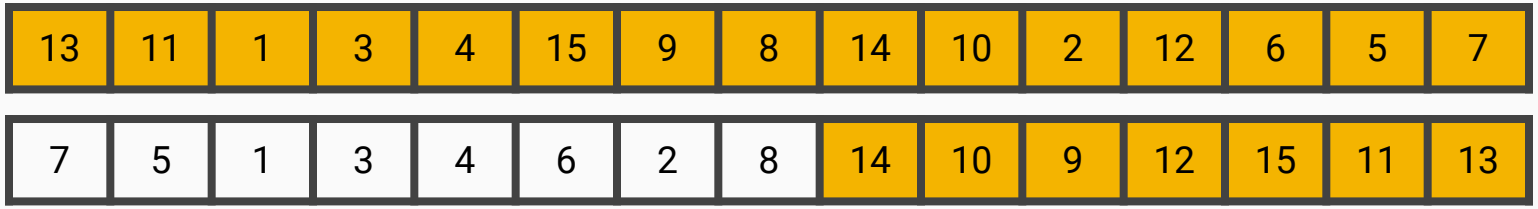
Partition



Quickselect

$k = 9$

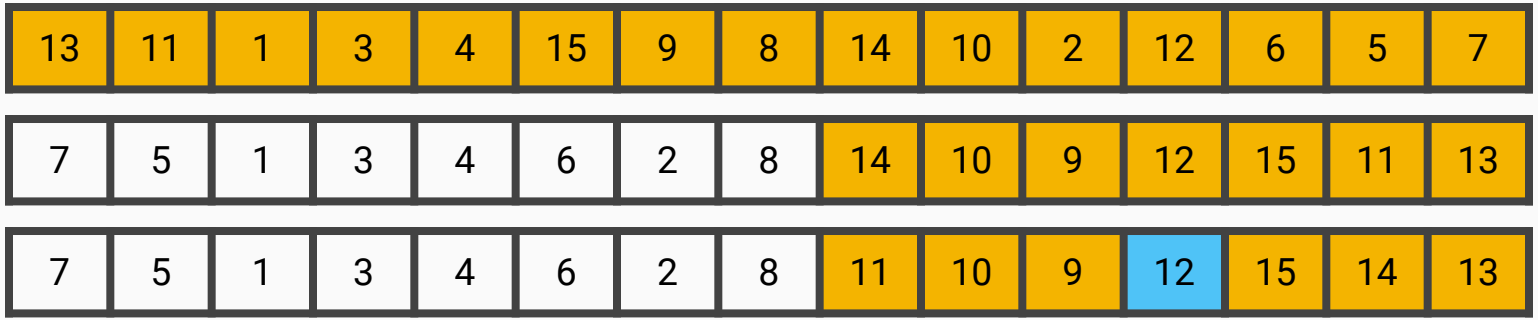
9th smallest on the right half. Find 1st smallest element on right half.



Quickselect

$k = 9$

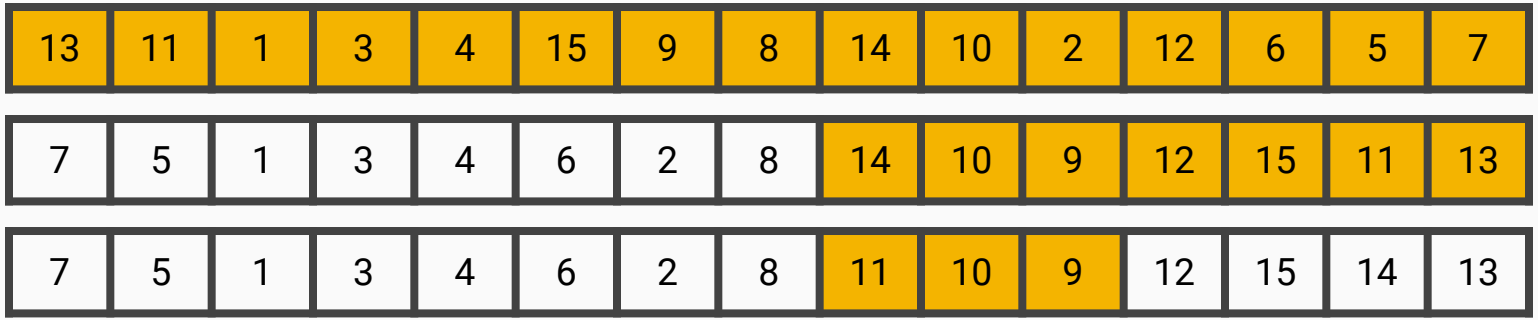
Partition.



Quickselect

$k = 9$

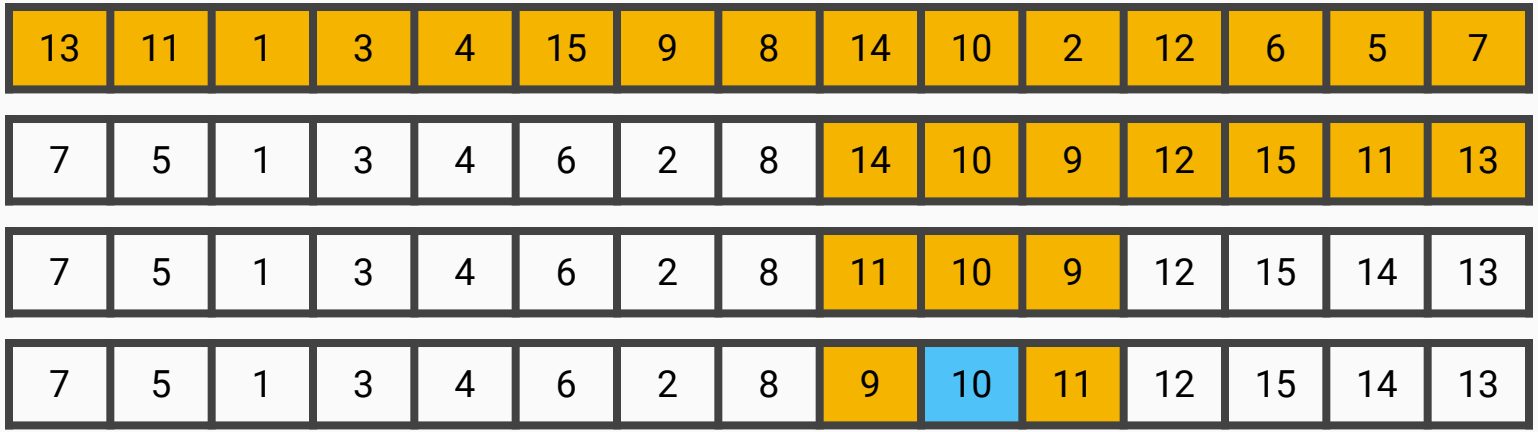
1st smallest element on left half. Find 1st smallest element on left half.



Quickselect

$k = 9$

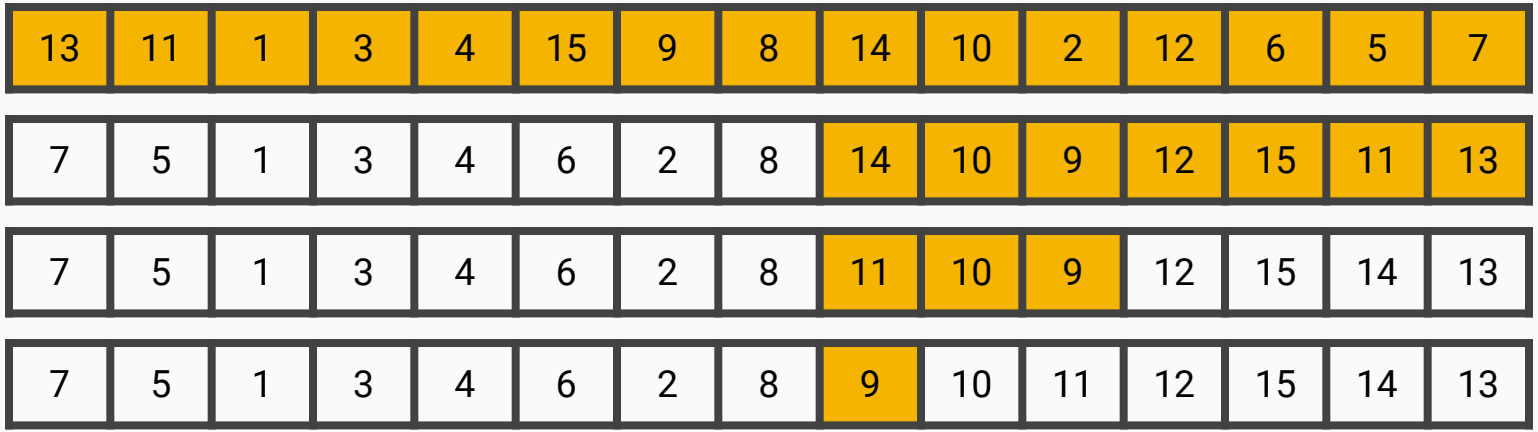
Partition.



Quickselect

$k = 9$

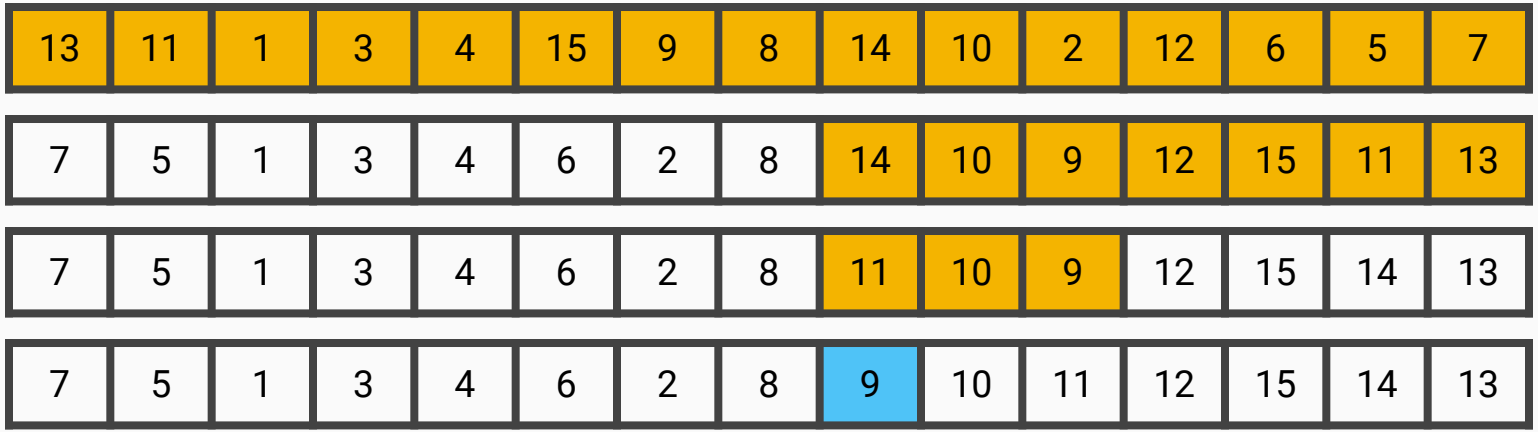
1st smallest element on left half. Find 1st smallest element on left half.



Quickselect

$k = 9$

Only one element left. We have found the 9th smallest element!



Question 3

Given N patients, where the ith patient requires T_i time to be served, minimise waiting time of all patients.

Question 3

Given N patients, where the ith patient requires T_i time to be served, minimise waiting time of all patients.

Greedy

Sort all the patients by waiting time. $O(N \log N)$.

Serve patients in increasing T_i. $O(N)$.

Total: $O(N \log N)$.

Question 3

Given N patients, where the ith patient requires T_i time to be served, minimise waiting time of all patients.

Proof (For CS3230)

Suppose serving patients in increasing T_i does not lead to an optimal solution.

Then in the optimal solution, there exists patients A and B such that $T_A < T_B$ but we serve T_B first.

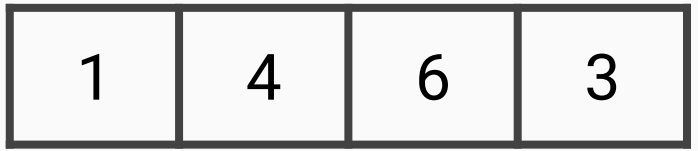
If we swap patients A and B, then we reduce the waiting time of at least one patient by $T_B - T_A$. This leads to a overall decrease in total waiting time.

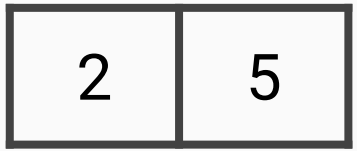
Contradiction. Hence, it is optimal to serve patients in increasing T_i.

Question 4

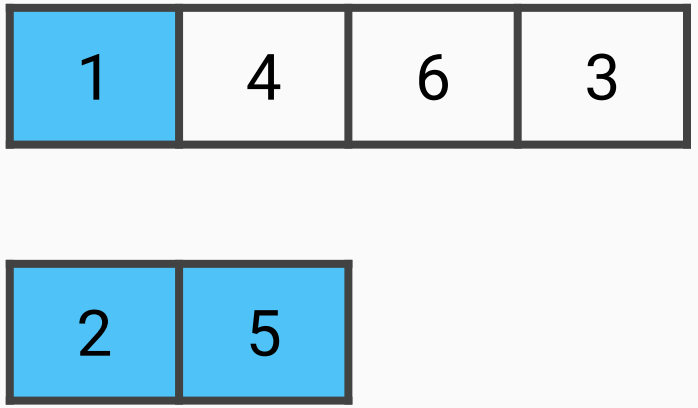
There is a permutation S of length n. Some elements have been deleted from S. You are given A, the elements remaining in the permutation. Find the lexicographically smallest permutation that could have been S.

Question 4

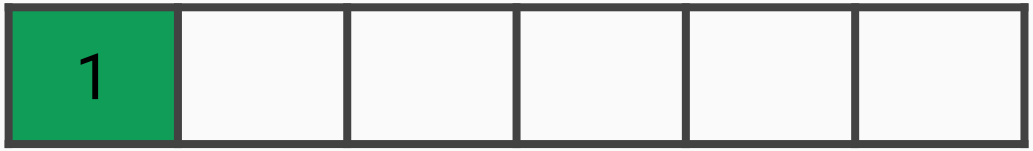


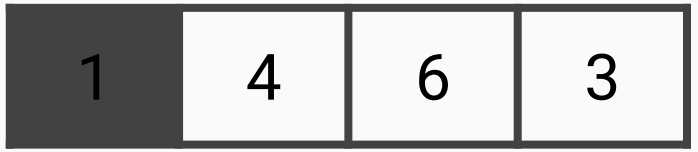


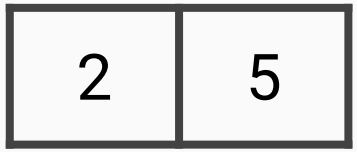
Question 4



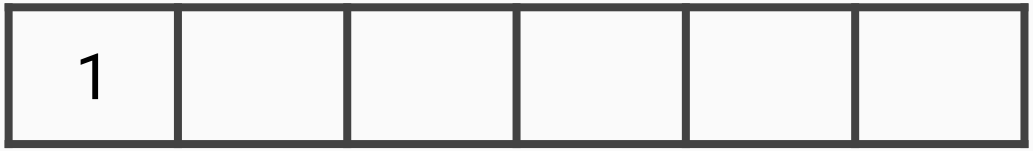
Question 4

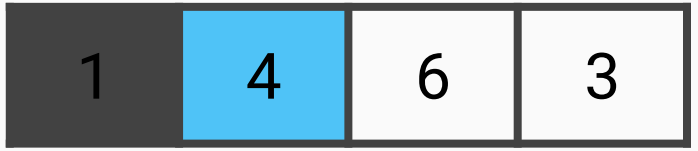


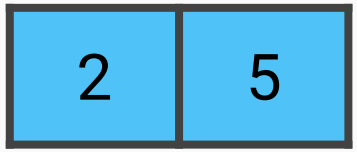




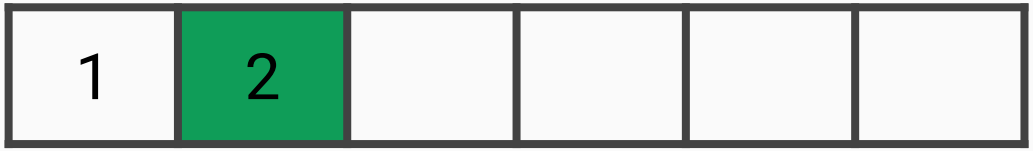
Question 4

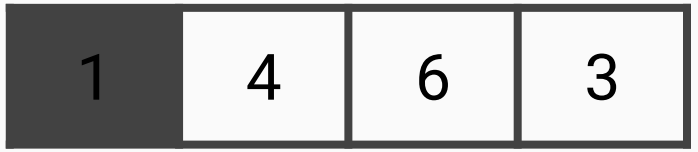




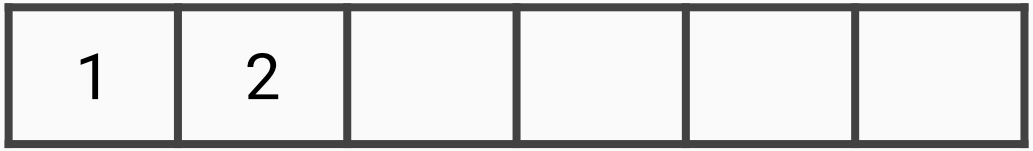


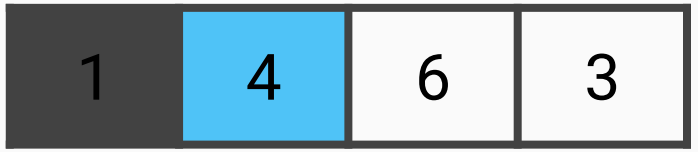
Question 4



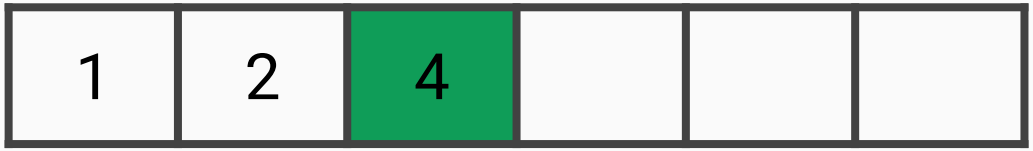


Question 4

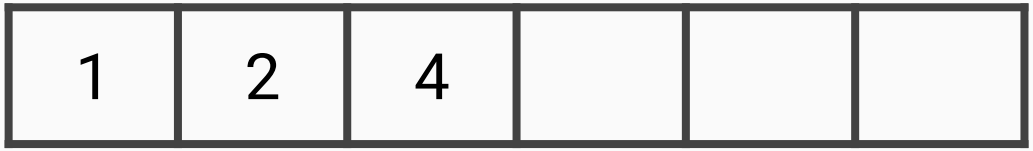


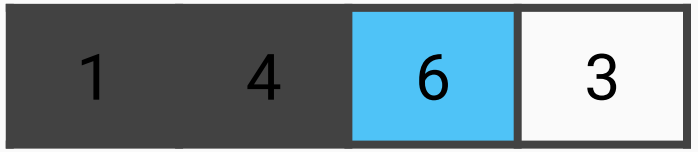


Question 4

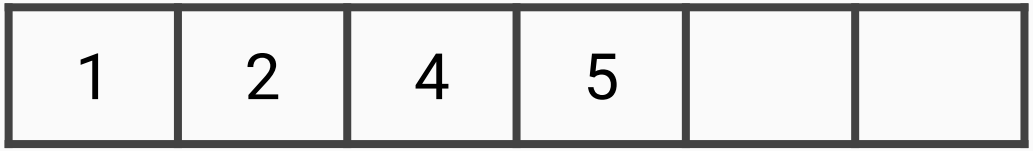


Question 4

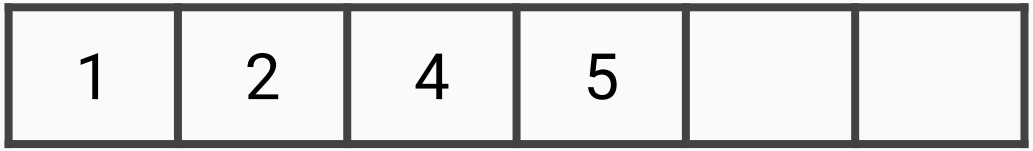




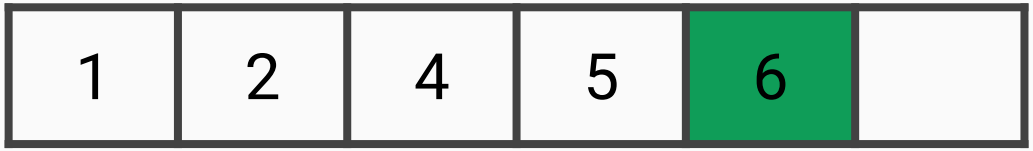
Question 4



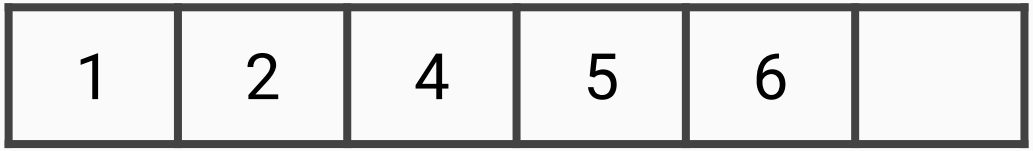
Question 4



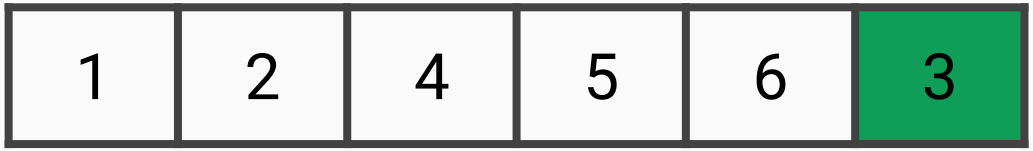
Question 4



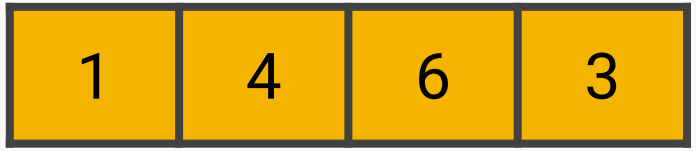
Question 4

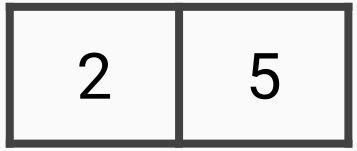


Question 4

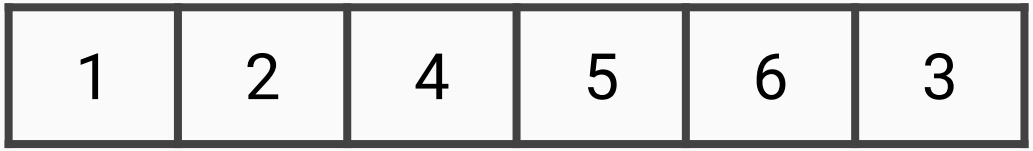


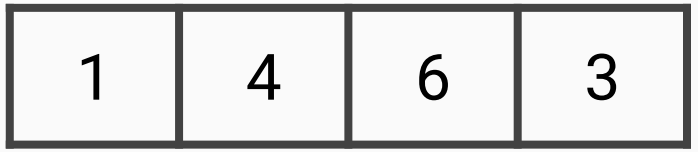
Question 4

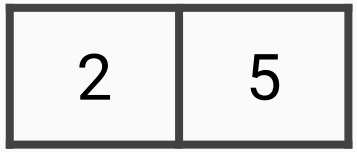




Question 4

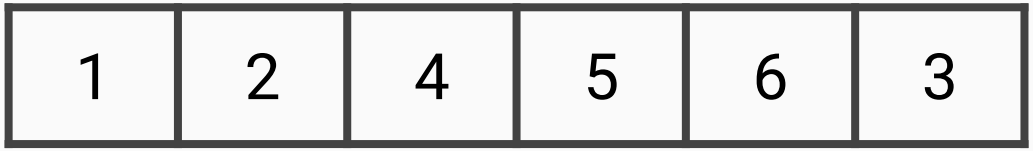


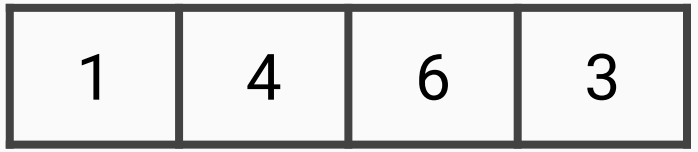


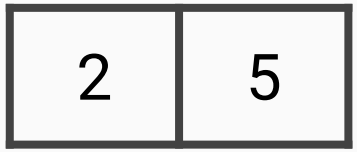


What does this procedure look like?

Question 4







What does this procedure look like?

‘Merge’ in mergesort!

CS2040 Tutorial 2 Appendix

Week 4, AY 19/20 Sem 2

Wang Zhi Jian
wzhijian@u.nus.edu

Extra Questions

Some extra questions here for you to try out, with solutions.

Do give them a try!

k-way Merge Sort

(AY17/18 Sem 2 Midterms) The merge sort that we discussed in lecture divides the items into two halves, recursively sorts the two halves and merges the two sorted halves. We called this 2-way merge sort.

A *k*-way merge sort algorithm divides the items into *k* almost-equal portions, recursively sorts the *k* portions and merges the *k* sorted portions. Is the *k*-way merge sort more efficient that the 2-way mergesort? Show your analysis of the time complexity of the *k*-way merge sort to justify your answer.

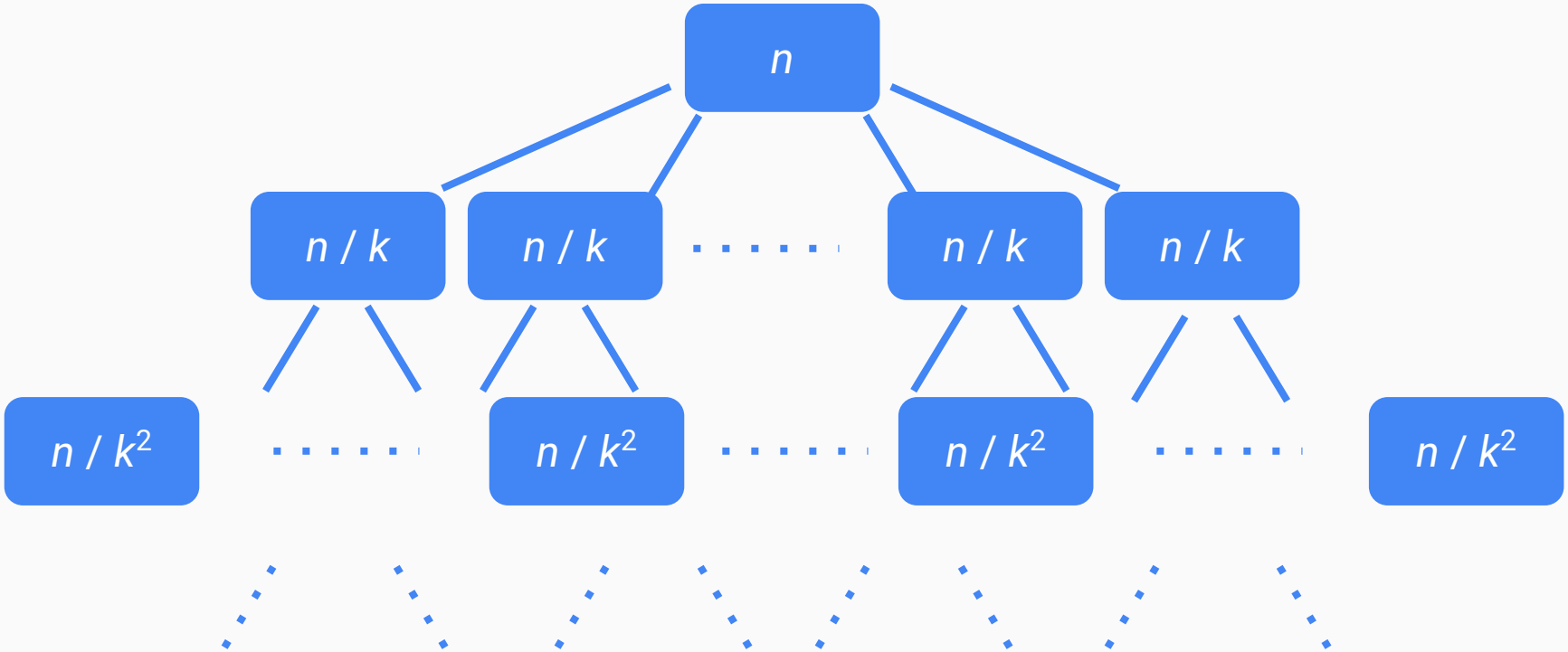
Recursion Tree

For big-O questions involving recursion, it would be useful to

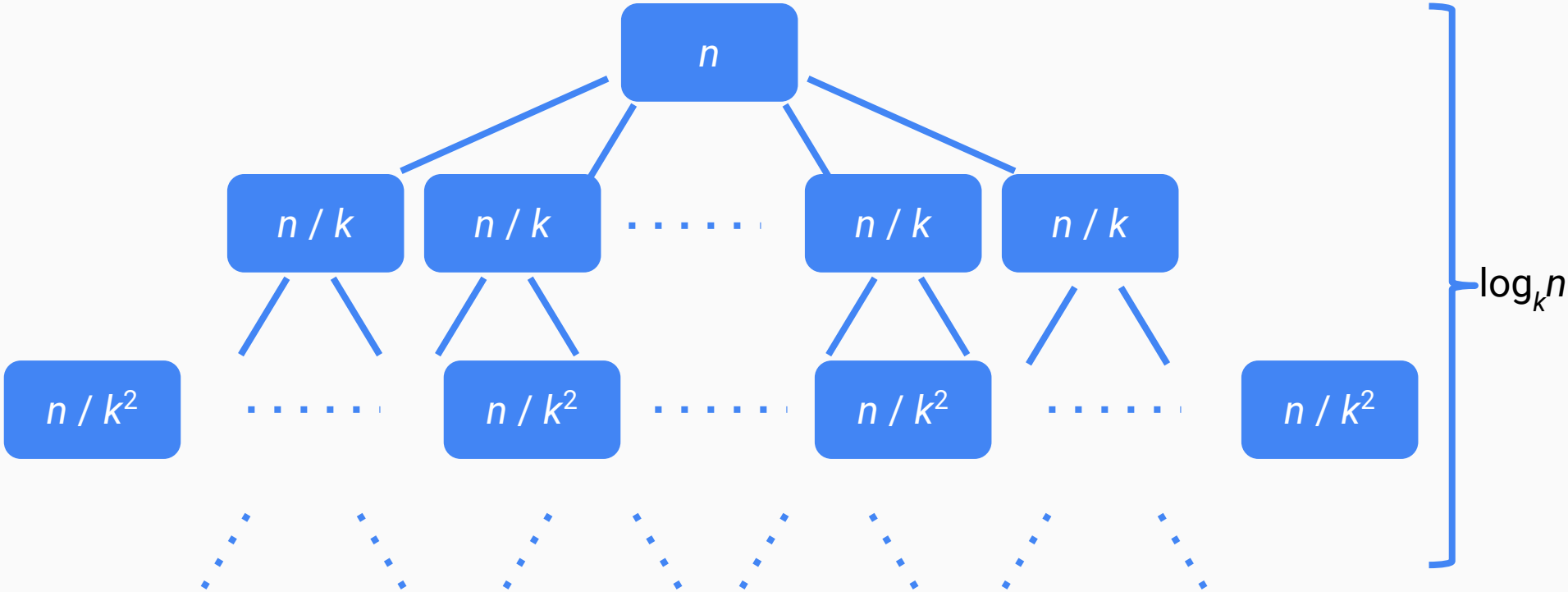
1. Draw the recursion tree out
2. Find the height of the tree
    a. The height of the tree usually corresponds to the **number of terms** you need to sum
3. Find the work done at every node in the recursion tree
4. Find the work done at every layer of the recursion
    a. This is just the sum of the work done by every node in each layer
    b. See you can spot some kind of pattern

k-way Merge Sort

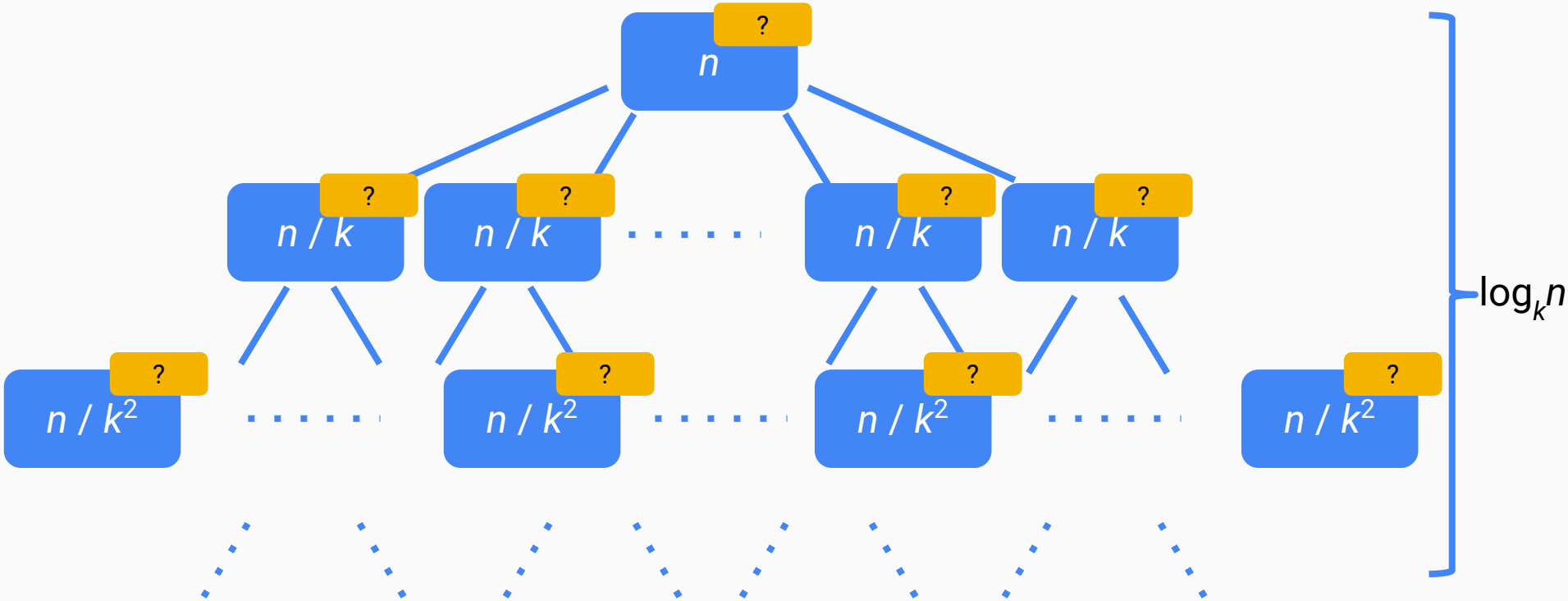
Step 1: Draw the recursion tree



Step 2: Find the height of the tree



Step 3: Find the work done for every node



k-way Merge Sort

Work Done

Suppose the array that an arbitrary node is responsible for sorting has size m.

Then, this node needs to merge k sorted arrays of size m / k.

k-way Merge

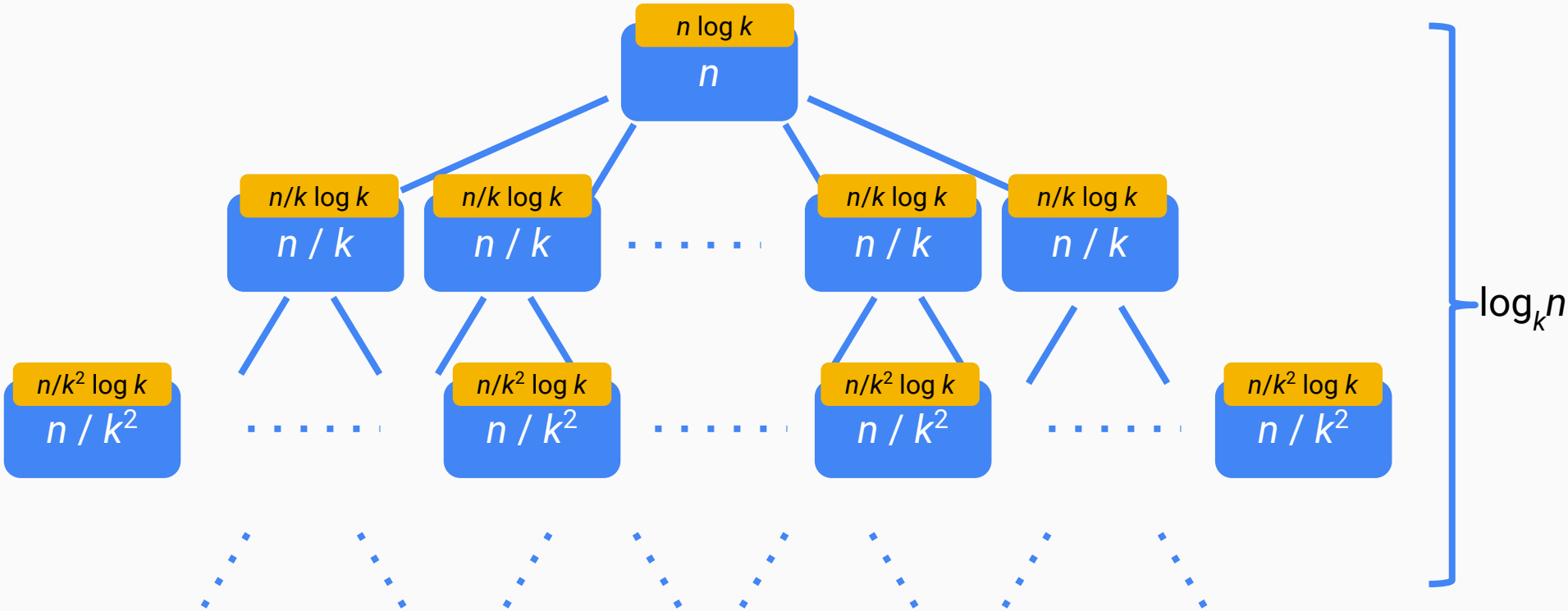
(AY18/19 Sem 1 Quiz 1) We want to merge k sorted lists and return one list comprising all their elements in sorted order. There are m/k elements in each sorted list and all of them have the same number of elements.

k-way Merge

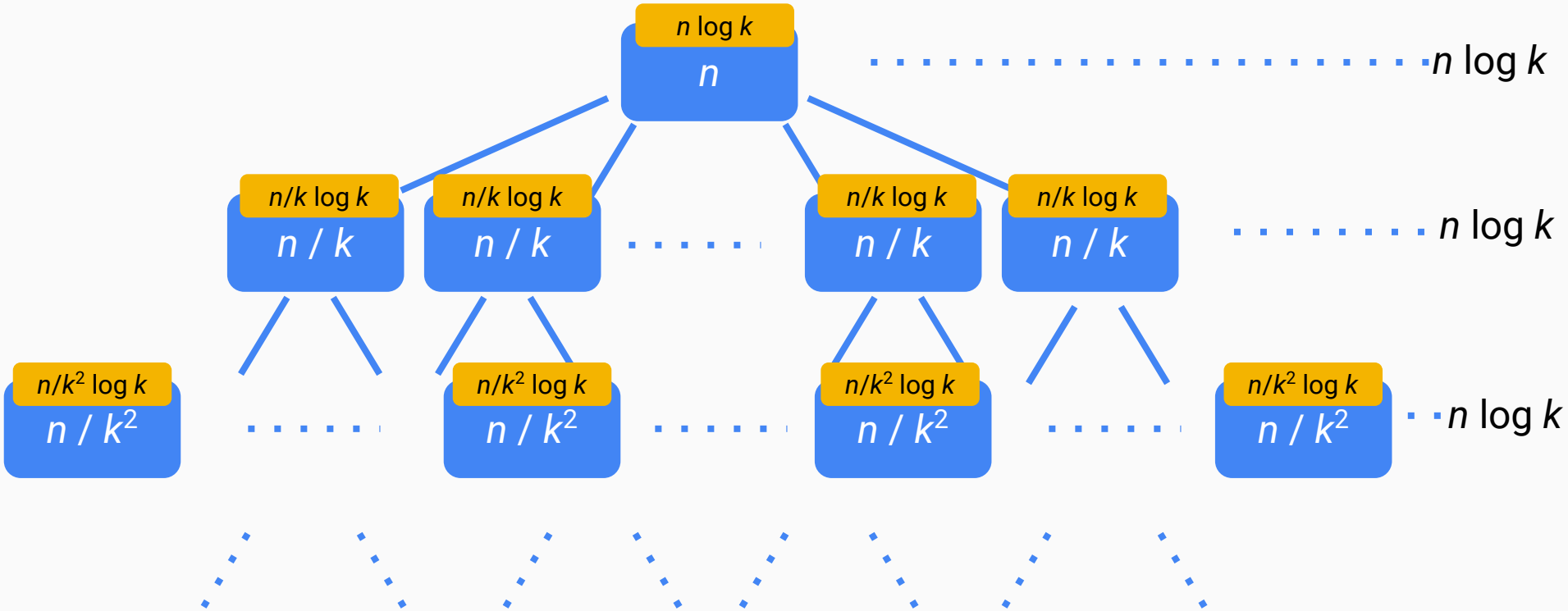
(AY18/19 Sem 1 Quiz 1) We want to merge k sorted lists and return one list comprising all their elements in sorted order. There are m/k elements in each sorted list and all of them have the same number of elements.

Refer to the solutions for AY18/19 Quiz 1: This problem can be solved in $O(m \log k)$.

Step 3: Find the work done for every node



Step 4: Find the work done at every layer



k-way Merge Sort

Finding a Pattern

At the 1st layer of recursion, $n \log k$ operations are performed.

At the 2nd layer of recursion, $n \log k$ operations are performed.

At the 3rd layer of recursion, $n \log k$ operations are performed.

…

At the ith layer of recursion, $n \log k$ operations are performed.

k-way Merge Sort

Finding a Pattern

At the 1st layer of recursion, $n \log k$ operations are performed.

At the 2nd layer of recursion, $n \log k$ operations are performed.

At the 3rd layer of recursion, $n \log k$ operations are performed.

...

At the $\log_k n$ th layer of recursion, $n \log k$ operations are performed.

We have $\log_k n$ layers of recursion.

k-way Merge Sort

Summing Everything Up

$$\underbrace{n \log k + n \log k + n \log k + \ldots + n \log k}_{\log_k n \text{ copies}} = (n \log k) \cdot (\log_k n)$$

$$= (n \log k) \cdot (\log n / \log k) \text{ (change of base)}$$

$$= n \log n$$

$$= O(n \log n)$$

Conclusion: Asymptotically, *k*-way merge sort is just as efficient as 2-way merge sort!