INFO1105/1905
Data Structures
Week 8

THE UNIVERSITY OF SYDNEY

Announcements

This week

- Quiz 3 (no hashing content!)
- Assignment 1 is due

Next week

- University mid-semester break

Lecture Outline

- Reflection on progress in 1st half of semester
- Reasoning about scalability
- Review/revision of Hash Tables content (week 7)

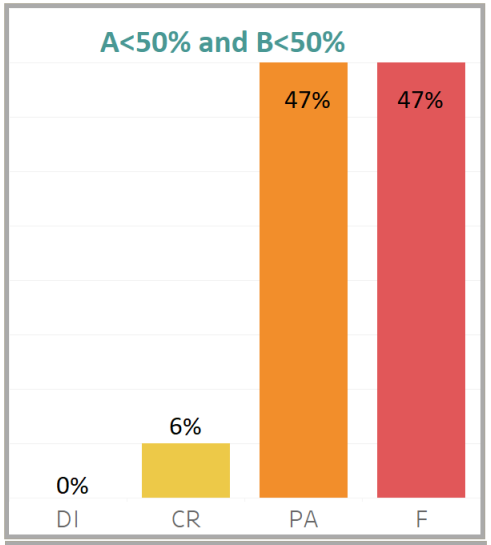
Course progress

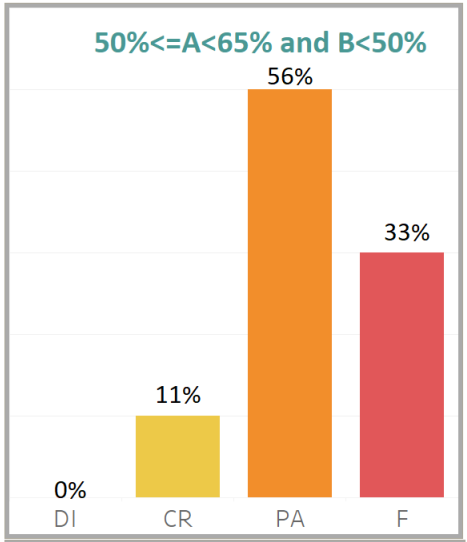
Reflect on your progress to date

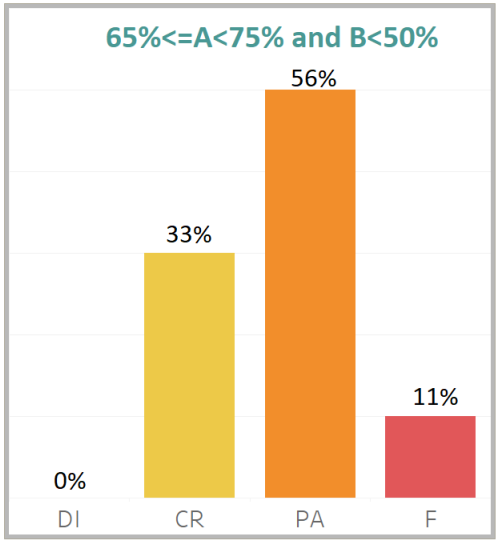
Discussion points:

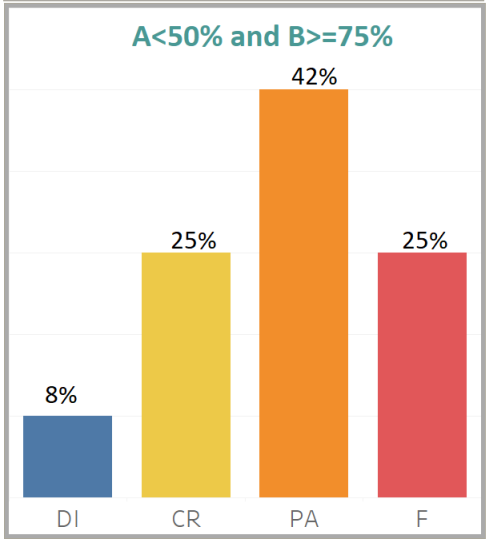
- If you are not doing well, is it too late to pass?
- If you are doing well, can you relax?

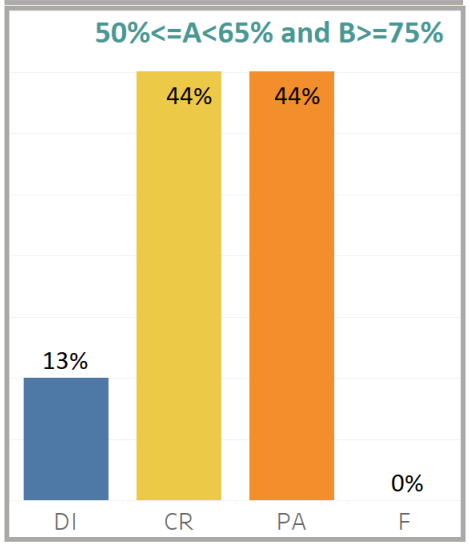
Statistics from last year's cohort

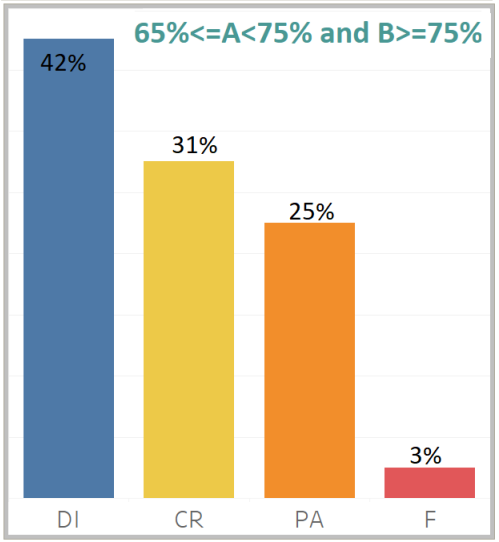












What is A & B?

A: Progress in first half of semester

B: Measure of engagement in second half of semester

Colour Legend

Grade
HD
DI
CR
PA
F

Grade Distribution

Grade	
HD	54
DI	98
CR	92
PA	114
F	36
Grand Total	394

Observations

For students with a progressive mark <50% at this point:

- If you engage with the course (**attempt** exercises, attend tutorials, lectures etc.) then you will probably pass, and you can still get a great grade

- Students that do **not** engage with the course have a very high (50/50) chance of failing

Observations

For students with a decent progressive mark (CR to D):

- If you engage with the course, your result is likely to get even better

- Students at this level who don't engage with the rest of the course *still* have significant chance of failing, or only securing a pass grade

Big ideas: analysis of runtime

- The particular choice of data structure and algorithms for the operations has a huge influence on how quickly the program runs
  - Especially, on how the running time grows, as we have more and more data items in the collection
- There is a mathematical language that we can use to describe how runtime (or other measurements) grow as a feature increases
  - "big-Oh" notation
  - allows to distinguish constant time, logarithmic time, linear time, quadratic time, exponential time algorithms
- There is mathematical theory that allows us to reason, to work out what growth a particular algorithm has, based on knowledge of how the algorithm works

Big-Oh notation

(**Very**) informal definition:

We say a function **f(n)** is **O(g(n))** if **f(n)** *asymptotically grows* no faster than **g(n)**.

i.e. if we *double* the size of **n**, then **f(n)** will not increase by a larger factor than **g(n)** does.

Example:

2n is **O(n)**, because if we double the size of **n**, both will double in size. These are both **linear** functions.

Big-Oh notation: common functions

Big-Oh	Name	Examples
$O(2^n)$	Exponential	Calculating Fibonacci without remembering previously calculated values
$O(n^2)$	Quadratic	Looking at every possible pair of n objects, Repeating an O(n) operation O(n) times Insertion Sort
O(n log n)		Many sorting algorithms, e.g. Merge Sort Inserting n elements into a balanced binary search tree
O(n)	Linear	Finding an element in a List Printing out everything from a list of n objects
O(log n)	Logarithmic	Searching in a balanced binary search tree
O(1)	Constant	Basic arithmetic Simple print statements

Big-Oh notation: why it matters

Table 2.1 The running times (rounded up) of different algorithms on inputs of increasing size, for a processor performing a million high-level instructions per second. In cases where the running time exceeds 10^{25} years, we simply record the algorithm as taking a very long time.

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} years
$n = 50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years	very long
$n = 100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1{,}000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long	very long
$n = 10{,}000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long	very long
$n = 100{,}000$	< 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1{,}000{,}000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long

Big-Oh notation: worst/average/best

When we analyse code, we usually talk about the **worst case** running time. Why?

- It's usually easy to reason about
- It's nice to know 'how bad' an operation is, at worst

Sometimes we talk about **average case.**

- Often **much** harder to reason about, or even define
- Sometimes more useful, where the worst case is extraordinarily rare (e.g. HashMaps)

We will talk about **worst case** for the rest of this section.

Big-Oh notation: analysing code

If we perform a fixed sequence of steps, the Big-Oh running time is the same as the running time of the worst of those steps.

Example:

Find the position of an element in a balanced BST: O(log n)

Remove that position from the BST: O(log n)

Increase the value of the element: O(1)

Store the new value in the BST: O(log n)

In total, this code is still only O(log n)

Big-Oh notation: analysing code

A simple approach to analysing code, is to think about how much work is done on each line, and how many times each line is executed.

Example:

if (x is in some List of size n) then

    insert x into a balanced BST of size n

Checking existence of x in a List is O(n)

Inserting x into a balanced BST is O(log n)

How long does the code take in total?

Big-Oh notation: analysing code

A simple approach to analysing code, is to think about how much work is done on each line, and how many times each line is executed.

Example:

if (x is in some List of size n) then

    insert x into a balanced BST of size n

Checking existence of x in a List is O(n)

Inserting x into a balanced BST is O(log n)

How long does the code take in total?

    at worst, O(n) once + O(log n) once = just O(n) in total.

Big-Oh notation: analysing code

A simple approach to analysing code, is to think about how much work is done on each line, and how many times each line is executed.

Example:

for each element x in some List of size n then

    insert x into a balanced BST

How long does this code take in total?

Big-Oh notation: analysing code

A simple approach to analysing code, is to think about how much work is done on each line, and **how many times each line is executed.**

Example:

for each element x in some List of size n then

    insert x into a balanced BST

How long does this code take in total?

    each insertion is O(log n), but we do this line **n** times!

    so, n*O(log n) = O(n log n)

Analysing recursive code

A simple approach to analysing code, is to think about how much work is done on each line, and **how many times each line is executed.**

This is a bit harder to deduce for recursive code!

We express the running time with a **recurrence** formula.

Analysing recursive code

Recall MergeSort, from earlier in the semester:

MergeSort(list):

    If the list has one or less elements, return the list

    Otherwise, split the list into two halves A and B

    List sortedA = MergeSort(A)

    List sortedB = MergeSort(B)

    Merge the sorted lists sortedA and sortedB

    Return the merged list

Analysing recursive code

MergeSort(list):

If the list has one or less elements, return the list	O(1)
Otherwise, split the list into two halves A and B	O(n)
List sortedA = MergeSort(A)	T(n/2)
List sortedB = MergeSort(B)	T(n/2)
Merge the sorted lists sortedA and sorted	O(n)
Return the merged list	

Let T(n) be the time to run MergeSort on a list of size n

Then **T(n) = 2*T(n/2) + O(n)**

Analysing recursive code: solving recurrences

So, we know MergeSort takes time:

$$T(n) = 2*T(n/2) + O(n)$$

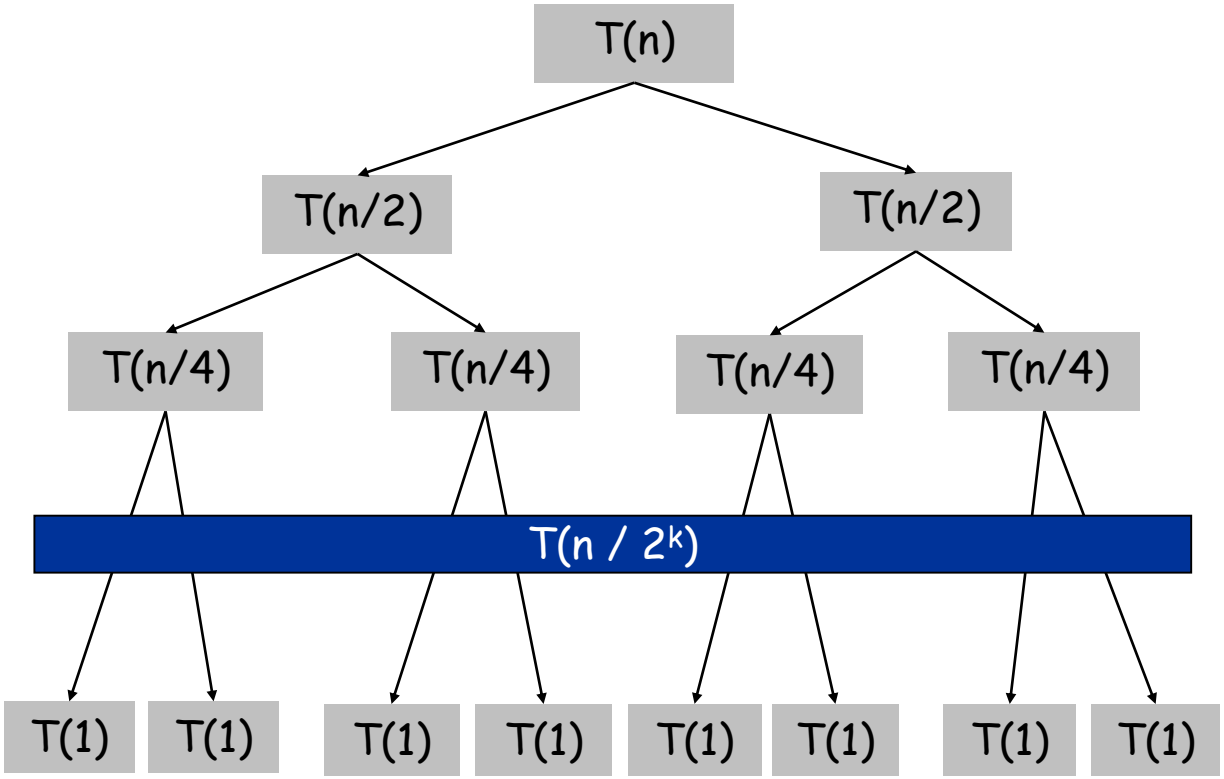
We need to try to solve this recurrence relation so that it only depends on n, not on T(n/2).

In this course, it is sufficient to be able to recognise certain patterns and guess the solution (i.e. you do **not** need to be able to solve these yet – you'll learn more about that next year!)

However, for MergeSort, I will show you how to solve it, for fun.

Analysing recursive code: solving recurrences

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ \underbrace{2T(n/2)}_{\text{sorting both halves}} + \underbrace{cn}_{\text{merging}} & \text{otherwise} \end{cases}$$



1 (of size n)

2 (of size n/2)

4 (of size n/4)

. . .

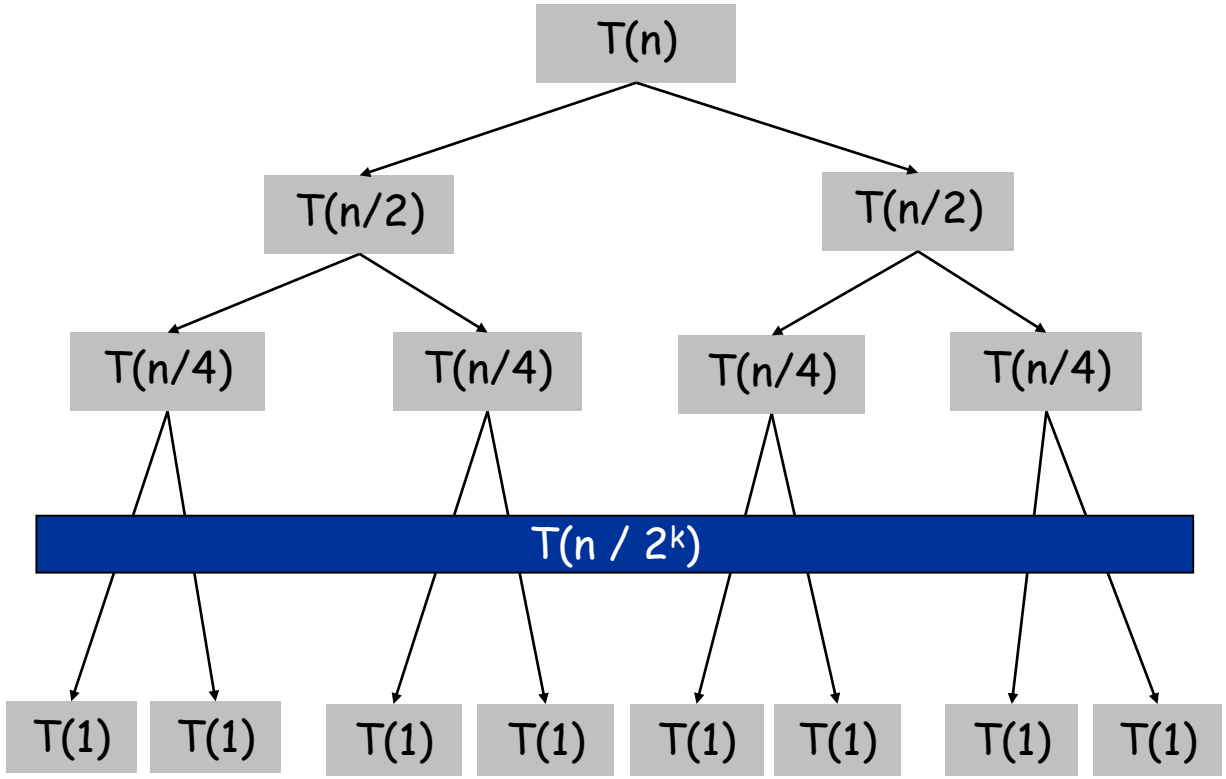
2^k (of size $n/2^k$)

. . .

n (of size 1)

Analysing recursive code: solving recurrences

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ \underbrace{2T(n/2)}_{\text{sorting both halves}} + \underbrace{cn}_{\text{merging}} & \text{otherwise} \end{cases}$$



1 (of size n) → O(n)

2 (of size n/2) → O(n)

4 (of size n/4) → O(n)

$\log_2 n$

. . .

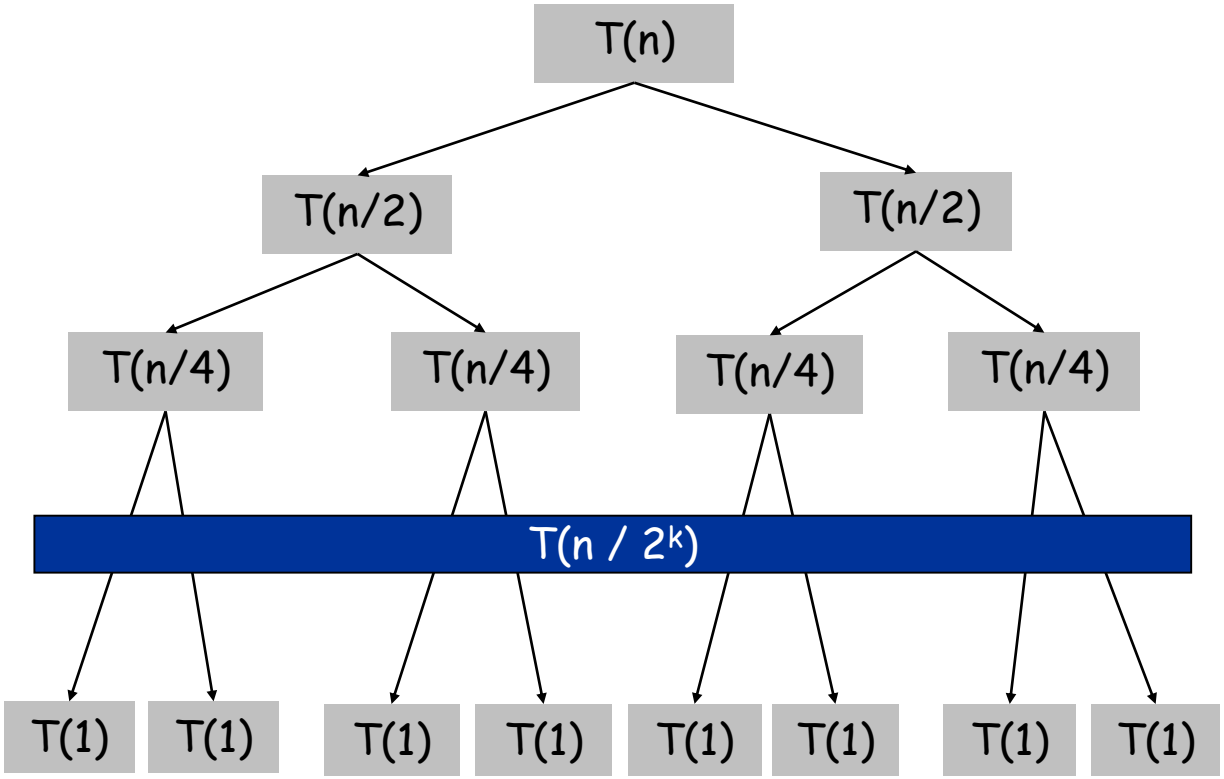
2^k (of size $n/2^k$) → O(n)

. . .

n (of size 1) → O(n)

Analysing recursive code: solving recurrences

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ \underbrace{2T(n/2)}_{\text{sorting both halves}} + \underbrace{cn}_{\text{merging}} & \text{otherwise} \end{cases}$$



1 (of size n) → O(n)

2 (of size n/2) → O(n)

4 (of size n/4) → O(n)

$\log_2 n$

. . .

2^k (of size $n/2^k$) → O(n)

. . .

n (of size 1) → O(n)

In total? log n of size O(n) → O(n log n)

Analysing recursive code: solving recurrences

You don't need to be able to mathematically solve them in this course. For now, it's sufficient to be able to recognise some common patterns and know their solutions.

Recurrence	Solution	Example
$T(n) = 2*T(n/2) + O(n)$	O(n log n)	Merge Sort
$T(n) = 2*T(n/2) + O(1)$	O(n)	Binary tree traversal (e.g. preorder)
$T(n) = T(n/2) + O(1)$	O(log n)	Binary search on a sorted array
$T(n) = T(n-1) + O(n)$	$O(n^2)$	Selection Sort
$T(n) = T(n-1) + T(n-2) + O(1)$	$O(1.618^n)$	Fibonacci (recursively)
$T(n) = T(n-1) + O(1)$	O(n)	Fibonacci (iteratively), n!

Summary

Big-Oh shows how well the code scales as the input size doubles

Analysing code line-by-line

- Think about how many times each line executes
  - if blocks execute once in the worst case
  - for blocks can execute many times

Analysing recursive code

- Find a recurrence in terms of T(n)
  - Solve the recurrence (if it is similar to one you recognise)

Hash Tables

Questions?