

# 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

# FIT2004: Algorithms and Data Structures

---

## Week 1: Introduction, and Proof of Correctness

These slides are prepared by Nathan Compane and are based on the material developed by [M. A. Cheema](#), [Arun Konagurthu](#) and [Lloyd Allison](#).

# Outline

---

- Part 1: Introduction to the unit
- Part 2:
  - Proving Correctness of Algorithms
  - Complexity Analysis (Recap)
  - Solving Recurrence Relations

# What's this unit about

---

- Solving problems with computers – efficiently
- Developing your algorithm toolbox
- Training your problem solving skills
- Deepening your understanding of how the workings of a computer relate to algorithm speed/space usage

# What's this unit about

---

- The subject is not (mainly) about **programming**.
- The subject just happens to use **Python** as the programming language in which lab work (etc.) is done. This subject is really **language agnostic**.
- Algorithms in this courseware will be presented/describe in English, pseudo-code, procedural set of instructions or Python (as convenient)

# Expectations

---

- The subject is very important for computer and technology related careers
  - Big tech companies actively hunt for people good at algorithms and data structures
  - Many coding interview questions will be based on algorithms from this unit
  - Many studio questions are very similar to questions you could be asked in coding interviews
  - The things you learn will help you throughout your career
  - Expertise in algorithms and data structures is a must if you want to do research in computer science

# Expectations

- This unit is **CHALLENGING**
  - Assignments will test your understanding of the topics as you go
  - You have to be on top of it from week 1 – you cannot pass if you think “I can brush up on the material close to the assessment deadlines”
  - Missing lectures or studios will require double the efforts to recover
- Good news: The unit wants you to succeed and understand

	Responses	Median	%Strongly Agree/Agree
The Learning Outcomes for this unit were clear to me	53	4.55	94.34%
The instructions for Assessment tasks were clear to me	53	4.65	94.34%
The Assessment in this unit allowed me to demonstrate the learning outcomes	53	4.59	90.57%
The Feedback helped me achieve the Learning Outcomes for the unit	53	4.47	79.25%
The Resources helped me achieve the Learning Outcomes for the unit	53	4.55	86.79%
The Activities helped me achieve the Learning Outcomes for the unit	53	4.52	88.68%
I attempted to engage in this unit to the best of my ability	53	4.62	90.57%
Overall, I was satisfied with this unit	52	4.69	92.31%

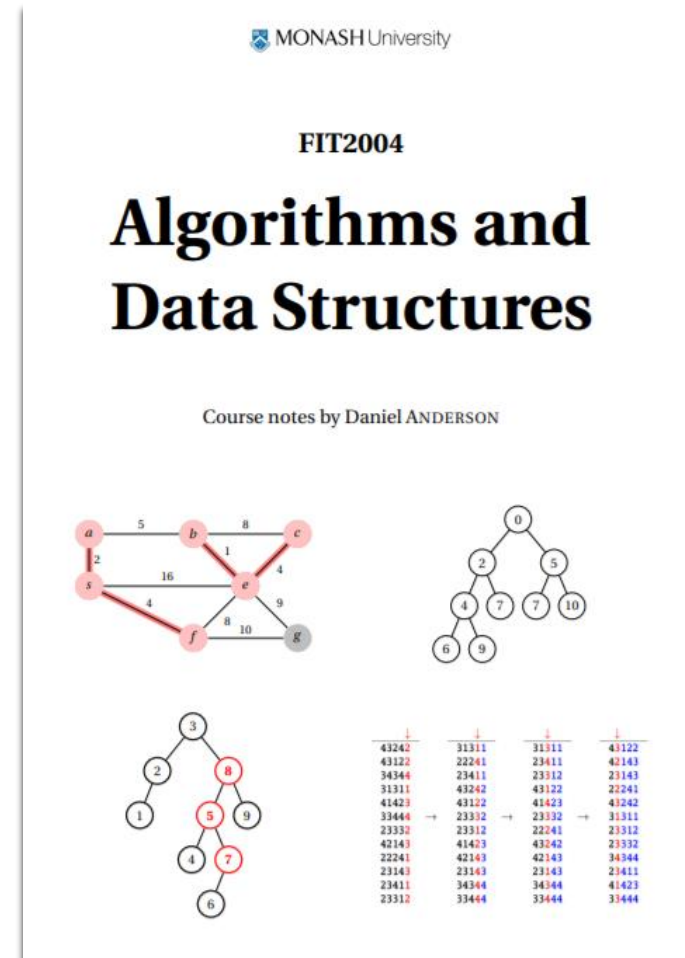
# Overview of the content

- **General problem-solving strategies and techniques**
  - Useful paradigms
  - Analysis of algorithms and data structures
- **A selection of important computational problems**
  - Sorting
  - Retrieval/Lookup
  - Compression
  - Connectivity
  - Shortest path
  - Minimum spanning tree
  - Maximum flow
- **A selection of important algorithms and data structures**
  - as a tool kit for the working programmer
  - as example solutions to problems
  - as examples of solved problems, to gain insight into concepts



# Unit Notes

- Unit notes are written based on the material covered in lecture notes
- Notes for all 12 weeks are available in a single PDF file on Moodle
  - Click on “Unit Information”
  - Scroll down to “unit resources”
  - Click on [“FIT2004 Notes”](#)



# FIT2004 Staff

---

- **Chief examiner:** Nathan Companez
- **Clayton Lecturers:** Nathan Companez, Bernd Meyer
- **Malaysia lecturer:** Lim Wern Han
  
- **Tutors:**
  - Aashna Jain
  - Ben Di Stefano
  - Chait Manapragada
  - Chaluka Salgado
  - James Ryan
  - Logan Yip
  - Rebecca Young
  - Saman Ahmadi
  - Santa Maiti
  - Shams Rahman
  - Magnus Bradley

**Contact details can be found on Moodle**

# Course Material

- Your main portal will be, as you already know, the unit's Moodle page:  
<http://moodle.vle.monash.edu/>
- Material available on Moodle will include:
  - Lecture slides
  - Lecture recordings
  - Assignments
  - Unit Notes
  - Studio sheets
  - Video solutions
- Ed Discussion Platform (Not moodle forums!)
  - General Q & A
  - Assignment amendments
  - Lecture error correction
  - Changes to class time
  - Changes to consultation time

# Course Structure

---

- Lecture 2hrs
  - All classes start at :00 and finish at :50
- Studio 3hrs (see allocate+)
  - No studio in week 01
  - There is a weekly studio solution video, along with written solutions
  - These used to be called tutorials, so if you see tutorial anywhere, it is referring to these classes

# Asking Questions During Lectures

---

- Please do it (you can use flux, **flux.qa/YTJMAZ**)
- If I move on from a point/topic and you are unsure about **anything**, ask!
- I would prefer to answer the same question more than once, than to never be asked
- I want to help you understand the material, but I can only help you if you tell me when you need help

# Give me feedback!

---

I imagine that as the unit runs, most of you will have things you dislike/think should be done differently

## **Tell me about it!**

- Send me an email
- Post anonymously on Ed

## **A few examples of changes which were motivated by student feedback**

- Reduction in exam percentage to 50% (new this semester!)
- Video solutions to tutorials
- Forum section specifically for sharing test cases and code for assignments
- Releasing studio solutions one week earlier

# Assessment Summary

---

- Assignments 1-4 40%
- Studio participation 10%
- Final Exam 50%

# Prac Assignments 1-4 (40%)

- Four practical assignments (each worth 10%)
  - Due: 20th aug, 10th sep, 1st Oct, 22nd oct
- Focus is on implementing algorithms **satisfying certain complexity** requirements
- If the complexity requirements are not met, you may simply receive a 0 for the task
- **Dictionaries** and **sets** are banned, unless specifically allowed by that question. Assignment has a note on this
- Must be implemented in Python
- Start early!!!



# How to approach assignments

---

- Many students have good in semester scores but fail the exam
- Many students focus all their effort in this unit on completing assignments
- The assignment are designed to be done by people with a strong grasp of the content
- They are **heavily** based on the content taught in the unit
- If you spend time understanding the content, the assignments will be **far** easier
- So will the exam!

# Studio Participation (10%)

- We have 3 hrs weekly studios starting from week 2
  - Studio sheet for week 1 is uploaded and you are expected to complete it in your own time
- Each studio is worth 1 mark . There are 11 weeks, excluding week 1. So the best 10 studios will be counted. (total marks capped at 10).
- In each studio:
  - Hurdle: (no marks awarded if hurdle not met)
    - You must answer questions in the section (assessed preparation) in your studio sheet. Due at the start of each week. You cannot work on this during class

# Final Exam (50%)

---

- 2 hours + 10 minutes reading time

# Hurdles

- To pass this unit a student must obtain:
  - **45% or more in the unit's final exam (i.e., at least 22.5 out of 50 marks for final exam), and**
  - **45% or more in the unit's total non-examination assessment (i.e., at least 22.5 out of total 50 marks for in-semester assessments), and**
  - **an overall unit mark of 50% or more.**
- If a student does not pass these hurdles then a mark of no greater than **45N** will be recorded for the unit.

# Submission of Assignments

- Submission details will be specified on each assignment sheet
  - You will submit your assignments to Moodle.
  - Late submission will incur a penalty of 10% **of the total** per day
  - Assignments submitted 7 calendar days after the due date will receive 0
- Extensions
  - Extensions up to 5 days can be granted at the discretion of the CE
  - Use this google form: <https://forms.gle/cndtb5QFJs1i3cF8A>
  - For longer extensions, you need to apply through Monash connect:
  - <https://www.monash.edu/connect/forms/modules/course/special-consideration>

# Cheating, Collusion, Plagiarism

---

- **Cheating:** Seeking to obtain an unfair advantage in an examination or in other written or practical work required to be submitted or completed for assessment.
- **Collusion:** Unauthorised collaboration on assessable work with another person or persons.
- **Plagiarism:** To take and use another person's ideas and or manner of expressing them and to pass them off as one's own by failing to give appropriate acknowledgement. This includes material from any source, staff, students or the Internet – published and un-published works.

<http://infotech.monash.edu.au/resources/student/assignments/policies.html>

# How to avoid collusion for FIT2004

- Do not discuss the assignment with other students
- It is possible to discuss the assignment in a way which is not collusion but...
- Most academic integrity cases I review turn out to be:
  - I was trying to help my friend
  - I gave them my code so they would have the right idea, I didn't think they would use it
  - I let them look at my code, but they took a photo of it and copied it
  - Etc

## What can you do?

- Help people understand the task, and understand what they would need to do to solve it
- Share test cases! This is encouraged, feel free to post your test cases on the forums and to use other people's and give them feedback
- If you need help, **come to consultations**
- **The consultation timetable is on Moodle, and online consultations are available for off-campus students**

# Short break

---





# Part 2: Outline

---

- Proving Correctness of Algorithms
  - Finding Minimum
  - Binary Search
- Complexity Analysis (Recap)
- Solving Recurrence Relations

# Algorithmic Analysis

---

In algorithmic analysis, one is interested in (at least) two things:

- An algorithm's correctness.
  - The amount of resources used by the algorithm
- 
- In this lecture we will see how to prove correctness.
  - Next week, we will analyse the resources used by the algorithm, a.k.a complexity analysis.

# Proving correctness of algorithms

- Commonly, we write programs and then test them.
- Testing detects inputs for which the program has incorrect output
- There are infinitely many possible inputs (for most programs) so we cannot test them all, therefore...
- Testing cannot guarantee that the program is always correct!
- Logic **can** prove that a program is always correct. This is usually achieved in two parts:
  1. Show that the program always terminates, and
  2. Show that a program produces correct results when it terminates

# Part 2: Outline

---

- Proving Correctness of Algorithms
  - Finding Minimum
  - Binary Search
- Complexity Analysis (Recap)
- Solving Recurrence Relations

# Finding minimum value

```
//Find minimum value in an unsorted array of N>0 elements
min = array[1] //in this unit, we assume index starts at 1
index = 2

while index <= N

    if array[index] < min
        min = array[index]
    index = index + 1

return min
```

# Does it always terminate?

```
//Find minimum value in an unsorted array of N>0 elements
min = array[1]
index = 2

while index <= N

    if array[index] < min
        min = array[index]
    index = index + 1

return min
```

# Correct result at termination?

```
//Find minimum value in an unsorted array of N>0 elements
min = array[1]
index = 2

while index <= N

    if array[index] < min
        min = array[index]
    index = index + 1

return min
```

# Correctness using Loop Invariant

- Invariant should be true at three points:
  1. Before the loop starts (initialisation)
  2. During each loop (maintenance)
  3. After the loop terminates (termination)
- The easiest way to do this is often to consider the invariant just **before** the loop condition is checked



# Correctness using Loop Invariant

```
min = array[1]
index = 2
//LI: min equals the minimum value in array[1 .. index - 1]
while index <= N
    if array[index] < min
        min = array[index]
    index = index + 1
return min
```

## Initialisation

- `min = array[1]`
- `index = 2`
- `array[1 .. index - 1] = array[1...1]`

As required

# Correctness using Loop Invariant

```
min = array[1]
index = 2
//LI: min equals the minimum value in array[1 ... index - 1]
while index <= N
    if array[index] < min
        min = array[index]
    index = index + 1
return min
```

## Maintenance

- If LI was true when `index` was  $k$ , is it still true when `index` is  $k+1$ ?
- To prove this, examine the code in the body of the loop and reason about the invariant.
- Remember to use the assumption that LI was true when `index` was  $k$

# Correctness using Loop Invariant

```
min = array[1]
index = 2
//LI: min equals the minimum value in array[1 ... index - 1]
while index <= N
    if array[index] < min
        min = array[index]
    index = index + 1
return min
```

## Termination

- We have shown that LI is true for all values of `index` (in the previous step)
- Set `index` to the value which causes termination ( $N+1$ )
- Check if the statement of LI with this `index` value is what we want
- //LI: min equals the minimum value in array[1 ... N] as required

See lecture 1 appendix (week 1) for complete proof

# Part 2: Outline

---

- Proving Correctness of Algorithms
  - Finding Minimum
  - Binary Search
- Complexity Analysis (Recap)
- Solving Recurrence Relations

# A harder problem

```
lo = 1 (we are 1 indexed here)
```

```
hi = N
```

```
while ( lo <= hi )
```

```
    mid = floor( (lo+hi)//2 )
```

```
    if target > array[mid]
```

```
        lo=mid+1
```

```
    else
```

```
        hi=mid-1
```

```
    else
```

```
        return mid
```

```
Return False
```

Always terminates (why?)



Finds **an** instance of target, if it exists

We don't know **which** one it will find

Let's solve the problem of finding the **rightmost** instance of target

# Rightmost instance



 = "active" subarray  
 = target items



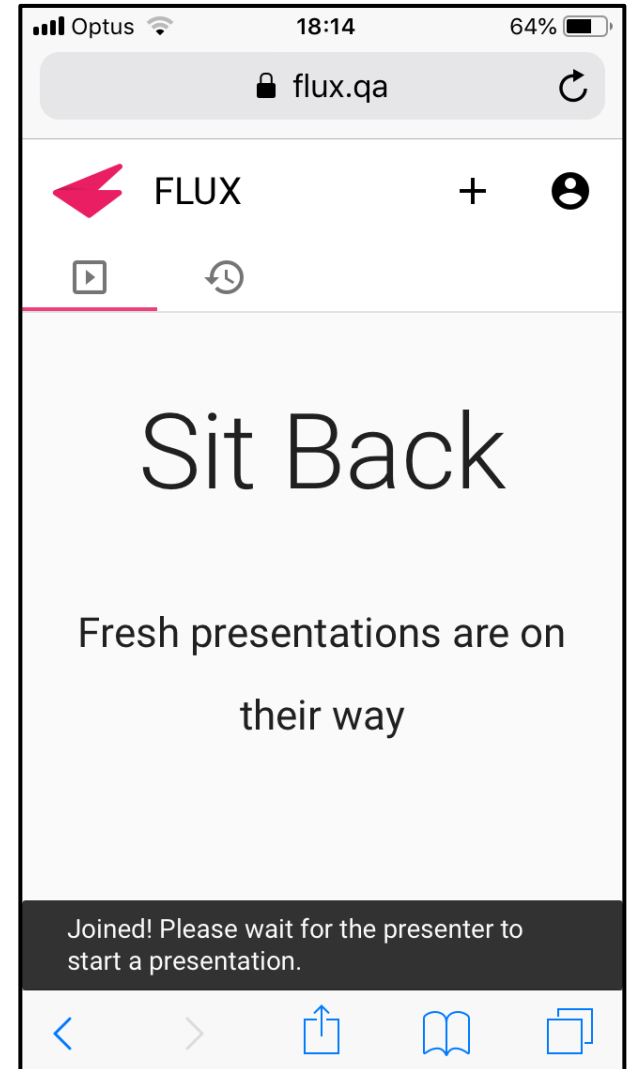
# Binary Search Invariant

5	10	15	20	25	30	35	40	45	50
1	2	3	4	5	6	7	8	9	10

- Invariant must relate to relevant variables (`lo`, `hi`, `array`)
- At termination, the invariant should tell us that the algorithm works
- Example: `min` is the minimum of `array[1...N]`

# Quiz time

1. Visit <https://flux.qa>
2. Log in (your Authcate details)
3. Touch the + symbol
4. Enter the audience code **SUK7DX**
5. Answer questions





# Binary Search Invariant

- One possible invariant:
  - Rightmost instance is between  $lo$  (inclusive) and  $hi$  (exclusive)
- Is this true at the start?
- Yes, if it exists...
- Invariant: if target is in the list, the rightmost instance is in  $list[lo...hi-1]$

# Rightmost instance

- We want to move **lo** when the rightmost instance of target is to the right of mid
- We want to move **hi** when the rightmost instance of target is to the left of mid
- We want to move **lo** when a target **is at** mid (since we are looking for the rightmost)
- This suggest we only have 2 cases,
  - $t < L[mid]$
  - $t \geq L[mid]$

# Design from Invariant

Invariant: If key is in L, it is in L[lo...hi-1]

```
lo = 1
```

```
hi = N+1
```

```
while ( lo ? hi )
```

When does the “active” section of the list have one element?

i.e. when is L[lo...hi-1] a single item?

When  $lo = hi-1$  (or  $lo \geq hi-1$ )

The while condition should be the opposite

i.e. we should continue the loop as long as there the active section has more than one item

While  $lo < hi-1$

# Design from Invariant

Invariant: If key is in L, it is in  $L[lo \dots hi-1]$

```
lo = 1
```

```
hi = N+1
```

```
while ( lo < hi - 1 )
```

When does the “active” section of the list have one element?

i.e. when is  $L[lo \dots hi-1]$  a single item?

When  $lo = hi-1$  (or  $lo \geq hi-1$ )

The while condition should be the opposite

i.e. we should continue the loop as long as there the active section has more than one item

While  $lo < hi-1$

# Design from Invariant

Invariant: If key is in L, it is in  $L[lo \dots hi-1]$

```
lo = 1
hi = N+1
while ( lo < hi - 1 )
    mid = (lo + hi) / 2
    if t ≥ L[mid]
```

Where should we move lo to?

We know that the element we are looking for is **at or to the right of** mid

We should set  $lo=mid$ , since lo is inclusive

# Design from Invariant

Invariant: If key is in L, it is in  $L[lo \dots hi-1]$

```
lo = 1
hi = N+1
while ( lo < hi - 1 )
    mid = (lo + hi) / 2
    if t ≥ L[mid]
        lo = mid
```

Where should we move lo to?

We know that the element we are looking for is **at or to the right of** mid

We should set  $lo=mid$ , since lo is inclusive

# Design from Invariant

Invariant: If key is in L, it is in  $L[\text{lo} \dots \text{hi}-1]$

```
lo = 1
hi = N+1
while ( lo < hi - 1 )
    mid = (lo + hi) / 2
    if t ≥ L[mid]
        lo = mid
    if t < L[mid]
        hi = mid
```

Where should we move hi to?

Remember,  $L[\text{hi}]$  is excluded from the active range

We can move hi to mid, since we know  $L[\text{mid}]$  is not what we are looking for

# Design from Invariant

Invariant: If key is in L, it is in  $L[lo \dots hi-1]$

```
lo = 1
hi = N+1
while ( lo < hi - 1 )
    mid = (lo + hi) / 2
    if t ≥ L[mid]
        lo = mid
    if t < L[mid]
        hi = mid
```

When the loop ends,  $lo = hi-1$ .

The only element in the active range is  $L[lo]$

This means that either  $L[lo]$  is what we are looking for, or it isn't



# Design from Invariant

Invariant: If key is in L, it is in  $L[lo \dots hi-1]$

```
lo = 1
hi = N+1
while ( lo < hi - 1 )
    mid = (lo + hi) / 2
    if t ≥ L[mid]
        lo = mid
    if t < L[mid]
        hi = mid
if L[lo] == t
    return lo
return False
```

When the loop ends,  $lo = hi-1$ .

The only element in the active range is  $L[lo]$

This means that either  $L[lo]$  is what we are looking for, or it isn't

Note that this algorithm improves over the “standard” 3-check algorithm since it does less work each iteration

# Design from Invariant

Invariant: If key is in L, it is in L[lo...hi-1]

```
lo = 1
hi = N+1
while ( lo < hi - 1 )
    mid = (lo + hi) / 2
    if t ≥ L[mid]
        lo = mid
    if t < L[mid]
        hi = mid
if L[lo] == t
    return lo
return False
```

Try to think about what would happen if we **included** hi (i.e. we try the invariant  
If key is in L, it is in L[lo...hi])

The issue is in the while condition.

It would need to be while (lo < hi)

What happens when lo = hi - 1 and  
t ≥ L[mid]?

# Algorithm for Binary Search

Invariant: If key is in L, it is in  $L[lo \dots hi-1]$

```
BinarySearch(L[1..n], t)
```

```
  lo = 1
```

```
  hi = N+1
```

```
  while ( lo < hi - 1 )
```

```
    mid = (lo + hi) / 2
```

```
    if t ≥ L[mid]
```

```
      lo = mid
```

```
    if t < L[mid]
```

```
      hi = mid
```

```
  if L[lo] == t
```

```
    return lo
```

```
  return False
```

Is this algorithm correct?

To prove correctness, we need to show that

1. it **always** terminates, and
2. it returns correct result when it terminates

# Correctness using Loop Invariant

Invariant: If key is in L, it is in  $L[lo \dots hi-1]$

```
BinarySearch(L[1..n], t)
```

```
  lo = 1
```

```
  hi = N+1
```

```
  while ( lo < hi - 1 )
```

```
    mid = (lo + hi) / 2
```

```
    if t ≥ L[mid]
```

```
      lo = mid
```

```
    if t < L[mid]
```

```
      hi = mid
```

```
  if L[lo] == t
```

```
    return lo
```

```
  return False
```

## Always terminates

- Algorithm terminates either when  $lo \geq hi - 1$
- Because of the while condition, each iteration, after mid is calculated,  $lo < mid$ ,  $mid < hi$  (why?)
- When we set lo to mid, lo must therefore increase by at least 1
- Similarly, when we set hi to mid, hi must decrease by at least 1
- So every iteration, either lo increases or hi decreases.
- Eventually,  $lo \geq hi-1$

# Correctness using Loop Invariant

Invariant: If key is in L, it is in  $L[lo \dots hi-1]$

```
BinarySearch(L[1..n], t)
```

```
  lo = 1
```

```
  hi = N+1
```

```
  while ( lo < hi - 1 )
```

```
    mid = (lo + hi) / 2
```

```
    if t ≥ L[mid]
```

```
      lo = mid
```

```
    if t < L[mid]
```

```
      hi = mid
```

```
  if L[lo] == t
```

```
    return lo
```

```
  return False
```

## Initialisation

- If rightmost target is in L, then  
target is in  $L[lo \dots hi-1] =$   
 $L[1 \dots N] = L$

As required

# Correctness using Loop Invariant

Invariant: If key is in L, it is in  $L[lo...hi-1]$

```
BinarySearch(L[1..n], t)
```

```
  lo = 1
```

```
  hi = N+1
```

```
  while ( lo < hi - 1 )
```

```
    mid = (lo + hi) / 2
```

```
    if t ≥ L[mid]
```

```
      lo = mid
```

```
    if t < L[mid]
```

```
      hi = mid
```

```
  if L[lo] == t
```

```
    return lo
```

```
  return False
```

## Maintenance

- Assume target in  $L[lo...hi-1]$  before some loop iteration
- If  $t \geq L[mid]$ , then it must be located in the range  $L[mid...hi-1]$ , so we should set `lo` to `mid`
- If  $t < L[mid]$ , then it must be in the range  $L[lo...mid-1]$ , so we should set `hi` to `mid` (since  $L[hi]$  is excluded)
- Since this is exactly what we do, at the start of the next iteration the invariant is still true

# Correctness using Loop Invariant

Invariant: If key is in L, it is in  $L[lo...hi-1]$

```
BinarySearch(L[1..n], t)
```

```
  lo = 1
```

```
  hi = N+1
```

```
  while ( lo < hi - 1 )
```

```
    mid = (lo + hi) / 2
```

```
    if t ≥ L[mid]
```

```
      lo = mid
```

```
    if t < L[mid]
```

```
      hi = mid
```

```
  if L[lo] == t
```

```
    return lo
```

```
  return False
```

## Termination

- $lo == hi - 1$ . Since the invariant has been correctly maintained...
- Rightmost target must be in  $L[lo...hi-1]$ , or it is not in the list
- $L[lo...hi-1]$  is a single element,  $L[lo]$ .
- If the element is found at  $L[lo]$ , then return that index
- Otherwise return False
- See lecture 1 appendix (week 1) for complete proof

# Part 2: Outline

---

- Proving Correctness of Algorithms
  - Finding Minimum
  - Binary Search
- Complexity Analysis (Recap)
- Solving Recurrence Relations



# Quiz time

---

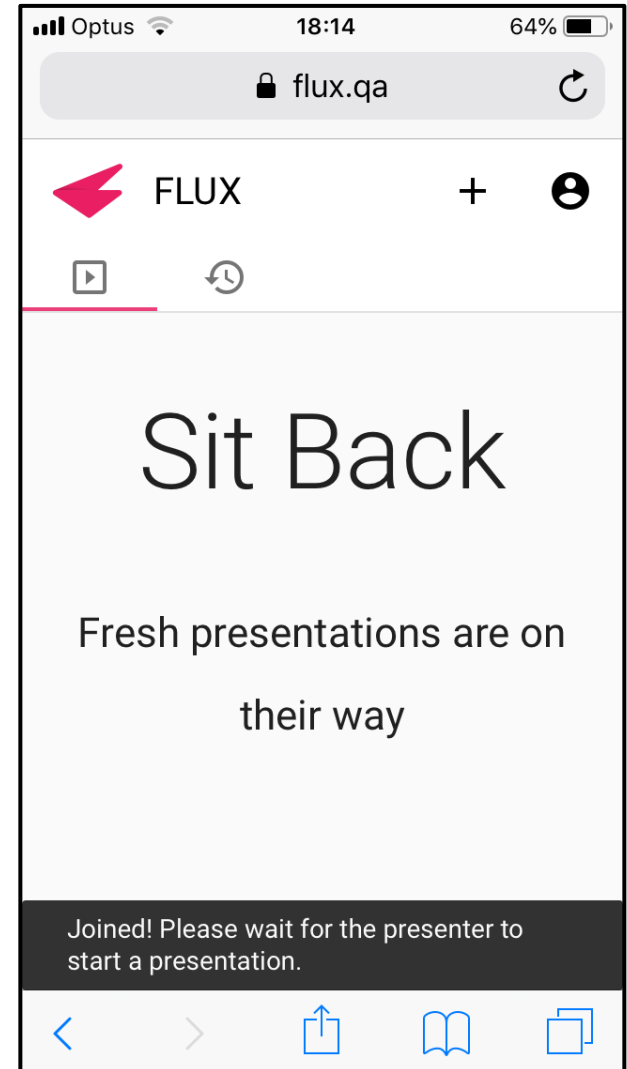
What is the complexity of the following code?

```
If x == y:  
    Return True
```

Think carefully about the most accurate answer

# Quiz time

1. Visit <https://flux.qa>
2. Log in (your Authcate details)
3. Touch the + symbol
4. Enter the audience code **I12NCB**
5. Answer questions



# Complexity Analysis

- Time complexity
  - The amount of time taken by an algorithm to run as a function of the input size
- Space complexity
  - The amount of space taken by an algorithm to run as a function of the input size
  - This is further divided into total space and auxiliary space, which will be discussed later
- Worst-case complexity
- Best-case complexity
- Average-case complexity (we don't discuss this much)

# Part 2: Outline

---

- Proving Correctness of Algorithms
  - Finding Minimum
  - Binary Search
- Complexity Analysis (Recap)
- Solving Recurrence Relations

# Recurrence Relations

A recurrence relation is an equation that recursively defines a sequence of values, and one or more initial terms are given.

E.g.,

$$T(1) = b$$

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

- Complexity of recursive algorithms can be analysed by writing its recurrence relation and then solving it

# Solving Recurrence Relations

// Compute Nth power of x

power(x,N)

{

if (N==0)

return 1

if (N==1)

return x

else

return x \* power(x, N-1)

}

## Time Complexity

Cost when  $N = 1$ :  $T(1) = b$  (b&c are constant)

Cost for general case:  $T(N) = T(N-1) + c$  (A)

Cost for N-1:  $T(N-1) = T(N-2) + c$

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

$T(N) = (T(N-2) + c) + c = T(N-2) + 2*c$  (B)

Cost for N-2:  $T(N-2) = T(N-3) + c$

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

$T(N) = T(N-3) + c+c+c = T(N-3) + 3*c$

Our goal is to reduce this term to  $T(1)$

Do you see the pattern?

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

# Solving Recurrence Relations

```
// Compute Nth power of x
power(x,N)
{
    if (N==0)
        return 1
    if (N==1)
        return x
    else
        return x * power(x, N-1)
}
```

## Time Complexity

Cost when  $N = 1$ :  $T(1) = b$  ( $b$  &  $c$  are constant)

Cost for general case:  $T(N) = T(N-1) + c$  (A)

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

Find the value of  $k$  such that  $N-k = 1 \rightarrow k = N-1$

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

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

Hence, the complexity is  $O(N)$

# Solving Recurrence Relations

```
// Compute Nth power of x
power(x,N)
{
    if (N==0)
        return 1
    if (N==1)
        return x
    else
        return x * power(x, N-1)
}
```

## Time Complexity

Cost when  $N = 1$ :  $T(1) = b$  ( $b$  &  $c$  are constant)

Cost for general case:  $T(N) = T(N-1) + c$  (A)

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

Check by substitution:

$$\begin{aligned} T(N-1) + c &= c * (N-1) + b - c + c \\ &= c * (N-1) + b \\ &= c * N + b - c \\ &= T(N) \end{aligned}$$

As required

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

As required



# Solving Recurrence Relations

```
// Compute Nth power x
power2(x,N)
{
    if (N==0)
        return 1
    if (N==1)
        return x
    if (N is even)
        return power2( x * x, N/2 )
    else
        return power2( x * x, N/2 ) * x
}
```

## Time Complexity

Cost when  $N = 1$ :  $T(1) = b$  ( $b$  &  $c$  are constant)

Cost for general case:  $T(N) = T(N/2) + c$  (A)

Now you try!

Cost for  $N/2$ :  $T(N/2) = T(N/4) + c$

Replacing  $T(N/2)$  in (A)

$T(N) = T(N/4) + c + c = T(N/4) + 2*c$  (B)

Cost for  $N/4$ :  $T(N/4) = T(N/8) + c$

Replacing  $T(N/4)$  in (B)

$T(N) = T(N/8) + c + c + c = T(N/8) + 3*c$

Do you see the pattern?

$T(N) = T(N/2^k) + k*c$

# Solving Recurrence Relations

```
// Compute Nth power x
power2(x,N)
{
    if (N==0)
        return 1
    if (N==1)
        return x
    if (N is even)
        return power2( x * x, N/2 )
    else
        return power2( x * x, N/2 ) * x
}
```

## Time Complexity

Cost when  $N = 1$ :  $T(1) = b$  ( $b$  &  $c$  are constant)

Cost for general case:  $T(N) = T(N/2) + c$  (A)

$$T(N) = T(N/2^k) + k*c$$

Find the value of  $k$  such that  $N/2^k = 1 \rightarrow k = \log_2 N$

$$T(N) = T(N/2^{\log_2 N}) + c * \log N = T(1) + c * \log_2 N$$

$$T(N) = b + c * \log_2 N$$

Hence, the complexity is  $O(\log N)$

# Solving Recurrence Relations

```
// Compute Nth power x
power2(x,N)
{
    if (N==0)
        return 1
    if (N==1)
        return x
    if (N is even)
        return power2( x * x, N/2)
    else
        return power2( x * x, N/2 ) * x
}
```

## Time Complexity

Cost when  $N = 1$ :  $T(1) = b$  ( $b$  &  $c$  are constant)

Cost for general case:  $T(N) = T(N/2) + c$  (A)

$$T(N) = b + c * \log_2 N$$

Check by substitution:

$$\begin{aligned} T(N/2) + c &= b + c * \log_2 (N/2) + c \\ &= b + c * [\log_2 (N) - \log_2 (2)] + c \\ &= b + c * \log_2 (N) \end{aligned}$$

As required

$$T(1) = b + c * \log_2 1 = b + c * 0 = b$$

As required

# Recurrence and complexity

---

Recurrence relation:

$$T(N) = T(N/2) + c$$

$$T(1) = b$$

Example algorithm?

Binary search

Solution:

$$O(\log N)$$

# Recurrence and complexity

Recurrence relation:

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

$$T(1) = b$$

Example algorithm?

Linear search

Solution:

$$O(N)$$

# Recurrence and complexity

Recurrence relation:

$$T(N) = 2 * T(N/2) + c * N$$

$$T(1) = b$$

Example algorithm?

Merge sort

Solution:

$$O(N \log N)$$

# Recurrence and complexity

Recurrence relation:

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

$$T(1) = b$$

Example algorithm?

Selection sort

Solution:

$$O(N^2)$$

# Recurrence and complexity

Recurrence relation:

$$T(N) = 2 * T(N-1) + c$$

$$T(0) = b$$

Example algorithm?

Naïve recursive Fibonacci

Solution:

$$O(2^N)$$



# Revision: Proof by induction

## 2 parts

1. Prove the base case, e.g., for the first state
2. **Assume** the proof holds for a state **k**. **Show** that it also holds for the next state **k+1**.

**We want to prove:**  $\text{something}(n) = \text{something\_else}(n)$

- **Step 0:** give a name to the left and right of statement you wish to prove (L, R)
- **Step 1:** For the smallest value **b** for which the statement must hold, Check that  $L(b) = R(b)$
- **Step 2:** Assume  $L(k) = R(k)$  for some  $k \geq b$
- **Step 3:** Using the assumption from step 2, prove  $L(k+1) = R(k+1)$
- **Step 4:** State that  $L(n) = R(n)$  for all  $n \geq b$

# Revision: Proof by induction

Prove that  $1+2+3+\dots+n = n(n+1)/2$  for all  $n \geq 1$

- **Step 0:** give a name to the left and right of statement you wish to prove (L, R)
- **Step 1:** For the smallest value  $b$  for which the statement must hold, Check that  $L(b) = R(b)$
- **Step 2:** Assume  $L(k) = R(k)$  for some  $k \geq b$
- **Step 3:** Using the assumption from step 2, prove  $L(k+1) = R(k+1)$
- **Step 4:** State that  $L(n) = R(n)$  for all  $n \geq b$

Let  $L(n) = 1+2+3+\dots+n$ ,  $R(n) = n(n+1)/2$

# Revision: Proof by induction

Prove that  $1+2+3+\dots+n = n(n+1)/2$  for all  $n \geq 1$

- **Step 0:** give a name to the left and right of statement you wish to prove (L, R)
- **Step 1:** For the smallest value  $b$  for which the statement must hold, Check that  $L(b) = R(b)$
- **Step 2:** Assume  $L(k) = R(k)$  for some  $k \geq b$
- **Step 3:** Using the assumption from step 2, prove  $L(k+1) = R(k+1)$
- **Step 4:** State that  $L(n) = R(n)$  for all  $n \geq b$

Let  $L(n) = 1+2+3+\dots+n$ ,  $R(n) = n(n+1)/2$

**Base Case:**  $n = 1$ . WTS  $L(1) = R(1)$

$L(1) = 1$ ,  $R(1) = 1 \cdot (2) / 2 = 1$ .  $L(1) = R(1)$  as required

# Revision: Proof by induction

Prove that  $1+2+3+\dots+n = n(n+1)/2$  for all  $n \geq 1$

- **Step 0:** give a name to the left and right of statement you wish to prove (L, R)
- **Step 1:** For the smallest value  $b$  for which the statement must hold, Check that  $L(b) = R(b)$
- **Step 2:** Assume  $L(k) = R(k)$  for some  $k \geq b$
- **Step 3:** Using the assumption from step 2, prove  $L(k+1) = R(k+1)$
- **Step 4:** State that  $L(n) = R(n)$  for all  $n \geq b$

Let  $L(n) = 1+2+3+\dots+n$ ,  $R(n) = n(n+1)/2$

**Base Case:**  $n = 1$ . WTS  $L(1) = R(1)$

$L(1) = 1$ ,  $R(1) = 1 \cdot (2) / 2 = 1$ .  $L(1) = R(1)$  as required

**Inductive step:** Assume  $L(k) = R(k)$ .

# Revision: Proof by induction

Prove that  $1+2+3+\dots+n = n(n+1)/2$  for all  $n \geq 1$

- **Step 0:** give a name to the left and right of statement you wish to prove (L, R)
- **Step 1:** For the smallest value b for which the statement must hold, Check that  $L(b) = R(b)$
- **Step 2:** Assume  $L(k) = R(k)$  for some  $k \geq b$
- **Step 3:** Using the assumption from step 2, prove  $L(k+1) = R(k+1)$
- **Step 4:** State that  $L(n) = R(n)$  for all  $n \geq b$

Let  $L(n) = 1+2+3+\dots+n$ ,  $R(n) = n(n+1)/2$

**Base Case:**  $n = 1$ . WTS  $L(1) = R(1)$

$L(1) = 1$ ,  $R(1) = 1 \cdot (2) / 2 = 1$ .  $L(1) = R(1)$  as required

**Inductive step:** Assume  $L(k) = R(k)$ .

WTS  $L(k+1) = R(k+1)$

$$\begin{aligned} L(k+1) &= 1+2+3+\dots+k+(k+1) \\ &= L(k) + (k+1) \\ &= R(k) + (k+1) \text{ by assumption} \\ &= k(k+1)/2 + (k+1) \\ &= [k(k+1) + 2(k+1)] / 2 \\ &= (k+2)(k+1)/2 = R(k+1) \text{ as required} \end{aligned}$$

# Revision: Proof by induction

Prove that  $1+2+3+\dots+n = n(n+1)/2$  for all  $n \geq 1$

- **Step 0:** give a name to the left and right of statement you wish to prove (L, R)
- **Step 1:** For the smallest value b for which the statement must hold, Check that  $L(b) = R(b)$
- **Step 2:** Assume  $L(k) = R(k)$  for some  $k \geq b$
- **Step 3:** Using the assumption from step 2, prove  $L(k+1) = R(k+1)$
- **Step 4:** State that  $L(n) = R(n)$  for all  $n \geq b$

Let  $L(n) = 1+2+3+\dots+n$ ,  $R(n) = n(n+1)/2$

**Base Case:**  $n = 1$ . WTS  $L(1) = R(1)$

$L(1) = 1$ ,  $R(1) = 1 \cdot (2) / 2 = 1$ .  $L(1) = R(1)$  as required

**Inductive step:** Assume  $L(k) = R(k)$ .

WTS  $L(k+1) = R(k+1)$

$L(k+1) = 1+2+3+\dots+k+(k+1)$

$= L(k) + (k+1)$

$= R(k) + (k+1)$  by assumption

$= k(k+1)/2 + (k+1)$

$= [k(k+1) + 2(k+1)] / 2$

$= (k+2)(k+1)/2 = R(k+1)$  as required

Therefore  $1+2+3+\dots+n = n(n+1)/2$  for all  $n \geq 1$

# Revision: Proof by induction

## Theorem

Prove that  $1+2+3+\dots+n = n(n+1)/2$  for all  $n \geq 1$

Step 0: Let  $L(n) = 1+2+3+\dots+n$ ,  $R(n) = n(n+1)/2$

Step 1: Base Case:  $n = 1$ . WTS  $L(1) = R(1)$

$L(1) = 1$ ,  $R(1) = 1*(2)/2 = 1$ .  $L(1) = R(1)$  as required

Step 2: Inductive step: Assume  $L(k) = R(k)$ .

Step 3: WTS  $L(k+1) = R(k+1)$

$$\begin{aligned} L(k+1) &= 1+2+3+\dots+k+(k+1) \\ &= L(k) + (k+1) \\ &= R(k) + (k+1) \text{ by assumption} \\ &= k(k+1)/2 + (k+1) \\ &= [k(k+1) + 2(k+1)] / 2 \\ &= (k+2)(k+1)/2 = R(k+1) \text{ as required} \end{aligned}$$

Step 4: Therefore  $1+2+3+\dots+n = n(n+1)/2$  for all  $n \geq 1$

For more details, watch it on Khan Academy ([click here](#))

# Concluding Remarks

## Summary

- This unit demands your efforts from week 1
- Testing cannot guarantee correctness; Requires logical reasoning to formally prove correctness

## Coming Up Next

- Analysis of algorithms
- Non-comparison based sorting (Counting Sort, Radix Sort) – related to Assignment 1

## **IMPORTANT: Preparation required before the next lecture**

- Revise computational complexity covered in earlier units (FIT1045, FIT1008). You may also want to watch [videos](#) or read [other online resources](#)
- **Complete tute 1 (in your own time) and assessed prep for tute 2**