Faculty of Information Technology, Monash University

FIT2004: Algorithms and Data Structures

Week 1: Introduction, and Proof of Correctness

These slides are prepared by M. A. Cheema and are based on the material developed by 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

- The subject is about problem-solving with computers: algorithms and data structures.
- 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

You must take this unit seriously and work diligently.

- The subject is arguably the most important for computer and technology related careers
  - Big companies (e.g., Google, Microsoft, Facebook etc.) actively hunt for people good at algorithms and data structures
  - 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

- This unit is **CHALLENGING**
  - 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 tutorials will require double the efforts to recover

Good News: If you work diligently, you will learn a lot and enjoy this unit

Overview of the content

- General problem-solving strategies and techniques, e.g.
  - Useful paradigms, e.g.
    - dynamic programming
    - divide and conquer, etc.
  - Analysis of algorithms and data structures, e.g.
    - program proof / correctness
    - analysis and estimation of space and time complexity, etc.

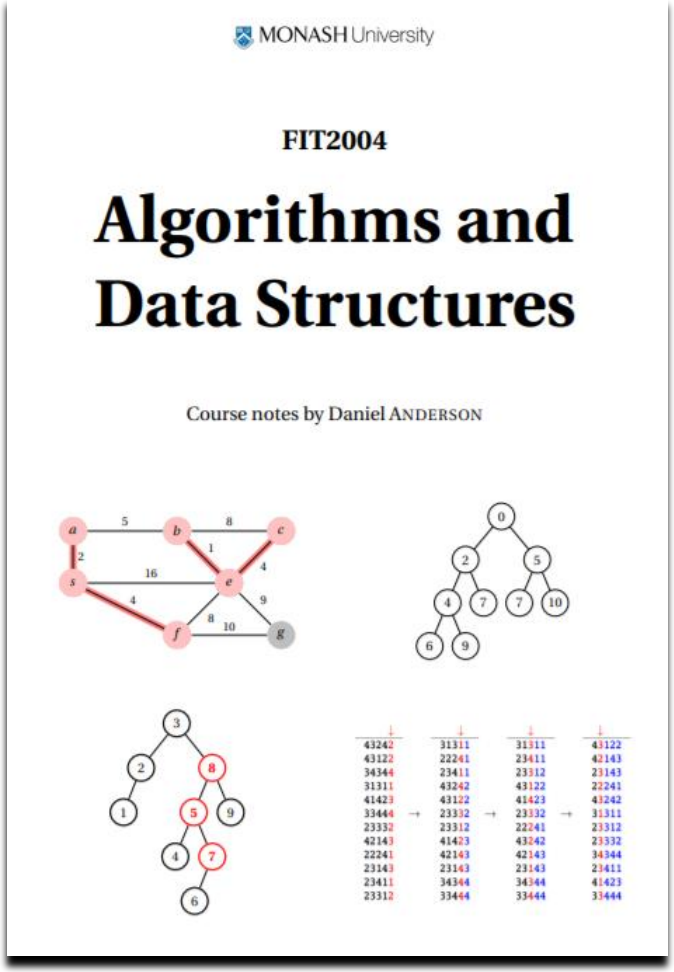
- A selection of important computational problems, e.g.
  - sorting
  - retrieval/searching, etc.

- 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 the bottom of the page
  - Click on “Lecture Notes”



FIT2004 Staff

- **Chief examiner:** Nathan Companez
- **Lecturer:** Nathan Companez
- **Malaysia lecturer:** Dr. Wern Han Lim

- **Tutors:**

  - Chaluka Salgado
  - Kiana Zeighami
  - Steven Fan
  - Shams Rahman
  - Saman Ahmadi
  - Tharindu Warnakula
  - Shahid Iqbal
  - Ammar Sohail
  - Steve Shahbazi
  - Nathan Companez

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
  - Unit Notes
  - Tutorial sheets

- Remember(!) to keep following the forum posts, they contain
  - Assignment amendments
  - Lecture error correction
  - Changes to class time
  - Changes to consultation time

Course Structure

- Lecture 2hrs (Monday 9:00 to 11:00, CL_16Rnf/S4)
  - Live Streaming!
  - All classes start at :00 and finish at :50
- Tutorial 3hrs (see allocate+)
  - No tutorial in week 01
  - Off-campus students can watch the weekly tutorial walkthrough video

Asking Questions During Lectures

- Please do it (you can use flux)
- 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

Assessment Summary

- Prac assignments 1-4 (week 4, 7, 9 and 12) 30%
- Tutorial participation 10%
- Final Exam 60%

Prac Assignments 1-4 (30%)

- Four practical assignments (each worth 7.5%)
  - Due: Week 4, 7, 9, 12
- 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!!!

Tutorial Participation (10%)

- We have 3 hrs weekly tutorials starting from week 2
  - Tutorial sheet for week 1 is uploaded and you are expected to complete it in your own time
- Each tutorial is worth 1 mark . There are 11 weeks. So the best 10 tutorials will be counted. (total marks capped at 10).
- In each lab/tutorial:
  - Hurdle: (no marks awarded if hurdle not met)
    - You must answer questions in the section (assessed preparation) in your Tutorial sheet before the class starts. You cannot work on this during class
  - Active participation:
    - Actively engaging in discussing the tutorials in-class
    - 1 mark awarded if hurdle met **and** actively participated (not applicable to off campus students)

Final Exam (60%)

- 2 hours + 10 minutes reading time

Hurdles

- To pass this unit a student must obtain:
  - **40% or more in the unit's final exam (i.e., at least 24 out of 60 marks for final exam), and**
  - **40% or more in the unit's total non-examination assessment (i.e., at least 16 out of total 40 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 **49N** will be recorded for the unit.

Submission of Assignments

- Submission details will be specified on each assignment/laboratory sheet
  - You will submit your assignments to Moodle.
  - Whenever you submit an assignment you must complete an assignment submission form.
  - Late submission will have 10% off the total assignment marks per day (including weekends).
  - Assignments submitted 5 days after the due date will normally not be accepted.
- Extensions
  - Genuine and compelling reasons only.
  - Supporting documentations are needed BEFORE any consideration would be given.
  - Submit spec. con. BEFORE deadline
  - (fit2004.allcampuses-x@monash.edu)

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

- Definition of collusion is different in each unit
- E.g. if group work is required, then group work is allowed

For this unit, avoid the following

- Writing down code while talking about the assignment
- Looking at people's screens/code
- Giving other people your code
- Tell someone which algorithm to use

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**

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

Cheating, Collusion, Plagiarism

MOSS

/home/ubuntu/Projects/work/2015/uct-csc1010h/tutorials/4/raw/[redacted]/ (68%)		/home/ubuntu/Projects/work/2015/uct-csc1010h/tutorials/4/raw/[redacted]/ (73%)	
4-71		2-66	
95-111		90-106	
74-91		69-86	
115-132		110-127	

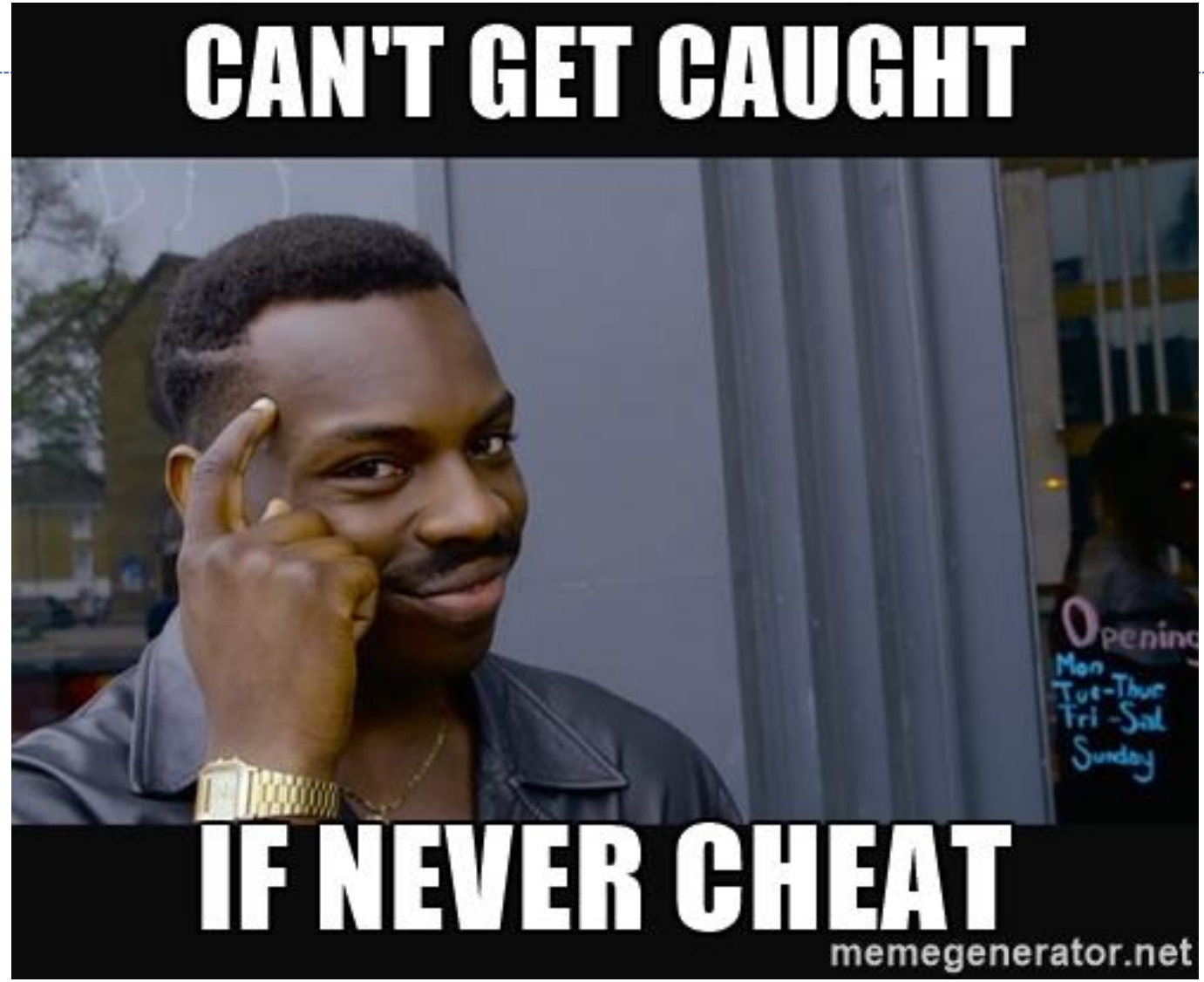
/home/ubuntu/Projects/work/2015/uct-csc1010h/tutorials/4/raw/[redacted]

```
>>>> file: LongJump.py
#S[redacted]a

print("***** Long Jump Information System *****")
print("Please enter the names of competitors. (Press return when done.)")
print("Competitor no. 1:")
competitor = input()
b,c,g,h,d,k = 1,0,0,0,[],0
maxi,competitors = [],[competitor]
while True:
    b += 1
    print("Competitor no. "+str(b)+":")
    competitor = input()
    if competitor == "":break
    else:
        competitors.append(competitor)
print("Please enter the distances for each competitor.")
for each in competitors:
    print("Competitor " each ":" sep="")
    at1 = input("Attempt 1:\n")
    at2 = input("Attempt 2:\n")
    at3 = input("Attempt 3:\n")
    x = (at1+at2+at3).lower()
    if (at1+at2+at3).find("oul") != -1:
        x = (at1+at2+at3).lower()
    d.append(at1)
    d.append(at2)
    d.append(at3)
    if x.find("foul") == -1:
        maxi.append(max(eval(at1),eval(at2),eval(at3)))
```

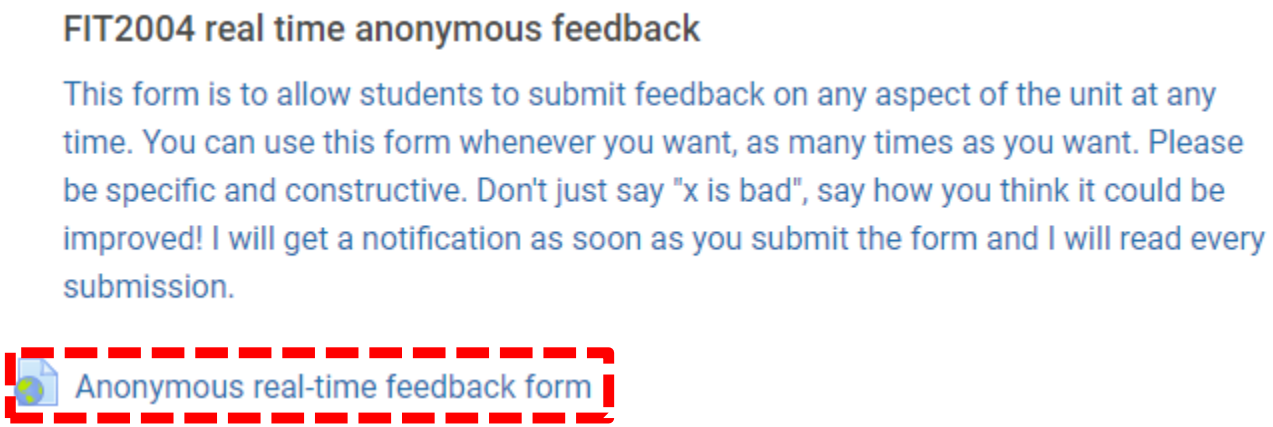
/home/ubuntu/Projects/work/2015/uct-csc1010h/tutorials/4/raw/[redacted]

```
>>>> file: LongJump.py

print("***** Long Jump Information System *****")
print("Please enter the names of competitors. (Press return when done.)")
print("Competitor no. 1:")
competitor = input()
b,c,g,h,d,k = 1,0,0,0,[],0
maximums,competitors = [],[competitor]
while True:
    b += 1
    print("Competitor no. "+str(b)+":")
    competitor = input()
    if competitor == "":break
    else:
        competitors.append(competitor)
print("Please enter the distances for each competitor.")
for each in competitors:
    print("Competitor " each ":" sep="")
    attempt1 = input("Attempt 1:\n")
    attempt2 = input("Attempt 2:\n")
    attempt3 = input("Attempt 3:\n")
    g = (attempt1+attempt2+attempt3).lower()
    if (attempt1+attempt2+attempt3).find("oul") != -1:
        g = (attempt1+attempt2+attempt3).lower()
    d.append(attempt1)
    d.append(attempt2)
    d.append(attempt3)
    if g.find("foul") == -1:
        maximums.append(max(eval(attempt1),eval(attempt2),eval(attempt3)))
    else:
        d.remove("foul")
    if not "foul" in d:
```

http://lightonphiri.org/wp-content/uploads/2015/09/moss_sample-initial_result-masked-021.png


CAN'T GET CAUGHT
IF NEVER CHEAT
memegenerator.net

Real-time anonymous feedback



- In the “Unit update” page on Moodle
- Google form to submit anonymous feedback any time
- I will receive a notification as soon as feedback is submitted
- Please submit only constructive and actionable feedback

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

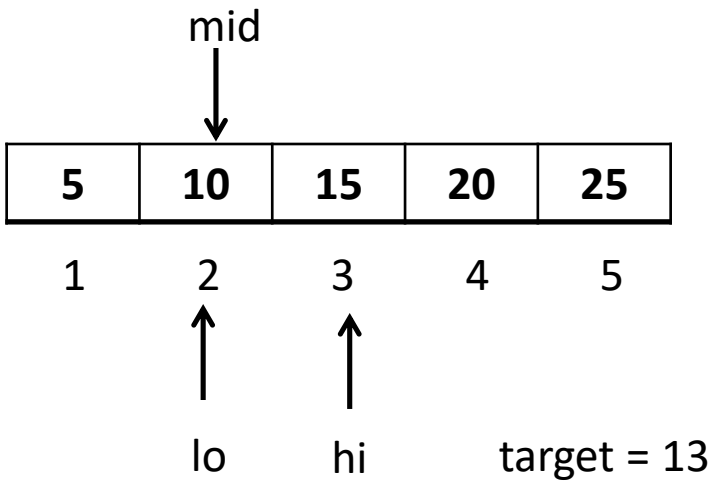
Part 2: Outline

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

Does this algorithm always terminate?

```
lo = 1
hi = N

while ( lo < hi )
    mid = floor( (lo+hi)//2 )
    if target >= array[mid]
        lo=mid
    else
        hi=mid

if array[lo] == target:
    return lo
return False
```

This algorithm may never terminate in **some** cases



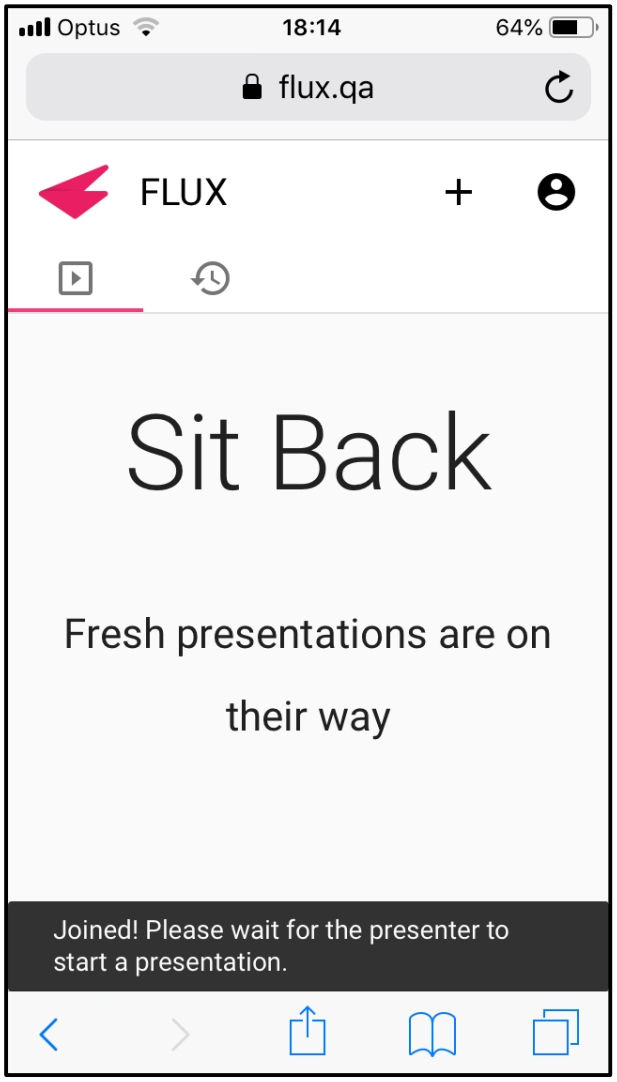
Binary Search Invariant

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 92A2WY
5. Answer questions



Binary Search Invariant

- Target item is between lo and hi (inclusive)
- Is this true at the start?
- Yes, if it exists…
- Invariant: If target is in the list, it is in list[lo…hi]

Algorithm for Binary Search

Invariant: If target is in the list, it is in list[lo…hi]

```
lo = 1
hi = N
while ( lo ? hi )
```

Algorithm for Binary Search

Invariant: If target is in the list, it is in list[lo…hi]

```
lo = 1
hi = N
while ( lo <= hi )
    mid = floor( (lo+hi)/2 )
    if key == array[mid]
        return mid
    if key >= array[mid]
        lo=mid+1
    else
        hi=mid-1
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 target is in the list, it is in list[lo…hi]

```
lo = 1
hi = N
while ( lo <= hi )
    mid = floor( (lo+hi)/2 )
    if key == array[mid]
        return mid
    if key >= array[mid]
        lo=mid+1
    else
        hi=mid-1
return False
```

Always terminates

- Algorithm terminates either when the key is found, or when lo > hi
- Each iteration, after mid is calculated, lo <= mid, mid <= hi
- When we set lo to mid+1, lo must therefore increase by at least 1
- Similarly, when we set hi to mid-1, hi must decrease by at least 1
- So every iteration, either lo increases or hi decreases.
- Eventually, lo passes hi

Correctness using Loop Invariant

Invariant: If target is in the list, it is in list[lo…hi]

```
lo = 1
hi = N
while ( lo <= hi )
    mid = floor( (lo+hi)/2 )
    if key == array[mid]
        return mid
    if key >= array[mid]
        lo=mid+1
    else
        hi=mid-1
return False
```

Initialisation

- Assume `key` is in `array`
- Key in `array[lo…hi]` = `array[1…N]` = `array`

As required

Correctness using Loop Invariant

Invariant: If target is in the list, it is in list[lo…hi]

```
lo = 1
hi = N
while ( lo <= hi )
    mid = floor( (lo+hi)/2 )
    if key == array[mid]
        return mid
    if key >= array[mid]
        lo=mid+1
    else
        hi=mid-1
return False
```

Maintenance

- If `key` in `Array[lo…hi]` before some loop iteration
- If `key` to the right of `mid`, then `key` must be between `mid+1` and `hi`
- But this is exactly the range that we adjust `lo` and `hi` to after the iteration
- Same argument for the other side
- If `key` is at `mid`, then we terminate

Correctness using Loop Invariant

Invariant: If target is in the list, it is in list[lo…hi]

```
lo = 1
hi = N
while ( lo <= hi )
    mid = floor( (lo+hi)/2 )
    if key == array[mid]
        return mid
    if key >= array[mid]
        lo=mid+1
    else
        hi=mid-1
return False
```

Termination

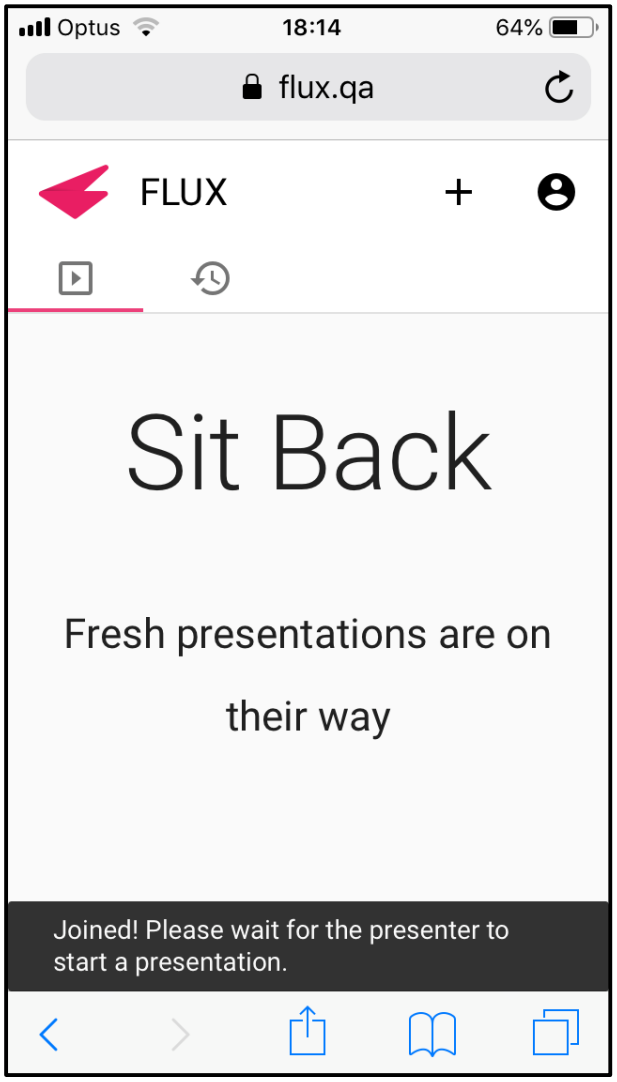
- Either we hit the `return mid` line, in which case we terminate correctly
- Or `lo > hi`. Since the invariant has been correctly maintained…
- `Key` must be in `array[lo…hi]` which is empty
- So we return `False`

Part 2: Outline

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

Quiz time

1. Visit https://flux.qa
2. Log in (your Authcate details)
3. Touch the + symbol
4. Enter the audience code 92A2WY
5. Answer questions



Quiz time

```
for i in a_list:
    if i in b_list:
        c_list.append(i)
```

len(a_list) = A

len(b_list) = B

- Outer loop runs **A** times
- Inner if needs to check every element in b_list, so that's **B** work
- Or is it?
- Append is O(1)
- Or is it?

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
- Worst-case complexity
- Best-case complexity
- Average-case complexity

Recap from FIT1045: Complexity

How to compute time complexity?

- Count the number of steps taken by the algorithm as a function of input size, e.g., $2N^2 + 10N + 100$
- The big-O notation of this function is its complexity, e.g., $2N^2 + 10N + 100 \rightarrow O(N^2)$

Recap from FIT1045: Big-O notation

Let N be the number of days

- # of rabbits → $2N^2$
- # of goats → 10N
- # of lions → 100
- #total population → $2N^2$ +10N + 100

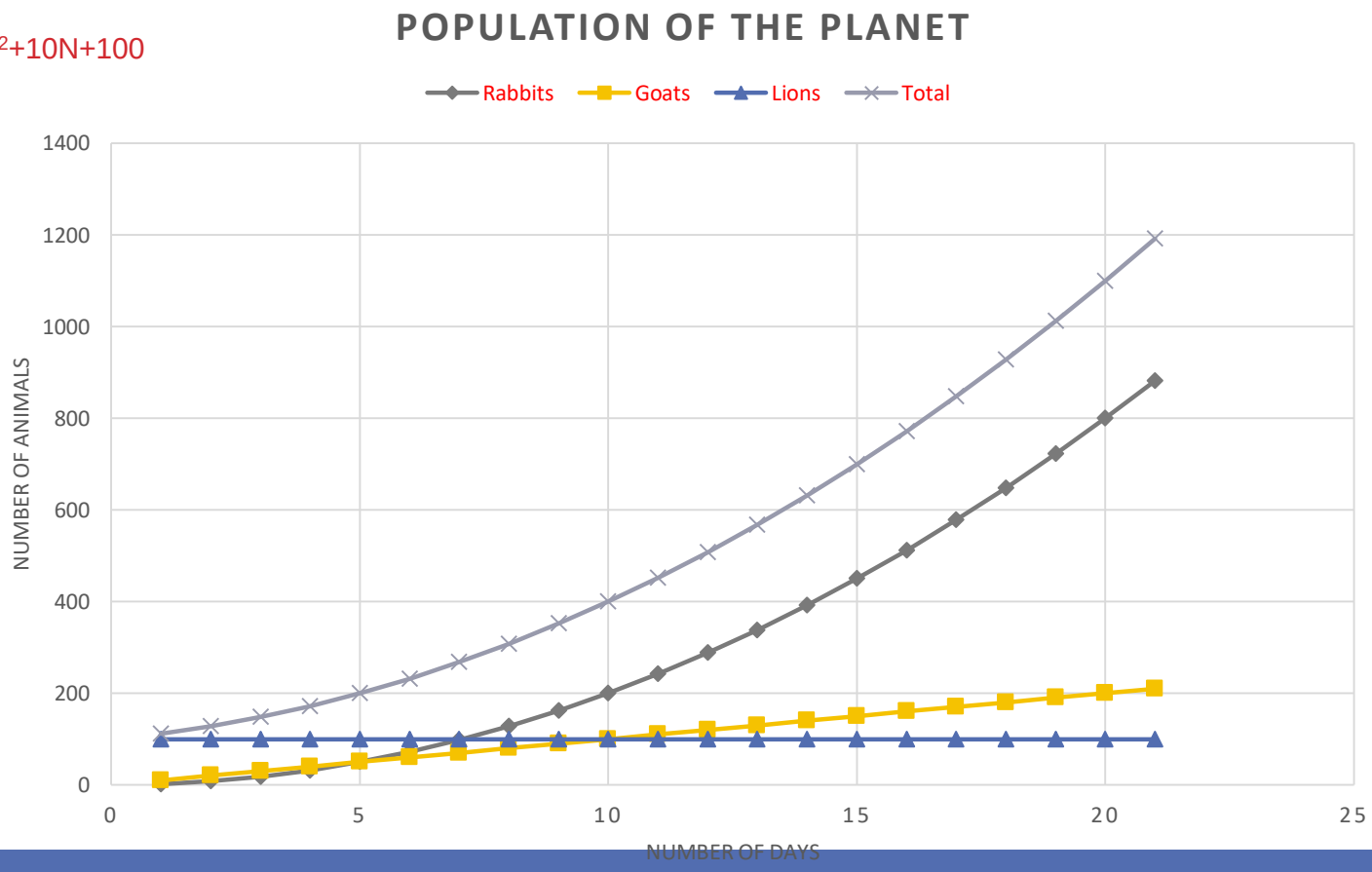
- ✓ The population grows mainly because of growth rate of rabbits
- ✓ For large values of N, # of other animals is insignificant as compared to the # of rabbits
- ✓ We can say population grows in *"**O**rder of rabbits' growth"*
- ✓ i.e., population growth is $O(N^2)$

N	rabbits	goats	lions	Total
1	2	10	100	112
2	8	20	100	128
3	18	30	100	148
4	32	40	100	172
5	50	50	100	200
6	72	60	100	232
7	98	70	100	268
8	128	80	100	308
9	162	90	100	352
10	200	100	100	400
11	242	110	100	452
12	288	120	100	508
13	338	130	100	568
14	392	140	100	632
15	450	150	100	700
16	512	160	100	772
17	578	170	100	848
1000	2000000	10000	100	2010100

Recap from FIT1045: Big-O notation

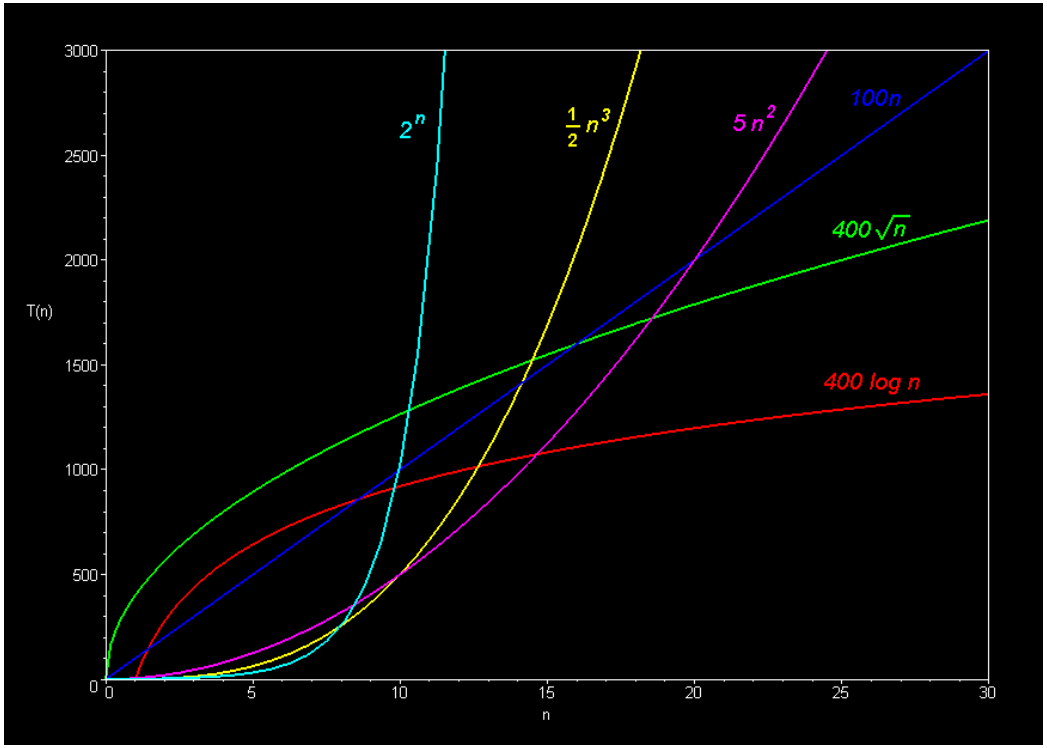
Let N be the number of days

- # of rabbits → $2N^2$
- # of goats → 10N
- # of lions → 100
- #total → $2N^2+10N+100$



Recap from FIT1045: Big-O notation

- Typically, we use the following simplification rules
  - If function is a product of several terms, any constants that do not depend on N can be ignored
  - If function is a sum of several terms, if there is one with the largest growth rate, it can be kept and others can be omitted
- E.g.,
  - **$12 N^2 + 4 N^3$**
    - → **$O(N^3)$**
  - **$12 N^2 + 3 N \log(N)$**
    - → **$O(N^2)$**
  - **$8N^4 + N^2 \log(N) + 12000$**
    - → **$O(N^4)$**
  - **1000 + 5000**
    - → **$O(N^0)$** → **O(1)**



What is the complexity of an algorithm in Big-O notation that runs in $30N \log (N^2) + 10 \log N + 8N$?

A. **O(N log N)**

B. **O(N log (N^2))**

C. **O(N log (N^2) + N + log N)**

D. **Option D (?)**

1. Visit https://flux.qa
2. Log in (your Authcate details)
3. Touch the + symbol
4. Enter the audience code 92A2WY
5. Answer questions

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

Do you see the pattern?

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

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

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 → 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:

T(N-1) + c = c*(N-1) + b – c + c

= c*(N-1) + b

= c*N + b – c

= T(N)

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 $N/2^k = 1 \rightarrow k = \log_2 N$

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

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

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:

$T(N/2) + c = b + c^* \log_2 (N/2) + c$

$= b + c^* [\log_2 (N) - \log_2 (2)] + c$

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

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: something(n) = 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 >= 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 >=b

Revision: Proof by induction

Prove that 1+2+3+…+n = n(n+1)/2 for all n >= 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 >= 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 >=b

```
Let L(n) = 1+2+3+….+n, R(n) = n(n+1)/2
```

Revision: Proof by induction

Prove that 1+2+3+…+n = n(n+1)/2 for all n >= 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 >= 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 >=b

Let L(n) = 1+2+3+….+n, R(n) = n(n+1)/2

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

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

Revision: Proof by induction

Prove that 1+2+3+…+n = n(n+1)/2 for all n >= 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 >= 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 >=b

Let L(n) = 1+2+3+….+n, R(n) = n(n+1)/2

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

L(1) = 1, R(1) = 1*(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+…+n = n(n+1)/2 for all n >= 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 >= 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 >=b

Let L(n) = 1+2+3+….+n, R(n) = n(n+1)/2

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

L(1) = 1, R(1) = 1*(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+…+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

Revision: Proof by induction

Prove that 1+2+3+…+n = n(n+1)/2 for all n >= 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 >= 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 >=b

Let L(n) = 1+2+3+….+n, R(n) = n(n+1)/2

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

L(1) = 1, R(1) = 1*(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+…+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+…+n = n(n+1)/2 for all n>=1

Revision: Proof by induction

Theorem

Prove that 1+2+3+…+n = n(n+1)/2 for all n >= 1

Step 0: Let L(n) = 1+2+3+….+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)

L(k+1) = 1+2+3+…+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

Step 4: Therefore 1+2+3+…+n = n(n+1)/2 for all n>=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**