Carpe Diem

Suggestions for teaching *"formal" methods*

to first-years
without formal logic
the benefit is huge

when to introduce it
how to present it
why it's important

UNSW SYDNEY

Trustworthy Systems

Carpe Diem

Suggestions for teaching "formal" methods

when to introduce it

how to present it

why it's important

(a practical issue)

and how to assess it

Carroll Morgan

BCS-FACS seminar

Thu 25 June, 2026

PART I

Teaching *"formal"* methods

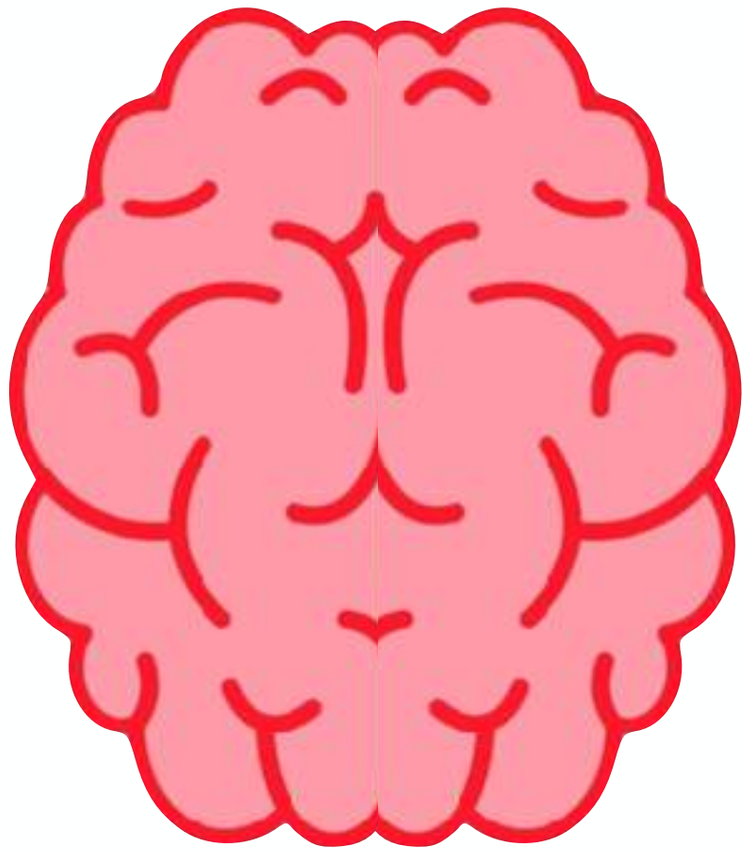
PART I When

Teaching "formal" methods

When

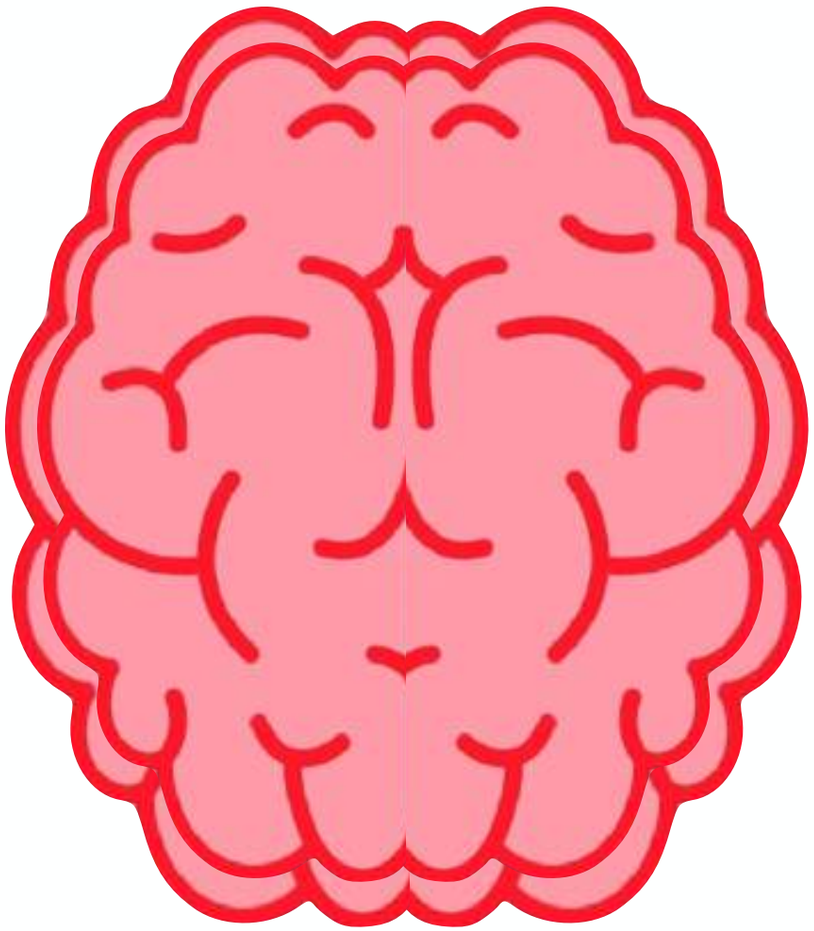
In the last year of high-school, students' brains are still flexible.

And <u>why</u> then?



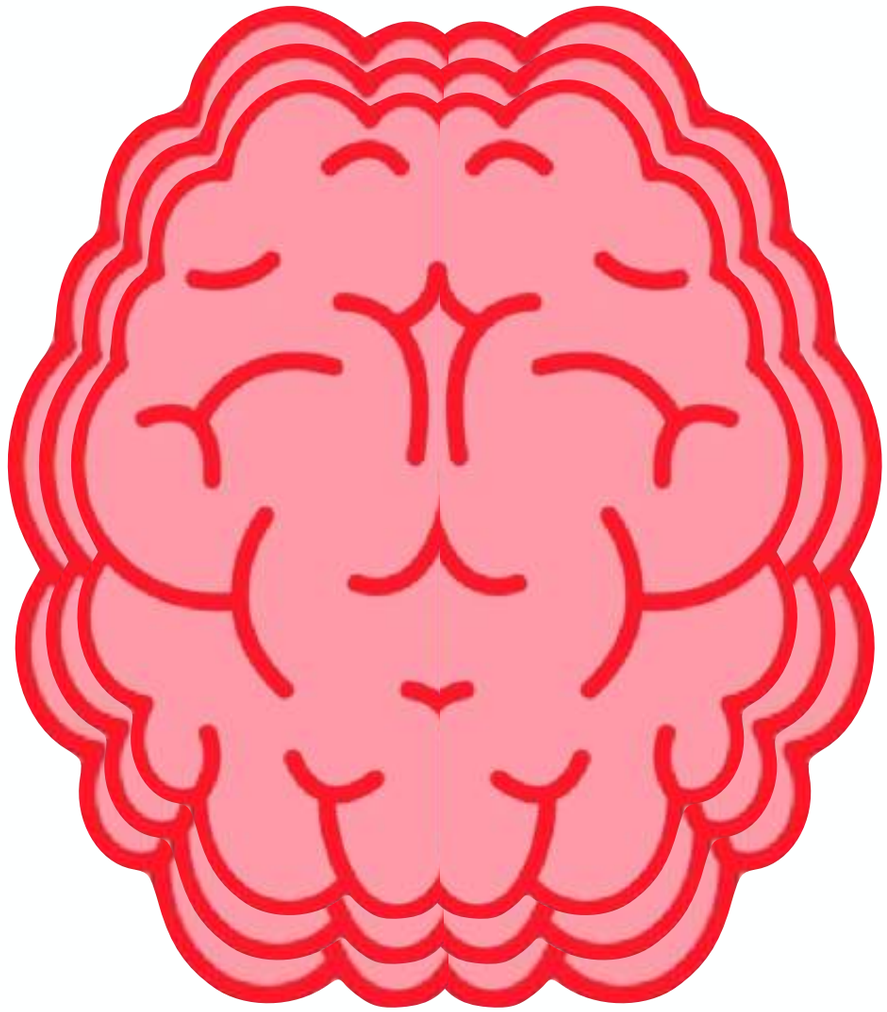
When

They're excited at what's coming, at university.



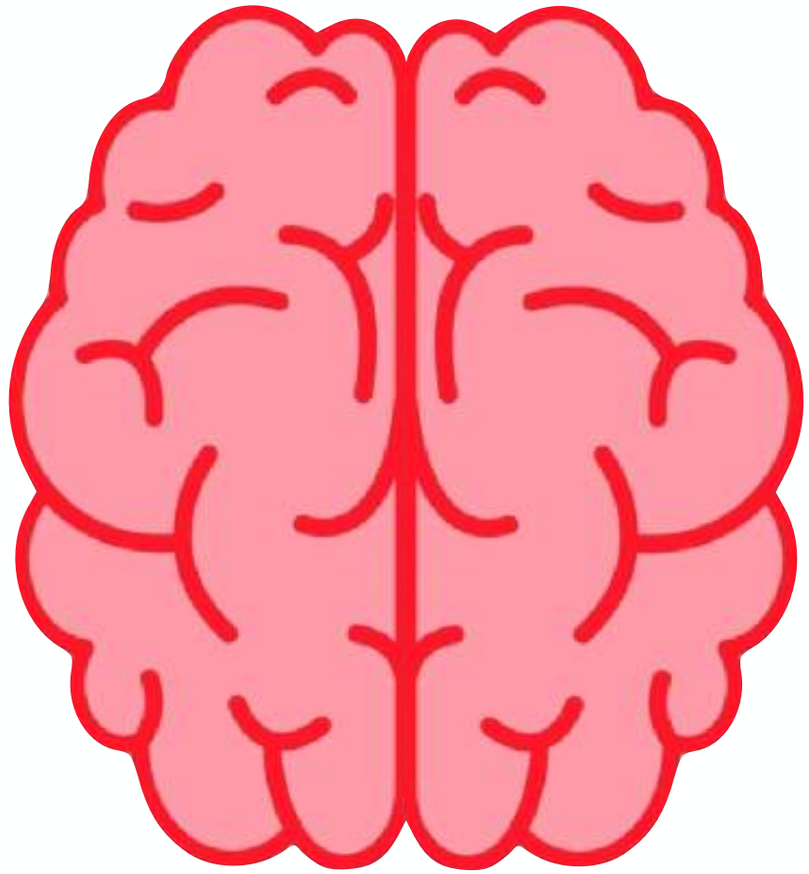
When

They're excited at what's coming; but they're not sure what it will be.



When

In their **first year** of university, they are still **vulnerable** to believing what you tell them.



When I was at high-school, some of my teachers were so ignorant I could hardly stand to have them around.

(apologies to Mark Twain)

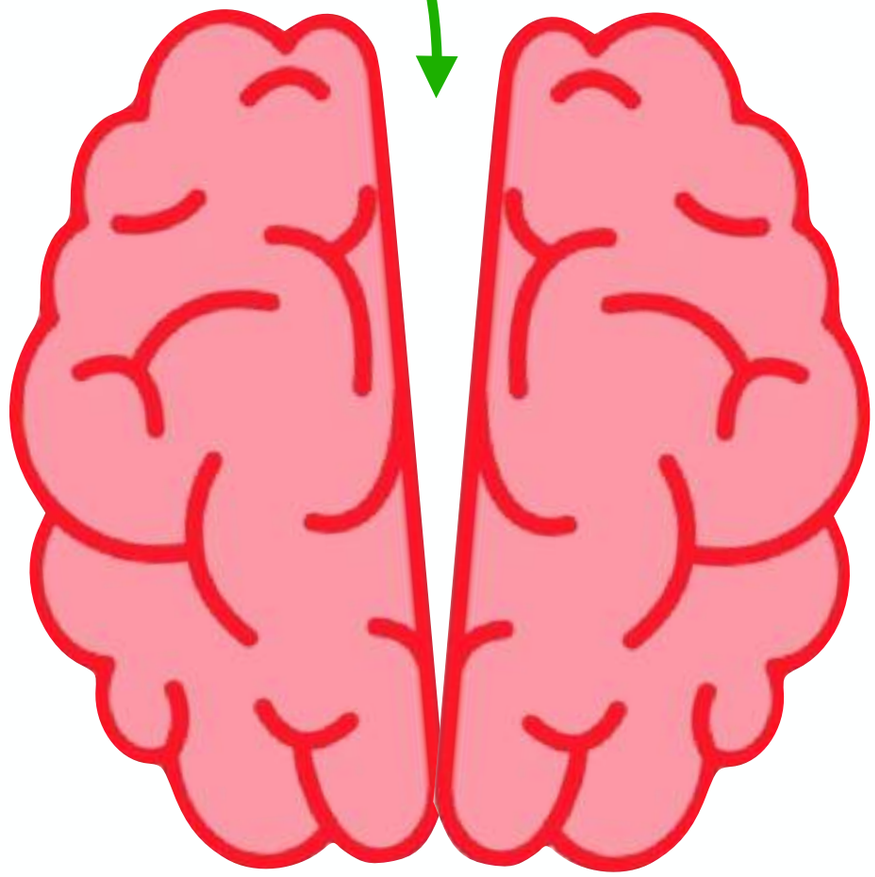
When I was in first year at university, I was astonished at how much teachers must have improved in those four years.

When? Now!

"Their brains are open."
(apologies to Paul Erdős)
That's "why <u>now</u>".

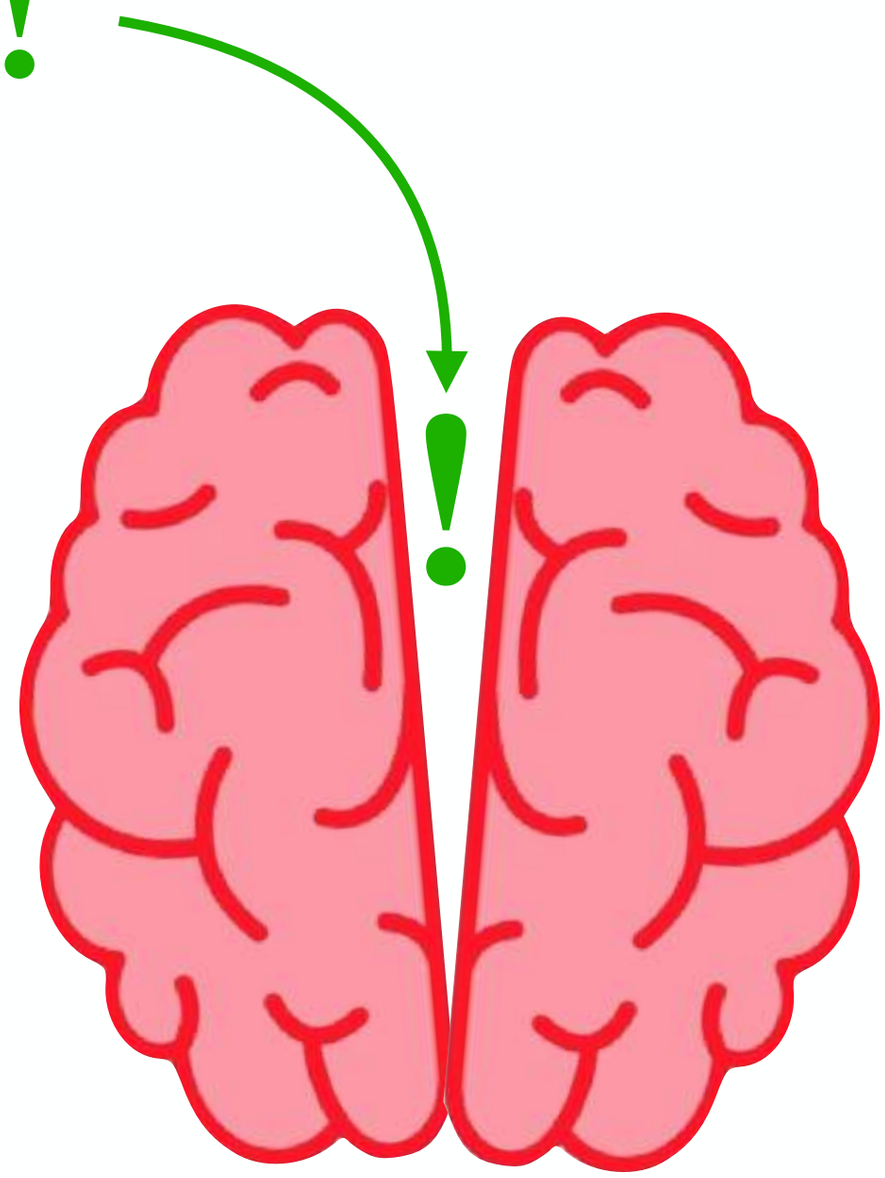
first year

In their first year of university, they are still vulnerable to believing what you tell them.



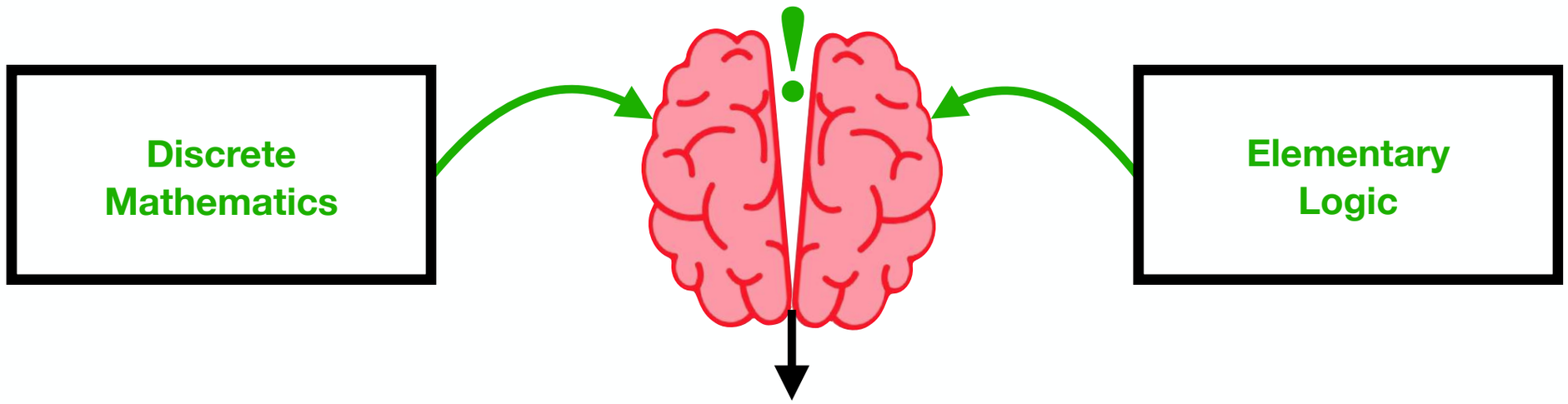
Take advantage of that!

Seize the day!



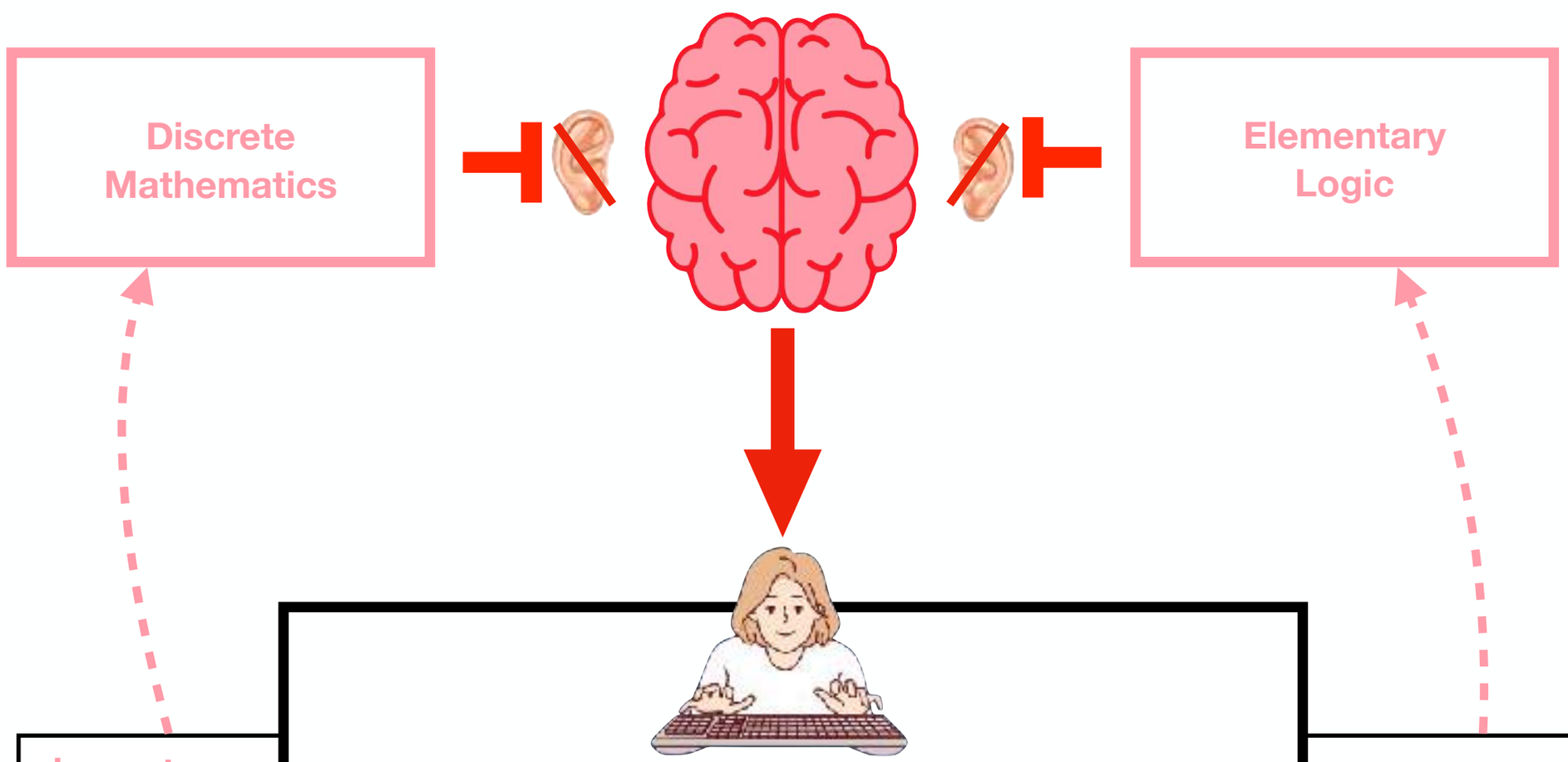
as early as possible

What we hope will happen in 1st year



Introduction to Programming

What often happens in 1st year



Pick one

"Just copy-in the red stuff for now."

Introduction to Programming in … ???

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

```
print("Hello, World!")
```

```
#include <stdio.h>
int main() {
    printf("Hello, World!\n");
    return 0;
}
```

```
#include <iostream>
int main() {
    std::cout << "Hello World!" << std::endl;
    return 0;
}
```

```
main = putStrLn "Hello, World!"
```

Pick a programming language. Then…

"An expression is evaluated, and stored in a named location."

"A loop goes 'round and 'round until its condition is false."

"Loops' e*dge-cases* are particularly important."

Typical 1st-year exercise: *sum an array*

```
// Sum the array numbers into s.
#define N 10
int numbers[N];
int s= 0;

int main() {
    int n= 0;
    while (n!=N) {
        s= s+numbers[n];
        n= n+1;
    }

}
```

Needs some documentation however!

Typical 1st-year exercise: *sum an array*

```
// Sum the array numbers into s.
#define N 10
int numbers[N];
int s= 0;

int main() {
     int n= 0;
     while (n!=N) {
          s= s+numbers[n];
          n= n+1;
     }

}
```

Needs some documentation however!

PART II

Start simple

Typical first-year exercise: sum an array

```
// Sum the array numbers into s.
#define N 10
int numbers[N];
int s= 0; // Initialise s.

int main() {
    int n= 0;
    while (n!=N) {
        s= s+numbers[n]; // Add current number,
        n= n+1;          // and move to next one.
    }
    // The sum of numbers is now in s.
}
```

100%

Nicely laid out.
Well documented!

b.b.b.but...

Typical first-year exercise: sum an array

```
// Sum the array numbers into s.
#define N 10
int numbers[N];
int s= 0; // Initialise s.

int main() {
    int n= 0;
    while (n!=N) {
        s= s+numbers[n]; // Add current number,
        n= n+1;          // and move to next one.
    }
    // The sum of numbers is now in s.
}
```

100%

Nicely laid out.
Well documented!

…but how do you check that your program is correct?

Typical first-year exercise: sum an array

```
// Sum the array numbers into s.
#define N 10
int numbers[N];
int s= 0; // Initialise s.

int main() {
    int n= 0;
    while (n!=N) {
        s= s+numbers[n]; // Add current number,
        n= n+1;          // and move to next one.
    }
    // The sum of numbers is now in s.
}
```

100%

Nicely laid out.
Well documented!

Code review!

Typical first-year exercise: sum an array

```
// Sum the array numbers into s.
#define N 10
int numbers[N];
int s= 0; // Initialise s.

int main() {
    int n= 0;
    while (n!=N) {
        s= s+numbers[n]; // Add current number,
        n= n+1;          // and move to next one.
    }
    // The sum of numbers is now in s.
}
```

100%

Nicely laid out.
Well documented!

Is this code-review enough?

Code review!

Typical first-year exercise: sum an array

```
// Sum the array numbers into s.
#define N 10
int numbers[N];
int s= 0; // Initialise s.

int main() {
    int n= 0;
    while (n!=N) {
        s= s+numbers[n]; // Add current number,
        n= n+1;          // and move to next one.
    }
    // The sum of numbers is now in s.
}
```

100%

Nicely laid out.
Well documented!

No!! It's *important*, but it's not enough. You must also…

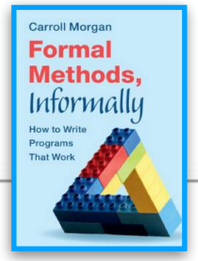
***Test it*, *of course!* But first let's look at code reviews.**

Extract from

[A code review]

It's much the same.

In Python:



p.5

1.3 When does a program "work"?

Suppose we have a sequence `A[0:N]` of numbers, indexed from `0` inclusive to `N` exclusive, and we want to calculate their sum. Does this program work?

```
s= 0 # Start from zero.
for n in range(N):                                    (1.2)
    s= s+A[n] # Add the current element.
```

To answer that question, we must check various things.

Is the intended sum `s` correctly initialised? *Yes.* Are all the sequence elements `A[n]` of `A` considered, exactly once? *Yes.* And does the `n`-indexing remain within the bounds of the sequence? (We remind ourselves that the elements of `A[0:N]` are indexed from `0` up to `N-1`, and that `n in range(N)` iterates over `0,1,...,N-1`.) Again *Yes.* Is the correct operator `+` being used in the loop body? *Yes.*

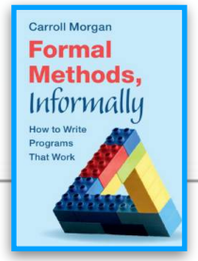
So —overall— *Yes.* It does work. For a program of that (tiny) size, we can be sure we've checked everything.

Extract from

A code review

It's much the same.

In Python:



p.5

1.3 When does a program "work"?

1.3 When does a program "work"?

Suppose we have a sequence `A[0:N]` of numbers, indexed from `0` inclusive to `N` exclusive, and we want to calculate their sum. Does this program work?

```
s= 0 # Start from zero.
for n in range(N):                                    (1.2)
    s= s+A[n] # Add the current element.
```

To answer that question, we must check various things.

Is the intended sum `s` correctly initialised? *Yes.* Are all the sequence elements `A[n]` of `A` considered, exactly once? *Yes.* And does the `n`-indexing remain within the bounds of the sequence? (We remind ourselves that the elements of `A[0:N]` are indexed from `0` up to `N-1`, and that `n in range(N)` iterates over `0,1,...,N-1`.) Again *Yes.* Is the correct operator `+` being used in the loop body? *Yes.*

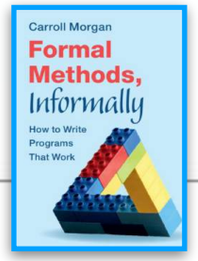
So—overall—*Yes.* It does work. For a program of that (tiny) size, we can be sure we've checked everything.

Extract from

A code review

It's much the same.

In Python:



p.5

1.3 When does a program "work"?

1.3 When does a program "work"?

Suppose we have a sequence `A[0:N]` of numbers, indexed from `0` inclusive to `N` exclusive, and we want to calculate their sum. Does this program work?

```
s= 0 # Start from zero.
for n in range(N):                                    (1.2)
    s= s+A[n] # Add the current element.
```

To answer that question, we must check various things.

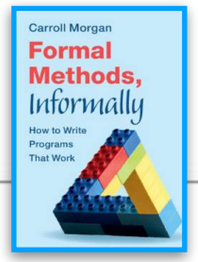
Is the intended sum `s` correctly initialised? *Yes.* Are all the sequence elements `A[n]` of `A` considered, exactly once? *Yes.* And does the `n`-indexing remain within the bounds of the sequence? (We remind ourselves that the elements of `A[0:N]` are indexed from `0` up to `N-1`, and that `n in range(N)` iterates over `0,1,...,N-1`.) Again *Yes.* Is the correct operator `+` being used in the loop body? *Yes.*

So—overall—*Yes.* It does work. For a program of that (tiny) size, we can be sure we've checked everything.

It's much the same.

In Python:

Extract from



p.5

This applies to both code reviews and testing.

1.3 When does a program "work"?

1.3 When does a program "work"?

Or does it work, really... *What does that mean?* It does run without crashing. But you can't say that something works (or doesn't) *unless* you know what it's supposed to do. And there's nothing "officially" associated with (1.2) that says what it's actually *for*. All we have is the text "...we have a sequence `A[0:N]` of numbers, and we want to calculate their sum." written in a paragraph nearby (in this case, just above the program code). What if there had been a page break in the source-text file, in between? When asking whether a program works we have to point not only to the program, but also to a description of what it's supposed to do. (See Sec. 18.2.3.)

So it's probably a good idea to include a (rough) description of the program's purpose as *part of the program itself*, say as a comment at its beginning. That would give us

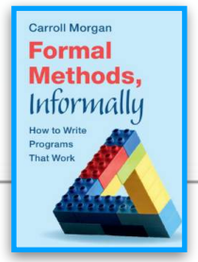
```
# --- Sum the elements of sequence A[0:N].
s= 0                                                    (1.3)
for n in range(N): s= s+A[n]    .
```

Extract from

This applies to both code reviews and testing.

It's much the same.

In Python:



p.5

1.3 When does a program "work"?

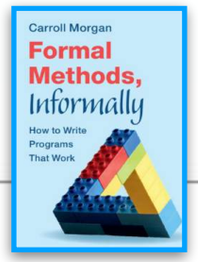
1.3 When does a program "work"?

Or does it work, really... *What does that mean?* It does run without crashing. But you can't say that something works (or doesn't) *unless* you know what it's supposed to do. And there's nothing "officially" associated with (1.2) that says what it's actually *for*. All we have is the text "...we have a sequence `A[0:N]` of numbers, and we want to calculate their sum." written in a paragraph nearby (in this case, just above the program code). What if there had been a page break in the source-text file, in between? When asking whether a program works we have to point not only to the program, but also to a description of what it's supposed to do. (See Sec. 18.2.3.)

So it's probably a good idea to include a (rough) description of the program's purpose as *part of the program itself*, say as a comment at its beginning. That would give us

```
# --- Sum the elements of sequence A[0:N].
s= 0                                                    (1.3)
for n in range(N): s= s+A[n]    .
```

Extract from



p.5

This applies to both code reviews and testing.

It's much the same.

In Python:

1.3 When does a program "work"?

1.3 When does a program "work"?

Or does it work, really... *What does that mean?* It does run without crashing. But you can't say that something works (or doesn't) *unless* you know what it's supposed to do. And there's nothing "officially" associated with (1.2) that says what it's actually *for*. All we have is the text "...we have a sequence `A[0:N]` of numbers, and we want to calculate their sum." written in a paragraph nearby (in this case, just above the program code). What if there had been a page break in the source-text file, in between? When asking whether a program works we have to point not only to the program, but also to a description of what it's supposed to do. (See Sec. 18.2.3.)

So it's probably a good idea to include a (rough) description of the program's purpose as *part of the program itself*, say as a comment at its beginning. That would give us

```
# --- Sum the elements of sequence A[0:N].
s= 0                                                    (1.3)
for n in range(N): s= s+A[n]    .
```

The blue comments are essentially *"What this statement did."*

Actually, "what it was supposed to do."

```
// Sum the array numbers into s.
#define N 10
int numbers[N];
int s= 0; // Initialise s.

int main() {
    int n= 0;
    while (n!=N) {
        s= s+numbers[n]; // Add current number,
        n= n+1;          // and move to next one.
    }
    // The sum of numbers is in s.
}
```

But the blue could *instead* have been…

"What was made true here?"

```
// Sum the array numbers into s.
#define N 10
int numbers[N];
int s= 0; // Initialise s.

int main() {
    int n= 0;
    while (n!=N) {
        s= s+numbers[n]; // Add current number,
        n= n+1;          // and move to next one.
    }
    // The sum of numbers is in s.
}
```

The cost-free alternative

~~*What it did.*~~

"What's true here?"

```
#define N 10
int numbers[N];
int s= 0; // Initialise s.

int main() {
    int n= 0;
    // Invariant: s == SUM numbers[0:n]
    while (n!=N) {
        s= s+numbers[n]; // Add current number,
        n= n+1;          // and move to next one.
    } // n==N
}
// Postcondition: s == SUM numbers[0:N]
```

The cost-free alternative *"What's true here?"*

```
#define N 10
int numbers[N];
int s= 0;

int main() {
    int n= 0;
    // Invariant: s == SUM numbers[0:n]
    while (n!=N) {
        s= s+numbers[n];
        n= n+1;
    }
}   // n==N
// Postcondition: s == SUM numbers[0:N]
```

AND

Given the BLUE, you can *generate* the GREY.

The cost-free alternative

```
// Sum the array numbers into s.
#define N 10
int numbers[N];
int s= 0;

int main() {
    int n= 0;
    // Invariant: s == SUM numbers[0:n]
    while (n!=N) {
        s= s+numbers[n];
        n= n+1;
    }
}   // n==N
// Postcondition: s == SUM numbers[0:N]
```

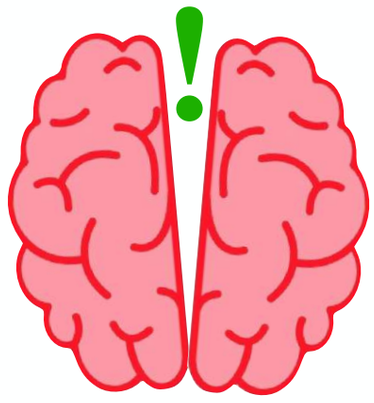
(and 0<=n<=N)

Given the BLUE, you can generate the BLACK.

150%

The cost-free alternative

```
// Sum the array numbers into s.
#define N 10
int numbers[N];
```



they

Given the BLUE, ~~you~~ can generate the BLACK.

”After taking this course I felt as if I was privy to a powerful ancient magic. Everything that was introduced was so useful yet not yet known to most CS students.”

```
// Postcondition: s == SUM numbers[0:N]
```

Summary so far

when to introduce it

how to present it

assessment ???

coming later

Start at the ***same*** time as the existing first programming course; use the ***same*** course structure; set the ***same*** exercises…

but explain and document them with…

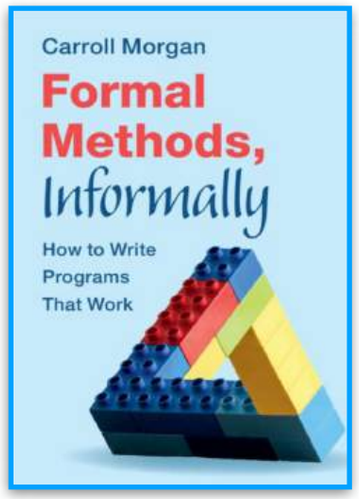
…***"What's true here?"***

more detailed example follows

How

A more detailed example of **how to present** "how"

Chapter 2 of this:



p.25

2

Using invariants to design loops

2.1 Introduction

In this chapter, we will go through a program-design process in three different ways, but all three for the *same* problem. The first two ways will use a conventional approach (although we do mention invariants, after the fact, to help check them); but the third uses an invariant *before* the fact, to help design the program in the first place.[1]

[1] This problem was taken from a first-year university course "Introduction to Programming in *C*".

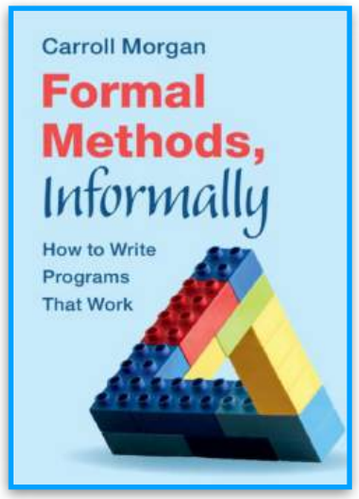
@ UNSW

Gain *their* confidence; **and yours:**

How

A more detailed example of **how to present** "how"

Chapter 2 of this:



p.25

2

Using invariants to design loops

2.1 Introduction

In this chapter, we will go through a program-design process in three different ways, but all three for the *same* problem. The first two ways will use a conventional approach (although we do mention invariants, after the fact, to help check them); but the third uses an invariant *before* the fact, to help design the program in the first place.[1]

[1] This problem was taken from a first-year university course "Introduction to Programming in *C*".

@ UNSW

PART III Gain *their* confidence; **and yours:**

How

Lots of detail here, but  will mean "just skim this…"

2.2 The longest *Good* subsegment problem

Assume we have a property $Bad(a, b, c)$ of three integers, for some (not yet made precise) property of "badness". Here are some examples of what $Bad(a, b, c)$ might be:

- All equal: $a = b = c$ is bad.
- In a run, up or down: either $a+1 = b$ and $b+1 = c$, or $a-1 = b$ and $b-1 = c$.
- Able to make a triangle: thus $a+b+c \geq 2\,(a \max b \max c)$.
- The negation (i.e. opposite) of any of the above.

Suppose further we are working (as usual) with a sequence `A[0:N]` and that we have implemented a Boolean function `Bad(n)` that determines $Bad($`A[n], A[n+1], A[n+2]`$)$ for us. (Note that the function `Bad` has just one argument `n` but examines `A` in three places.) It is a programming error –index out of range– if `n<0` or `n+3>N`.

[1] This problem was taken from a first-year university course "Introduction to Programming in *C*".

How

Make sure the requirements are clear!

The requirements for this program might be written as follows:

```
# --- Determine the length of the longest Good subsegment
# --- of sequence A[0:N], where a Good subsegment has no
# --- subsegment of three (consecutive) elements inside it
# --- that is Bad.

# --- Assume that the Boolean function  Bad(n)  determines
# --- whether A[n:n+3] is Bad, for 0<=n<=N-3.
```

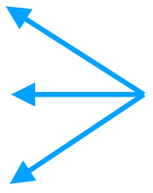
Gave this to ***chatGPT*** later!

The program's specification, making the requirements more precise, might be

```
# PRE: Bad(n) is well defined for 0<=n and n+3<=N.

... # Program text goes here.

# POST: good == "a largest value such that
#       there is a low with 0<=low and low+good<=N and
#       for no n with low<=n and n+3<=low+good
#       does Bad(n) hold."
```



This practice is actually pretty good already.

How

Takes about 1/2 hour to **do this on the board** —
with contributions from the class.

2.3 First attempt: operational intuition and diagrams

A straightforward approach to solving the above problem might be along these lines.

First go through all of sequence `A` and store its "bad-subsegment positions" in an extra array `B`. Then find the differences between adjacent elements of `B`. Finally –after some careful thought– set good to the largest of those differences.[3]

Just skim this…

```
1  # Sequence B must have length at least N-2.
2  bads= 0 # Number of Bad subsegments found.
3  for b in range(N-2): # Find starting points of Bad subsegments.
4    if Bad(b): B[bads],bads= b,bads+1
5  # Now we know where the Bad subsegments are.
6
7  if bads==0: good= N # Special case.
8  else:
9    good= B[0]+2 # First Bad subsegment gives first Good subsegment.
10   for n in range(1,bads): # More Bad subsegments... except last.
11     good= good max (B[n]-B[n-1]+1) # ...give subsequent Good's.
12   good= good max (N-B[bads-1]-1) # Very last Bad subsegment.
```

Find the bad subsegments.

Measure their separations.

How

2.3 First attempt: operational intuition and diagrams

A straightforward approach to solving the above problem might be along these lines.

First go through all of sequence `A` and store its "bad-subsegment positions" in an extra array `B`. Then find the differences between adjacent elements of `B`. Finally –after some careful thought– set `good` to the largest of those differences.[3]

in helper-sequence **B**

This too 

unfortunately

[3] The "careful thought" here –**which is actually fun**– is to figure out what the longest *Good* subsegment can be that includes neither *Bad* subsegment `A[b0:b0+3]` nor *Bad* subsegment `A[b1:b1+3]`, where `b0<b1` and they are the two adjacent bad positions with `diff == b1-b0`. It must start at-or-after `b0+1`, in order to leave out `A[b0:b0+3]`, and it must end at-or-before `b1+2-1` to leave out `A[b1:b1+3]`. So its greatest possible length is "where it starts" minus "where it ends" plus 1, that is `(b1+2-1)-(b0+1)+1` which simplifies to `b1-b0+1`. Since the largest such `b1-b0` is `diff` itself, the correct value for `good` is `diff+1` — and not `diff+2` as we initially guessed above.

This "guessing" and then **"tweaking until correct" is a common approach** — especially because it *is* fun. **But is it reliable?**

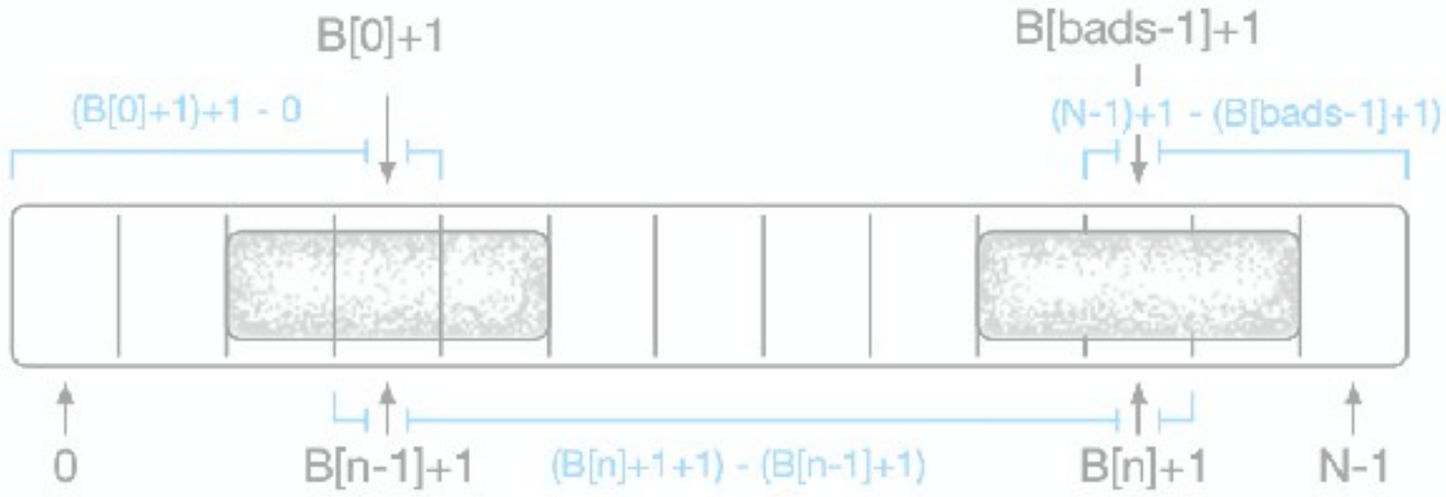
```
# Sequence B must have length at least N-2.
bads= 0 # Number of Bad subsegments found.
for b in range(N-2): # Find starting points of Bad subsegments.
  if Bad(b): B[bads],bads= b,bads+1
# Now we know where the Bad subsegments are.
if bads==0: good= N # Special case.
else:
  good= B[0]+2 # First Bad subsegment gives first Good subsegment.
  for n in range(1,bads): # More Bad subsegments...  except last.
    good= good max (B[n]-B[n-1]+1) # ...give subsequent Good's.
  good= good max (N-B[bads-1]-1) # Very last Bad subsegment.
```

And skim these too

BUT it *is* actually fun, because it feels to them like **rocket science**.

unfortunately

Figure 2.2 Sketches to check index calculations in (2.3) above



The length of subsegment `A[low:high]` is `high-low` in all cases.

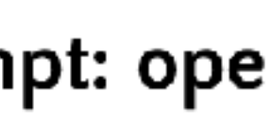
Not done in class

2.4 Second attempt: operational intuition and a caterpillar

~~9 lines of code~~

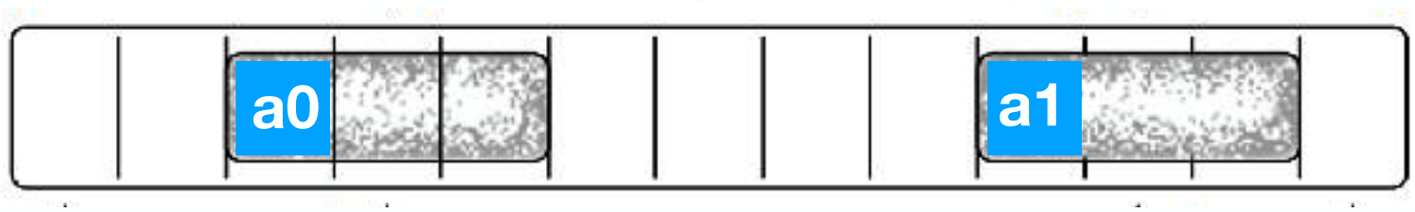
Figure 2.3 Second program — 8 lines of code, no extra sequence (Sec. 2.4)

```
1  a0,a1,good= -1,-1,0 # Pretend there is a Bad subsegment at -1.
2  while True:
3    a1= a1+1 # Look for the next Bad subsegment.
4    if a1+3>N: # There isn't one.
5      good= good max (N-a0-1) # Take last a0 into account.
6      break
7    if Bad(a1): # Found one: update good from a0,a1 and carry on.
8      good,a0= good max (a1-a0+1), a1
```



This second program improves the first in two major ways: it doesn't need an extra sequence B, and there are fewer special cases.

Remove **B**
Replace with **a0,a1.**



More "How"...?

Wait a minute...!

What does all this have to do with "informal methods"?

About 1/2 hour to do the 1st version
with audience participation
on the board...

...will have prepared them for a still better "how" that even 1st-years will *understand*.

2.5 Third attempt: use an invariant to design the program

Figure 2.4 Third program: recommended method

```
# PRE: Bad(n) is well defined for 0<=n and n+3<=N.
# Inv1(n) is "good is the length
#              of a longest Good subsegment of A[:n]."
# Inv2(n) is "A[low:n] is the longest Good suffix of A[:n]."
.....
# INV: Inv1(n) and Inv2(n) and 0<=n<=N
.....
.....
.....
# POST: Inv1(N).
```

Not 9 lines with an extra sequence "B".

Nor 8 lines once "B" is finessed away.

How?

Next slide

- No extra array "B" is used (2.3).
- No off-line sketches are required (Fig. 2.2 for (2.3)).
- There are no mysterious −1 initialisations (2.4).
- There is no special treatment needed for the beginning or the end of A (or both).
- max is used only once.
- It is the shortest (only half the size, at 4 lines — not 8 or 9).
- We do not need to check it afterwards (though we *must* test it: Sec. 19.3) since it was checked automatically by the construction process as we went along.[4]

[4] All three programs in this chapter were run on thousands of sample A's, generated by Python's hypothesis tester: it confirmed that their answers were equal in every generated case.

2.5 Third attempt: use an invariant to design the program

Figure 2.4 Third program: recommended method

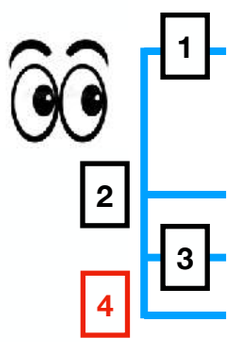
```
# PRE: Bad(n) is well defined for 0<=n and n+3<=N.
# Inv1(n) is "good is the length
#              of a longest Good subsegment of A[:n]."
# Inv2(n) is "A[low:n] is the longest Good suffix of A[:n]."
......
# INV: Inv1(n) and Inv2(n) and 0<=n<=N
......
......
......
# Post: Inv1(N).
```

Not 9 lines with an extra sequence "B".

Nor 8 lines once "B" is finessed away.

How?

This is where they really wake up.



Next slide

Next slide

Figure 2.4 Third program: recommended method

```
# PRE: Bad(n) is well defined for 0<=n and n+3<=N.
# Inv1(n) is "good is the length
#                of a longest Good subsegment of A[:n]."
# Inv2(n) is "A[low:n] is the longest Good suffix of A[:n]."
n,good,low= 0,0,0                                          1
# INV: Inv1(n) and Inv2(n) and 0<=n<=N                     2
while n!=N:                                                2
  if n>=2 and Bad(n-2): low= n-1                           3
  good,n= good max (n+1-low), n+1                          4
# POST: Inv1(N).
```

Only 4 lines long

A celebrity program?

(No **edge cases**)

- No extra array "B" is used (2.3). (It's **unnecessary**)
- No off-line sketches are required (Fig. 2.2 for (2.3)). (No **rocket science**)
- There are no mysterious -1 initialisations (2.4).
- There is no special treatment needed for the beginning or the end of A (or both).
- max is used only once.
- It is the shortest (only half the size, at 4 lines — not 8 or 9).
- We do not need to check it afterwards (though we *must* test it: Sec. 19.3) since it was checked automatically by the construction process as we went along.[4]

Ch.2

[4] All three programs in this chapter were run on thousands of sample A's, generated by Python's `hypothesis` tester: it confirmed that their answers were equal in every generated case.

Figure 2.4 Third program: recommended method

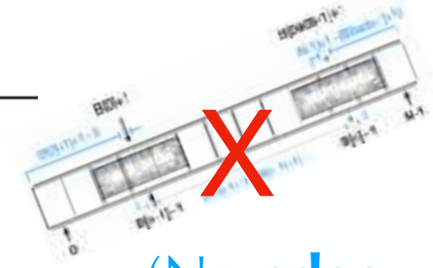
```
# PRE: Bad(n) is well defined for 0<=n and n+3<=N.
# Inv1(n) is "good is the length
#                of a longest Good subsegment of A[:n]."
# Inv2(n) is "A[low:n] is the longest Good suffix of A[:n]."
n,good,low= 0,0,0                                          1
# INV: Inv1(n) and Inv2(n) and 0<=n<=N                     2
while n!=N:                                                2
  if n>=2 and Bad(n-2): low= n-1                           3
  good,n= good max (n+1-low), n+1                          4
# POST: Inv1(N).
```

p.221

Not a celebrity program.

Anyone can do it.

- No extra array "A" is used (2.3). (It's **unnecessary**)
- No off-line sketches are required (Fig. 2.2 for (2.3)). (No **rocket science**)
- There are no mysterious —1 initialisations (2.4).
- There is no special treatment needed for the beginning or the end of A (or both).
- max is used only once.
- It is the shortest (only half the size, at 4 lines — not 8 or 9).
- We do not need to check it afterwards (though we *must* test it: Sec. 19.3) since it was checked automatically by the construction process as we went along.[4]



(No **edge cases**)

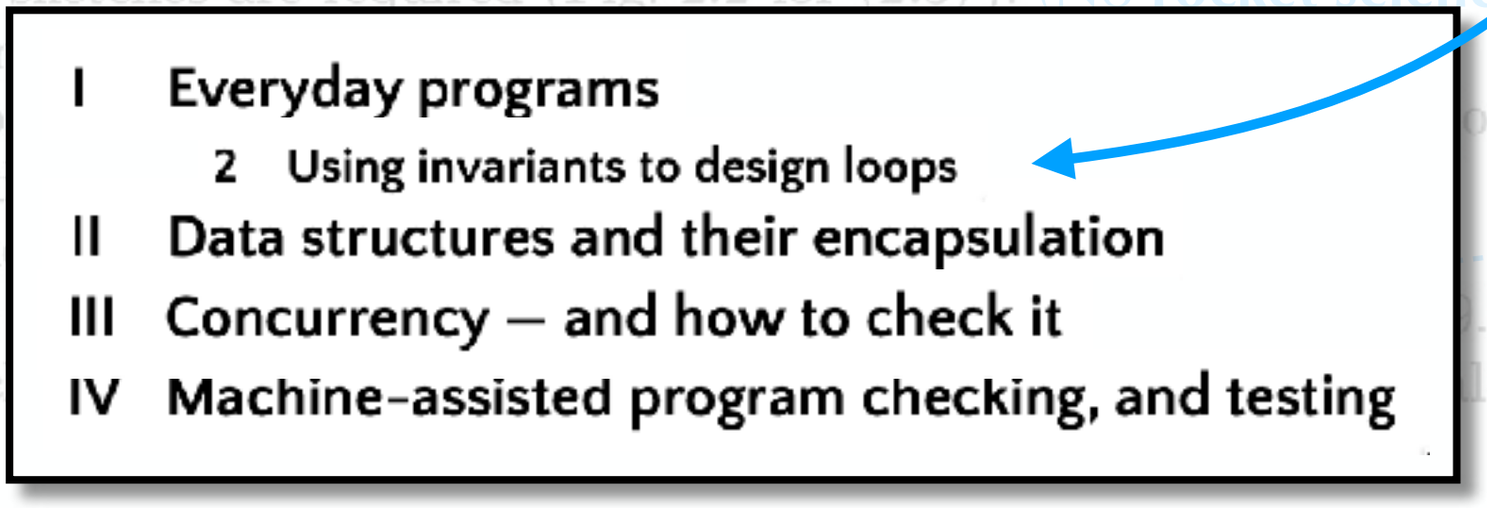
Ch.2

[4] All three programs in this chapter were run on thousands of sample A's, generated by `hypothesis` tester: it confirmed that their answers were equal in every generated case.

***Test it*, of course!**

"This course radically changed the way I view programming, and has certainly improved my programming skills immensely. This course should be compulsory for all Computer Science students, or have its contents integrated into first year."

Anyone can do it.



Anyone can do it?

Fourth attempt: ***chatGPT***,
after the May 2026 FMO$\mathbb{Z}$ meeting
at the University of Queensland
organised now 10 times annually by
Kirsten Winter and **Graeme Smith**.

```
Write a Python program which, given a sequence A
of numbers of length N, finds the length of the
longest subsegment of A that does not contain a
Bad subsegment, where Bad(n) is true just when
A[n:n+3] is Bad.
```

chatGPT tidied

Comments removed, to fit on one slide.

```
1    N= len(A)
2    if N<3: return N
3    bad_start= [False]*N
4    for n in range(N-2):
5        if Bad(n): bad_start[n]= True
6    max_length= 0
7    left= 0
8    last_bad_start= -1
9    for right in range(N):
10       if right>=2 and bad_start[right-2]:
11           last_bad_start= right-2
12       if last_bad_start >= left:
13           left= last_bad_start+1
14       current_length= right-left+ 1
15       if current_length > max_length:
16           max_length= current_length
```

PART IV

biting the bullet

Assessment

PART IV

biting the bullet

Assessment

PART IV

biting the bullet

1,000 students?

Assessment

Disclaimer: This section is rather dense, and **so I will skim it**.

It is *not* a "How it *must* be done." but rather an example of "How it *can* be done."

There will be other ways, and I hope you might suggest some.

Assessment

PART IV

biting the bullet

Usually in programming courses –especially when the numbers are large– marks for programming exercises are based on "auto-marking".

1,000 students?

Thus can't auto-mark here.

Running against many test-cases is not evaluating what the course is supposed to teach!

And there's no formal logic that can be bulk-verified with e.g. Dafny (e.g. SMT)!

Can't auto-mark here either.

Assessment

i.e. as well as whether their programs give the right answers under test.

The students must be graded *also* on how they *construct* programs.

That "by construction" focus must somehow be auto-marked even when the student numbers are large.

Following are two ways in that was done: in **2022** and (differently) in **2023**.

In earlier years (from **2010**), the numbers were much smaller.

UNSW course
(In-)Formal Methods: the Lost Art
COMP 6721

Assessment in 2022

Briefly

The project for the whole term in 2022 was TeX's paragraph layout program (essentially dynamic programming), arguably too complex for 1st-years of course. But at that stage we were dealing with 2nd-years and higher.

A large-ish program template was given, with "What *should* be true here?" assertion-like statements already embedded in it.

The students had to fill-in or complete code that would make those assertions true. (They were not allowed to change the assertions, of course.)

There were also *weekly quizzes* (explained in the 2023 section below).

Assessment in 2023

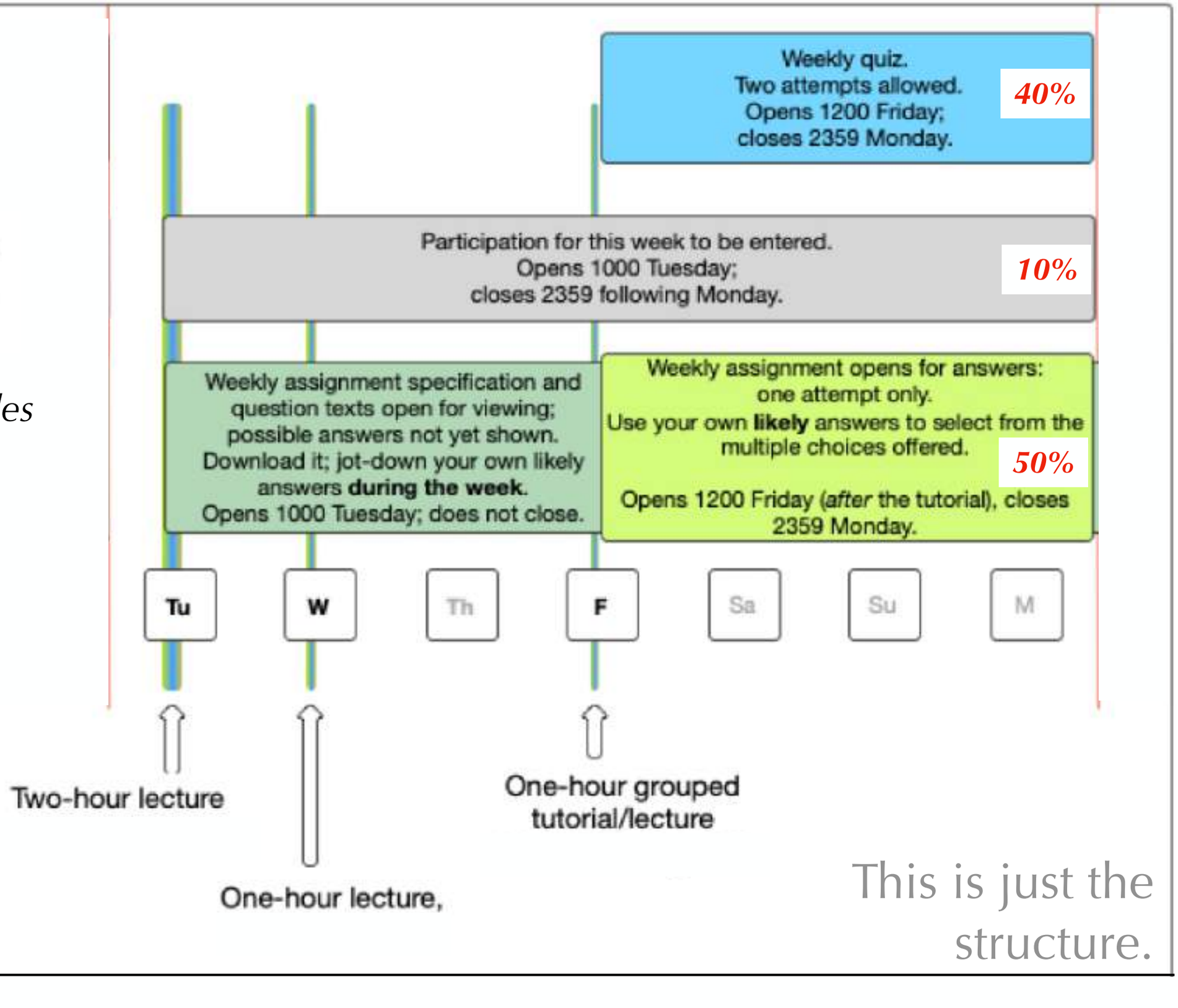
There were two principal assessments: **quizzes** and **project**.

The auto-marked Moodle quizzes were every week, randomised, and on "basic stuff". Best 7 out of 9 taken, (so messing one up is not fatal…)

and worth 40% overall.

Overall 2023 assessment per week

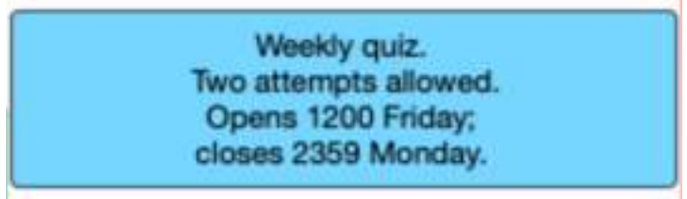
Next slides



This is just the structure.

quizzes

Weekly quiz, two attempts allowed.
Opens 1200 Friday, closes 2359 Monday.



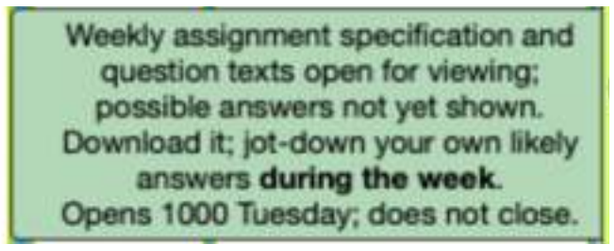
Fri, Sat, Sun, Mon

40%

Weekly assignment-portion

***specification*, and question texts**,
open for viewing from 1000 Tuesday;
does not close.

But possible answers are <u>not</u> shown (yet).
Download it, jot-down your own likely
answers during the week.

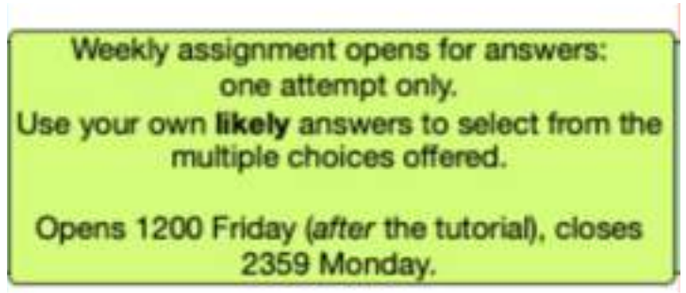


Tue, does not close

assignment ← linked →

in 9 portions

Weekly assignment-portion opens for ***answers***
1200 Friday (after the tutorial), closes 2359

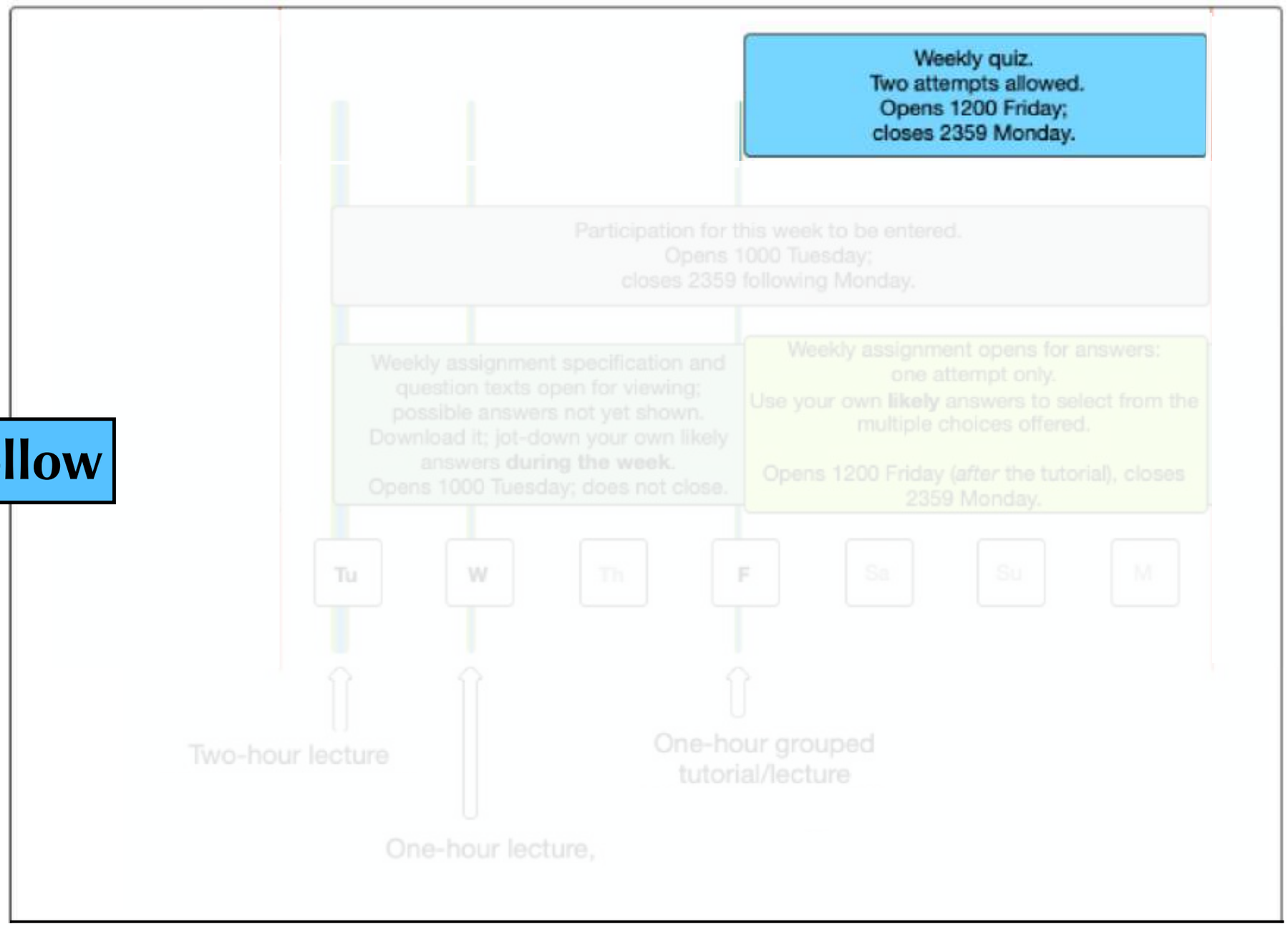


Fri, Sat, Sun, Mon

50%

Weekly quiz, two attempts allowed.
Opens 1200 Friday, closes 2359 Monday.

Fri, Sat, Sun, Mon



Two examples follow

Randomised quiz: Week 3

This program assumes sequence A is in ascending order, and finds two elements of it that sum to x --- if there are any. (The two elements can be the same element in A.) If there are none, it indicates that as well. (See the print statement at the end.)

```
# Post:  high<low and "None there."
     or  0<=low,high<N and A[low]+A[high]==x
```

Make sure you study the invariant before you begin! It says that if there is any x-summing pair in A at all, then there must be one in the part of A that has not yet been examined. Note that the pair does not have to be two different elements: it can be a single element added to itself.

Assertions WTH1 and WTH2 are for you to figure out (but you do not need to say here what they are). They are written there as hints, to indicate that they are important.

Too hard for beginners to do from scratch.

Remember! A statement is incorrect if it does not establish the assertion just after it, even if by some miracle the entire program works anyway. And any statements commented "Decrease variant." must actually do so.

You must “obey” the “what‘s true comments” (assertions) that are there.

Quiz for Week 3...

Screenshot from Moodle

Three different templates for this program, delivered randomly

Weekly quiz.
Two attempts allowed.
Opens 1200 Friday;
closes 2359 Monday.

Week 3...Quiz for...

Program and "what's-true-here" comments	Commentary
`# Pre: 0<=N and A[0:N] ascending`	Precondition.
[⌄]	Initialisation.

`# Inv: If  A[i]+A[j]==x`
`for some 0 <= i,j < N`
`then A[i]+A[j]==x`
`for some low<=i,j<=high .`
	Inv an implication.

`while` [⌄] 	Pair still there?
`# Inv and WTH1`	Inv and Grd.
`# Var: V == high-low`	Capture variant.
`if  A[low]+A[high]` [⌄]	Case 1...

Quiz for Week 3...

Many (randomised) possible answers, only one correct

Program and "what's-true-here" comments	Commentary
`# Pre: 0<=N and A[0:N] ascending`	Precondition.
[ ]	Initialisation.

`# Inv: If A[i]+A[j]==x`
`for some 0 <= i,j < N`
`then A[i]+A[j]==x`
`for some low<=i,j<=high .`
	Inv an implication.

`while` [ ] (low <= high: / low < high: / True: / low != high:)	Pair still there?
`# In` …	Inv and Grd.
`# Va` … `w`	Capture variant.
`if A[low]+A[high]` [ ]	Case 1...

More of Week 3.

```
                                                                    Decrease variant.
   # Invariant maintained.

elif   A[low]+A[high]                                               Case 2...

                                                                    Decrease variant.
   # Invariant maintained.

else: # WTH2                                                        What now?

   # ? <= high-low < V                                             Variant decreased.

   # Invariant still true.

# Inv and ((not WTH1) or WTH2).                                     Implies postcondition.

# Post:  high<low and "None there."
     or  0<=low,high<N and A[low]+A[high]==x

if                                                                  Print outcome.

else:
```

```
                                                    Decrease variant.

   # Invariant maintained.

elif   A[low]+A[high]                               Case 2...

                                                    Decrease variant.

   # Invariant maintained.

else: # WTH2                                        What now?

   # ? <= high-low < V                              Variant decreased.

   # Invariant still true.

# Inv and ((not WTH1) or WTH2).                     Implies postcondition.

# Post:  high<low and "None there."
     or  0<=low,high<N and A[low]+A[high]==x

if                                                  Print outcome.

else
```

A[low]+A[high]==x: print(low,high)
low <= high: print(low,high)
low < high: print(low,high)
A[low]+A[high]!=x: print("None.")

Note! No formality needed for this.

This is the quiz for Week 4

Notice this "formality" of elementary propositional calculus is *after* the previous Quiz (for Week 3).

In particular, it was not a prerequisite for the programming exercise in Week 3.

Place in each box on the right a correct formula from the choices below, so that "IF left-hand side THEN right-hand side". (Reason about each question independently of the others.)

Use (blank) if necessary to evict an answer so that you can put it back somewhere else. (Double-click should also work, however.)

If	(A=>D) and (D=>B)	then	
If	False	then	
If	A	then	
If	A or C	then	
If	B=>C	then	
If	(C=>A) and (C<=A)	then	
If	A=>(not A)	then	
If	not D	then	
If	B	then	

C | True | B=>A | A=>B | A<=D | not A | C or B or A | (blank) | C or B | C or not A

"Eyes" here means that this well be shown larger on the next slide.

Quiz for Week 4

Place in each box on the right a correct formula from the choices below, so that "IF left-hand side THEN right-hand side". (Reason about each question independently of the others.)

Use (blank) if necessary to evict an answer so that you can put it back somewhere else. (Double-click should also work, however.)

If	(A=>D) and (D=>B)	then	
If	False	then	!!!
If	A	then	
If	A or C	then	
If	B=>C	then	
If	(C=>A) and (C<=A)	then	
If	A=>(not A)	then	
If	not D	then	
If	B	then	

C | True | B=>A | A=>B | A<=D | not A | C or B or A | (blank) | C or B | C or not A

Note! No program associated with this.

This is randomised, and indeed "False" implies anything.

But there's only *one way* to get them *all* right.

The emphasis here is *not* on using sequent calculus, or natural deduction, or truth tables, or tableaux, or…

If	(A=>D) and (D=>B)	then	A=>B
If	False	then	C
If	A	then	B=>A
If	A or C	then	C or B or A
If	B=>C	then	True
If	(C=>A) and (C<=A)	then	C or not A
If	A=>(not A)	then	not A
If	not D	then	A<=D
If	B	then	C or B

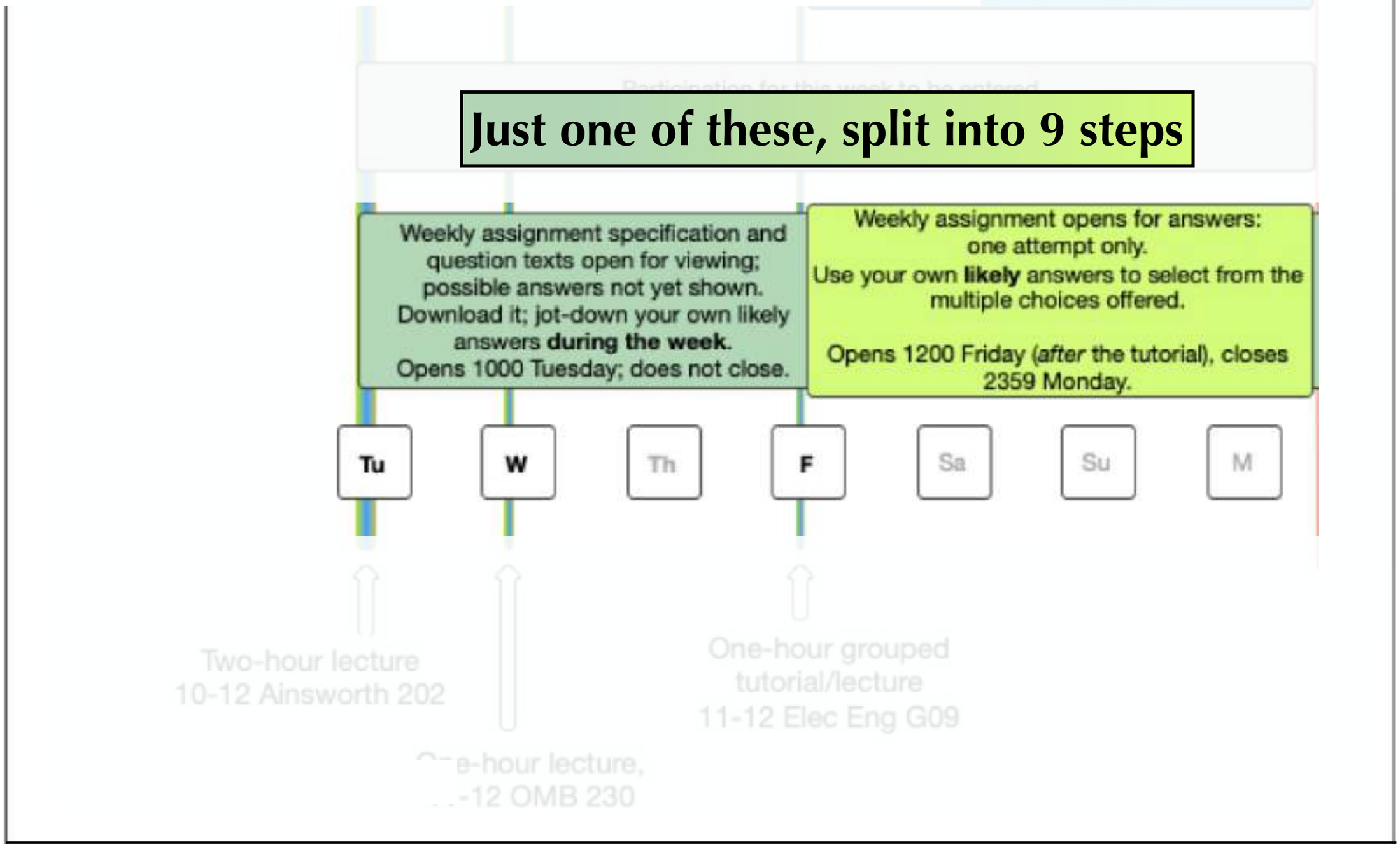
It's not about formality.

C | True | B=>A | A=>B | A<=D | not A | C or B or A | (blank) | C or B | C or not A

It is *instead* about making your propositional reasoning as familiar as your reasoning about arithmetic. You don't need PEANO to check your shopping bill.

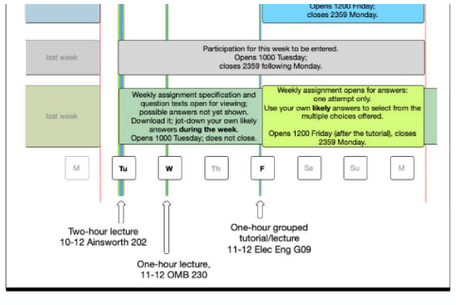
Weekly assignment-portion

Questions revealed on **Mon**, but possible answers revealed only on **Fri.**



Weekly stages of the whole-term assignment

Questions revealed on **Mon**, but possible answers revealed only on **Fri.**



Univ. of New South Wales
Computer Science
COMP 6721

Assignment Part I
Weeks 1–5

This is Part I of COMP 6721's single Programming Assignment, delivered one portion per week for Weeks 1–5. It will be followed, after flex week, by Part II for Weeks 7-10. It is (thus) divided into 5+4 = 9 portions overall.

Every Tuesday morning during term (excluding flex), the coming week's portion of the Assignment text and its questions –but not yet their possible multi-choice answers– will be added here, and students should from that point begin to think about what their answers to the current week's portion should be. Clarification may *of course* be sought during the week — but Friday's lecture/tutorial will *not* focus on the Assignment. Most of Friday's lecture will instead be based on other exercises, from the course textbook or elsewhere, and will be done in small groups with lecturer/tutor assistance and encouragement.

Only on Friday afternoon (i.e. after the lecture/tutorial) will the week's Assignment multiple-choice answers become available on Moodle: from then until the deadine on the following Monday at midnight there will be a fixed time-limit for answering (say 30 minutes, but possibly varying from week to week), and only *one* submission is allowed.

Note that some of the multi-choice questions are "select all that apply", and others are "select one only". *Be careful not to select only one answer in Moodle when several might be sought.*

To get the best result, you should print the current week's Assignment portion as soon as it is available (i.e. Tuesday) and work on it gradually, on your own throughout the week, by jotting down your best answers in the spaces provided. The tutors and the lecturer will be very happy to give (appropriately limited) advice: the Forum is a good place to ask for clarification if necessary.

Beginning only on Friday afternoon is definitely not recommended. Start on Tuesday, and pose questions –if you have them– during the week. Remember however that your answers should come from you alone.

Each portion has equal weight, with all 9 (weeks) together taken to be the overall Assignment submission for the course as a whole.

The Assignment has so-called "optional questions" which are **not assessed** –you do not hand them in– and in particular your answers to them do not count at all towards participation (though asking questions about them might).

Attempting the optional questions however will definitely help you to understand the material that *is* assessed.

[0] The course-text is now called *Formal Methods, Informally: Writing programs that work.*

…the (big) assignment: a *single* program constructed step-by-step throughout the whole term.

Assignment Part I: Weeks 1–5

This is Part I of COMP 6721's single Programming Assignment, delivered one portion per week for Weeks 1–5. It will be followed, after flex week, by Part II for Weeks 7-10. It is (thus) divided into 5+4 = 9 portions overall.

Every Tuesday morning during term (excluding flex), the coming week's portion of the Assignment text and its questions –but not yet their possible multi-choice answers– will be added here, and students should from that point begin to think about what their answers to the current week's portion should be. Clarification may *of course* be sought during the week — but Friday's lecture/tutorial will *not* focus on the Assignment. Most of Friday's lecture will instead be based on other exercises, from the course textbook or elsewhere, and will be done in small groups with lecturer/tutor assistance and encouragement.

Only on Friday afternoon (i.e. after the lecture/tutorial) will the week's Assignment multiple-choice answers become available on Moodle: from then until the deadine on the following Monday at midnight there will be a fixed time-limit for answering (say 30 minutes, but possibly varying from week to week), and only *one* submission is allowed.

1 Simple searching: is character x in the string target ?

1.1 Do what comes naturally

We begin with perhaps the of the simplest programming examples of all: finding whether some character occurs in a string. Program 1 (below) does that in just a single line of Python:

Program 1

```
# -- Find whether a single character in some character string.

found= x in target # This is the whole program!

# Post: Boolean 'found' is whether 'x' occurs in 'target'.
```

The "# --" comment at the beginning of the program is an informal description of what the program is for. Think of it as advertising, or what might be a table-of-contents entry in a program library. With it, you could decide whether this program could be useful for you; but that comment alone is not enough to tell you exactly *how* to use the program.[1]

More significantly (for us) is the "# Post:" comment at the end of the program: it is the "post condition", a so-called "what's true here" -style statement — which style we abbreviate WTH. It articulates what the program has made true at that point. (But it does *not* explain how it did it, nor why.)

Optional Exercise 1 *(not marked)* Type Prog. 1 in, plus whatever else you need to set the inputs x and target, and print the output found. See for yourself that you really can use the operator "in" for that purpose in Python. (Section 14 might be helpful.)

(Remember that Optional Exercises are not handed in, thus not marked.) ⇐⇐⇐ □

Weekly assignment specification and question texts open for viewing; possible answers not yet shown. Download it; jot-down your own likely answers **during the week**. Opens 1000 Tuesday; does not close.

While we're here, and just to get started: are you tempted to add parentheses to Prog. 1, writing "found= (x in target)" ?

Resist the temptation: it's far more important to be comfortable with Booleans as first-class variables. Would you include parentheses in the assignment statement "x= (1+2)" ? The Boolean values True and False are just as legitimate as the integers 1 and 2; and "+" and "in" are both simple infix operators. Unless there's a good reason, all types should be treated in the same way.

[1] It would be useful for a computer-literate secretary whom you have asked to find such a program for you: but he would not need to know Python at all.

1 Simple searching: is character x in the string target?

1.1 Do what comes naturally

We begin with perhaps the of the simplest programming examples of all: finding whether some character occurs in a string. Program 1 (below) does that in just a single line of Python:

Program 1

```
# -- Find whether a single character in some character string.

found= x in target # This is the whole program!

# Post: Boolean ‘found’ is whether ‘x’ occurs in ‘target’.
```

Program 1

```
# -- Find whether a single character in some character string.

found= x in target # This is the whole program!

# Post: Boolean ‘found’ is whether ‘x’ occurs in ‘target’.
```

(Remember that Optional Exercises are not handed in, thus not marked.) □

While we're here, and just to get started: are you tempted to add parentheses to Prog. 1, writing "`found= (x in target)`"?

Resist the temptation: it's far more important to be comfortable with Booleans as first-class variables. Would you include parentheses in the assignment statement "`x= (1+2)`"? The Boolean values `True` and `False` are just as legitimate as the integers 1 and 2; and "`+`" and "`in`" are both simple infix operators. Unless there's a good reason, all types should be treated in the same way.

[1] It would be useful for a computer-literate secretary whom you have asked to find such a program for you: but he would not need to know Python at all.

Question 1 In everyday prose (like this), "Boolean" is often written capitalised; but "integer" is usually not. Why is that? □

Question 2 Some say that commutative operators should have symmetric symbols for them. Consider the infix bitwise-**and** operator "`&`" in *C* — is it commutative? Is its "`&`" symbol symmetric?[2] □

Question 3 As for Question 2, but this time for "`&&`" in *C*. Is the "`&&`" operator commutative? Is the "`&&`" symbol symmetric? □

Question 4 Why do I (as a personal preference) write "`=`" left-adjusted in assignment statements, for example "`x= y`" rather than "`x = y`"? □

Question 5 Why don't we need to write "`found==True`" in the postcondition of Prog. 1? □

`Python` –for our purposes here– could be called a "wide spectrum" language, because it has all the usual (low level) features — but it has also what could be called "high level" features built-in, like the "`in`" operator (used above) that can do quite sophisticated things — like searching for a character in a string.[3]

An advantage of such a "high and low" combination in a single language is that it is easy to develop an intricate program by introducing any complications only gradually, *testing* each of the resulting "prototypes" as you go. (The complications are sometimes for efficiency, sometimes because your ultimate target language does not have the features you are using at the moment.)

1.2 What if our programming language is *not* `Python`?

If our eventual implementation language is not `Python`, then indeed Prog. 1 above can be only a prototype, because we cannot run it "as is" on our target system. Rather than `Python`, we might require something more basic, for example *C* or even even assembly-code deep in an operating system, neither (perhaps) having a built-in operator "`in`". Should we start off our programming project in *that* language instead, say with something like Prog. 3 below? (We'll use *C* rather than assembly to make this point.[4])

[2] The "`&`" here is not the prefix "address of" operator, although it has the same symbol. And see the course-text for what "commutative" means, if you're not sure: use the index.

[3] As we'll see later, `Python`'s "`in`" operator (also called a "relation" because its result-type is Boolean) can do much more that that. Also, many other languages (e.g. *C*) have such features too, but supplied by standard libraries rather than built-in to the language.

[4] Historically *C* was designed just for that purpose, to avoid having to use assembly code for the original Unix operating system, but still to be close to the (PDP-11) hardware on which the system ran — which hardware, incidentally, had "auto-inc" and "auto-dec" address-indexing modes. What does that remind you of?

Further, with the operating system writen (mostly) in *C*, rather than PDP-11 assembly, it became possible to "port" Unix from one machine architecture to another simply by writing a *C* compiler for the other architecture. The first *in the world* to do that was Richard Miller at the **University of Wollongong**: the target machine was an "Interdata", and examples of both machines can be seen in the stairwell of Building K17.

What's the use of these !?

Question 1 In everyday prose (like this), "Boolean" is often written capitalised; but "integer" is usually not. Why is that? □

Question 2 Some say that commutative operators should have symmetric symbols for them. Consider the infix bitwise-**and** operator "`&`" in *C* — is it commutative? Is its "`&`" symbol symmetric?[2] □

Question 3 As for Question 2, but this time for "`&&`" in *C*. Is the "`&&`" operator commutative? Is the "`&&`" symbol symmetric? □

Question 4 Why do I (as a personal preference) write "`=`" left-adjusted in assignment statements, for example "`x= y`" rather than "`x = y`"? □

Question 5 Why don't we need to write "`found==True`" in the postcondition of Prog. 1? □

To encourage curiosity in general.
They are *not* about writing a particular program.

3 More ambitious searching, this time for one *string* in another

3.1 Do what comes naturally: again

Looking back at Sec. 1.1, we recall that the algorithmic pattern was

general scheme

```
initialise an index to the beginning of 'target'
while 'target' is defined at the index, but is not what we're looking for :
    increment the index
```

We now use the same idea to search for a source *string* rather than just a single character. After all, it's essential to be able to be inspired by algorithmic *ideas* — nothing wrong with that. What we're trying to reduce is reliance on algorithmic "in-our-head simulation" for checking our programs' correctness, and to increase the benefit we get by thinking "WTH?" as well. [13]

As before, the module provided (Sec. 14) can fetch our `source` and `target` strings from the command-line. Copying it to `Parameters.py` somewhere allows your test-programs to use it like this:

Import Parameters

```
import Parameters # See Sec. 14.
source,target= Parameters.Get() # Get source and target string.
S,T= len(source),len(target)
```

But you can of course use your own input-code instead.

If we follow the algorithmic pattern from earlier, using `source` and `target` instead of `x` and `target`, we'd get something like

Program 9

```
t= 0
while DDD and source!=target[t:t+S]: t= t+1      # [14]
```

In the second conjunct, we have used the fact that `Python` can compare whole strings as a primitive operation, as in `source!=target[t:t+S]`. Note however that CCC (from Prog. 4) has changed, to become DDD.

[13] In both cases, however, we must test our programs afterwards — or even during.

[14] In fact in `Python` you can write just the Boolean "`source in target`" for the result of the whole program. But we use a loop, because we are trying to reach lower-level code, avoiding "`in`", even while remaining in `Python`.

3 More ambitious searching, this time for one *string* in another

If we follow the algorithmic pattern from earlier, using `source` and `target` instead of `x` and `target`, we'd get something like

Program 9

```
t= 0
while DDD and source!=target[t:t+S]: t= t+1      #14
```

In the second conjunct, we have used the fact that `Python` can compare whole strings as a primitive operation, as in `source!=target[t:t+S]`. Note however that CCC (from Prog. 4) has changed, to become DDD.

which they will remember was `t<T and x!=target[t]`

4 Program Algebra: a more basic program

Week 4

4.1 Transform while preserving correctness

We now (finally!) address the issue of the string-to-string comparison in the loop-guard: how do we reduce it to character-to-character comparisons only? Here is Program 11 again, without its WTH annotations:

Program 13, that is Prog. 11 without its WTH annotations.

```
t= 0
# Inv: as in Prog. 10 earlier.
while DDD and source!=target[t:t+S]: t= t+1
```

Since (we imagine) comparisons of whole strings will not be allowed in our (final) programming language, the loop-guard's "`source!=target[t:t+S]`" will not be "transliteratable". So we have to replace it by something more basic.

Suppose –worse– that we have been given that job, of transforming Prog. 13 into an equivalent program (without string-to-string comparison),

!!! *but we don't even know what Prog. 13 is supposed to do.*

We have simply been asked to transform it — without knowing its purpose.[19] What's important is (only) that the new version does whatever the old version did (whatever that was), but without using string-to-string comparisons.

We'll use program *algebra*, so called because $x+y = y+x$, an example of *arithmetic* algebra, is true no matter which numbers x and y stand for, or what physical things those numbers might describe: we can make that substitution anywhere. Similarly, in *program* algebra we want to replace one program above by another one that does just what the original program did, *no matter what that was*; and then use it in place of the original — *anywhere we like*.

[19] "Transform" is being used here somewhat informally: it's somewhere between "translate" and "transliterate". The main point remains that we do not need to think about the program's purpose.

!!!

Program 13, that is Prog. 11 without its WTH annotations.

```
t= 0
# Inv: as in Prog. 10 earlier.
while DDD and source!=target[t:t+S]: t= t+1
```

```
# Inv: 0<=t<=T
# and  (forall n | 0<=n<t and EEE · source!=target[n:n+S])
```

It's OK to use advanced idioms *in the assertions*!

n+S<=T

Program 13, that is Prog. 11 without its WTH annotations.

```
t= 0
# Inv: as in Prog. 10 earlier.
while DDD and source!=target[t:t+S]: t= t+1
```

```
# Inv: 0<=t<=T
# and  (forall n | 0<=n<t and EEE · source!=target[n:n+S])
```

Some program algebra takes Prog. 13 to Program 14.

```
        t= 0
        while DDD and source!=target[t:t+S]: t= t+1

=       t= 0
        while DDD:
            if source==target[t:t+S]: break # <-- Replace this with ↓.
            t= t+1

=       # The grey text remains the same.
        t= 0
        # Inv: as above in Prog. 10.
        while DDD:

            # This replaces ↑.
            s= 0
            # Inv1: 0<=s<=S and source[:s]==target[t:t+s]
            while s!=S and source[s]==target[t+s]: s= s+1
            if FFF: break

            t= t+1
```

Program 13, that is Prog. 11 without its WTH annotations.

```
t= 0
# Inv: as in Prog. 10 earlier.
while DDD and source!=target[t:t+S]: t= t+1
```

```
# Inv: 0<=t<=T
# and  (forall n | 0<=n<t and EEE · source!=target[n:n+S])
```

not found yet

Now we have *two* invariants, which we *discovered* simply by being methodical.

Some program algebra takes Prog. 13 to Program 14.

```
        t= 0
        while DDD and source!=target[t:t+S]: t= t+1

=       t= 0
        while DDD:
            if source==target[t:t+S]: break # <-- Replace this with ↓.
            t= t+1

=       # The grey text remains the same.
        t= 0
        # Inv: as above in Prog. 10.
        while DDD:

            # This replaces ↑.
            s= 0
            # Inv1: 0<=s<=S and source[:s]==target[t:t+s]
            while s!=S and source[s]==target[t+s]: s= s+1
            if FFF: break

            t= t+1
```

!!!

equal so far

And the first one we had already, from earlier (simpler) work.

Program 21.py

```
# WTH-comments omitted.

import Parameters
source,target= Parameters.Get()
S,T= len(source),len(target)

s,t= 0,0
while GGG:
    if source[s]==target[t+s]: s,t= HHH,III
    else: s,t= JJJ,KKK

if s==S:
    print(f"Found '{source}' at target[{t}:{t+S}]=='{target[t:t+S]}'.")
else: print(f"Did not find '{source}' in '{target}'.")
```

The code needed here is generated by the invariants we already have.

No nested loop necessary!!!

Mid-term break coming: and *already they have done something many could not have done before*.

The cost is minimal: **just a different way of thinking**.

This way of thinking is **not hard**. It has (unfortunately) merely been forgotten.

Program 21.py

```
# WTH-comments omitted.

import Parameters
source,target= Parameters.Get()
S,T= len(source),len(target)

s,t= 0,0
while GGG:
    if source[s]==target[t+s]: s,t= HHH,III
    else: s,t= JJJ,KKK

if s==S:
    print(f"Found '{source}' at target[{t}:{t+S}]=='{target[t:t+S]}'.")
else: print(f"Did not find '{source}' in '{target}'.")
```

The code needed here is generated by the invariants we already have.

No nested loop necessary!!!

Mid-term break coming: and *already they have done something many could not have done before*.

The cost is minimal: **just a different way of thinking**.

This way of thinking is **not hard**. It has (unfortunately) merely been forgotten.

It is, after all, the principal message from Floyd's ground-breaking paper in 1967:

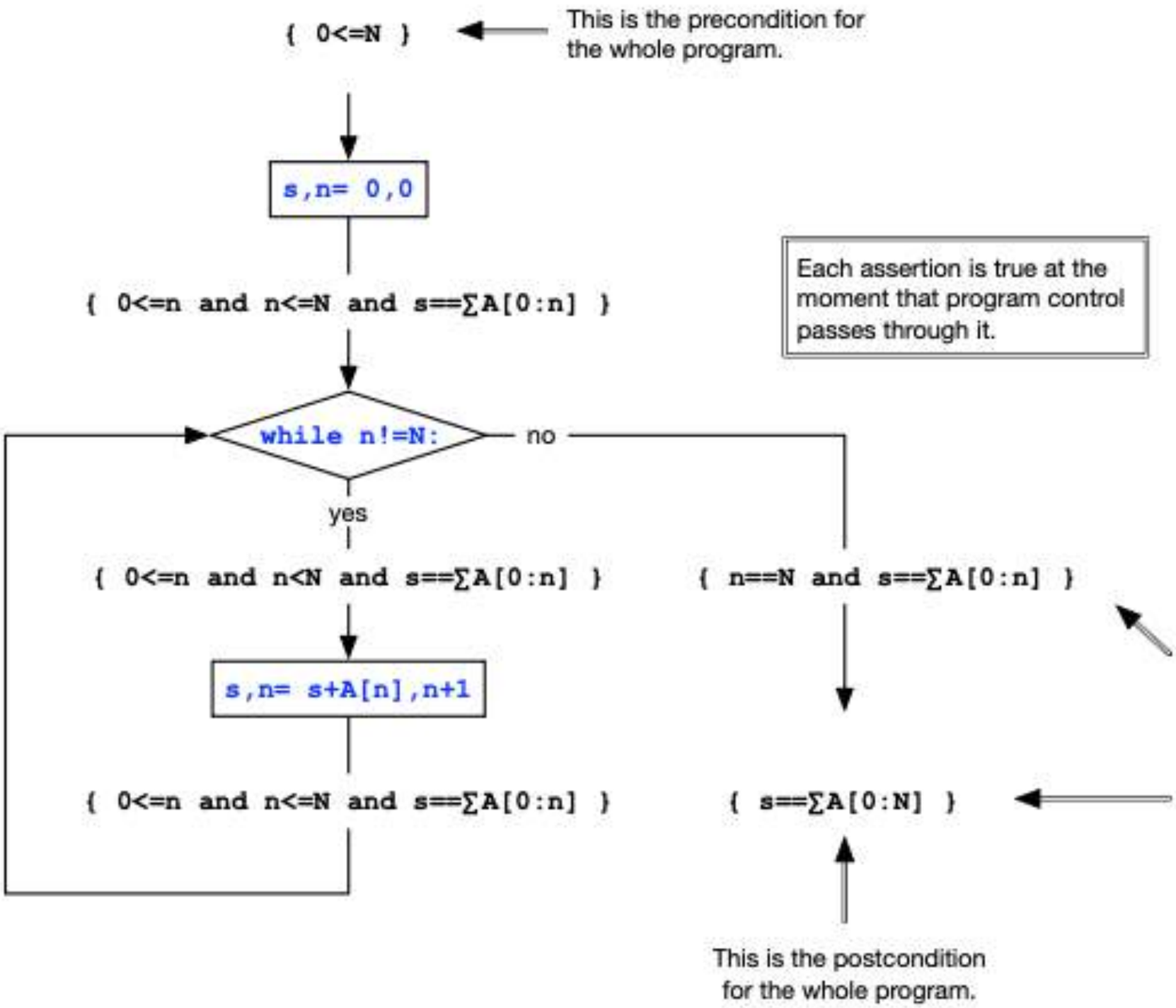
*"Annotate the **lines**, not the boxes".*

First-year students ***can*** understand that, ***if*** it's pointed out to them.

1967!

We've had 60 years to do that.

Figure 1.1 Flowchart for program (1.17)



It is, after all, the principal message from Floyd's ground-breaking paper in 1967:

"Annotate the ***lines***, not the boxes".

1967!

We've had 60 years to do that.

First-year students ***can*** understand that, ***if*** it's pointed out to them.

6 Can it go faster?

We will finish off by discussing the whole program Prog. 21 of Sec. 5 just above: it can be made *drastically* more efficient in some cases, and here are two examples.

Suppose `source` is "`abac`" and `target` is "`ababac`". Program 21 will start with `s,t= 0,0`, going on to match "`aba-`" against "`aba---`"; but then the comparison of "`c`" at `source[3]` against "`b`" at `target[0+3]` will fail. At that point we'll have `s,t==3,0`, and Prog. 21 will take its `else`-branch.

Comparisons will then restart after the `else`'s assignment `s,t= JJJ,KKK`. But (at least) the first new comparison is pointless: it is already known that `target[1]` is "`b`" (because previously it matched `source[1]`) and so `source[0]` (which is "`a`") cannot possibly match `target[0+1]`. Thus `t` can be set to some higher value than `KKK` in this case, leading to a faster progression through `target`.

A second, different efficiency opportunity is illustrated when `source` is "`aaab`" and `target` is "`aaac`". Again Prog. 21 will start with `s,t= 0,0`, this time matchin `source[3]` against "`c`"

And again the normal KKK *does* ap hey are both "`aa`". Here just above): the new `s` sl because it is already know ed.

Understar the topic of Part II of this Assignment.

Program 21.py

```
# WTH-comments omitted.

import Parameters
source,target= Parameters.Get()
S,T= len(source),len(target)

s,t= 0,0
while GGG:
    if source[s]==target[t+s]: s,t= HHH,III
    else: s,t= JJJ,KKK

if s==S:
      print(f"Found '{source}' at target[{t}:{t+S}]=='{target[t:t+S]}'.")
else: print(f"Did not find '{source}' in '{target}'.")
```

See you in Week 7: have a nice Flex!

Flexibility week

This is where *curiosity* pays off: they will spend part of their "off week" wondering *about this course*

Five weeks later…

End of term

"Secret" — the students do not know until this point what program they have made.

13 The complete program

Optional Exercise 8 *(not marked)*

Congratulations! You've now finished coding the complete algorithm, which searches for `source` in `target` *in time linear in the sum of the string lengths.*

Enjoy it by putting all of our work together: the main-program code from Prog. 25 and the longest-*prepo*-memoising code from Prog. 30 made into a function as above, `Prepoes(string)` whose returned value is assigned to `jumps`.

Be sure to test your program.[36] □

Summary *Seize the day*

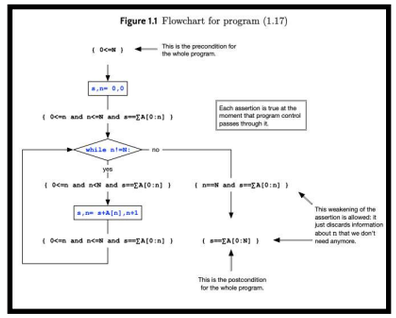
Teach *rigorous* **methods**

---

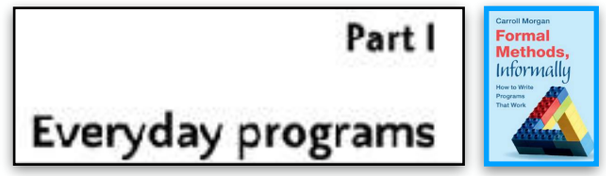
to first-years
without formal logic
the benefit is huge

It *can* be done.

annotate lines,
not boxes

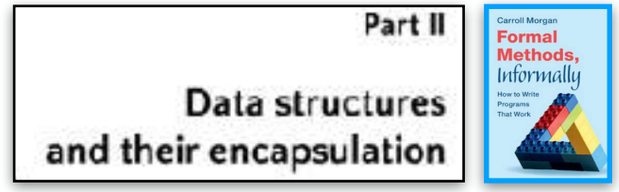


Teach rigorous, Floyd-style sequential reasoning *without* requiring formal logic first.

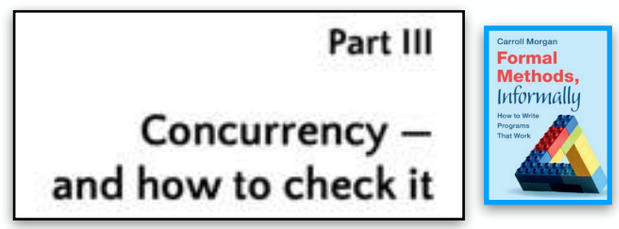


Teach data-refinement and encapsulation *without* requiring formally expressed "commuting diagrams" between layers.

(not covered in this talk)



Teach concurrency *without* requiring labelled transition systems and (co-)simulations.



Educate –and recruit– your teaching assistants, and assess all that using auto-marking as far as possible.

COMP 6721

"After taking this course I felt as if I was privy to a powerful ancient magic. Everything that was introduced was so useful yet not yet known to most CS students."

"This course radically changed the way I view programming, and has certainly improved my programming skills immensely. This course should be compulsory for all Computer Science students, or have its contents integrated into first year."

Thanks for your attention!

Zhiming Liu
Zili Zhang
Jonathan Bowen
SETSS Chongqing

Teacher's manual 2014

UNSW
Student view 2023

Dongol et al
ACM paper 2024

CUP text 2026

Another student's-eye view

I did COMP6721 (Informal Methods) in **Term 2 2023**.

The subject is a great introduction to formal verification, focusing on high-level concepts and their practical applications and providing a solid foundation for future mathematically heavy verification courses. Overall, the subject has a moderate workload and great potential for high marks when consistent effort is made to stay up-to-date and revise the content throughout the term. I would highly recommend this course to anyone who wants to write better code and be more aware of the fundamental logic of basic control structures…

(my emphasis)

(there's more)

The entire term has one big assignment and each week is one segment of this that slowly builds up to a really elegant final solution. The assignment walks you through everything you need, laying out a set of premises and asking you what the conclusion would be.

For example, some assignment questions give you a template for a program to achieve a specific purpose and ask you what the loop guard or invariant is that results in it achieving its purpose.

chatGPT and Chapter 2

me

```
Write a Python program which, given a sequence A
of numbers of length N, finds the length of the
longest subsegment of A that does not contain a
Bad subsegment, where Bad(n) is true just when
A[n:n+3] is Bad.
```

chatGPT

```
"To solve this problem, we need to find the longest
continuous subsegment of an array A that does not
contain any "Bad" subsegment of length 3. A
subsegment of A from index i to j (inclusive) is
valid if it does not contain any index n such that
A[n:n+3] is Bad and i \le n \le j-2.

"We can solve this efficiently in O(N) time using a
sliding window (or two-pointer) approach.Here is the
complete Python program. You will need to replace
the placeholder is_bad_three_elements function
with your specific definition of what makes three
elements "Bad".pythondef
is_bad_three_elements(a,b,c):"
```

```
def find_longest_good_subsegment(A):
    """
    Finds the length of the longest subsegment of A that
    does not contain any Bad subsegment of length 3.
    """
    N = len(A)
    if N < 3:
        return N

    # Step 1: Precalculate which starting indices 'n'
    # trigger a Bad subsegment
    # bad_start[n] will be True if A[n:n+3] is Bad
    bad_start = [False] * N
    for n in range(N - 2):
        if is_bad_three_elements(A[n], A[n+1], A[n+2]):
            bad_start[n] = True

    # Step 2: Sliding window to find the longest valid subsegment
    max_length = 0
    left = 0

    # We maintain the latest (rightmost) index
    # where a Bad subsegment starts
    # inside or just before our current window.
    last_bad_start = -1
```

```
# We maintain the latest (rightmost) index
# where a Bad subsegment starts
# inside or just before our current window.
last_bad_start = -1

for right in range(N):
    # If the window expands to include a new 3-element
    # block ending at 'right',
    # check if the start of this block (right - 2)
    # is marked as Bad.
    if right >= 2 and bad_start[right - 2]:
        last_bad_start = right - 2

    # If a Bad subsegment starts inside our current window,
    # we must shrink the window by moving 'left' past that
    # Bad start index.
    if last_bad_start >= left:
        left = last_bad_start + 1

    # Calculate the length of the current valid window
    current_length = right - left + 1
    if current_length > max_length:
        max_length = current_length

return max_length
```

Convince them that they *need* this course

In the very first lecture, there was a surprise quiz on "Binary Search". It was marked later –by hand– and returned only at the very end of term, but it did not contribute to assessment.

It was more to show *the students* how much they had learned.

Like *Outward Bound*

COMP 6721
2019 T2

(In-) Formal Methods
The Lost Art

Student number

Your program reads from three variables:

- a sorted array A of integers, of size N, indexed from 0 (inclusive) to N (exclusive). Thus the array elements are
$$A[0] \leq A[1] \leq \cdots \leq A[N-1] .$$
- an integer x to be sought in the array.

The program writes to one variable n, which is to be the index of x in A if it is there. If x is not there, then n should index the first value in A that is greater than x. If no value in A is greater than x, then n should be set to N.

You can use other, temporary variables.

"Eyes" here means that this will be shown larger on the next slide.

Hand-written, informal

Hand-written, informal

Your program reads from three variables:

- a sorted array A of integers, of size N, indexed from 0 (inclusive) to N (exclusive).

  Thus the array elements are

  $$A[0] \leq A[1] \leqslant \cdots \leq A[N-1] .$$

- an integer x to be sought in the array.

Hand-written, informal

The program writes to one variable n, which is to be the index of x in A if it is there. If x is not there, then n should index the first value in A that is greater than x. If no value in A is greater than x, then n should be set to N.

You can use other, temporary variables.

Binary Search

COMP 6721
2019 T2

(In-) Formal Methods
The Lost Art

sorted:
A[0] ≤ A[1] ... etc.

last element is A[N-1]

looking for

```
var A[0,N), x, n
```

where found (if there)

```
var l,h,m
```

low, high, middle

assignment

```
l,h := [          ]
```

multiple assignment: two values separated by a comma

```
while l [ ] h do
```

<? ≤? ≠? which?

```
m:= [          ]
```

use ÷ for integer division

```
if A[m] [ ] x then [          ]
```

<? ≤? ≠?

```
else [          ]
```

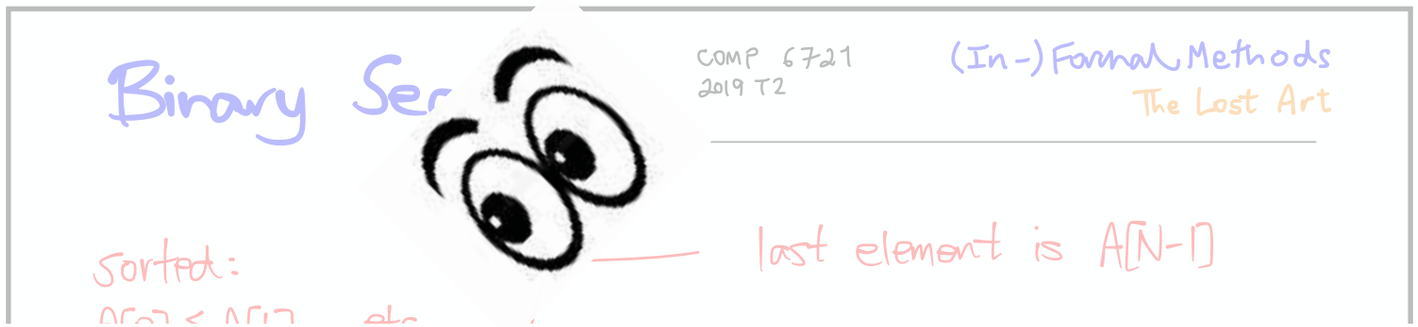
```
end
[          ]
```

change l? change h?

set n to the correct value

CCM
Week 1

Just fill-in the blanks

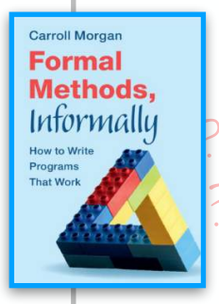


[5] In *Programming Pearls*, Jon Bentley wrote that professional programmers who wer
of hours", to code binary search, in "the language of their choice", at the end –in
reported that they had found correct code for the task. But once they tested the
about half-an-hour each), apparently some 90% of them found bugs [8, p. 36].

Bentley goes on to say that Knuth, in his *Sorting and Searching*, wrote that the fi
program was published in 1946, but the first published *bug-free* version did not a
[33, Sec. 6.2.1].

Bug free? Perhaps not yet: the binary-search *overflow* bug was discovered on

Ex. 18.1.



p.199

For the students, the success rate was about the same as in Bentley's case:

only ~10% .

Binary Search

COMP 6721
2019 T2

(In-) Formal Methods
The Lost Art

sorted:
A[0] ≤ A[1] ... etc.

last element is A[N-1]

looking for

where found (if there)

```
var A[0,N), x, n
var l,h,m
```

low, high, middle

assignment

```
l,h := 
```

```
while l 
```

> In *Programming Pearls*, Jon Bentley wrote that professional programmers who were given "a couple of hours", to code binary search, in "the language of their choice"; at the end –in almost all cases– reported that they had found correct code for the task. But once they tested their programs (for about half-an-hour each), apparently some 90% of them found bugs [8, p. 36]. Bentley goes on to say that Knuth, in his *Sorting and Searching*, wrote that the first binary-search program was published in 1946, but the first published bug-free version did not appear until 1962 [33, Sec. 6.2.1]. Bug free? Perhaps not yet: the binary-search overflow bug was discovered only in 2006. See Ex. 18.1.

change l? change h?

set n to the correct value

CCM
Week 1

For the students, the success rate was about the same as in Bentley's case:

only ~10% .