M269 Revision 2019

Exam 2017J

Contents

1 M269 Exam Revision Agenda & Aims

1. Welcome and introductions
2. Revision strategies
3. M269 Exam — Part 1 has 15 questions 65%
4. M269 Exam — Part 2 has 2 questions 35%
5. M269 Exam — 3 hours, Part 1 80 mins, Part 2 90 mins
6. M269 2017J exam (June 2018)
7. Topics and discussion for each question

8. Exam techniques
9. These slides and notes are at http://www.pmolyneux.co.uk/OU/M269/M269ExamRevision/

1.1 Introductions & Revision Strategies

- Introductions
- What other exams are you doing this year ?
- Each give one exam tip to the group

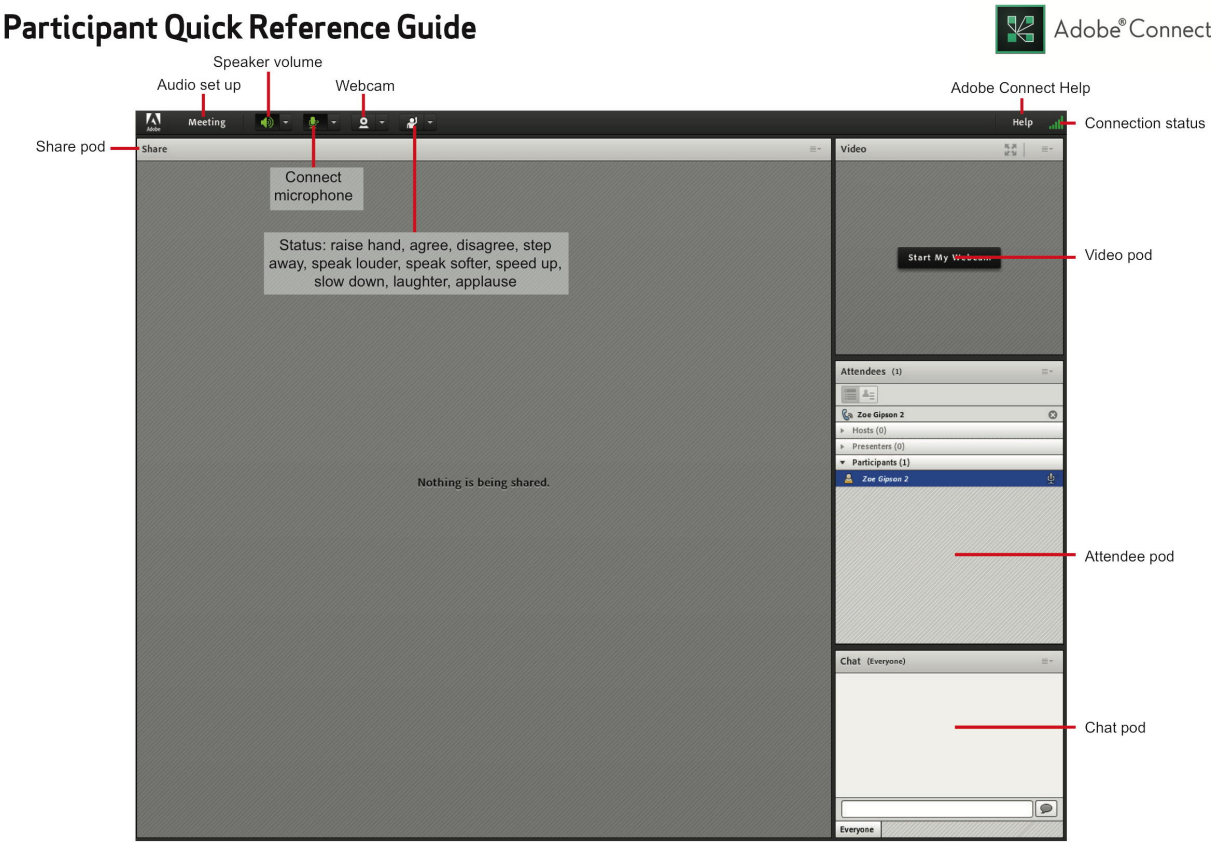
1.2 M269 Exam 2017J

- Not examined this presentation:
- Unit 4, Section 2 String search
- Unit 7, Section 2 Logic Revisited
- Unit 7, Section 4 Beyond the Limits

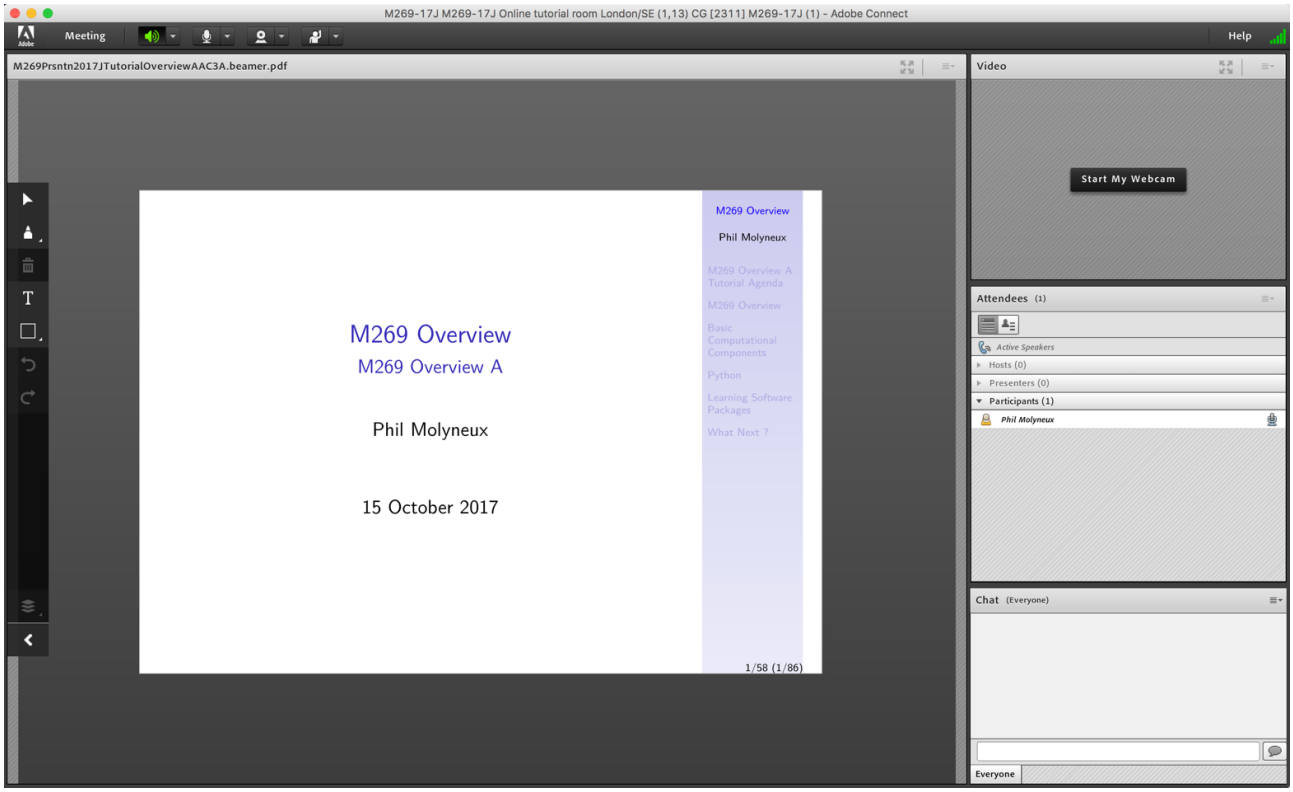
2 Adobe Connect Interface and Settings

2.1 Adobe Connect Interface — Student View

Adobe Connect Interface — Student Quick Reference



Adobe Connect Interface — Student View



2.2 Adobe Connect Settings

Adobe Connect Settings

- **Everybody: Audio Settings** Meeting › Audio Setup Wizard...
- **Audio** Menu bar › Audio › Microphone rights for Participants ✔
- Do not *Enable single speaker mode*
- **Drawing Tools** Share pod menu bar › Draw (1 slide/screen)
- Share pod menu bar › Menu icon › Enable Participants to draw ✔ gray
- Meeting › Preferences › Whiteboard › Enable Participants to draw ✔
- Cancel hand tool
- Do not enable green pointer...
- Meeting › Preferences › Attendees Pod Disable *Raise Hand notification*
- **Cursor** Meeting › Preferences › General tab › Host Cursors › Show to all attendees ✔ (default *Off*)
- Meeting › Preferences › Screen Share › Cursor › Show Application Cursor
- **Webcam** Menu bar › Webcam › Enable Webcam for Participants ✔
- **Recording** Meeting › Record Meeting... ✔

Adobe Connect — Access

- **Tutor Access**
- TutorHome › M269 Website › Tutorials
- Cluster Tutorials › M269 Online tutorial room
- Tutor Groups › M269 Online tutor group room
- Module-wide Tutorials › M269 Online module-wide room
- **Attendance**

  TutorHome › Students › View your tutorial timetables
- **Beamer Slide Scaling** 440% (422 x 563 mm)
- **Clear Everyone's Status**

  Attendee Pod › Menu › Clear Everyone's Status
- **Grant Access**

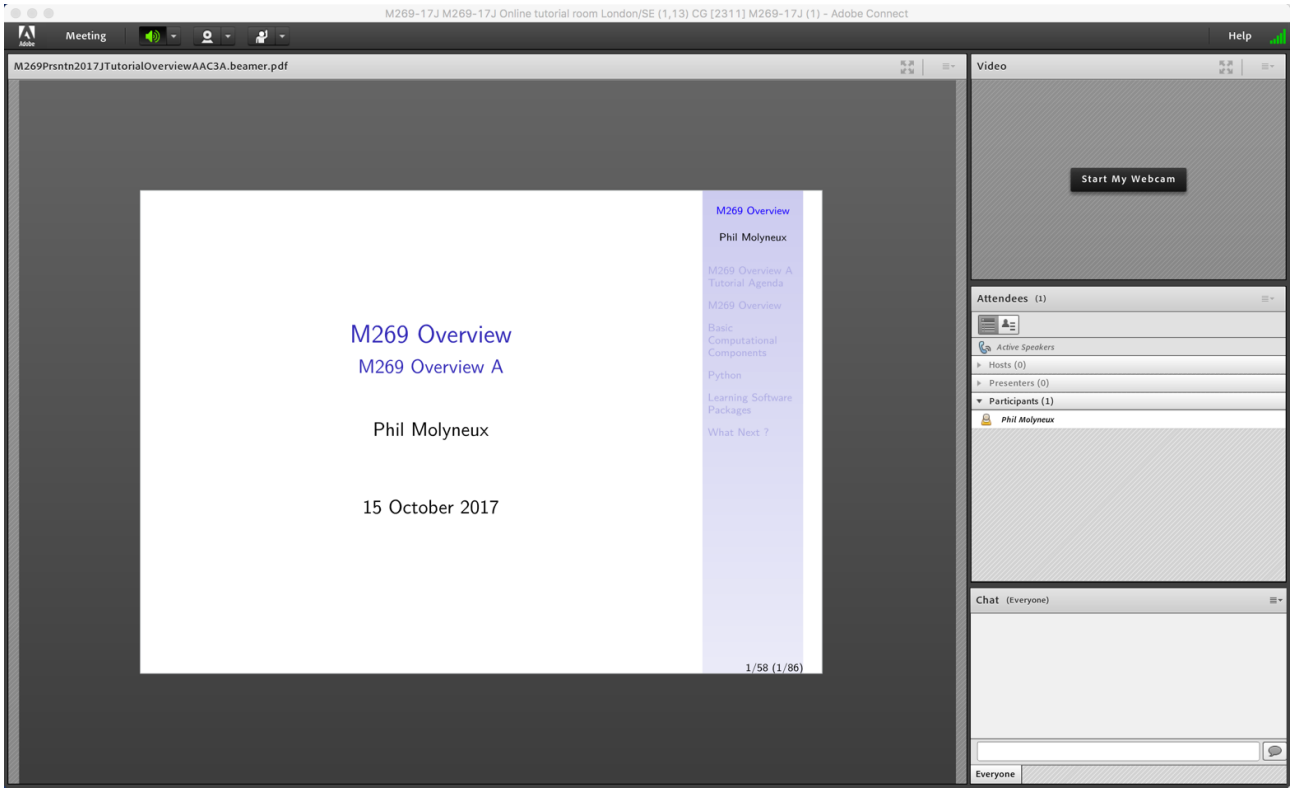
  Meeting › Manage Access & Entry › Invite Participants... and send link via email

Adobe Connect — Keystroke Shortcuts

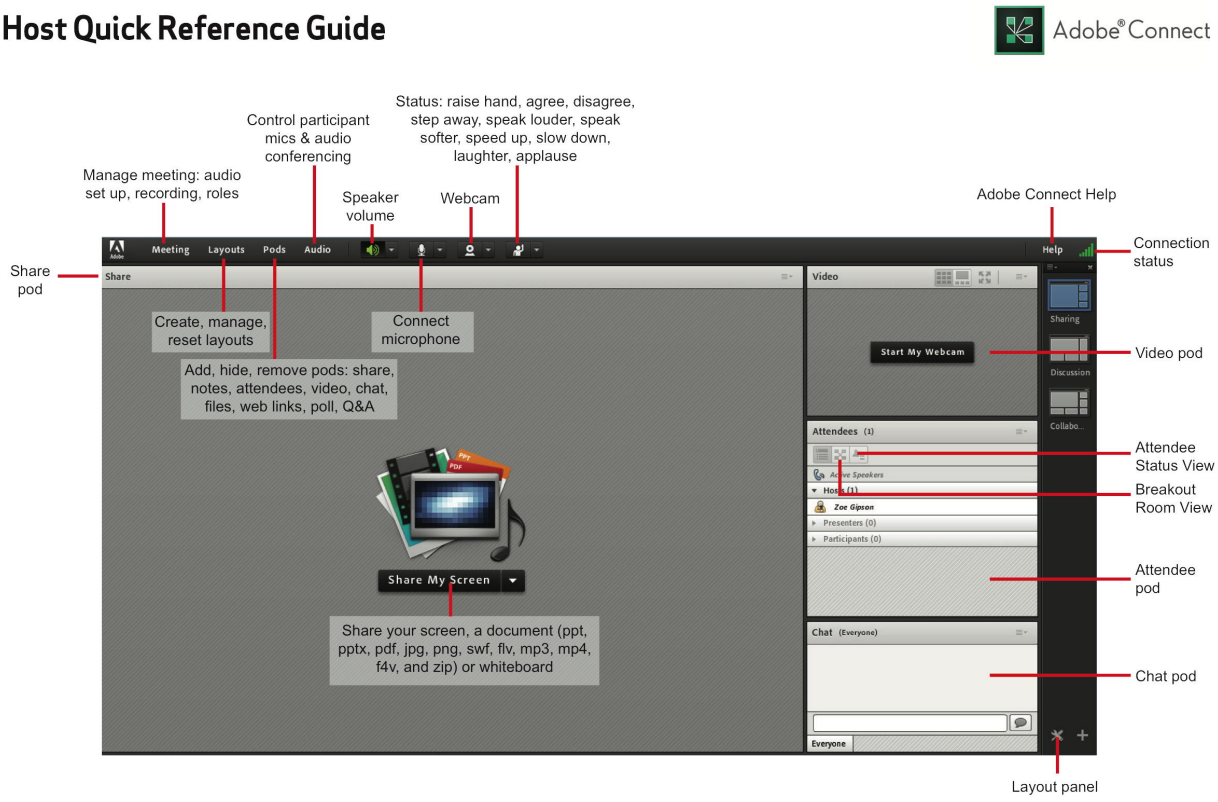
- Keyboard shortcuts in Adobe Connect
- **Toggle Mic** ⌘ + M (Mac), Ctrl + M (Win) (On/Disconnect)
- **Toggle Raise-Hand status** ⌘ + E
- **Close dialog box** ⎋ (Mac), Esc (Win)
- **End meeting** ⌘ + \

2.3 Adobe Connect Interface — Student & Tutor Views

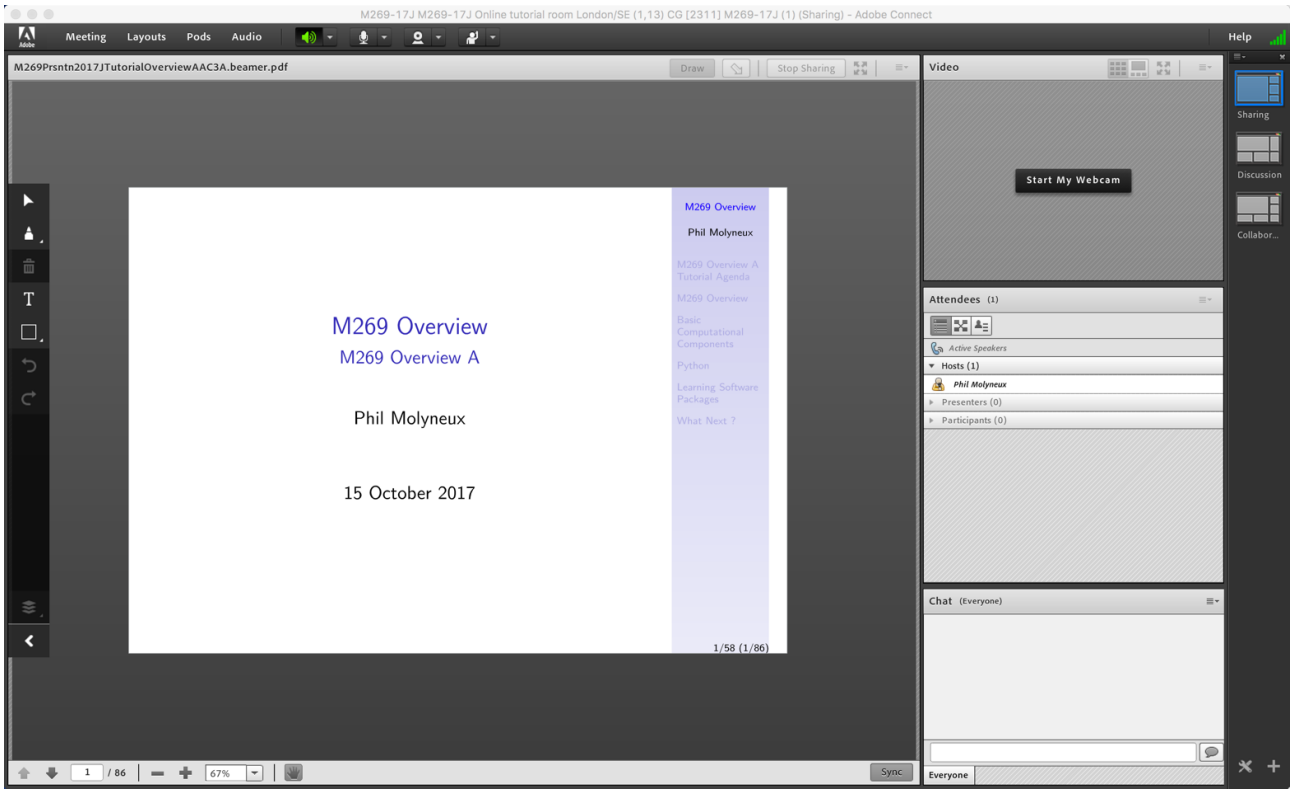
Adobe Connect Interface — Student View (default)



Adobe Connect Interface — Tutor Quick Reference



Adobe Connect Interface — Tutor View



2.4 Adobe Connect — Sharing Screen & Applications

- Share My Screen › Application tab › Terminal for Terminal
- Share menu › Change View › Zoom in for mismatch of screen size/resolution (Participants)
- Leave the application on the original display
- Beware blued hatched rectangles — from other (hidden) windows or contextual menus
- Presenter screen pointer affects viewer display — beware of moving the pointer away from the application
- First time: System Preferences › Security & Privacy › Privacy › Accessibility

2.5 Adobe Connect — Ending a Meeting

- *Notes for the tutor only*
- **Student:** Meeting › Exit Adobe Connect
- **Tutor:**
- **Recording** Meeting › Stop Recording ✔
- **Remove Participants** Meeting › End Meeting... ✔
  - Dialog box allows for message with default message:
  - *The host has ended this meeting. Thank you for attending.*
- **Recording availability** *In course Web site for joining the room, click on the eye icon in the list of recordings under your recording* — edit description and name
- **Meeting Information** Meeting › Manage Meeting Information — can access a range of information in Web page.
- **Attendance Report** see course Web site for joining room

Go to Table of Contents

3 M269 Prsntn 2017J Exam Qs

3.1 M269 2017J Exam Qs

- M269 Algorithms, Data Structures and Computability
- Presentation 2017J Exam
- Date Thursday, 7 June 2018 Time 10:00–13:00
- There are **TWO** parts to this examination. You should attempt all questions in **both** parts
- **Part 1** carries **65 marks** — **80 minutes**

- **Part 2** carries **35 marks** — **90 minutes**
- **Note** see the original exam paper for exact wording and formatting — these slides and notes may change some wording and formatting
- **Note** The 2015J exam and before had Part 1 with 60 marks (100 minutes), Part 2 with 40 marks (70 minutes)

3.2 M269 2017J Exam Q Part1

- Answer every question in this part.
- The marks for each question are given below the question number.
- Answers to questions in this Part should be written **on this paper** in the spaces provided, or in the case of multiple-choice questions you should tick the appropriate box(es).
- If you tick **more** boxes than indicated for a multiple choice question, you will receive **no** marks for your answer to that question.
- Use the provided answer books for any rough working.

4 Units 1 & 2

4.1 Unit 1 Introduction

- Unit 1 Introduction
- Computation, computable, tractable
- Introducing Python
- What are the three most important concepts in programming ?
    1. Abstraction
    2. Abstraction
    3. Abstraction
- Quote from Paul Hudak (1952–2015)

4.2 M269 2017J Exam Q 1

- Which **one** of the following statements is true? (Tick **one** box.) **(2 marks)**

A. An Abstract Data Type is the definition of a data structure in terms of the pre- and postconditions on the data structure.

B. A more complex algorithm will always take more time to execute than a less complex one.

C. Abstraction as modelling involves two layers — the interface and the implementation.

D. A problem is computable if it is possible to build an algorithm which solves any instance of the problem in a finite number of steps.

Go to Soln 1

4.3 M269 2017J Exam Soln 1

A. An Abstract Data Type is the definition of a data structure in terms of the pre- and postconditions on the data structure. **No** *ADT defined by operations that may be performed on it and the pre- and postconditions on the operations*

B. A more complex algorithm will always take more time to execute than a less complex one. **No** *The less complex one could have a bigger problem*

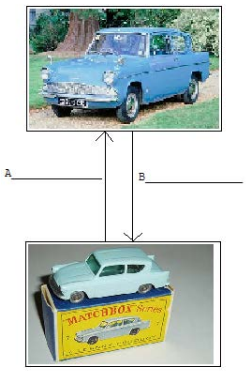
C. Abstraction as modelling involves two layers — the interface and the implementation. **No** *Models represent reality in sufficient detail*

D. A problem is computable if it is possible to build an algorithm which solves any instance of the problem in a finite number of steps. **Yes**

Go to Q 1

4.4 M269 2017J Exam Q 2

- The general idea of abstraction as modelling can be shown with the following diagram.
- The picture in the top is of a Ford Anglia in the real world, and the picture in the bottom is of a Matchbox model of a Ford Anglia.



- Complete the diagram by adding an appropriate label in the space indicated by **A** and one in the space indicated by **B**. **(2 marks)**

Go to Soln 2

4.5 M269 2017J Exam Soln 2

- **A** (Model) ignores detail of

- **B** (Actual car) represented by

Go to Q 2

4.6 Unit 2 From Problems to Programs

- Unit 2 From Problems to Programs
- Abstract Data Types
- Pre and Post Conditions
- Logic for loops

4.6.1 Example Algorithm Design — Searching

- Given an ordered list (xs) and a value (val), return
  - Position of val in xs or
  - Some indication if val is not present
- *Simple strategy:* check each value in the list in turn
- *Better strategy:* use the ordered property of the list to reduce the range of the list to be searched each turn
  - Set a range of the list
  - If val equals the mid point of the list, return the mid point
  - Otherwise half the range to search
  - If the range becomes negative, report not present (return some distinguished value)

Binary Search Iterative

```
def binarySearchIter(xs,val):
  lo = 0
  hi = len(xs) - 1

  while lo <= hi:
    mid = (lo + hi) // 2
    guess = xs[mid]

    if val == guess:
      return mid
    elif val < guess:
      hi = mid - 1
    else:
      lo = mid + 1

  return None
```

Binary Search Recursive

```
1  def binarySearchRec(xs,val,lo=0,hi=-1):
2    if (hi == -1):
3      hi = len(xs) - 1
4
5    mid = (lo + hi) // 2
6
7    if hi < lo:
8      return None
9    else:
10     guess = xs[mid]
11     if val == guess:
12       return mid
13     elif val < guess:
14       return binarySearchRec(xs,val,lo,mid-1)
15     else:
16       return binarySearchRec(xs,val,mid+1,hi)
```

Binary Search Recursive — Solution

```
xs = [12,16,17,24,41,49,51,62,67,69,75,80,89,97,101]
binarySearchRec(xs, 67)
xs = [12,16,17,24,41,49,51,62,67,69,75,80,89,97,101]
binarySearchRec(xs,67,8,14)  by line 15
xs = [12,16,17,24,41,49,51,62,67,69,75,80,89,97,101]
binarySearchRec(xs,67,8,10)  by line 13
xs = [12,16,17,24,41,49,51,62,67,69,75,80,89,97,101]
binarySearchRec(xs,67,8,8)  by line 13
xs = [12,16,17,24,41,49,51,62,67,69,75,80,89,97,101]
Return value: 8  by line 11
```

Binary Search Iterative — Miller & Ranum

```
1  def binarySearchIterMR(alist, item):
2    first = 0
3    last = len(alist)-1
4    found = False
5
6    while first<=last and not found:
7      midpoint = (first + last)//2
8      if alist[midpoint] == item:
9        found = True
10     else:
11       if item < alist[midpoint]:
12         last = midpoint-1
13       else:
14         first = midpoint+1
15
16   return found
```

Miller and Ranum (2011, page 192)

Binary Search Recursive — Miller & Ranum

```
1  def binarySearchRecMR(alist, item):
2    if len(alist) == 0:
3      return False
4    else:
5      midpoint = len(alist)//2
6      if alist[midpoint]==item:
7        return True
8      else:
```

```
9         if item<alist[midpoint]:
10          return binarySearchRecMR(alist[:midpoint],item)
11        else:
12          return binarySearchRecMR(alist[midpoint+1:],item)
```

Miller and Ranum (2011, page 193)

4.7 M269 2017J Exam Q 3

- A binary search is being carried out on the list shown below for item 41: **(4 marks)**

  [2,16,17,25,31,39,41,52,67,69,77,83,89,91,99]

- For each pass of the algorithm, draw a box around the items in the partition to be searched during that pass, continuing for as many passes as you think are needed.
- We have done the first pass for you showing that the search starts with the whole list. Draw your boxes below for each pass needed; you may not need to use all the lines below. *(The question had 8 rows)*

  (Pass 1) [2,16,17,25,31,39,41,52,67,69,77,83,89,91,99]
  (Pass 2) [2,16,17,25,31,39,41,52,67,69,77,83,89,91,99]
  (Pass 3) [2,16,17,25,31,39,41,52,67,69,77,83,89,91,99]

Go to Soln 3

4.8 M269 2017J Exam Soln 3

- The complete binary search:

  (Pass 1) [2,16,17,25,31,39,41,52,67,69,77,83,89,91,99]
  (Pass 2) [2,16,17,25,31,39,41 ,52,67,69,77,83,89,91,99]
  (Pass 3) [2,16,17,25, 31,39,41 ,52,67,69,77,83,89,91,99]
  (Pass 4) [2,16,17,25,31,39, 41 ,52,67,69,77,83,89,91,99]

Go to Q 3

4.9 M269 2017J Exam Q 4

- A Python program contains a loop with the following guard

```
while not (x >= 2 or y <= 2) or (x < 2 and y > 2):
```

- Complete the following truth table, where:

  P represents x < 2

  Q represents y > 2

P	Q	¬P	¬Q	¬P ∨ ¬Q	¬(¬P ∨ ¬Q)	P ∧ Q	¬(¬P ∨ ¬Q) ∨ (P ∧ Q)
F	F						
F	T						
T	F						
T	T						

- Use the results from your truth table to choose which one of the following expressions could be used as the simplest equivalent to the above guard. (Tick **one** box.) **(5 marks)**

A. **not** (x < 2 **and** y > 2)

B. (x >= 2 **or** y <= 2)

C. (x < 2 **and** y > 2)

D. (x >= 2 **and** y <= 2)

E. (x < 2 **and** y <= 2)

Go to Soln 4

4.10 M269 2017J Exam Soln 4

P	Q	¬P	¬Q	¬P ∨ ¬Q	¬(¬P ∨ ¬Q)	P ∧ Q	¬(¬P ∨ ¬Q) ∨ (P ∧ Q)
F	F	T	T	T	F	F	F
F	T	T	F	T	F	F	F
T	F	F	T	T	F	F	F
T	T	F	F	F	T	T	T

- The equivalent expression is C.

A. **not** (x < 2 **and** y > 2)

→ **not** P **and not** Q

B. (x >= 2 **or** y <= 2)

→ **not** P **or not** Q

C. (x < 2 **and** y > 2) → P **and** Q

D. (x >= 2 **and** y <= 2)

→ **not** P **and not** Q

E. (x < 2 **and** y <= 2)

→ P **and not** Q

- **not** (**not** P **or not** Q) **or** (P **and** Q)

→ (P **and** Q) **or** (P **and** Q)

→ (P **and** Q)

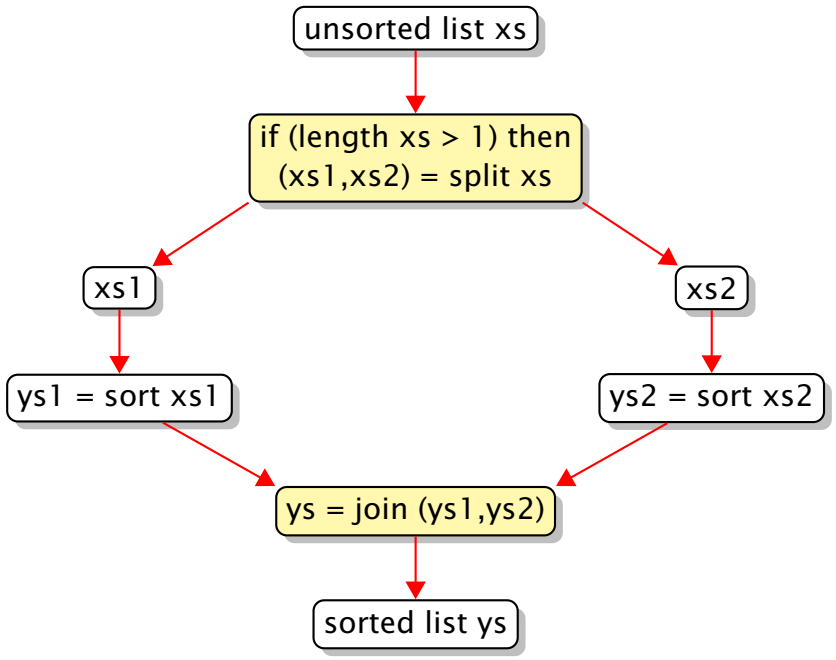
Go to Q 4

5 Units 3, 4 & 5

5.1 Unit 3 Sorting

- Unit 3 Sorting
- Elementary methods: Bubble sort, Selection sort, Insertion sort
- Recursion — base case(s) and recursive case(s) on smaller data
- Quicksort, Merge sort
- Sorting with data structures: Tree sort, Heap sort
- See sorting notes for abstract sorting algorithm

Abstract Sorting Algorithm



Sorting Algorithms

Using the *Abstract sorting algorithm*, describe the *split* and *join* for:

- Insertion sort
- Selection sort
- Merge sort
- Quicksort
- Bubble sort (the odd one out)

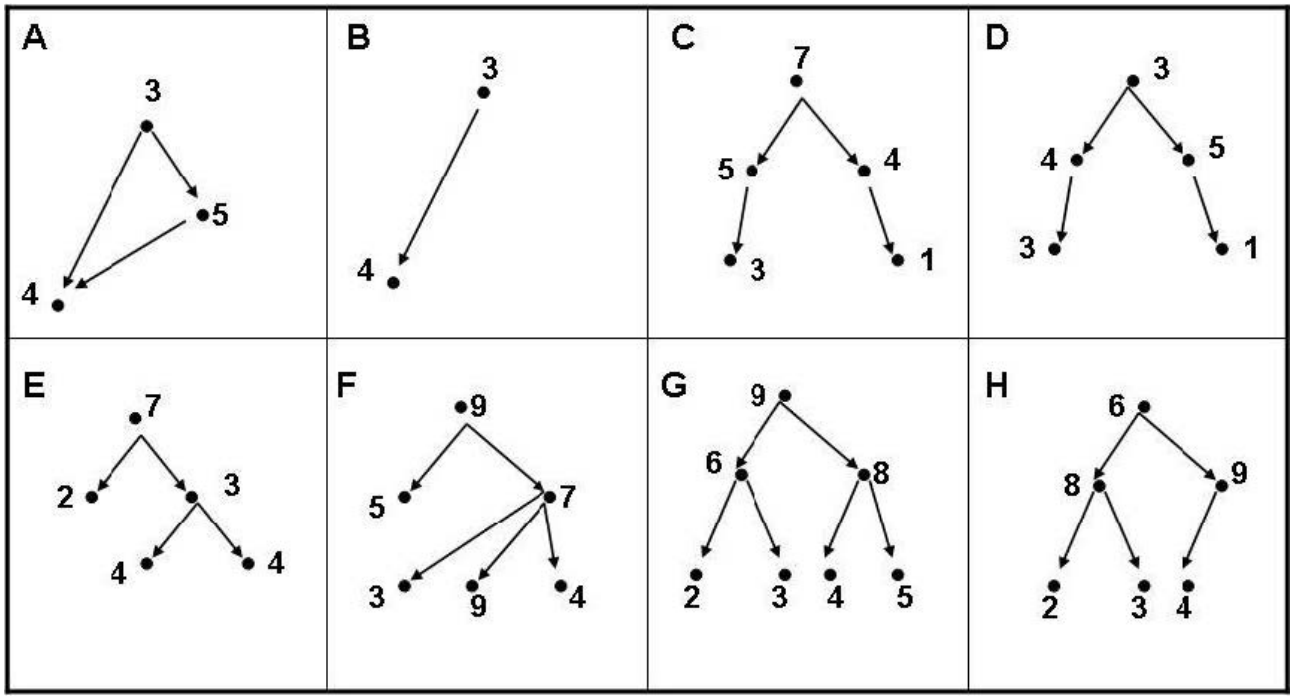
5.2 Unit 4 Searching

- Unit 4 Searching

- String searching: Quick search Sunday algorithm, Knuth-Morris-Pratt algorithm
- Hashing and hash tables
- Search trees: Binary Search Trees
- Search trees: Height balanced trees: AVL trees

5.3 M269 2017J Exam Q 5

- Consider the diagrams in A–H, where nodes are represented by black dots and edges by arrows. The numbers are the keys for the corresponding nodes.



- On each line, write one or more letters, or write *None*. **(4 marks)**

(a) Which of **A**, **B**, **C** and **D**, if any, are **not** a tree?

(b) Which of **E**, **F**, **G** and **H**, if any, are binary trees?

(c) Which of **C**, **D**, **G** and **H**, if any, are complete binary trees?

(d) Which of **C**, **D**, **G** and **H**, if any, are not a heap?

Go to Soln 5

5.4 M269 2017J Exam Soln 5

(a) Which of **A**, **B**, **C** and **D**, if any, are **not** a tree?

A is not a tree since 4 has two parents

(b) Which of **E**, **F**, **G** and **H**, if any, are binary trees?

E, **G** and **H** — **F** is not a binary tree since 7 has three sub-trees — note **E** has duplicate nodes

(c) Which of **C**, **D**, **G** and **H**, if any, are complete binary trees?

G and **H** — **E** is not a complete binary tree since the last level is not filled from left to right

(d) Which of **C**, **D**, **G** and **H**, if any, are not a heap?

C (since not a complete binary tree), **D** (since misses both properties), **H** (since does not have ordering property)

Go to Q 5

5.5 M269 2017J Exam Q 6

- Consider the following function, which takes a non-empty list as an argument.

```
def variance(aList):
    n = len(aList)
    total = 0
    for item in aList:
        total = total + item
    mean = total / n
    ssdev = 0
    for item in aList:
        deviation = item - mean
        ssdev = ssdev + (deviation * deviation)
    var = ssdev / n
    return var
```

- From the options below, select the two that represent the correct combination of T(n) and Big-O complexity for this function.

  You may assume that a step (i.e. the basic unit of computation) is the assignment statement. **(6 marks)**

  (Tick **one** box for T(n) and **one** box for Big-O complexity.)

A. $T(n) = 2n + 5$	i. $O(n)$
B. $T(n) = 3n + 5$	ii. $O(2n)$
C. $T(n) = 3n + 6$	iii. $O(3n)$
D. $T(n) = n^2 + 5$	iv. $O(n^2)$
E. $T(n) = 3n^2 + 6$	v. $O(3n^2)$

- Explain how you arrived at T(n) and the associated Big-O

Go to Soln 6

5.6 M269 2017J Exam Soln 6

- Options **B** and **i**
- There are two loops (not nested) with 3 assignments which contribute 3n to T(n)
- The remainder of the code has 5 assignments
- Hence T(n) = 3n + 5
- and complexity is O(n) from the leading term

Go to Q 6

5.7 M269 2017J Exam Q 7

(a) Which **one** of the following statements are true? (Tick **one** box.)

A. Hash tables store unique (i.e. non-duplicate) keys in an arbitrary order and are therefore an implementation of the Set ADT.

B. A hash function maps a value to a key in the table.

C. The higher the load factor on a hash table, the higher the risk of collisions.

D. Linear Probing is a chaining technique designed to resolve collisions.

Go to Soln 7

(b) Calculate the load factor for the hash table below. Show your working. **(4 marks)**

	Alice			Nisha	Bob				Ali		
0	1	2	3	4	5	6	7	8	9	10	11

Go to Soln 7

5.8 M269 2017J Exam Soln 7

A. Hash tables store unique (i.e. non-duplicate) keys in an arbitrary order and are therefore an implementation of the Set ADT. **No** — the order is not arbitrary, it is a result of the hash function and any collision resolution

B. A hash function maps a value to a key in the table. **No** — a hash function maps values to integer indices of a table, but that position may be occupied.

C. The higher the load factor on a hash table, the higher the risk of collisions. **Yes** — a high load factor means a high proportion of the hash table is occupied

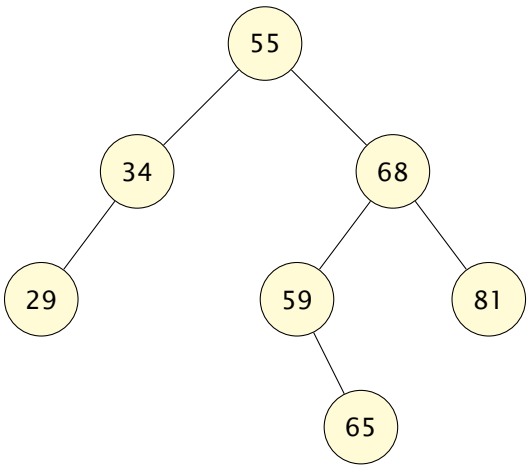
D. Linear Probing is a chaining technique designed to resolve collisions. **No** — Linear probing and chaining are different techniques

(b) The load factor is 4/12 or 0.3333

Go to Q 7

5.9 M269 2017J Exam Q 8

- In the following binary search tree, label each node with its balance factor.

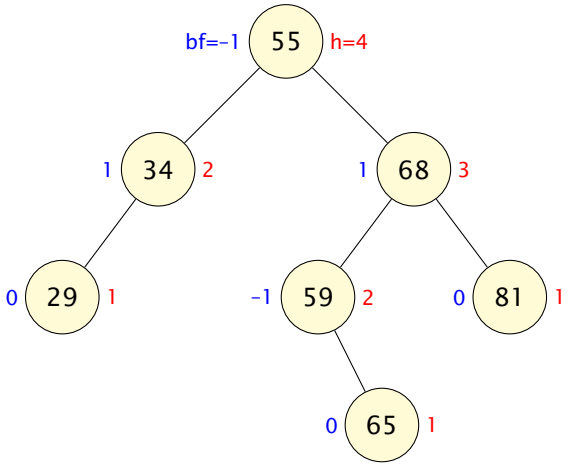


- Would this tree need to be rebalanced to be a valid AVL tree? Explain your answer. **(4 marks)**

Go to Soln 8

5.10 M269 2017J Exam Soln 8

- Binary tree with balance factors and heights — note: here empty trees have height 0 (not -1)



- The tree would not need rebalancing to be an AVL tree — the tree is a binary search tree and every node has balance factor in the range $\{-1, 0, +1\}$

Go to Q 8

5.11 Unit 5 Optimisation

- Unit 5 Optimisation
- Graphs searching: DFS, BFS
- Distance: Dijkstra's algorithm
- Greedy algorithms: Minimum spanning trees, Prim's algorithm
- Dynamic programming: Knapsack problem, Edit distance

- See Graphs Tutorial Notes

5.12 M269 2017J Exam Q 9

- A water distribution network can be represented as a weighted directed graph.
- The nodes represent the reservoirs, water treatment centres and consumers (homes, factories, etc.).
- The directed edges represent the water pipes, showing the flow of water, from the reservoirs to the consumers, via the treatment centres.
- The edge weights indicate the maximum flow (in cubic metres per second) of the pipes.
- Complete the following statements, and include in the justification any assumptions you make. **(4 marks)**
- For a typical water distribution network, the graph is (*choose from CYCLIC/ACYCLIC*) because:
- and it is (*choose from SPARSE/DENSE*) because:

Go to Soln 9

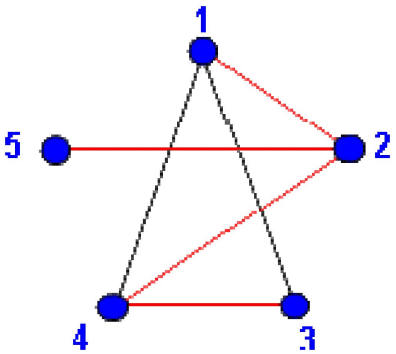
5.13 M269 2017J Exam Soln 9

- The network is *acyclic* since water does not return to the sources (in this network) — no mention is made of waste water and sewerage collection and recycling.
- A *sparse* network since most nodes are only connected to one other node.

Go to Q 9

5.14 M269 2017J Exam Q 10

- Consider the following undirected graph: **(4 marks)**



- Complete the table below to show **one** order in which the vertices of the above graph could be visited in a **Breadth** First Search (BFS) starting at vertex 3:

Vertices visited	3				

Go to Soln 10

5.15 M269 2017J Exam Soln 10

- Possible answers:

Vertices visited	3	1	4	2	5

Vertices visited	3	4	1	2	5

Go to Q 10

6 Units 6 & 7

6.1 Propositional Logic

M269 Specimen Exam Q11 Topics

- Unit 6
- Sets
- Propositional Logic
- Truth tables
- Valid arguments
- Infinite sets

6.2 M269 2017J Exam Q 11

- In propositional logic, what does it mean to say that a well-formed formula is *contingent*?
- Is the well-formed formula $(P \rightarrow Q) \rightarrow (\neg Q \rightarrow \neg P)$ contingent? Explain. **(4 marks)**

Go to Soln 11

6.3 M269 2017J Exam Soln 11

- A WFF is *contingent* if it is true in some interpretations and false in others — a *tautology* is true in every interpretation, a *contradiction* is false in every interpretation.
- $(P \rightarrow Q) \rightarrow (\neg Q \rightarrow \neg P)$ is a *tautology*

  $\equiv \neg(\neg P \vee Q) \vee (\neg\neg Q \vee \neg P)$ by rewriting $\rightarrow$

  $\equiv \neg(\neg P \vee Q) \vee (\neg P \vee Q)$ by negation and commutativity

  $\equiv$ True by negation

P	Q	(P → Q)	(¬Q → ¬P)	(P → Q) → (¬Q → ¬P)
T	T	T	T	T
T	F	F	F	T
F	T	T	T	T
F	F	T	T	T

Go to Q 11

6.4 Predicate Logic

- Unit 6
- Predicate Logic
- Translation to/from English
- Interpretations

6.5 M269 2017J Exam Q 12

- Consider the following particular interpretation $\mathcal{I}$ for predicate logic allowing facts to be expressed about people and the computer games they own and play. **(6 marks)**
- The domain of individuals is $\mathcal{D}$ = {Jane, John, Saira, Gran Turismo, Kessen, Pacman, The Sims, Pop Idol}.
- The constants *jane*, *john*, *saira*, *gran_turismo*, *kessen*, *pacman*, *the_sims* and *pop_idol* are assigned to the corresponding individuals.
- Two predicate symbols are assigned binary relations as follows:
- $\mathcal{I}$(*owns*) = {(Jane, Gran Turismo), (Jane, Kessen), (John, Pacman), (John, The Sims), (John, Pop Idol), (Saira, Pop Idol), (Saira, Kessen)}
- $\mathcal{I}$(*has_played*) = {(Jane, Gran Turismo), (Jane, Pop Idol), (Jane, Kessen), (John, The Sims), (John, Pop Idol), (Saira, Gran Turismo), (Saira, The Sims)}

(a) Consider the sentence in English: *Jane owns all the games she has played*.

Which **one** of these well-formed formulae is a translation of the sentence into predicate logic?

A. $\forall X.(owns(jane, X) \rightarrow has\_played(jane, X))$

B. $\forall X.(has\_played(jane, X) \rightarrow owns(jane, X))$

C. $\forall X.(has\_played(jane, X) \wedge owns(jane, X))$

(b) Give an appropriate translation of the well-formed formula below into English

$\exists X.(\neg owns(saira, X) \wedge has\_played(jane, X))$

- This formula is (*choose from TRUE/FALSE*), under the interpretation given on the previous page.
- Explain why in the box below.

You need to consider any relevant values for the variables, and show, using the domain and interpretation on the previous page, whether they make the formula TRUE or FALSE.

In your explanation, make sure that you use formal notation.

For example, instead of stating *John doesn't own Kessen* you need to write (John, Kessen) $\notin \mathcal{I}(owns)$

Go to Soln 12

6.6 M269 2017J Exam Soln 12

(a) *Jane owns all the games she has played* means

If Jane has played X *then Jane owns* X

so the answer is

B. $\forall X.(has\_played(jane, X) \rightarrow owns(jane, X))$

- A. $\forall X.(owns(jane, X) \rightarrow has\_played(jane, X))$ means

  Jane has played all the games she owns

- B. $\forall X.(has\_played(jane, X) \wedge owns(jane, X))$ means

  Jane owns all games and has played all of them

(b) $\exists X.(\neg owns(saira, X) \wedge has\_played(jane, X))$ means

There is at least one game that Saira does not own that Jane has played

- **True**

  because Jane has played Gran Turismo but Saira does not own it

- (Saira, Gran Turismo) $\notin \mathcal{I}(owns)$
  $\wedge$ (Jane, Gran Turismo) $\in \mathcal{I}(has\_played)$

Go to Q 12

6.7 SQL Queries

M269 Specimen Exam Q13 Topics

- Unit 6
- SQL queries

6.8 M269 2017J Exam Q 13

- A database contains the following tables: **(6 marks)**

oilfield

name	production
Warga	3
Lolli	5
Tolstoi	0.5
Dakhun	2
Sugar	3

operator

company	field
Amarco	Warga
Bratape	Lolli
Rosbif	Tolstoi
Taqar	Dakhun
Bratape	Sugar

(a) For the following SQL query, give the table returned by the query.

```
SELECT name, company
FROM oilfield CROSS JOIN operator
WHERE name = field ;
```

- Write the question that the above query is answering.

(b) Write an SQL query that answers the question

What is the name and the operating company of each oil field operated by Bratape?

Your query should return the following table.

company	field
Bratape	Lolli
Bratape	Sugar

Go to Soln 13

6.9 M269 2017J Exam Soln 13

```
SELECT name, company
FROM oilfield CROSS JOIN operator
WHERE name = field ;
```

- Table returned by the query

Warga	Amarco
Lolli	Bratape
Tolstoi	Rosbif
Dakhun	Taqar
Sugar	Bratape

- SQL for *What is the name and the operating company of each oil field operated by Bratape?*

```
SELECT company, field
FROM operator
WHERE company = 'Bratape'
```

Go to Q 13

6.10 Logic

M269 Exam — Q14 topics

- Unit 7

- Proofs
- Natural deduction

Logicians, Logics, Notations

- A plethora of logics, proof systems, and different notations can be puzzling.
- Martin Davis, Logician When I was a student, even the topologists regarded mathematical logicians as living in outer space. Today the connections between logic and computers are a matter of engineering practice at every level of computer organization
- Various logics, proof systems , were developed well before programming languages and with different motivations,

References: Davis (1995, page 289)

Logic and Programming Languages

- Turing machines, Von Neumann architecture and procedural languages Fortran, C, Java, Perl, Python, JavaScript
- Resolution theorem proving and logic programming — Prolog
- Logic and database query languages — SQL (Structured Query Language) and QBE (Query-By-Example) are syntactic sugar for first order logic
- Lambda calculus and functional programming with Miranda, Haskell, ML, Scala

Reference: Halpern et al. (2001)

Validity and Justification

- There are two ways to model what counts as a logically good argument:
  - the **semantic** view
  - the **syntactic** view
- The notion of a valid argument in propositional logic is rooted in the semantic view.
- It is based on the semantic idea of interpretations: assignments of truth values to the propositional variables in the sentences under discussion.
- A *valid argument* is defined as one that preserves truth from the premises to the conclusions
- The syntactic view focuses on the syntactic form of arguments.
- Arguments which are correct according to this view are called *justified arguments*.

Proof Systems, Soundness, Completeness

- Semantic validity and syntactic justification are different ways of modelling the same intuitive property: whether an argument is logically good.

- A proof system is *sound* if any statement we can prove (justify) is also valid (true)
- A proof system is *adequate* if any valid (true) statement has a proof (justification)
- A proof system that is sound and adequate is said to be *complete*
- Propositional and predicate logic are *complete* — arguments that are valid are also justifiable and vice versa
- Unit 7 section 2.4 describes another logic where there are valid arguments that are not justifiable (provable)

Reference: Chiswell and Hodges (2007, page 86)

Valid arguments

- Unit 6 defines valid arguments with the notation $\dfrac{\begin{array}{c}P_1\\ \vdots \\ P_n\end{array}}{C}$
- The argument is *valid* if and only if the value of C is *True* in each interpretation for which the value of each premise P_i is *True* for $1 \leq i \leq n$
- In some texts you see the notation $\{P_1, \ldots, P_n\} \models C$
- The expression denotes a *semantic sequent* or *semantic entailment*
- The $\models$ symbol is called the *double turnstile* and is often read as *entails* or *models*
- In LaTeX $\vDash$ and $\models$ are produced from `\vDash` and `\models` — see also the *turnstile* package
- In Unicode $\models$ is called *TRUE* and is U+22A8, HTML ⊨
- The argument $\{\} \models C$ is valid if and only if C is *True* in all interpretations
- That is, if and only if C is a tautology
- Beware different notations that mean the same thing
  - Alternate symbol for empty set: $\varnothing \models C$
  - Null symbol for empty set: $\models C$
  - Original M269 notation with null axiom above the line:

    $\dfrac{}{C}$

Justified Arguments and Natural Deduction

- Definition 7.1 An argument $\{P_1, P_2, \ldots, P_n\} \vdash C$ is a justified argument if and only if either the argument is an instance of an axiom or it can be derived by means of an inference rule from one or more other justified arguments.
- Axioms

$$\Gamma \cup \{\mathbf{A}\} \vdash \mathbf{A} \text{ (axiom schema)}$$

- This can be read as: any formula **A** can be derived from the assumption (premise) of {**A**} itself

- The $\vdash$ symbol is called the *turnstile* and is often read as *proves*, denoting *syntactic entailment*
- In LaTeX $\vdash$ is produced from \vdash
- In Unicode $\vdash$ is called *RIGHT TACK* and is U+22A2, HTML ⊢

See (Thompson, 1991, Chp 1)

- Section 2.3 of Unit 7 (not the Unit 6, 7 Reader) gives the inference rules for $\rightarrow$, $\wedge$, and $\vee$ — only dealing with *positive propositional logic* so not making use of negation — see List of logic systems
- Usually (Classical logic) have a functionally complete set of logical connectives — that is, every binary Boolean function can be expressed in terms the functions in the set

Inference Rules — Notation

- Inference rule notation:

$$\frac{\mathit{Argument}_1 \quad \ldots \quad \mathit{Argument}_n}{\mathit{Argument}} \ (\mathit{label})$$

Inference Rules — Conjunction

- $\frac{\Gamma \vdash \mathbf{A} \quad \Gamma \vdash \mathbf{B}}{\Gamma \vdash \mathbf{A} \wedge \mathbf{B}}$ ($\wedge$-introduction)
- $\frac{\Gamma \vdash \mathbf{A} \wedge \mathbf{B}}{\Gamma \vdash \mathbf{A}}$ ($\wedge$-elimination left)
- $\frac{\Gamma \vdash \mathbf{A} \wedge \mathbf{B}}{\Gamma \vdash \mathbf{B}}$ ($\wedge$-elimination right)

Inference Rules — Implication

- $\frac{\Gamma \cup \{\mathbf{A}\} \vdash \mathbf{B}}{\Gamma \vdash \mathbf{A} \rightarrow \mathbf{B}}$ ($\rightarrow$-introduction)
- The above should be read as: If there is a proof (justification, inference) for **B** under the set of premises, Γ, augmented with **A**, then we have a proof (justification. inference) of $\mathbf{A} \rightarrow \mathbf{B}$, *under the unaugmented set of premises,* Γ.

  The unaugmented set of premises, Γ may have contained **A** already so we cannot assume

  $$(\Gamma \cup \{\mathbf{A}\}) - \{\mathbf{A}\} \text{ is equal to } \Gamma$$

- $\frac{\Gamma \vdash \mathbf{A} \quad \Gamma \vdash \mathbf{A} \rightarrow \mathbf{B}}{\Gamma \vdash \mathbf{B}}$ ($\rightarrow$-elimination)

Inference Rules — Disjunction

- $\frac{\Gamma \vdash \mathbf{A}}{\Gamma \vdash \mathbf{A} \vee \mathbf{B}}$ ($\vee$-introduction left)
- $\frac{\Gamma \vdash \mathbf{B}}{\Gamma \vdash \mathbf{A} \vee \mathbf{B}}$ ($\vee$-introduction right)
- Disjunction elimination

$$\frac{\Gamma \vdash \mathbf{A} \vee \mathbf{B} \quad \Gamma \cup \{\mathbf{A}\} \vdash \mathbf{C} \quad \Gamma \cup \{\mathbf{B}\} \vdash \mathbf{C}}{\Gamma \vdash \mathbf{C}} \ (\vee\text{-elimination})$$

- The above should be read: if a set of premises Γ justifies the conclusion $\mathbf{A} \vee \mathbf{B}$ and Γ augmented with each of **A** or **B** separately justifies **C**, then Γ justifies **C**

Proofs in Tree Form

- The syntax of proofs is recursive:
- A proof is either an axiom, or the result of applying a rule of inference to one, two or three proofs.
- We can therefore represent a proof by a tree diagram in which each node have one, two or three children
- For example, the proof of $\{P \wedge (P \to Q)\} \vdash Q$ in *Question 4* (in the Logic tutorial notes) can be represented by the following diagram:

$$\dfrac{\dfrac{\dfrac{\{P \wedge (P \to Q)\} \vdash P \wedge (P \to Q)}{\{P \wedge (P \to Q)\} \vdash P}\ (\wedge\text{-E left}) \quad \dfrac{\{P \wedge (P \to Q)\} \vdash P \wedge (P \to Q)}{\{P \wedge (P \to Q)\} \vdash P \to Q}\ (\wedge\text{-E right})}{\{P \wedge (P \to Q)\} \vdash Q}\ (\to\text{-E})$$

Self-Assessment activity 7.4 — tree layout

- Let $\Gamma = \{P \to R, Q \to R, P \vee Q\}$
- $\dfrac{\Gamma \vdash P \vee Q \quad \Gamma \cup \{P\} \vdash R \quad \Gamma \cup \{Q\} \vdash R}{\Gamma \vdash R}\ (\vee\text{-elimination})$
- $\dfrac{\Gamma \cup \{P\} \vdash P \quad \Gamma \cup \{P\} \vdash P \to R}{\Gamma \cup \{P\} \vdash R}\ (\to\text{-elimination})$
- $\dfrac{\Gamma \cup \{Q\} \vdash Q \quad \Gamma \cup \{Q\} \vdash Q \to R}{\Gamma \cup \{Q\} \vdash R}\ (\to\text{-elimination})$
- Complete tree layout
- $$\dfrac{\Gamma \vdash P \vee Q \quad \dfrac{\dfrac{\Gamma \cup \{P\} \vdash P \quad \Gamma \cup \{P\} \vdash P \to R}{\Gamma \cup \{P\} \vdash R}\ (\to\text{-E}) \quad \dfrac{\Gamma \cup \{Q\} \vdash Q \quad \Gamma \cup \{Q\} \vdash Q \to R}{\Gamma \cup \{Q\} \vdash R}\ (\to\text{-E})}{\Gamma \vdash R}\ (\vee\text{-E})$$

Self-assessment activity 7.4 — Linear Layout

1. $\{P \to R, Q \to R, P \vee Q\} \vdash P \vee Q$ [Axiom]
2. $\{P \to R, Q \to R, P \vee Q\} \cup \{P\} \vdash P$ [Axiom]
3. $\{P \to R, Q \to R, P \vee Q\} \cup \{P\} \vdash P \to R$ [Axiom]
4. $\{P \to R, Q \to R, P \vee Q\} \cup \{Q\} \vdash Q$ [Axiom]
5. $\{P \to R, Q \to R, P \vee Q\} \cup \{Q\} \vdash Q \to R$ [Axiom]
6. $\{P \to R, Q \to R, P \vee Q\} \cup \{P\} \vdash R$ [2, 3, $\to$-E]
7. $\{P \to R, Q \to R, P \vee Q\} \cup \{Q\} \vdash R$ [4, 5, $\to$-E]
8. $\{P \to R, Q \to R, P \vee Q\} \vdash R$ [1, 6, 7, $\vee$-E]

6.11 M269 2017J Exam Q 14

- Consider the following decision problems: **(6 marks)**

1. The Equivalence Problem
2. Is a given list not empty?
3. The Halting Problem
4. Is a given binary tree balanced?

- On each line, write one or more of the above problem numbers, or write *None*.
- Which problems, if any, are decidable?
- Which problems, if any, are tractable?
- Which problems, if any, are NP-hard?

Go to Soln 14

6.12 M269 2017J Exam Soln 14

- Decidable: 2. (Empty list), 4. (Balanced binary tree)
- Tractable: 2. (Empty list), 4. (Balanced binary tree)
- NP-hard: 3. (Halting problem)

  See StackOverflow: Proof that the halting problem is NP-hard?

Go to Q 14

6.13 Computability

M269 Specimen Exam — Q15 Topics

- Unit 7
- Computability and ideas of computation
- Complexity
- P and NP
- NP-complete

Ideas of Computation

- The idea of an algorithm and what is effectively computable
- **Church-Turing thesis** Every function that would naturally be regarded as computable can be computed by a deterministic Turing Machine. (Unit 7 Section 4)
- See Phil Wadler on computability theory performed as part of the Bright Club at The Strand in Edinburgh, Tuesday 28 April 2015

Reducing one problem to another

- To reduce problem P_1 to P_2, invent a construction that converts instances of P_1 to P_2 that have the same answer. That is:
  - any string in the language P_1 is converted to some string in the language P_2
  - any string over the alphabet of P_1 that is not in the language of P_1 is converted to a string that is not in the language P_2
- With this construction we can solve P_1
  - Given an instance of P_1, that is, given a string w that may be in the language P_1, apply the construction algorithm to produce a string x
  - Test whether x is in P_2 and give the same answer for w in P_1

(Hopcroft et al., 2007, page 322)

- The direction of reduction is important
- If we can reduce P_1 to P_2 then (in some sense) P_2 is at least as hard as P_1 (since a solution to P_2 will give us a solution to P_1)
- So, if P_2 is decidable then P_1 is decidable
- To show a problem is undecidable we have to reduce from an known undecidable problem to it
- $\forall x(dp_{P_1}(x) = dp_{P_2}(reduce(x)))$
- Since, if P_1 is undecidable then P_2 is undecidable

Computability — Models of Computation

- In automata theory, a *problem* is the question of deciding whether a given string is a member of some particular language
- If Σ is an alphabet, and L is a language over Σ, that is $L \subseteq \Sigma^*$, where Σ^* is the set of all strings over the alphabet Σ then we have a more formal definition of *decision problem*
- Given a string $w \in \Sigma^*$, decide whether $w \in L$
- Example: Testing for a prime number — can be expressed as the language L_p consisting of all binary strings whose value as a binary number is a prime number (only divisible by 1 or itself)

(Hopcroft et al., 2007, section 1.5.4)

Computability — Church-Turing Thesis

- **Church-Turing thesis** Every function that would naturally be regarded as computable can be computed by a deterministic Turing Machine.
- **physical Church-Turing thesis** Any finite physical system can be simulated (to any degree of approximation) by a Universal Turing Machine.

- **strong Church-Turing thesis** Any finite physical system can be simulated (to any degree of approximation) with polynomial slowdown by a Universal Turing Machine.
- Shor's algorithm (1994) — quantum algorithm for factoring integers — an NP problem that is not known to be P — also not known to be NP-complete and we have no proof that it is not in P

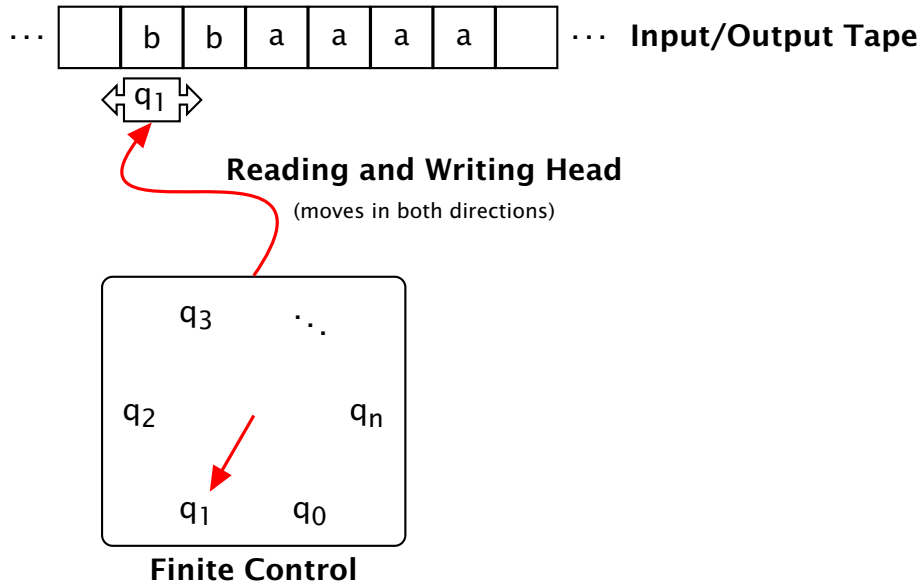
Reference: Section 4 of Unit 6 & 7 Reader

Computability — Turing Machine

- **Finite control** which can be in any of a finite number of *states*
- **Tape** divided into cells, each of which can hold one of a finite number of symbols
- Initially, the **input**, which is a finite-length string of symbols in the *input alphabet*, is placed on the tape
- All other tape cells (extending infinitely left and right) hold a special symbol called *blank*
- A **tape head** which initially is over the leftmost input symbol
- A **move** of the Turing Machine depends on the state and the tape symbol scanned
- A move can change state, write a symbol in the current cell, move left, right or stay

References: Hopcroft et al. (2007, page 326), Unit 6 & 7 Reader (section 5.3)

Turing Machine Diagram



Source: Sebastian Sardina http://www.texample.net/tikz/examples/turing-machine-2/
Date: 18 February 2012 (seen Sunday, 24 August 2014)
Further Source: Partly based on Ludger Humbert's pics of Universal Turing Machine at https://haspe.homeip.net/projekte/ddi/browser/tex/pgf2/turingmaschine-schema.tex (not found) — http://www.texample.net/tikz/examples/turing-machine/

Turing Machine notation

- Q finite set of states of the finite control
- Σ finite set of *input symbols* (M269 S)
- Γ complete set of *tape symbols* $\Sigma \subset \Gamma$
- δ Transition function (M269 instructions, I)
  $\delta :: Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R, S\}$
  $\delta(q, X) \mapsto (p, Y, D)$
- $\delta(q, X)$ takes a state, q and a tape symbol, X and returns (p, Y, D) where p is a state, Y is a tape symbol to overwrite the current cell, D is a direction, Left, Right or Stay
- q_0 *start state* $q_0 \in Q$
- B *blank symbol* $B \in \Gamma$ and $B \notin \Sigma$
- F set of *final or accepting states* $F \subseteq Q$

Computability — Decidability

- **Decidable** — there is a TM that will halt with yes/no for a decision problem — that is, given a string w over the alphabet of P the TM with halt and return yes.no the string is in the language P (same as *recursive* in Recursion theory — old use of the word)
- **Semi-decidable** — there is a TM will halt with yes if some string is in P but may loop forever on some inputs (same as *recursively enumerable*) — *Halting Problem*
- **Highly-undecidable** — no outcome for any input — *Totality, Equivalence Problems*

Undecidable Problems

- **Halting problem** — the problem of deciding, given a program and an input, whether the program will eventually halt with that input, or will run forever — term first used by Martin Davis 1952
- **Entscheidungsproblem** — the problem of deciding whether a given statement is provable from the axioms using the rules of logic — shown to be undecidable by Turing (1936) by reduction from the *Halting problem* to it
- **Type inference and type checking** in the second-order lambda calculus (important for functional programmers, Haskell, GHC implementation)
- Undecidable problem — see link to list

(Turing, 1936, 1937)

Why undecidable problems must exist

- A *problem* is really membership of a string in some language
- The number of different languages over any alphabet of more than one symbol is uncountable

- Programs are finite strings over a finite alphabet (ASCII or Unicode) and hence countable.
- There must be an infinity (big) of problems more than programs.
- **Computational problem** — defined by a function
- **Computational problem is computable** if there is a Turing machine that will calculate the function.

Reference: Hopcroft et al. (2007, page 318)

Computability and Terminology

- The idea of an *algorithm* dates back 3000 years to Euclid, Babylonians...
- In the 1930s the idea was made more formal: which *functions* are *computable*?
- A *function* a set of pairs $f = \{(x, f(x)) : x \in X \wedge f(x) \in Y\}$ with the *function property*
- *Function property:* $(a, b) \in f \wedge (a, c) \in f \Rightarrow b == c$
- *Function property:* Same input implies same output
- Note that maths notation is deeply inconsistent here — see Function and History of the function concept
- *What do we mean by computing a function — an algorithm ?*
- In the 1930s three definitions:
- λ-Calculus, simple semantics for computation — Alonzo Church
- General recursive functions — Kurt Gödel
- Universal (Turing) machine — Alan Turing
- Terminology:
  - Recursive, recursively enumerable — Church, Kleene
  - Computable, computably enumerable — Gödel, Turing
  - Decidable, semi-decidable, highly undecidable
  - In the 1930s, *computers* were *human*
  - Unfortunate choice of terminology
- Turing and Church showed that the above three were equivalent
- Church-Turing thesis — function is intuitively computable if and only if Turing machine computable

Sources on Computability Terminology

- Soare (1996) on the history of the terms *computable* and *recursive* meaning *calculable*
- See also Soare (2013, sections 9.9–9.15) in Copeland et al. (2013)

6.13.1 Non-Computability — Halting Problem

Halting Problem — Sketch Proof

- Halting problem — is there a program that can determine if any arbitrary program will halt or continue forever ?
- Assume we have such a program (Turing Machine) `h(f,x)` that takes a program `f` and input `x` and determines if it halts or not

```
h(f,x)
  = if f(x) runs forever
      return True
    else
      return False
```

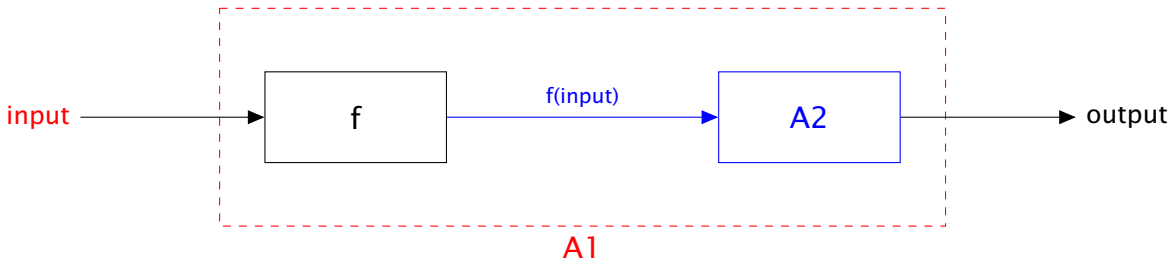
- We shall prove this cannot exist by contradiction
- Now invent two further programs:
- `q(f)` that takes a program `f` and runs `h` with the input to `f` being a copy of `f`
- `r(f)` that runs `q(f)` and halts if `q(f)` returns `True`, otherwise it loops

```
q(f)
  = h(f,f)

r(f)
  = if q(f)
      return
    else
      while True: continue
```

- What happens if we run `r(r)` ?
- If it loops, `q(r)` returns `True` and it does not loop — contradiction.

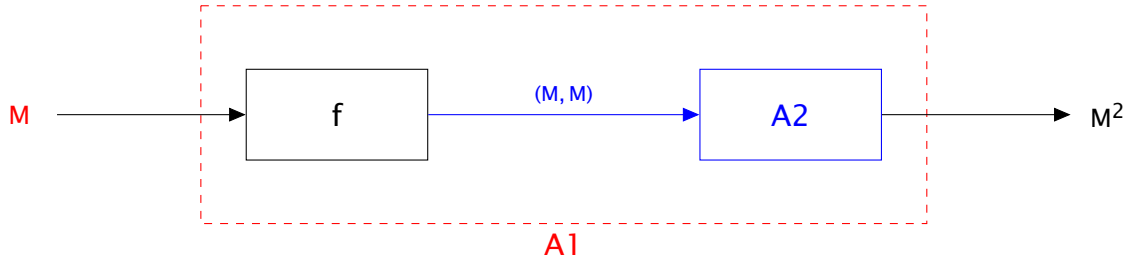
6.13.2 Reductions & Non-Computability



- A *reduction* of problem P_1 to problem P_2
  - transforms inputs to P_1 into inputs to P_2
  - runs algorithm A2 (which solves P_2) and
  - interprets the outputs from A2 as answers to P_1
- More formally: A problem P_1 is *reducible* to a problem P_2 if there is a function f that takes any input x to P_1 and transforms it to an input f(x) of P_2

  such that the solution of P_2 on f(x) is the solution of P_1 on x

Source: Bridge Theory of Computation, 2007

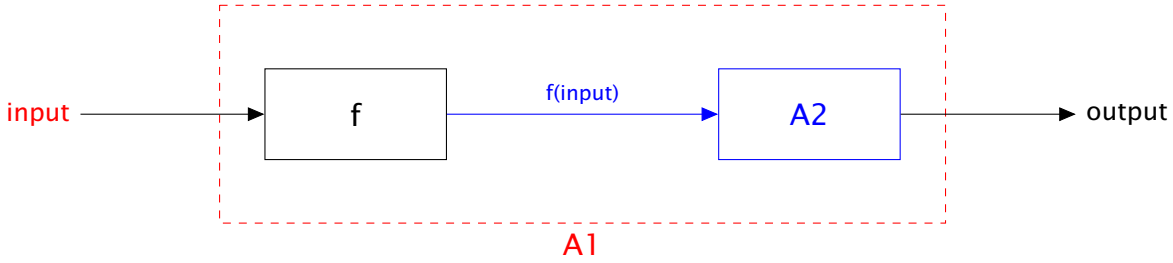


- Given an algorithm (A2) for matrix multiplication (P_2)
  - Input: pair of matrices, (M_1, M_2)
  - Output: matrix result of multiplying M_1 and M_2
- P_1 is the problem of squaring a matrix
  - Input: matrix M
  - Output: matrix M^2
- Algorithm A1 has

  f(M) = (M, M)

  uses A2 to calculate $M \times M = M^2$

Non-Computable Problems

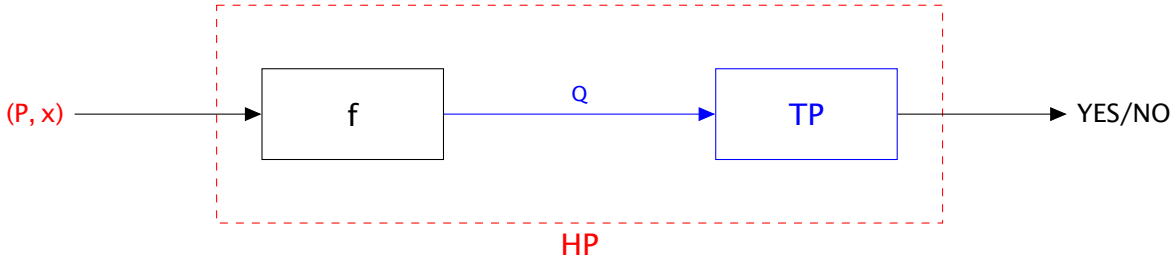


- If P_2 is computable (A2 exists) then P_1 is computable (f being simple or polynomial)
- Equivalently If P_1 is non-computable then P_2 is non-computable
- **Exercise:** show $B \rightarrow A \equiv \neg A \rightarrow \neg B$
- Proof by Contrapositive
- $B \rightarrow A \equiv \neg B \vee A$ by truth table or equivalences

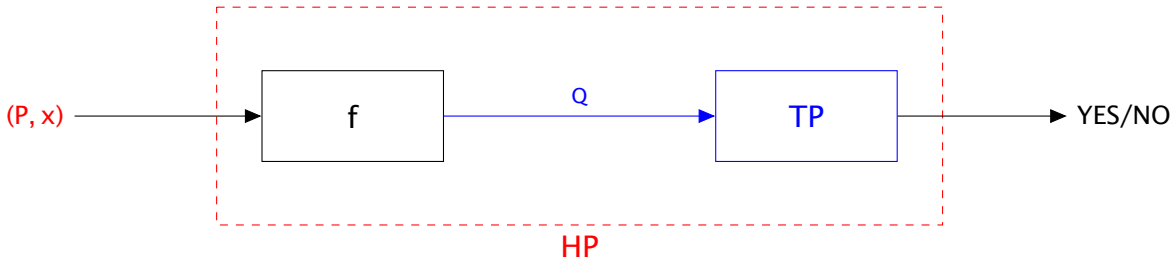
  $\equiv \neg(\neg A) \vee \neg B$ commutativity and negation laws

  $\equiv \neg A \rightarrow \neg B$ equivalences
- Common error: switching the order round

Totality Problem



- **Totality Problem**
  - Input: program Q
  - Output: YES if Q terminates for all inputs else NO
- Assume we have algorithm TP to solve the Totality Problem
- Now reduce the Halting Problem to the Totality Problem

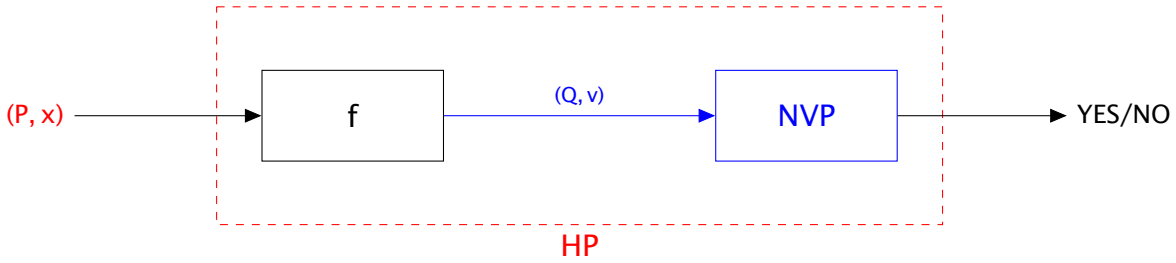


- Define f to transform inputs to HP to TP pseudo-Python

```
def f(P,x) :
  def Q(y):
    # ignore y
    P(x)
  return Q
```

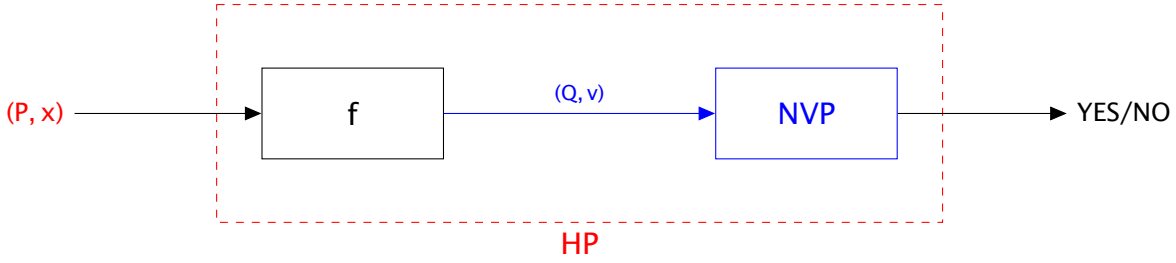
- Run TP on Q
  - If TP returns YES then P halts on x
  - If TP returns NO then P does not halt on x
- We have *solved* the Halting Problem — contradiction

Negative Value Problem



- **Negative Value Problem**
  - Input: program Q which has no input and variable v used in Q
  - Output: YES if v ever gets assigned a negative value else NO

- Assume we have algorithm NVP to solve the Negative Value Problem
- Now reduce the Halting Problem to the Negative Value Problem

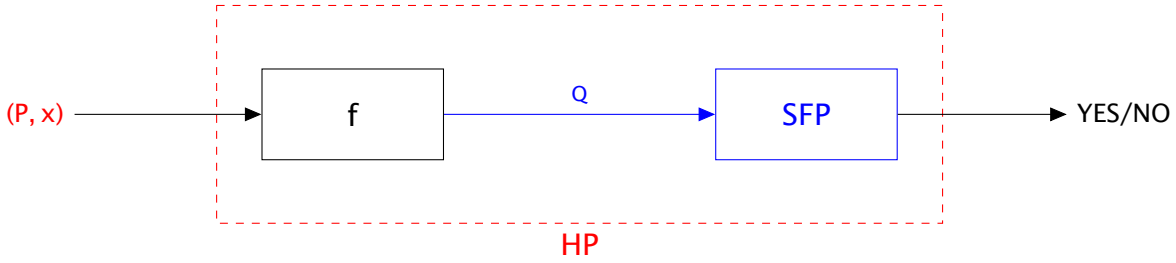


- Define f to transform inputs to HP to NVP pseudo-Python

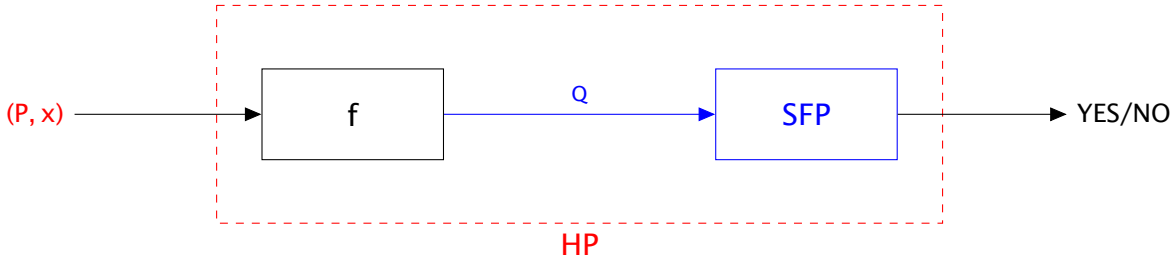
```
def f(P,x) :
  def Q(y):
    # ignore y
    P(x)
    v = -1
  return (Q,var(v))
```

- Run NVP on (Q, var(v)) `var(v)` gets the variable name
  - If NVP returns YES then P halts on x
  - If NVP returns NO then P does not halt on x
- We have *solved* the Halting Problem — contradiction

Squaring Function Problem



- **Squaring Function Problem**
  - Input: program Q which takes an integer, y
  - Output: YES if Q always returns the square of y else NO
- Assume we have algorithm SFP to solve the Squaring Function Problem
- Now reduce the Halting Problem to the Squaring Function Problem

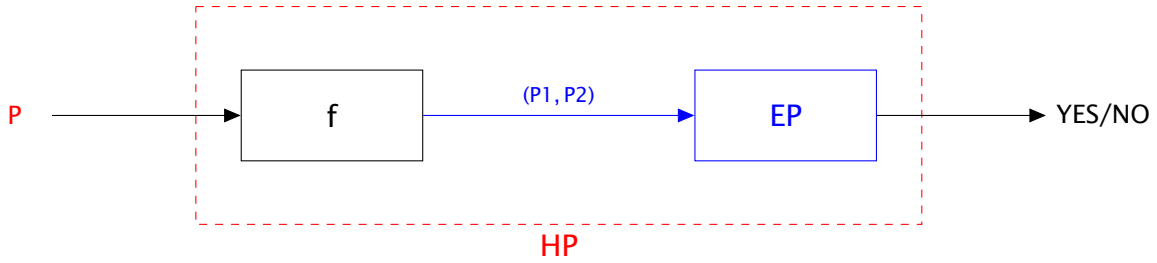


- Define f to transform inputs to HP to SFP pseudo-Python

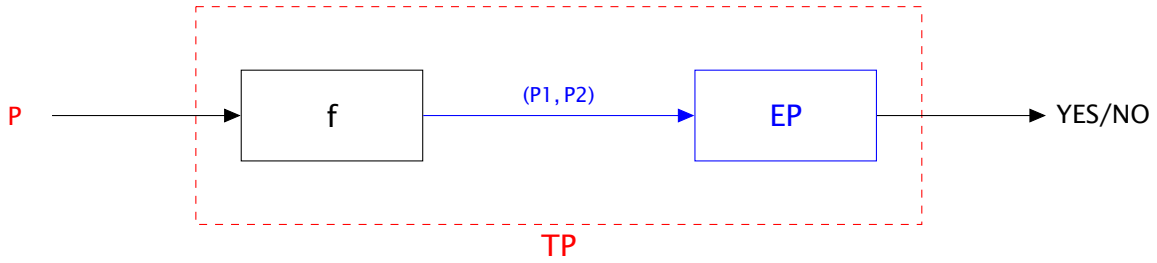
```
def f(P,x) :
  def Q(y):
    P(x)
    return y * y
  return Q
```

- Run SFP on Q
  - If SFP returns YES then P halts on x
  - If SFP returns NO then P does not halt on x
- We have *solved* the Halting Problem — contradiction

Equivalence Problem



- **Equivalence Problem**
  - Input: two programs P1 and P2
  - Output: YES if P1 and P2 solve the ame problem (same output for same input) else NO
- Assume we have algorithm EP to solve the Equivalence Problem
- Now reduce the Totality Problem to the Equivalence Problem

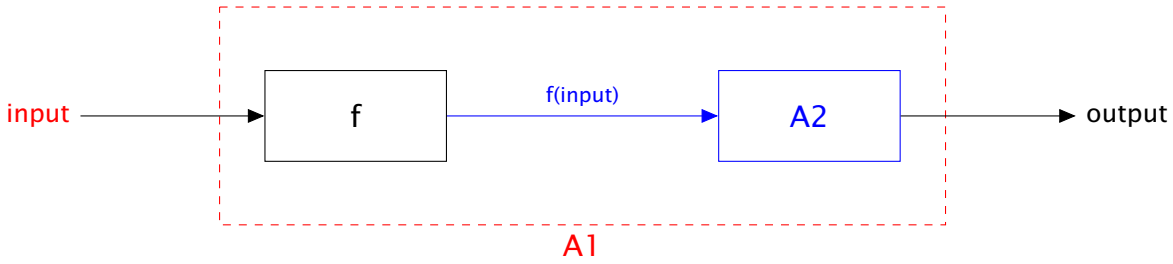


- Define f to transform inputs to TP to EP pseudo-Python

```
def f(P) :
  def P1(x):
    P(x)
    return "Same string"
  def P2(x)
    return "Same string"
  return (P1,P2)
```

- Run EP on (P1, P2)
  - If EP returns YES then P halts on all inputs
  - If EP returns NO then P does not halt on all inouts
- We have *solved* the Totality Problem — contradiction

Rice's Theorem



- Rice's Theorem all non-trivial, semantic properties of programs are undecidable. H G Rice 1951 PhD Thesis
- Equivalently: For any non-trivial property of partial functions, no general and effective method can decide whether an algorithm computes a partial function with that property.
- A property of partial functions is called trivial if it holds for all partial computable functions or for none.
- Rice's Theorem and computability theory
- Let S be a set of languages that is nontrivial, meaning
  - there exists a Turing machine that recognizes a language in S
  - there exists a Turing machine that recognizes a language not in S
- Then, it is undecidable to determine whether the language recognized by an arbitrary Turing machine lies in S.
- This has implications for compilers and virus checkers
- Note that Rice's theorem does not say anything about those properties of machines or programs that are not also properties of functions and languages.
- For example, whether a machine runs for more than 100 steps on some input is a decidable property, even though it is non-trivial.

6.14 M269 2017J Exam Q 15

- Which **two** of the following statements are true? (Tick **two** boxes.) **(4 marks)**

A. If a programming language, let's call it PL, is Turing complete, then any computational problem can be solved with a program written in PL.

B. The Equivalence Problem is not computable.

C. Problems in the class NP are defined as problems for which it is not known whether they're tractable.

D. There are non-computable computational problems because: There are more decision problems with the natural numbers as their domain (DPN) than Turing Machines that solve instances of DPN.

E. The Totality Problem is definitely in the class P.

Go to Soln 15

6.15 M269 2017J Exam Soln 15

A. **False** PL, Turing complete programming language can compute anything that is computable but there are some computational problems that are not computable

B. **True** Equivalence Problem is not computable — see Computability notes

C. **False** The class P is a subset of NP — we just do not know whether it is a proper subset or equal

D. **True** Programs are finite strings over a finite alphabet (ASCII or Unicode) hence countable — however the number of different languages over any alphabet of more than one symbol is uncountable — a problem is really membership of a string in some language

E. **False** Totality Problem is not computable — see Computability notes — so not in the class P

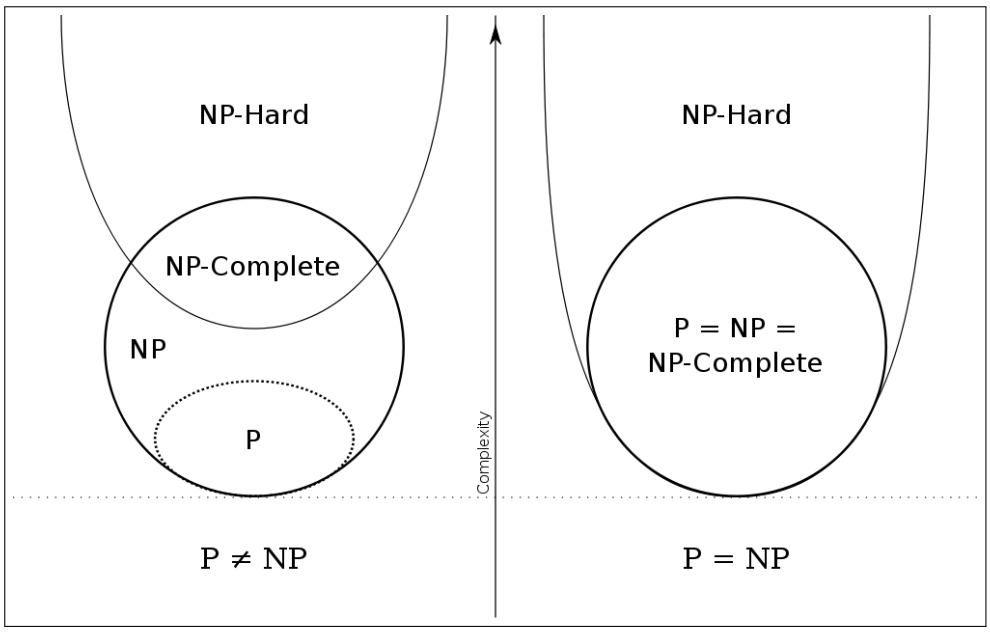
Go to Q 15

6.16 Complexity

P and NP

- P, the set of all decision problems that can be solved in polynomial time on a deterministic Turing machine
- NP, the set of all decision problems whose solutions can be verified (certificate) in polynomial time
- Equivalently, NP, the set of all decision problems that can be solved in polynomial time on a non-deterministic Turing machine
- A *decision problem*, *dp* is *NP-complete* if
  1. *dp* is in NP and
  2. Every problem in NP is reducible to *dp* in polynomial time
- *NP-hard* — a problem satisfying the second condition, whether or not it satisfies the first condition. Class of problems which are at least as hard as the hardest problems in NP. NP-hard problems do not have to be in NP and may not be decision problems

Euler diagram for P, NP, NP-complete and NP-hard set of problems

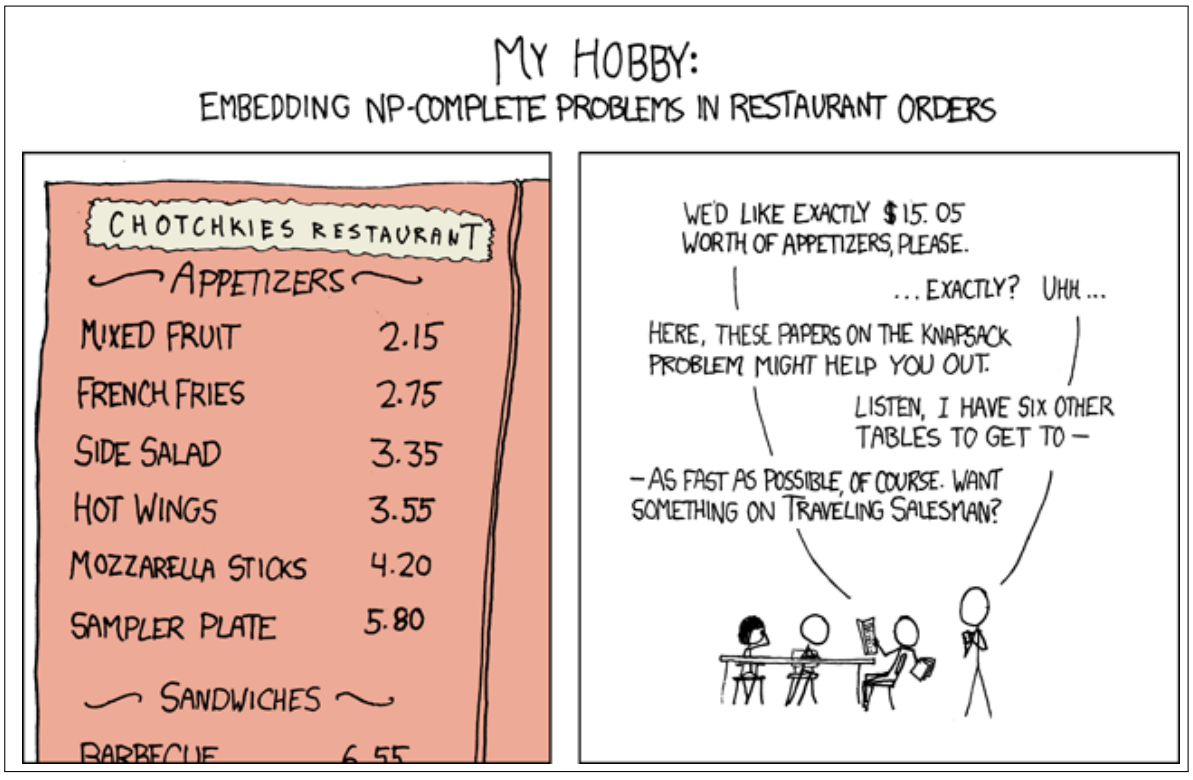


Source: Wikipedia NP-complete entry

NP-complete problems

- Boolean satisfiability (SAT) Cook-Levin theorem
- Conjunctive Normal Form 3SAT
- Hamiltonian path problem
- Travelling salesman problem
- NP-complete — see list of problems

XKCD on NP-Complete Problems



Source & Explanation: XKCD 287

6.16.1 NP-Completeness and Boolean Satisfiability

- The *Boolean satisfiability problem (SAT)* was the first decision problem shown to be *NP-Complete*
- This section gives a sketch of an explanation
- **Health Warning** different texts have different notations and there will be some inconsistency in these notes
- **Health warning** these notes use some formal notation *to make the ideas more precise* — computation requires precise notation and is about manipulating strings according to precise rules.

Alphabets, Strings and Languages

- Notation:
- Σ is a set of symbols — the alphabet
- Σ^k is the set of all string of length k, which each symbol from Σ
- Example: if $\Sigma = \{0, 1\}$
  - $\Sigma^1 = \{0, 1\}$
  - $\Sigma^2 = \{00, 01, 10, 11\}$
- $\Sigma^0 = \{\epsilon\}$ where ϵ is the empty string
- Σ^* is the set of all possible strings over Σ
- $\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \ldots$
- A *Language*, L, over Σ is a subset of Σ^*
- $L \subseteq \Sigma^*$

Language Accepted by a Turing Machine

- Language accepted by Turing Machine, M denoted by L(M)
- L(M) is the set of strings $w \in \Sigma^*$ accepted by M
- For *Final States* F = {Y, N}, a string $w \in \Sigma^*$ is accepted by M $\Leftrightarrow$ (if and only if) M starting in q_0 with w on the tape halts in state Y
- Calculating a function (function problem) can be turned into a decision problem by asking whether f(x) = y

The NP-Complete Class

- If we do not know if P $\neq$ NP, what can we say ?
- A language L is *NP-Complete* if:

  - $L \in NP$ and
  - for all other $L' \in NP$ there is a *polynomial time transformation* (Karp reducible, reduction) from L' to L
- Problem P_1 *polynomially reduces* (Karp reduces, transforms) to P_2, written $P_1 \propto P_2$ or $P_1 \leq_p P_2$, iff $\exists f : dp_{P_1} \rightarrow dp_{P_2}$ such that
  - $\forall I \in dp_{P_1} [I \in Y_{P_1} \Leftrightarrow f(I) \in Y_{P_2}]$
  - f can be computed in polynomial time
- More formally, $L_1 \subseteq \Sigma_1^*$ polynomially transforms to $L_2 \subseteq \Sigma_2^*$, written $L_1 \propto L_2$ or $L_1 \leq_p L_2$, iff $\exists f : \Sigma_1^* \rightarrow \Sigma_2^*$ such that
  - $\forall x \in \Sigma_1^* [x \in L_1 \Leftrightarrow f(x) \in L_2]$
  - There is a polynomial time TM that computes f
- *Transitivity* If $L_1 \propto L_2$ and $L_2 \propto L_3$ then $L_1 \propto L_3$
- If L is NP-Hard and $L \in P$ then P = NP
- If L is NP-Complete, then $L \in P$ if and only if P = NP
- If L_0 is NP-Complete and $L \in NP$ and $L_0 \propto L$ then L is NP-Complete
- Hence if we find one NP-Complete problem, it may become easier to find more
- In 1971/1973 Cook-Levin showed that the Boolean satisfiability problem (SAT) is NP-Complete

The Boolean Satisfiability Problem

- A propositional logic formula or Boolean expression is built from variables, operators: AND (conjunction, $\wedge$), OR (disjunction, $\vee$), NOT (negation, $\neg$)
- A formula is said to be *satisfiable* if it can be made True by some assignment to its variables.
- *The Boolean Satisfiability Problem* is, given a formula, check if it is satisfiable.
  - *Instance:* a finite set U of Boolean variables and a finite set C of clauses over U
  - *Question:* Is there a satisfying truth assignment for C ?
- A *clause* is is a disjunction of variables or negations of variables
- *Conjunctive normal form (CNF)* is a conjunction of clauses
- Any Boolean expression can be transformed to CNF
- Given a set of Boolean variable $U = \{u_1, u_2, \ldots, u_n\}$
- A literal from U is either any u_i or the negation of some u_i (written $\overline{u_i}$)
- A clause is denoted as a subset of literals from U — $\{u_2, \overline{u_4}, u_5\}$
- A clause is satisfied by an assignment to the variables if at least one of the literals evaluates to True (just like disjunction of the literals)
- Let C be a set of clauses over U — C is satisfiable iff there is some assignment of truth values to the variables so that every clause is satisfied (just like CNF)

- $C = \{\{u_1, u_2, u_3\}, \{\overline{u_2}, \overline{u_3}\}, \{u_2, \overline{u_3}\}\}$ is satisfiable
- $C = \{\{u_1, u_2\}, \{u_1, \overline{u_2}\}, \{\overline{u_1}\}\}$ is not satisfiable
- Proof that SAT is NP-Complete looks at the structure of NDTMs and shows you can transform any NDTM to SAT in polynomial time (in fact logarithmic space suffices)
- SAT is in NP since you can check a solution in polynomial time
- To show that $\forall L \in NP : L \propto SAT$ invent a polynomial time algorithm for each polynomial time NDTM, M, which takes as input a string x and produces a Boolean formula E_x which is satisfiable iff M accepts x
- See Cook-Levin theorem

Sources

- Garey and Johnson (1979, page 34) has the notation $L_1 \propto L_2$ for polynomial transformation
- Arora and Barak (2009, page 42) has the notation $L_1 \leq_p L_2$ for *polynomial-time Karp reducible*
- The sketch of Cook's theorem is from Garey and Johnson (1979, page 38)
- For the satisfiable C we could have assignments $(u_1, u_2, u_3) \in \{(T, T, F), (T, F, F), (F, T, F)\}$

Coping with NP-Completeness

- What does it mean if a problem is NP-Complete ?
  - There is a P time verification algorithm.
  - There is a P time algorithm to solve it iff P = NP (?)
  - No one has yet found a P time algorithm to solve any NP-Complete problem
  - So what do we do ?
- Improved exhaustive search — Dynamic Programming; Branch and Bound
- Heuristic methods — *acceptable* solutions in *acceptable* time — compromise on optimality
- Average time analysis — look for an algorithm with good average time — compromise on generality (see Big-O Algorithm Complexity Cheatsheet)
- Probabilistic or Randomized algorithms — compromise on correctness

Sources

- *Practical Solutions for Hard Problems* Rich (2007, chp 30)
- *Coping with NP-Complete Problems* Garey and Johnson (1979, chp 6)

7 M269 Exam 2017J Q Part2

- Answer every question in this Part.
- The marks for each question are given below the question number.
- Marks for a part of a question are given after the question.
- Answers to questions in this Part must be written in the additional answer books, which you should also use for your rough working.

Go to Soln Part2

7.1 M269 2017J Exam Q 16

- Consider an ADT for **undirected** graphs, named UGraph, that includes these operations:
- nodes, which returns a sequence of all nodes in the graph, in no particular order;
- has_edge, which takes two nodes and returns true if there is an edge between those nodes;
- edges, which returns a sequence of node-node pairs (tuples), in no particular order. Each edge only appears once in the returned sequence, i.e. if the pair (node1, node2) is in the sequence, the pair (node2, node1) is not.
- How each node is represented is irrelevant.
- You can assume the graph is connected and has no edge between a node and itself.

(a) The following stand-alone Python function checks if an undirected graph is complete, i.e. if each node is connected to every other node.

It assumes the ADT is implemented as a Python class.

```
def is_complete(graph):
  nodes = graph.nodes()
  for node1 in nodes:
    for node2 in nodes:
      edge_exists = graph.has_edge(node1, node2)
      if node1 != node2 and not edge_exists:
        return False
  return True
```

- Assume that graph.nodes has complexity O(n), where n is the number of nodes, and graph.has_edge has complexity O(1).
- State and justify a bestcase scenario and a worst-case scenario for the above function, and their corresponding Big-O complexities.
- Assume the basic computational step is the assignment.
- State explicitly any other assumptions you make. **(7 marks)**

(b) In graph theory, the number of nodes in a graph is called the order of the graph.

The term *order* is unrelated to sorting.

(i) Specify the problem of calculating the order of an undirected graph by completing the following template. Note that it is specified as an independent problem, not as a UGraph operation.

You may write the specification in English and/or formally with mathematical notation. **(4 marks)**

Name: order

Inputs

Preconditions:

Outputs:

Postconditions:

(ii) Give your initial insight for an algorithm that solves the problem.

Of the ADT operations given above you may only use edges. **(4 marks)**

(c) A city council is planning the city's bus routes.

It has decided which places will have a bus stop (schools, cinemas, hospital, etc.).

Each bus route will start from the train station, visit a number of bus stops, and then return through the same streets to the station, visiting the same bus stops in reverse order. Each bus stop has to be served by at least one bus route. The council wants to minimize the total amount of time that all buses are on the road when following their routes.

- State and justify which data structure(s) and algorithm(s) you would adopt or adapt to solve this problem.

State explicitly any assumptions you make. **(5 marks)**

Go to Soln 16

7.2 M269 2017J Exam Q 17

- Imagine you are working for a logistics company that currently uses heuristic algorithms to send their trucks on round trips that use as little fuel as possible.
- The morning paper reports that P=NP has been proved through the discovery of a tractable algorithm for the SAT problem.
- What does this news mean for the company?
- Write a brief memo with your advice on this matter to the board of the company, which doesn't include any computing experts.

The memo must have the following structure:

1. A suitable title.
2. A paragraph *setting the scene* and introducing the key question.
3. A paragraph in which you describe in layperson's terms what P=NP means.

4. A paragraph describing briefly how P=NP may impact on the company's main business objective (the cost-effective use of their trucks).
5. A conclusion on what you propose the company should do in face of this news, if anything.

- Some marks will be awarded for a clear coherent text that is appropriate for its audience, so avoid unexplained technical jargon and abrupt changes of topic, and make sure your sentences fit together to tell an overall *story*. **(15 marks)**

  As a guide, you should aim to write roughly two to five sentences per paragraph.

Go to Soln 17

8 M269 Exam 2017J Soln Part2

- Part 2 solutions

Go to Q Part2

8.1 M269 2017J Exam Soln 16

(a) Best case: First node in `nodes` has no edge to the second node in `nodes` (the first being itself) — hence returns `False` with only two calls in the inner loop — so O(n)

Worst case: The graph is complete and $O(n^2)$ since both loops fully traversed

(b) **(i)** Specification of `order` function

Name: order

Inputs: undirected graph, `g`

Preconditions: `g` is connected

Outputs: Integer, `n`

Postconditions: `n` is the size of the set of nodes in `g`

- **(ii)** Use `edges` to give a sequence of edges;

  extract a list of the first and second nodes in each edge;

  remove duplicates in the list (making a set);

  the size of the result is the order of the graph (assumes connected graph)

(c) Data structures: graph with bus stops as nodes and weighted edges as distance between stops;

- Algorithm(s): Some variant on Prim's algorithm for minimum spanning tree.

Go to Q 16

8.2 M269 2017J Exam Soln 17

- Follow the given structure:
- Title: *given at the end*
- Setting the scene:
- P as the class of problems with solutions that are found in time which is a fixed polynomial of the input size $O(n^k)$
- NP as the class of problems with solutions that can be checked in polynomial time
- Give examples of both:
- Pairing problem: given a group of students and knowledge of which are compatible, place them in compatible groups of 2 — Edmonds (1965) showed there is a polynomial time algorithm for this (so we do not have to use brute force search)
- Partition into Triangles: make groups of three with each pair in the group compatible
- Find a large group of students who are compatible — Clique problem
- Sit the students round a large table so that no incompatible students are next to each other (Hamiltonian Cycle)
- The first problem is in P, the others are in NP (we can check a solution) but it is not known if they are in P
- Define NP complete problems, dp: (a) In NP; (b) Every problem in NP is reducible to dp in polynomial time
- If P=NP then every NP problem would have a polynomial time solution — possibly via reduction to the SAT problem
- However proving P=NP (a) may not actually give an algorithm in polynomial time for solving an NP complete problem (the newspaper says there is a tractable algorithm for SAT) (b) Even with a tractable algorithm for SAT, the $O(n^k)$ may be very large.
- Give example of linear programming: standard simplex algorithm is exponential (worst case) while the ellipsoid algorithm is polynomial — however in practice simplex is used (because it is good enough) (see Wikipedia: LP)
- Implications: Good: all optimisation problems become tractable including vehicle routing
- Implications: Bad: Public key cryptography becomes impossible, banking transactions become tricky to carry out securely, the same applies to secure Web transactions
- Conclusion: prepare for huge disruption — this is bigger that the Internet or the Web
- Title: P=NP — a Disruptive Discovery
- **Reading**
- StackExchange: What would be the impact of P=NP?
- Lance Fortnow: The Status of the P Versus NP Problem readable article in 2009 CACM (Fortnow, 2009)

- The International SAT Competitions Web Page
- Lance Fortnow: *The Golden Ticket: P, NP and the Search for the Impossible* (2013,2017) (Fortnow, 2017)
- Lance Fortnow, Steve Homer: *A Short History of Computational Complexity* (Fortnow and Homer, 2003)
- Computational Complexity blog from Lance Fortnow and Bill Gasarch

Go to Q 17

9 Exam Reminders

- Read the Exam arrangements booklet
- Before the exam — check the date, time and location (and how to get there)
- At the exam centre – arrive early
- Bring photo ID with signature
- Use black or blue pens (not erasable and not pencil) — see Cult Pens for choices — pencils for preparing diagrams (HB or blacker)
- Practice writing by hand
- In the exam — Read the questions — carefully — before and after answering them
- Don't get stuck on a question — move on, come back later
- But do make sure you have attempted all questions
- . . . and finally **Good Luck**

10 White Slide

11 Web Sites & References

11.1 Web Sites

- **Logic**
  - WFF, WFF'N Proof online http://www.oercommons.org/authoring/1364-basic-wff-n-proof-a-teaching-guide/view
- **Computability**
  - Computability
  - Computable function
  - Decidability (logic)
  - Turing Machines

  - Universal Turing Machine
  - Turing machine simulator
  - Lambda Calculus
  - Von Neumann Architecture
  - Turing Machine XKCD http://www.explainxkcd.com/wiki/index.php/205:_Candy_Button_Paper
  - Turing Machine XKCD http://www.explainxkcd.com/wiki/index.php/505:_A_Bunch_of_Rocks
  - Phil Wadler Bright Club on Computability http://wadler.blogspot.co.uk/2015/05/bright-club-computability.html

- **Complexity**
  - Complexity class
  - NP complexity
  - NP complete
  - Reduction (complexity)
  - P versus NP problem
  - Graph of NP-Complete Problems

Note on References — the list of references is mainly to remind me where I obtained some of the material and is not required reading.

References

Adelson-Velskii, G M and E M Landis (1962). An algorithm for the organization of information. In *Doklady Akademia Nauk SSSR*, volume 146, pages 263–266. Translated from *Soviet Mathematics — Doklady*; 3(5), 1259–1263.

Arora, Sanjeev and Boaz Barak (2009). *Computational Complexity: A Modern Approach*. Cambridge University Press. ISBN 0521424267. URL http://www.cs.princeton.edu/theory/complexity/.

Chiswell, Ian and Wilfrid Hodges (2007). *Mathematical Logic*. Oxford University Press. ISBN 0199215626.

Church, Alonzo et al. (1937). Review: AM Turing, On Computable Numbers, with an Application to the Entscheidungsproblem. *Journal of Symbolic Logic*, 2(1):42–43.

Cook, Stephen A. (1971). The Complexity of Theorem-proving Procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, pages 151–158. ACM, New York, NY, USA. doi:10.1145/800157.805047. URL http://doi.acm.org/10.1145/800157.805047.

Copeland, B. Jack; Carl J. Posy; and Oron Shagrir (2013). *Computability: Turing, Gödel, Church, and Beyond*. The MIT Press. ISBN 0262018993.

Cormen, Thomas H.; Charles E. Leiserson; Ronald L. Rivest; and Clifford Stein (2009). *Introduction to Algorithms*. MIT Press, third edition. ISBN 0262533057. URL http://mitpress.mit.edu/books/introduction-algorithms.

Davis, Martin (1995). Influences of mathematical logic on computer science. In *The Universal Turing Machine A Half-Century Survey*, pages 289–299. Springer.

Davis, Martin (2012). *The Universal Computer: The Road from Leibniz to Turing*. A K Peters/CRC Press. ISBN 1466505192.

Dowsing, R.D.; V.J Rayward-Smith; and C.D Walter (1986). *First Course in Formal Logic and Its Applications in Computer Science*. Blackwells Scientific. ISBN 0632013087.

Fortnow, Lance (2009). The Status of the P Versus NP Problem. *Communications of the ACM*, 52(9):78–86. ISSN 0001-0782. doi:10.1145/1562164.1562186. URL http://doi.acm.org/10.1145/1562164.1562186.

Fortnow, Lance (2017). *The Golden Ticket: P, NP, and the Search for the Impossible*. Princeton University Press. ISBN 0691175780. URL https://press.princeton.edu/titles/9937.html.

Fortnow, Lance and Steve Homer (2003). A Short History of Computational Complexity. *Bulletin of the European Association for Theoretical Computer Science*, 80. URL https://lance.fortnow.com/.

Franzén, Torkel (2005). *Gödel's Theorem: An Incomplete Guide to Its Use and Abuse*. A K Peters, Ltd. ISBN 1568812388.

Fulop, Sean A. (2006). *On the Logic and Learning of Language*. Trafford Publishing. ISBN 1412023815.

Garey, Michael R. and David S. Johnson (1979). *Computers and Intractability: A Guide to the Theory of NP-completeness*. W.H.Freeman Co Ltd. ISBN 0716710455.

Halbach, Volker (2010). *The Logic Manual*. OUP Oxford. ISBN 0199587841. URL http://logicmanual.philosophy.ox.ac.uk/index.html.

Halpern, Joseph Y; Robert Harper; Neil Immerman; Phokion G Kolaitis; Moshe Y Vardi; and Victor Vianu (2001). On the unusual effectiveness of logic in computer science. *Bulletin of Symbolic Logic*, pages 213–236.

Hindley, J. Roger and Jonathan P. Seldin (1986). *Introduction to Combinators and λ-Calculus*. Cambridge University Press. ISBN 0521318394. URL http://www-maths.swan.ac.uk/staff/jrh/.

Hindley, J. Roger and Jonathan P. Seldin (2008). *Lambda-Calculus and Combinators: An Introduction*. Cambridge University Press. ISBN 0521898854. URL http://www-maths.swan.ac.uk/staff/jrh/.

Hodges, Wilfred (1977). *Logic*. Penguin. ISBN 0140219854.

Hodges, Wilfred (2001). *Logic*. Penguin, second edition. ISBN 0141003146.

Hopcroft, John E.; Rajeev Motwani; and Jeffrey D. Ullman (2001). *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, second edition. ISBN 0-201-44124-1.

Hopcroft, John E.; Rajeev Motwani; and Jeffrey D. Ullman (2007). *Introduction to*

Automata Theory, Languages, and Computation. Pearson, third edition. ISBN 0321514483. URL http://infolab.stanford.edu/~ullman/ialc.html.

Hopcroft, John E. and Jeffrey D. Ullman (2001). *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, first edition. ISBN 020102988X.

Lemmon, Edward John (1965). *Beginning Logic*. Van Nostrand Reinhold. ISBN 0442306768.

Levin, Leonid A (1973). Universal sorting problems. *Problemy Peredachi Informatsii*, 9(3):265–266.

Manna, Zohar (1974). *Mathematical Theory of Computation*. McGraw-Hill. ISBN 0-07-039910-7.

Miller, Bradley W. and David L. Ranum (2011). *Problem Solving with Algorithms and Data Structures Using Python*. Franklin, Beedle Associates Inc, second edition. ISBN 1590282574. URL http://interactivepython.org/courselib/static/pythonds/index.html.

Pelletier, Francis Jeffrey and Allen P Hazen (2012). A history of natural deduction. In Gabbay, Dov M; Francis Jeffrey Pelletier; and John Woods, editors, *Logic: A History of Its Central Concepts*, volume 11 of *Handbook of the History of Logic*, pages 341–414. North Holland. ISBN 0444529373. URL http://www.ualberta.ca/~francisp/papers/PellHazenSubmittedv2.pdf.

Pelletier, Francis Jeffry (2000). A history of natural deduction and elementary logic textbooks. *Logical consequence: Rival approaches*, 1:105–138. URL http://www.sfu.ca/~jeffpell/papers/pelletierNDtexts.pdf.

Rayward-Smith, V J (1983). *A First Course in Formal Language Theory*. Blackwells Scientific. ISBN 0632011769.

Rayward-Smith, V J (1985). *A First Course in Computability*. Blackwells Scientific. ISBN 0632013079.

Rich, Elaine A. (2007). *Automata, Computability and Complexity: Theory and Applications*. Prentice Hall. ISBN 0132288060. URL http://www.cs.utexas.edu/~ear/cs341/automatabook/.

Smith, Peter (2003). *An Introduction to Formal Logic*. Cambridge University Press. ISBN 0521008042. URL http://www.logicmatters.net/ifl/.

Smith, Peter (2007). *An Introduction to Gödel's Theorems*. Cambridge University Press, first edition. ISBN 0521674530.

Smith, Peter (2013). *An Introduction to Gödel's Theorems*. Cambridge University Press, second edition. ISBN 1107606756. URL http://godelbook.net.

Smullyan, Raymond M. (1995). *First-Order Logic*. Dover Publications Inc. ISBN 0486683702.

Soare, Robert Irving (1996). Computability and Recursion. *Bulletin of Symbolic Logic*, 2:284–321. URL http://www.people.cs.uchicago.edu/~soare/History/.

Soare, Robert Irving (2013). Interactive computing and relativized computability. In *Computability: Turing, Gödel, Church, and Beyond*, chapter 9, pages 203–260. The MIT Press. URL http://www.people.cs.uchicago.edu/~soare/Turing/shagrir.pdf.

Teller, Paul (1989a). *A Modern Formal Logic Primer: Predicate and Metatheory: 2*. Prentice-Hall. ISBN 0139031960. URL http://tellerprimer.ucdavis.edu.

Teller, Paul (1989b). *A Modern Formal Logic Primer: Sentence Logic: 1*. Prentice-Hall. ISBN 0139031707. URL http://tellerprimer.ucdavis.edu.

Thompson, Simon (1991). *Type Theory and Functional Programming*. Addison Wesley. ISBN 0201416670. URL http://www.cs.kent.ac.uk/people/staff/sjt/TTFP/.

Tomassi, Paul (1999). *Logic*. Routledge. ISBN 0415166969. URL http://emilkirkegaard.dk/en/wp-content/uploads/Paul-Tomassi-Logic.pdf.

Turing, Alan Mathison (1936). On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42:230–265.

Turing, Alan Mathison (1937). On computable numbers, with an application to the Entscheidungsproblem. A Correction. *Proceedings of the London Methematical Society*, 43:544–546.

van Dalen, Dirk (1994). *Logic and Structure*. Springer-Verlag, third edition. ISBN 0387578390.

van Dalen, Dirk (2012). *Logic and Structure*. Springer-Verlag, fifth edition. ISBN 1447145577.