# M269 Revision 2018

## Exam 2016J

## 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 2016J exam (June 2017)
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 2016J

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

# 2 M269 Prsntn 2016J Exam Qs

## 2.1 M269 2016J Exam Qs

- M269 Algorithms, Data Structures and Computability
- Presentation 2016J Exam
- Date Wednesday, 7 June 2017 Time 14:30–17:30
- 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** 2015J and before had Part 1 with 60 marks (100 minutes), Part 2 with 40 marks (70 minutes)

## 2.2 M269 2016J 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.

# 3 Units 1 & 2

## 3.1 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)

## 3.2 M269 2016J Exam Q 1

- Which **two** of the following statements are true? (Tick **two** boxes.) **(2 marks)**

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

B. An effective procedure is an algorithm which, for every instance of a given problem, solves that instance in the most efficient way — minimising the use of resources such as memory.

C. A decision problem is decidable if it is computable.

D. A decision problem is any problem stated in a formal language.

Go to Soln 1

## 3.3 M269 2016J Exam Soln 1

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

B. An effective procedure is an algorithm which, for every instance of a given problem, solves that instance in the most efficient way — minimising the use of resources such as memory. **No** *An effective procedure is an algorithm that solves any instance of a decision problem in a finite number of steps (Reader, page 91)*

C. A decision problem is decidable if it is computable. **Yes**

D. A decision problem is any problem stated in a formal language. **No** *Problems where the answer is yes or no (Unit 1)*

Go to Q 1

## 3.4 M269 2016J Exam Q 2

- Complete these paragraphs correctly using words or phrases from the list below. **(2 marks)**
- Abstraction as _____ can be understood in terms of the relationship between a _____ and a _____. The latter represents the details of interest and captures the essentials, ignoring certain irrelevant details.
- Abstraction as _____ generally involves two layers — the _____ (which is a layer through which users interact with the model) and the _____ (a layer that automates the model)
- Possible words and phrases to insert:

| | | | |
|---|---|---|---|
| encapsulation | model | modelling | procedural |
| algorithm | process | automation | interface |
| part of reality | data | simulation | implementation |

Go to Soln 2

## 3.5 M269 2016J Exam Soln 2

- Abstraction as **modelling** can be understood in terms of the relationship between a **part of reality** and a **model**.

  The latter represents the details of interest and captures the essentials, ignoring certain irrelevant details.
- Abstraction as **encapsulation** generally involves two layers — the **interface** (which is a layer through which users interact with the model) and the **implementation** (a layer that automates the model).

Go to Q 2

## 3.6 Unit 2 From Problems to Programs

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

### 3.6.1 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

```
1  def binarySearchIter(xs,val):
2    lo = 0
3    hi = len(xs) - 1
4
5    while lo <= hi:
6      mid = (lo + hi) // 2
7      guess = xs[mid]
8
9      if val == guess:
10       return mid
11     elif val < guess:
12       hi = mid - 1
13     else:
14       lo = mid + 1
15
16   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)

## 3.7 M269 2016J Exam Q 3

- This question is about bubble sort and selection sort, where we are sorting numbers in **ascending** order. **(6 marks)**

(a) Selection sort improves on bubble sort by making only one exchange for every pass through the list.

In selection sort, given the starting list below, indicate which two elements are to be swapped at each stage, and complete below as necessary.

You have space to indicate up to 5 swaps and the resulting list.

If selection sort requires fewer than 5 swaps for this list, leave any remaining step(s) blank.

| 1 | 6 | 2 | 3 | 5 |
|---|---|---|---|---|

1. Swap elements _____ and _____ to give

| | | | | |
|---|---|---|---|---|
| | | | | |

2. Swap elements _____ and _____ to give

| | | | | |
|---|---|---|---|---|
| | | | | |

3. Swap elements _____ and _____ to give

| | | | | |
|---|---|---|---|---|
| | | | | |

4. Swap elements _____ and _____ to give

| | | | | |
|---|---|---|---|---|
| | | | | |

5. Swap elements _____ and _____ to give

| | | | | |
|---|---|---|---|---|
| | | | | |

(b) Although both bubble sort and selection sort make the same number of comparisons for a list of the same length, they do not make the same number of swaps.

How many swaps are made in a **worst** case, with a list of length 5, for each of bubble sort and selection sort?

Explain how you arrived at the number of swaps for each. There is no need to refer to Big-O in your answer.

Go to Soln 3

## 3.8 M269 2016J Exam Soln 3

- Selection sort: sorting ascending and selecting largest first

```
def selSortAscByMax(xs):
  for fillSlot in range(len(xs) - 1, 0, -1):
    maxIndex = 0
    for index in range(1, fillSlot + 1):
      if xs[index] > xs[maxIndex]:
        maxIndex = index

    temp = xs[fillSlot]
    xs[fillSlot] = xs[maxIndex]
    xs[maxIndex] = temp
```

- Here is an informal version

```
for fillSlot = len(xs) - 1 down to 1 do
  find the maximum of
    xs[0] .. xs[fillSlot]
  and swap with xs[fillSlot]
```

| | | | | |
|---|---|---|---|---|
| 1 | 6 | 2 | 3 | 5 |

1. Swap elements *6* and *5* to give

| | | | | |
|---|---|---|---|---|
| 1 | *5* | 2 | 3 | 6 |

2. Swap elements *5* and *3* to give

| | | | | |
|---|---|---|---|---|
| 1 | *3* | 2 | 5 | 6 |

3. Swap elements *3* and *2* to give

| 1 | *2* | 3 | 5 | 6 |
|---|---|---|---|---|

4. Swap elements *2* and *2* to give

| 1 | 2 | 3 | 5 | 6 |
|---|---|---|---|---|

- Note the last swap would not be there if there was a test for `fillSlot == maxIndex`
- Selection sort: sorting ascending and selecting smallest first

```
def selectionSort(xs):
  for fillSlot in range(0,len(xs)-1):
    minIx = fillSlot
    for ix in range(fillSlot + 1, len(xs)):
      if xs[ix] < xs[minIx]:
        minIx = ix

    # if fillSlot != minIx: # swap if different
    xs[fillSlot],xs[minIx] = xs[minIx],xs[fillSlot]
```

- Here is an informal version

```
for fillSlot = 0 to (len(xs) - 2) do
  find the minimum of
    xs[fillSlot]..xs[len(xs) - 1]
  and swap with xs[fillSlot]
```

| 1 | 6 | 2 | 3 | 5 |
|---|---|---|---|---|

1. Swap elements *1* and *1* to give

| 1 | 6 | 2 | 3 | 5 |
|---|---|---|---|---|

2. Swap elements *6* and *2* to give

| 1 | 2 | *6* | 3 | 5 |
|---|---|---|---|---|

3. Swap elements *6* and *3* to give

| 1 | 2 | 3 | *6* | 5 |
|---|---|---|---|---|

4. Swap elements *6* and *5* to give

| 1 | 2 | 3 | 5 | *6* |
|---|---|---|---|---|

- Note the swap at stage 1. would not be there if there was a test for `fillSlot == maxIx`

(b) Bubble sort does 10 swaps in a worst case since it does n – 1 swaps iterating over n items so total = 4 + 3 + 2 + 1 = 10 swaps

- Selection sort does 4 swaps in a worst case since it does (at most) one swap per pass and n – 1 passes

## 3.9 M269 2016J Exam Q 4

- A Python program contains a loop with the following guard **(4 marks)**

```
    while a <= 3 or b > 8:
```

Make the following substitutions:

P represents a > 3

Q represents b <= 8

Complete the following table

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

- Based on the table, which of the following expressions is equivalent to the above guard? (Tick **one** box.)

A. `not a < 3`

B. `not b <= 8`

C. `not (a <= 3 and b > 8)`

D. `a > 3 and b <= 8`

E. `not (a > 3 and b <= 8)`

Go to Soln 4

### 3.10 M269 2016J Exam Soln 4

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

- The equivalent expression is E.

Go to Q 4

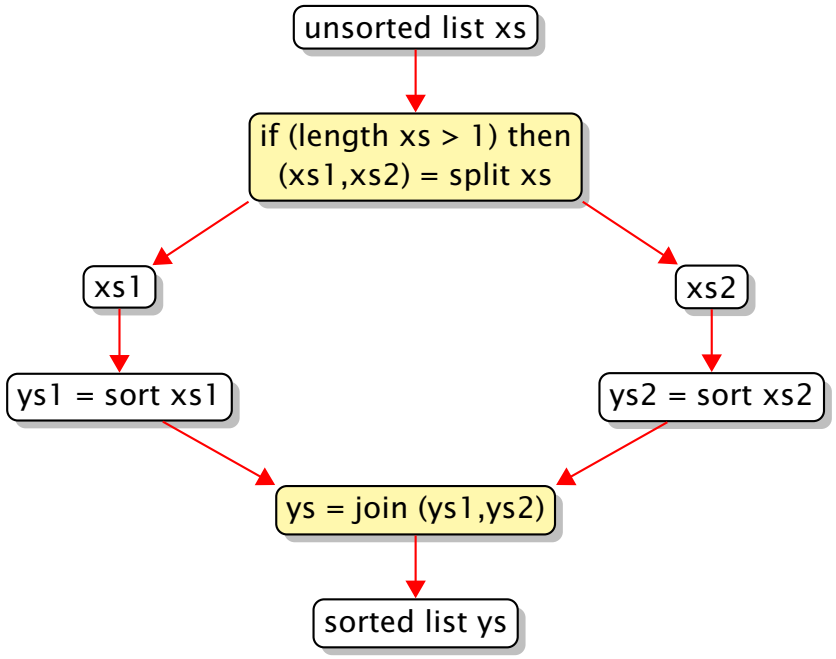
## 4 Units 3, 4 & 5

### 4.1 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:

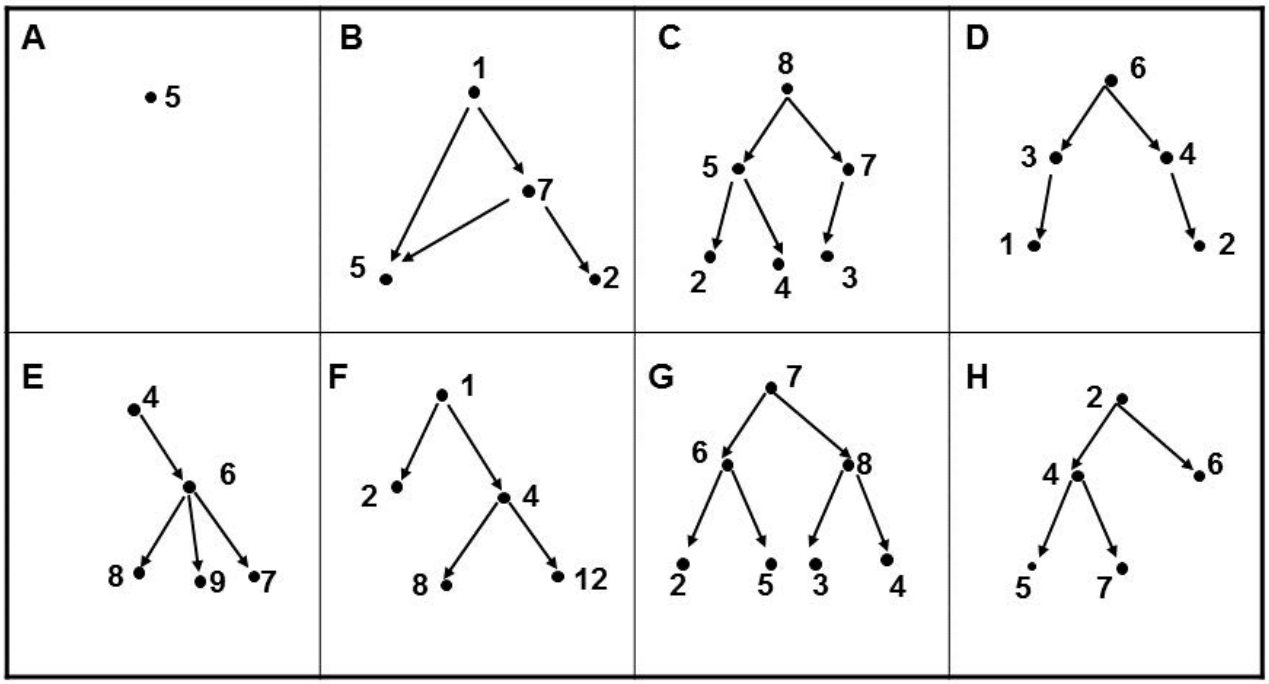
- Insertion sort
- Selection sort
- Merge sort
- Quicksort
- Bubble sort (the odd one out)

## 4.2 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

## 4.3 M269 2016J 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 the following lines, write the letter(s) of the diagram(s) that satisfies (satisfy) the condition, or write "None" if no diagram satisfies the condition. **(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 (min or max) heap?

Go to Soln 5

## 4.4 M269 2016J Exam Soln 5

(a) **B** is not a tree since node 5 has two parents — **A** is a node with two empty sub-trees

(b) **F**, **G**, **H** are binary trees — **E** is not a binary tree since node 6 has three sub-trees

(c) **C**, **G**, **H** are complete binary trees — **D** is not a complete binary tree since the last level is not filled from left to right

(d) **C** is a max heap, **H** is a min heap — **G** is not a heap since node 8 is greater than node 7

Go to Q 5

## 4.5 M269 2016J Exam Q 6

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

```
1  def someFunction(aList):
2    n = len(aList)
3    counterOne = 0
4    counterTwo = 0
5    for i in range(n):
```

```
6       counterOne = counterOne + 1
7       for j in range(n):
8         counterTwo = counterTwo + 1
9         for k in range(n):
10          counterOne = counterOne + 1
11          counterTwo = counterTwo + 1
12    return counterOne + counterTwo
```

- 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.

| | |
|---|---|
| A. $T(n) = 4n + 3$ | i. $O(1)$ |
| B. $T(n) = 2n^3 + n^2 + n + 3$ | ii. $O(n)$ |
| C. $T(n) = 2n^2 + n + 3$ | iii. $O(n^2)$ |
| D. $T(n) = n^3 + n^2 + n + 3$ | iv. $O(n^3)$ |
| E. $T(n) = 3 \log n + n^3 + n^2 + n + 3$ | v. $O(\log n)$ |

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

Go to Soln 6

## 4.6 M269 2016J Exam Soln 6

- Options **B** and **IV**
- There are three levels of nested loops with each loop executing n times.
- The innermost loop has 2 assignments giving $2n^3$ assignments
- The middle loop has one assignment giving a further $n^2$ assignments
- The outer loop has one assignment giving n assignments
- A further 3 assignments precedes all the loops
- Total $2n^3 + n^2 + n + 3$

Go to Q 6

## 4.7 M269 2016J Exam Q 7

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

A. Hash tables are an implementation of Map ADTs because they are searchable structures that contain key-value pairs, which allow searching for the key in order to find a value.

B. Chaining, where a slot in the hash table may be associated with a collection of items, is a standard way of implementing hash functions.

C. Clustering occurs when the number of unoccupied slots in a hash table exceeds the number of occupied slots.

D. The efficiency of inserting new items into a hash table decreases as the load factor becomes greater.

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

| A | Q | | S | F | | U | | | N |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

Go to Soln 7

## 4.8 M269 2016J Exam Soln 7

(a) **A** and **D** are true

- B is not true — chaining is a way of resolving collisions
- C is not true — see What is primary and secondary clustering in hash?, Primary clustering

(b) The load factor is 0.6 = $\frac{6}{10}$

Go to Q 7

## 4.9 M269 2016J Exam Q 8

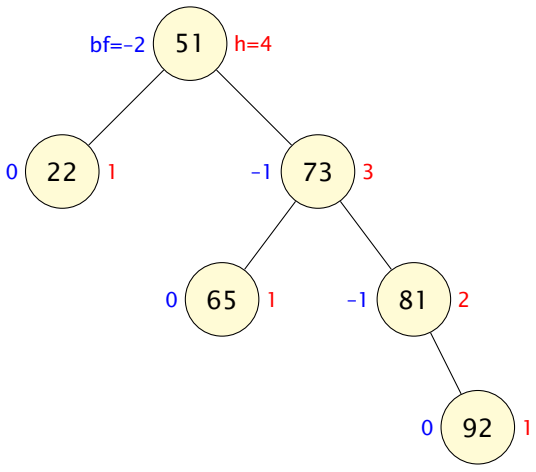
(a) Lay out the keys [51, 22, 73, 65, 81, 92] as a Binary Search Tree, adding the nodes in the order in which they appear in the list, i.e. starting with 51 as the root node.

(b) Label each node with its balance factor. Is the tree balanced? Explain. **(5 marks)**

Go to Soln 8

## 4.10 M269 2016J Exam Soln 8

(a)




(b) The tree is not balanced since node 51 has balance factor -2 which is outside -1,0,1

- Note the height definition here is from my notes not M269

Go to Q 8

## 4.11 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

## 4.12 M269 2016J Exam Q 9

(a) Consider the food web in a certain ecosystem. It can be modelled by a graph in which each node represents an animal or plant species, and where an edge indicates that one species eats another species.

For a **typical** food web, e.g. all animals and plants living in and around a lake, the graph is _____ (choose from UNDIRECTED/DIRECTED) because

*insert answer here*

(b) Is an adjacency matrix a good data structure for a sparse graph? Explain. **(4 marks)**

Go to Soln 9

## 4.13 M269 2016J Exam Soln 9

(a) For a typical food web, the graph is **directed** because the relation is not symmetric: if A eats B, B doesnâĂŹt necessarily eat A.

(b) An adjacency matrix is not a good data structure because it would waste memory: only few of the $n^2$ matrix cells would be non-zero

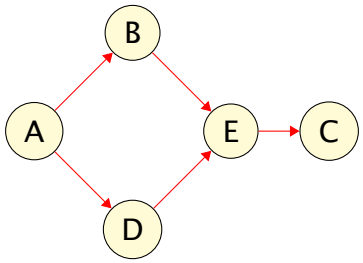
Go to Q 9

## 4.14 M269 2016J Exam Q 10

- The graph showing the dependencies of tasks in a project has been lost. The project manager remembers that there were 5 tasks (let's call them A, B, C, D and E) and that ABCDE and ABEDC were not possible schedules (i.e. topological sorts of the graph), but ABDEC and ADBEC were.
- Draw a directed acyclic graph that is compatible with the given information.
- Each node has to be connected to or from at least one other node. **(4 marks)**

Go to Soln 10

### 4.15 M269 2016J Exam Soln 10




- ABDEC, ADBEC are topological sorts
- ABCDE, ABEDC are not topological sorts
- The graph must be shown with directed edges (arrows)

Go to Q 10

# 5 Units 6 & 7

## 5.1 Propositional Logic

#### M269 Specimen Exam Q11 Topics

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

## 5.2 M269 2016J Exam Q 11

(a) In propositional logic, a tautology is a well-formed formula (WFF) that is TRUE in every possible interpretation.

- It follows that if a WFF is a tautology, it is satisfiable.
- Explain what "satisfiable" means, and why a tautology must be satisfiable.

(b) The following WFF is satisfiable. Complete the truth table.

$(P \vee Q) \rightarrow Q$

| P | Q | $(P \vee Q)$ | $(P \vee Q) \rightarrow Q$ |
|---|---|---|---|
| T | T | | |
| T | F | | |
| F | T | | |
| F | F | | |

- State whether the WFF is a tautology or not, and explain why. **(4 marks)**

Go to Soln 11

## 5.3 M269 2016J Exam Soln 11

(a) A WFF is *satisfiable* if there is at least one interpretation under which the formula is true — hence a tautology is satisfiable

(b) The WFF is not a tautology because the formula is not true under all interpretations — it is false when P is true and q is false

| P | Q | (P ∨ Q) | (P ∨ Q) → Q |
|---|---|---|---|
| T | T | T | T |
| T | F | T | F |
| F | T | T | T |
| F | F | F | T |

Go to Q 11

## 5.4 Predicate Logic

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

## 5.5 M269 2016J Exam Q 12

- A particular interpretation of predicate logic allows facts to be expressed about people and their pets. Some of the assignments in the interpretation are given below (where the symbol $\mathcal{I}$ is used to show assignment).
- The domain of individuals is $\mathcal{D}$ = {Clara, Nicky, Mark, Rex, Fifo, Henny, Admiral}.
- The constants *clara*, *nicky*, *mark*, *rex*, *fifo*, *henny* and *admiral* are assigned to the individuals Clara, Nicky, Mark, Rex, Fifo, Henny and Admiral respectively.
- Four unary predicate symbols are assigned to individuals as follows:
  - $\mathcal{I}$(*person*) = {Clara,Nicky,Mark}
  - $\mathcal{I}$(*pet*) = {Rex,Fifo,Henny,Admiral}
  - $\mathcal{I}$(*dog*) = {Rex,Fifo}
  - $\mathcal{I}$(*chicken*) = {Henny}
- Two further predicate symbols are assigned binary relations as follows:
  - $\mathcal{I}$(*has-pet*) = {(Nicky,Rex),(Nicky,Fifo),(Mark,Henny)}

- $\mathcal{I}(feeds)$ = {(Clara,Rex),(Nicky,Fifo)}

- On the next page, you will be asked whether a given sentence is true or false. In your explanation, you need to consider any relevant values for the variables, and show, using the domain and interpretation above, whether they make the quantified expression TRUE or FALSE.
- **In your answer, when you explain why a sentence is true or false, make sure that you use formal notation.** So instead of stating that *"Henny is a chicken in the interpretation"*, write Henny $\in \mathcal{I}(chicken)$. Similarly, instead of *"Henny is not a dog"* you would need to write Henny $\notin \mathcal{I}(dog)$ **(6 marks)**

(a) Consider the following sentence in English: *"All dogs are Nicky's pets"*. Which **one** well-formed formula is a translation of this sentence into predicate logic?

A. $\forall X.(\text{dog}(X) \wedge \text{has-pet}(\text{nicky}, X))$

B. $\forall X.(\text{dog}(X) \rightarrow \text{has-pet}(\text{nicky}, X))$

C. $\exists X.(\text{dog}(X) \wedge \text{has-pet}(\text{nicky}, X))$

(b) Give an appropriate translation of the well-formed formula $\forall X.\exists Y.(\text{dog}(X) \rightarrow \text{feeds}(Y, X))$ into English

- This well-formed formula is ____ (*choose from TRUE/FALSE*), under the interpretation on the previous page, because:

Go to Soln 12

## 5.6 M269 2016J Exam Soln 12

(a) **B.** *All dogs are Nicky's pets* translates to:

- $\forall X.(\text{dog}(X) \rightarrow \text{has-pet}(\text{nicky}, X))$
- **A.** $\forall X.(\text{dog}(X) \wedge \text{has-pet}(\text{nicky}, X))$ means
- *All objects are dogs and are Nicky's pets*
- **C.** $\exists X.(\text{dog}(X) \wedge \text{has-pet}(\text{nicky}, X))$ means
- *There is some object which is a dog and is Nicky's pet*

(b) $\forall X.\exists Y.(\text{dog}(X) \rightarrow \text{feeds}(Y, X))$ means

*All dogs are fed by someone*

- But not *Somebody feeds all dogs* which would be

  $\exists Y.\forall X.(\text{dog}(X) \rightarrow \text{feeds}(Y, X))$

- This is true because

(i) If X is not a dog then the implication is true

(ii) We have $\mathcal{I}(dog)$ = {Rex, Fifo} and we have (Clara,Rex) $\in \mathcal{I}(feeds)$ and (Nicky,Fifo) $\in \mathcal{I}(feeds)$

Go to Q 12

## 5.7 SQL Queries

### M269 Specimen Exam Q13 Topics

- Unit 6
- SQL queries

## 5.8 M269 2016J Exam Q 13

- A database contains the following tables, *lawnmower* and *brand*. **(6 marks)**

*lawnmower*

| *make* | *model* | *type* |
|---|---|---|
| MowIt | Bella | push |
| MowIt | Speedy | electric |
| Mamouth | Kodiak | petrol |
| Mamouth | Pachyderm | petrol |
| Blades | Meadow | petrol |
| Blades | Nibble | robot |
| Blades | Yard | electric |

*brand*

| *manufacturer* | *location* |
|---|---|
| Mamouth | France |
| MowIt | USA |
| Blades | China |
| Scythes | China |

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

```
SELECT make, model
FROM lawnmower
WHERE type = 'electric';
```

- Write the question that the above query is answering.

(b) Write an SQL query that answers the question *Which lawnmowers are from manufacturers located in China?* The answer should be the following table:

| *manufacturer* | *model* |
|---|---|
| Blades | Meadow |
| Blades | Nibble |
| Blades | Yard |

Go to Soln 13

## 5.9 M269 2016J Exam Soln 13

(a)

| *make* | *model* |
|---|---|
| MowIt | Speedy |
| Blades | Yard |

- *Which models of which makes are electric lawnmowers ?*

(b)

```
SELECT manufacturer, model
FROM lawnmower CROSS JOIN brand
WHERE make = manufacturer
AND location = 'China';
```

- Also allow

```
FROM lawnmower, brand
```

Go to Q 13

## 5.10 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 $\begin{array}{l} P_1 \\ \vdots \\ P_n \\ \hline C \end{array}$
- 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:

    $\overline{C}$

### 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:

$$\frac{Argument_1 \quad \ldots \quad Argument_n}{Argument}\ (label)$$

### Inference Rules — Conjunction

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

### Inference Rules — Implication

- $\dfrac{\Gamma \cup \{\mathbf{A}\} \vdash \mathbf{B}}{\Gamma \vdash \mathbf{A} \to \mathbf{B}}$ (→-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 **A** → **B**, *under the unaugmented set of premises,* Γ.

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

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

- $\dfrac{\Gamma \vdash \mathbf{A} \quad \Gamma \vdash \mathbf{A} \to \mathbf{B}}{\Gamma \vdash \mathbf{B}}$ (→-elimination)

### Inference Rules — Disjunction

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

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

- The above should be read: if a set of premises Γ justifies the conclusion **A** ∨ **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:

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

**Self-Assessment activity 7.4 — tree layout**

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

**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]

## 5.11 M269 2016J Exam Q 14

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

A. If a decision problem is in NP, then it is computable.

B. The complexity of an algorithm that solves a problem places a lower bound on the complexity of the problem itself.

C. If the best algorithm we currently have for solving a decision problem has complexity $O(2^n)$, then we know that problem can't be in P.

D. If an NP-hard problem A can be Karp-reduced to a problem B, then problem B is NP-hard too.

E. Every NP-hard problem is also NP-complete.

Go to Soln 14

## 5.12 M269 2016J Exam Soln 14

A. ✔ If a decision problem is in NP, then it is computable.

B. The complexity of an algorithm that solves a problem places a lower bound on the complexity of the problem itself.

C. If the best algorithm we currently have for solving a decision problem has complexity $O(2^n)$, then we know that problem can't be in P.

D. ✔ If an NP-hard problem A can be Karp-reduced to a problem B, then problem B is NP-hard too.

E. Every NP-hard problem is also NP-complete.

Go to Q 14

## 5.13 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 $\Sigma$ is an alphabet, and L is a language over $\Sigma$, that is $L \subseteq \Sigma^*$, where $\Sigma^*$ is the set of all strings over the alphabet $\Sigma$ 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

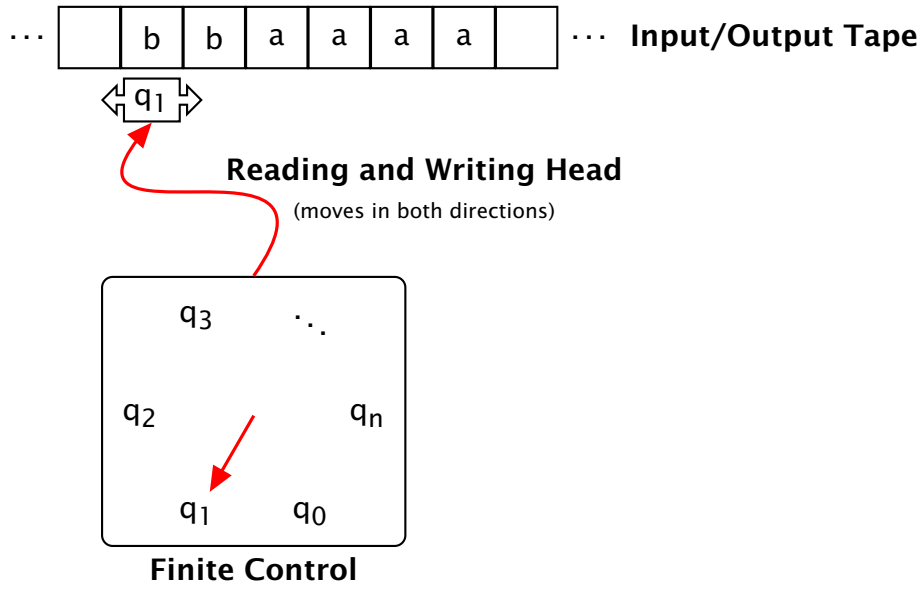
### Computability — Turing Machine

- **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
- $\Sigma$ finite set of *input symbols* (M269 S)
- $\Gamma$ complete set of *tape symbols* $\Sigma \subset \Gamma$
- $\delta$ Transition function (M269 instructions, I)
  $\delta :: Q \times \Gamma \to Q \times \Gamma \times \{L, R, S\}$
  $\delta(q, X) \mapsto (p, Y, D)$
- $\delta(q, X)$ 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)

### 5.13.1 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

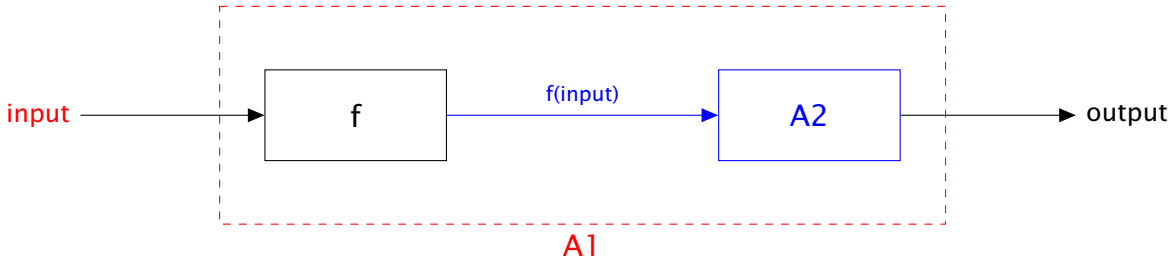
```
q(f)
  = h(f,f)

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.

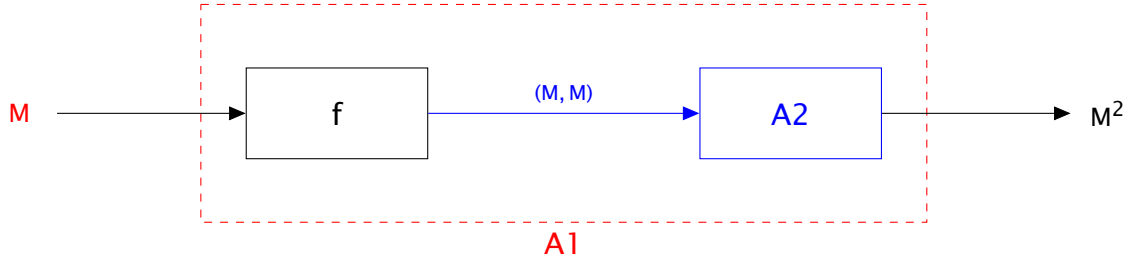
### 5.13.2 Reductions & Non-Computability




- 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

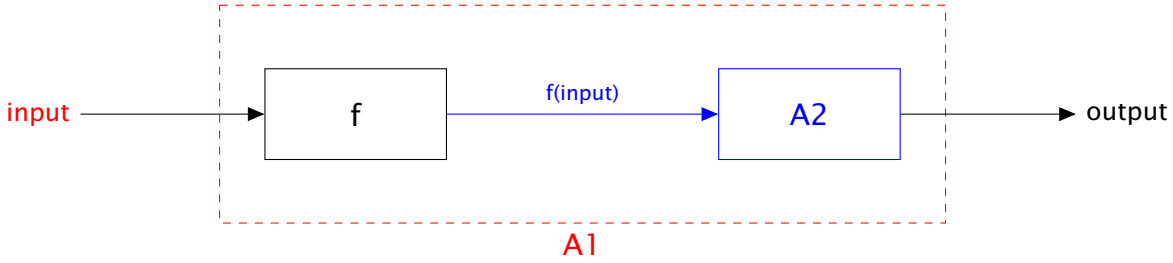



- Given an algorithm (A2) for matrix multiplication ($P_2$)
  - Input: pair of matrices, $(M_1, M_2)$
  - Output: matrix result of multiplying $M_1$ and $M_2$
- $P_1$ is the problem of squaring a matrix
  - Input: matrix M
  - Output: matrix $M^2$
- Algorithm A1 has

  f(M) = (M, M)

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

### Non-Computable Problems

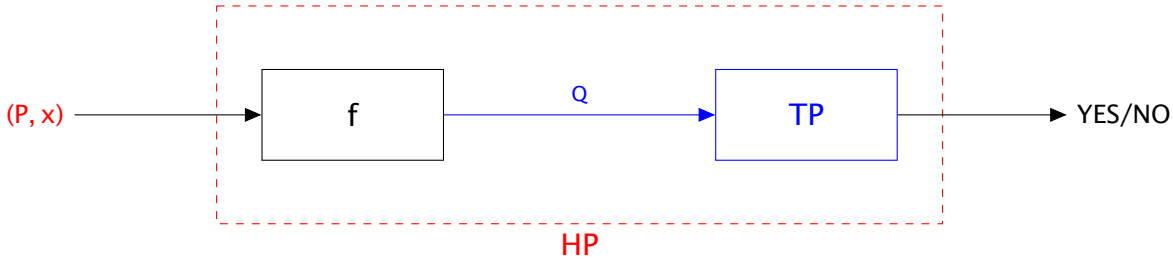



- If $P_2$ is computable (A2 exists) then $P_1$ is computable (f being simple or polynomial)
- Equivalently If $P_1$ is non-computable then $P_2$ is non-computable
- **Exercise:** show $B \rightarrow A \equiv \neg A \rightarrow \neg B$
- Proof by Contrapositive
- $B \rightarrow A \equiv \neg B \vee A$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

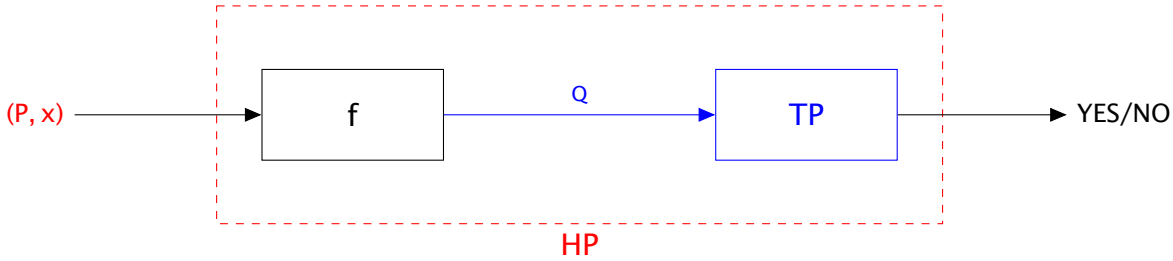
  $\equiv \neg(\neg A) \vee \neg B$ commutativity and negation laws

  $\equiv \neg A \rightarrow \neg B$ 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



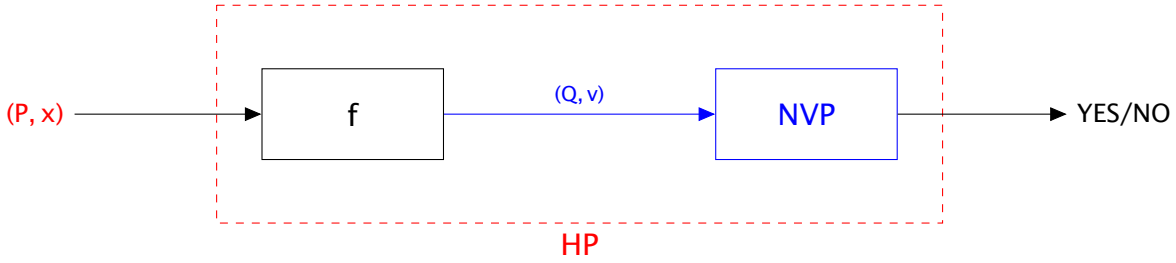

- Define f to transform inputs to HP to TP pseudo-Python

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

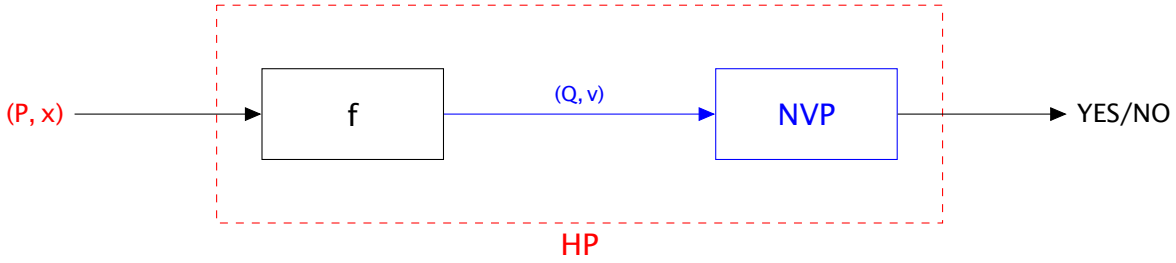
```
    return Q
```

- 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

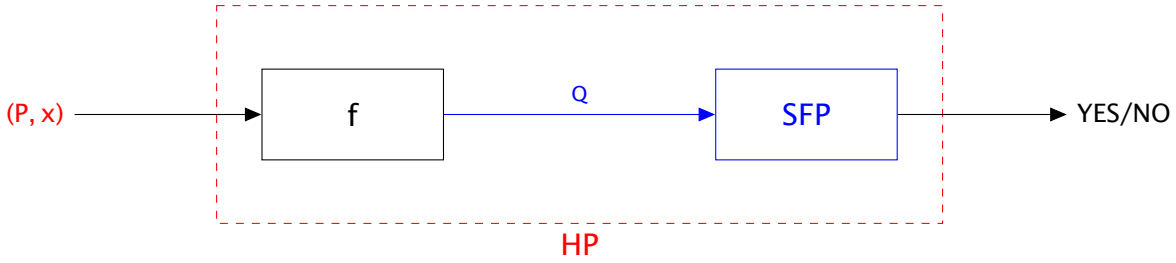



- Define f to transform inputs to HP to NVP pseudo-Python

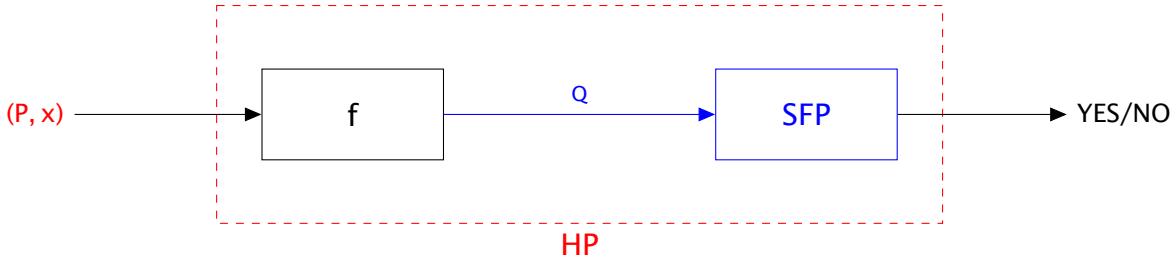
```
def f(P,x) :
  def Q(y):
    # ignore y
    P(x)
    v = -1
  return (Q,var(v))
```

- 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

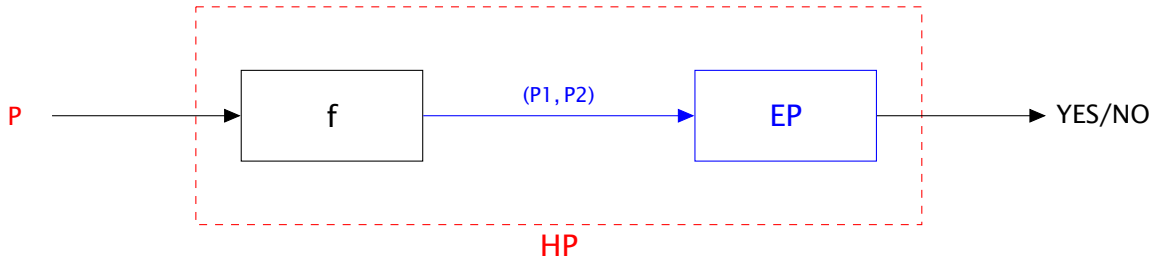



- Define f to transform inputs to HP to SFP pseudo-Python

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

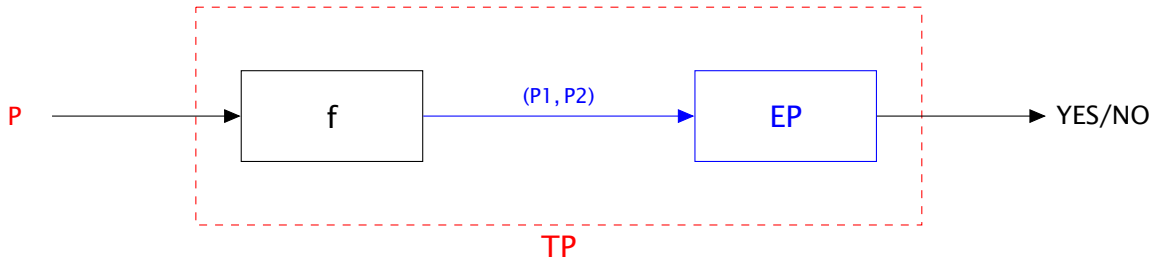
- Run SFP on Q
  - If SFP returns YES then P halts on x
  - If SFP returns NO then P does not halt on x
- We have *solved* the Halting Problem — contradiction

### 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

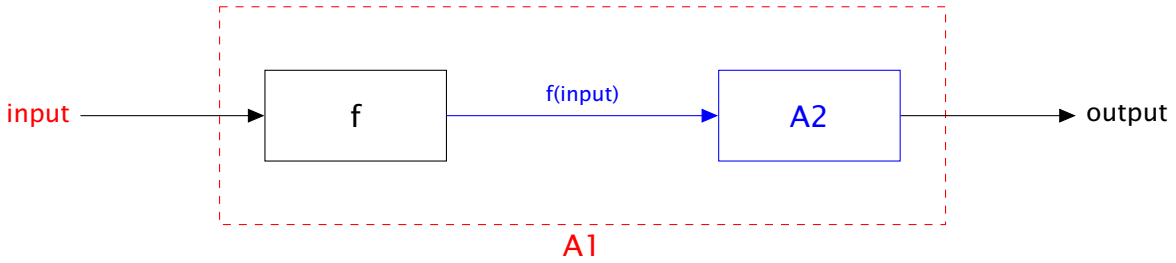



- Define f to transform inputs to TP to EP pseudo-Python

```
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.

## 5.14 M269 2016J Exam Q 15

- Consider the following decision problems: **(4 marks)**

1. The 3SAT Problem
2. Is a given list of numbers already sorted?
3. The Totality Problem
4. Is a given path from A to B in a given undirected graph the shortest path from A to B?

- For each of the following groups of problems, write on the line the numbers of any of the above problems that belong to that group, or write "*none*" if none of the above problems belongs to that group.

(a) undecidable

(b) tractable

(c) NP-complete

Go to Soln 15

## 5.15 M269 2016J Exam Soln 15

(a) Undecidable: **3.**Totality Problem

(b) Tractable: **2.** Sorted?, **4.** Path?

(c) NP-complete: **1.** 3SAT Problem

Go to Q 15

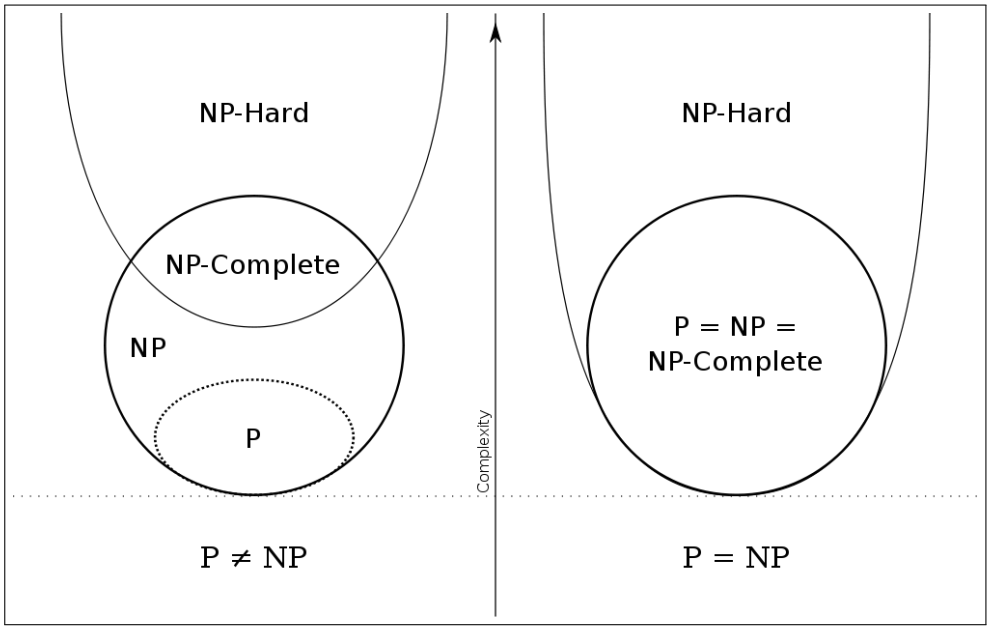
## 5.16 Complexity

### 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

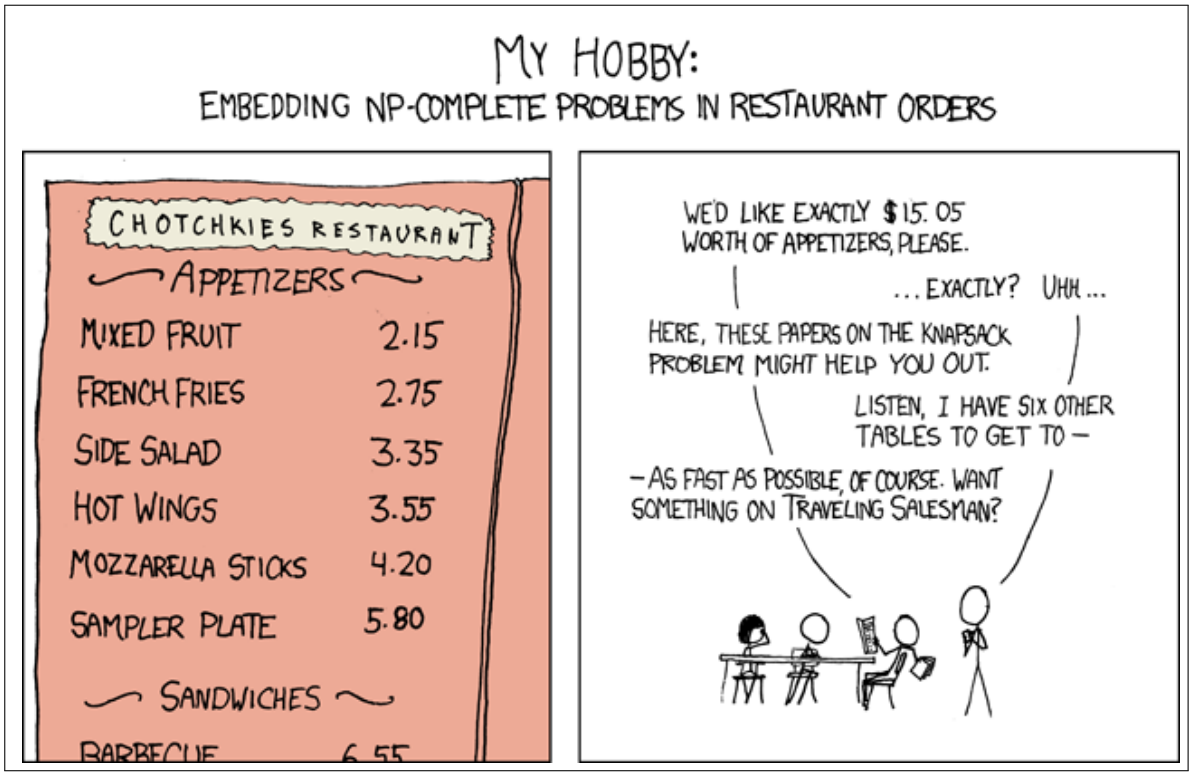



Source: Wikipedia NP-complete entry

**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

### 5.16.1 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:
- $\Sigma$ is a set of symbols — the alphabet
- $\Sigma^k$ is the set of all string of length k, which each symbol from $\Sigma$
- Example: if $\Sigma = \{0, 1\}$
  - $\Sigma^1 = \{0, 1\}$
  - $\Sigma^2 = \{00, 01, 10, 11\}$
- $\Sigma^0 = \{\epsilon\}$ where $\epsilon$ is the empty string

- $\Sigma^*$ is the set of all possible strings over $\Sigma$
- $\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \ldots$
- A *Language*, L, over $\Sigma$ is a subset of $\Sigma^*$
- $L \subseteq \Sigma^*$

**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)

# 6 M269 Exam 2016J 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

## 6.1 M269 2016J Exam Q 16

- **Question 16** **(20 marks)**
- Consider an ADT for **undirected** graphs, named UGraph, which includes these two operations:
- `nodes`, which returns a sequence of all nodes in the graph, in no particular order;
- `neighbours`, which takes a node and returns a sequence of all its adjacent nodes, in no particular order.
- How each node is represented is irrelevant.

(a) The following stand-alone Python function checks if a graph has a loop (an edge from a node to itself), assuming that UGraph is implemented as a Python class.

```
def hasLoop(graph):
  for node in graph.nodes():
    if node in graph.neighbours(node):
      return True
  return False
```

- Assume that the if-statement guard does a linear search of the sequence returned by `neighbours`.
- If the graph has no node with a loop, is that a best-, average-, or worst-case scenario for `hasLoop`?
- Assuming the graph has $n$ nodes and $e$ edges, what is the Big-O complexity of that scenario? Justify your answers.
- Note that the complexity is in terms of how many nodes and edges hasLoop visits, because it has no assignments. **(5 marks)**

(b) A node is *isolated* if it has no adjacent nodes. Isolated nodes cannot be reached from any other node and hence won't be processed by some graph algorithms.

- It is therefore useful to first check if a graph has isolated nodes.

(i) Specify the problem of finding all isolated nodes in an undirected graph by completing the following template.

- Note that `isolatedNodes` 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:** `isolatedNodes`
>
> **Inputs**:
>
> **Outputs:**
>
> **Preconditions**
>
> **Postconditions**

(ii) If instead of being an independent problem, `isolatedNodes` were an operation of the UGraph ADT, would it be a creator, inspector or modifier? Explain why. **(2 marks)**

(iii) Give your initial insight for an algorithm that solves the problem, using the ADT's operations. **(4 marks)**

(c) The ACME company used Prim's algorithm to connect its data centres with the least amount of fibre optic cable necessary.

- One of the centres is a gateway to the Internet.
- ACME wants to know the maximum latency for an Internet message to reach any centre.
- In other words, they want to know which centre is the furthest away from the gateway and what is the distance.
- State and justify which data structure(s) and algorithm(s) you would adopt or adapt to solve this problem efficiently.

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

Go to Soln 16

## 6.2 M269 2016J Exam Q 17

- Imagine you have been invited to write a guest post for a technology blog, aimed at interested readers who know little about computing.
- Write a draft of your blog post, which will explain relational databases and the formal logic that underpins them. **(15 marks)**
- It should have

1. A suitable title and a short paragraph *'setting the scene'* by explaining the practical importance of relational databases.
2. A paragraph describing in layperson's terms what a relational database is and how it's organised.
3. A paragraph describing in layperson's terms what predicate logic is and its relationship with relational databases.
4. A concluding paragraph stating your view on the importance, or not, of information technologies having a formal logic basis.

- Note that 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'* to the reader.
- You may wish to use examples in your text to help explain the concepts.
- As a guide, you should aim to write roughly three to five sentences per paragraph.

Go to Soln 17

# 7 M269 Exam 2016J Soln Part2

- Part 2 solutions

Go to Q Part2

## 7.1 M269 2016J Exam Soln 16

(a) It is a worst-case scenario since there is no early exit from the loop, before returning false.

- The complexity is O(n + e) since all nodes are visited by the outer loop, and all edges are visited by the linear search through the neighbours of each node.

(b) **(i) Name:** isolatedNodes

- **Inputs:** an undirected graph *theGraph* (or a Ugraph *theGraph*)

- **Outputs:** *isolated*, a set of nodes
- **Preconditions:** true
- **Postconditions:** all nodes without neighbours in *theGraph* are in *isolated*; each node in *isolated* has no neighbours in *theGraph*

  *Alternative:* a node is in *isolated* if and only if it has no neighbours in *theGraph*

(b) **(ii)** It would be an inspector because *theGraph* is not in the outputs.

*Alternative:* because the operation does not create or modify a graph.

- **(iii)** Initialise isolated to the empty set.

  Iterate over the nodes of *theGraph* and for each one check if its neighbours is the empty sequence.

  If so, add the node to *isolated*.

(c) The data structure is a weighted tree (alternative: acyclic graph).

Prim → Minimum Spanning Tree

The nodes represent the data centres.

The edges represent the cables.

The weights represent the cable lengths.

- To compute the longest path, do any traversal of the tree starting at the gateway node and add the weights of the edges visited.

  For an efficient, single-pass algorithm, when visiting a leaf, check if its distance is the maximum so far.

- *Alternative:* calculate the height of the tree with cable lengths

Go to Q 16

## 7.2 M269 2016J Exam Soln 17

- There is no definitive answer — here are some points:

1. *Setting the scene with the importance of relational databases:*

- All retailers need to keep data on their products, suppliers and clients, the properties of those entities (e.g. current stock of a product) and their relationships (e.g. who bought which product to issue invoices).
- Storing entities and their properties and relationships is such a generic need across business, government departments and other organisations that so-called relational databases were invented for that purpose.

2. *What are relational databases:*

- It is a data structure that represents each entity type as a table, with one column per property and one row per entity, e.g. a table to represent customers may have columns for their name and address.

- A table can also represent a relation, e.g. a table with customer names and product ids would store who bought what.
- A database can be queried to retrieve information from the database, e.g. which other customers bought a particular book

3. *What is predicate logic and its relation to relational databases:*

- Predicate logic is a formal language to represent unambiguously statements about entities and their properties and relations, e.g. *No customer in Yorkshire bought a polka dot dress.*
- Given information about the existing entities and their properties/relations, it is possible to prove whether a predicate logic statement is true or false.
- A database query is a particular form of a predicate logic statement.
- Running a query is an automated proof: it returns the entities stored in the database that make the statement true; if no entities are returned, the statement is false.

4. *Conclusion:*

- Formal logic helps verifying the correctness of systems, which is important for our daily reliance on them.
- There are limits on what is computable, and a system may be correct but not fit for purpose, so formal logic doesn't suffice for quality assurance.

Go to Q 17

# 8 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**

# 9 White Slide

# 10 Web Sites & References

## 10.1 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/bri 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.

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.