M269

Exam Revision

Contents

1 M269 Exam Revision Agenda & Aims

1. Welcome and introductions
2. Revision strategies
3. M269 Exam — Part 1 has 15 questions 60%
4. M269 Exam — Part 2 has 2 questions 40%
5. M269 Exam — 3 hours, Part 1 100 mins, Part 2 70 mins
6. M269 2015J exam (June 2016)
7. Topics and discussion for each question
8. Exam techniques
9. Two sessions
10. *OU Live* — if you or I get cut off, wait till we reconnect
11. These slides and notes are in Dropbox at `https://db.tt/WUMSBB4csL`

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 2015J Exam Qs

2.1 M269 2015J Exam Qs

- M269 Algorithms, Data Structures and Computability
- Presentation 2015J Exam
- Date Thursday, 2 June 2016 Time 14:30–17:30
- There are **TWO** parts to this examination. You should attempt all questions in **both** parts
- **Part 1** carries **60 marks** — **100 minutes**
- **Part 2** carries **40 marks** — **70 minutes**
- **Note** see the original exam paper for exact wording and formatting — these slides and notes may change some wording and formatting

2.2 M269 2015J 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 2015J Exam Q 1

- **Question 1** Which **two** of the following statements are true? **(2 marks)**

A. Computational thinking consists of the skills to formulate a problem as a computational problem and then construct a good computational solution to solve it or explain why there is no such solution.

B. Every computable problem can be solved in a practical way using existing computers.

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

D. An algorithm consists of a computer program that will solve a computable problem.

Go to Soln 1

3.3 M269 2015J Exam Soln 1

- A, C

Go to Q 1

3.4 M269 2015J Exam Q 2

- **Question 2** Which **two** of the following statements are true? **(2 marks)**

A. Abstraction allows us to manage complexity.

B. In abstraction as modelling, we hide the details of an implementation behind an interface.

C. Every algorithm can be expressed as some combination of sequence, iteration and selection.

D. If a polynomial algorithm is executed, it will quickly overwhelm the resources of a computer and exceed any reasonable time limits.

Go to Soln 2

3.5 M269 2015J Exam Soln 2

- A, C

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

```python
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

```
def binarySearchIterMR(alist, item):
  first = 0
  last = len(alist)-1
  found = False

  while first<=last and not found:
    midpoint = (first + last)//2
    if alist[midpoint] == item:
      found = True
    else:
      if item < alist[midpoint]:
        last = midpoint-1
      else:
        first = midpoint+1

  return found
```

Miller and Ranum (2011, page 192)

Binary Search Recursive — Miller & Ranum

```python
def binarySearchRecMR(alist, item):
  if len(alist) == 0:
    return False
  else:
    midpoint = len(alist)//2
    if alist[midpoint]==item:
      return True
    else:
      if item<alist[midpoint]:
        return binarySearchRecMR(alist[:midpoint],item)
      else:
        return binarySearchRecMR(alist[midpoint+1:],item)
```

Miller and Ranum (2011, page 193)

3.7 M269 2015J Exam Q 3

- **Question 3** In roughly three or four sentences (in total) explain what is meant by the following terms: **(4 marks)**
- Abstract data type (ADT)
- Encapsulation
- Data structure

Go to Soln 3

3.8 M269 2015J Exam Soln 3

- An *abstract data type* is a logical description of how we view the data and the operations that are allowed without regard to how they will be implemented. See Miller and Ranum chp 1 and WIkipedia: Abstract data type
- *Encapsulation* hides the implementation of an ADT so a user must only access data via the interface and not directly.
- A *data structure* is a concrete implementation of some ADT

Go to Q 3

3.9 M269 2015J Exam Q 4

- **Question 4** Consider the guard in the following Python `while` loop header: **(4 marks)**

```python
while (s < 5 and t > 3) or not(s >= 5 or t <= 3):
```

(a) Make the following substitutions:

P represents `s < 5`

Q represents `t > 3`

Then complete the following truth table:

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

(b) 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.

A. `not (s < 5 and t > 3)`

B. `(s >= 5 or t <= 3)`

C. `(s < 5 and t > 3)`

D. `(s >= 5 and t <= 3)`

E. `(s < 5 and t <= 3)`

Go to Exam Soln 4

3.10 M269 2015J Exam Soln 4

(a) Truth table

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

(b) C

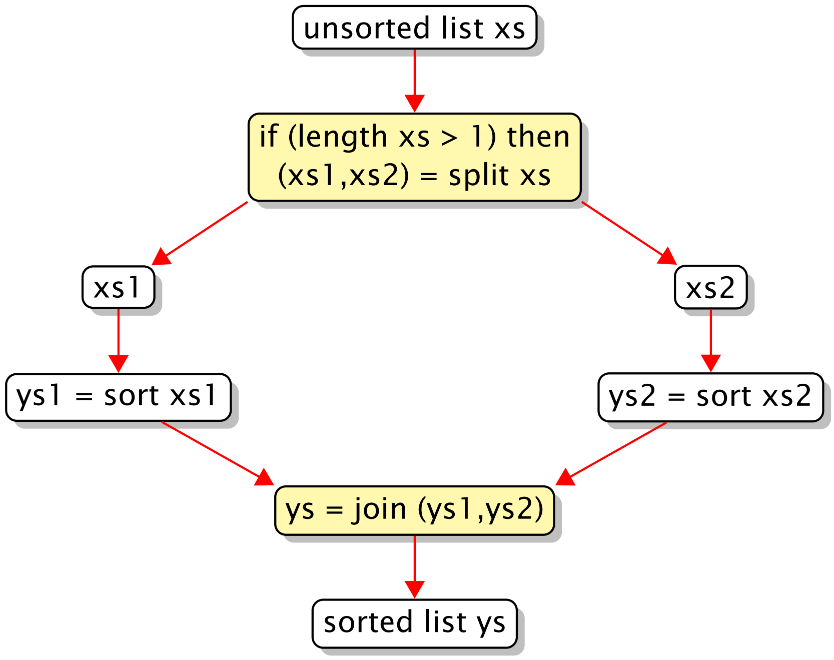
Go to Q 4

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:

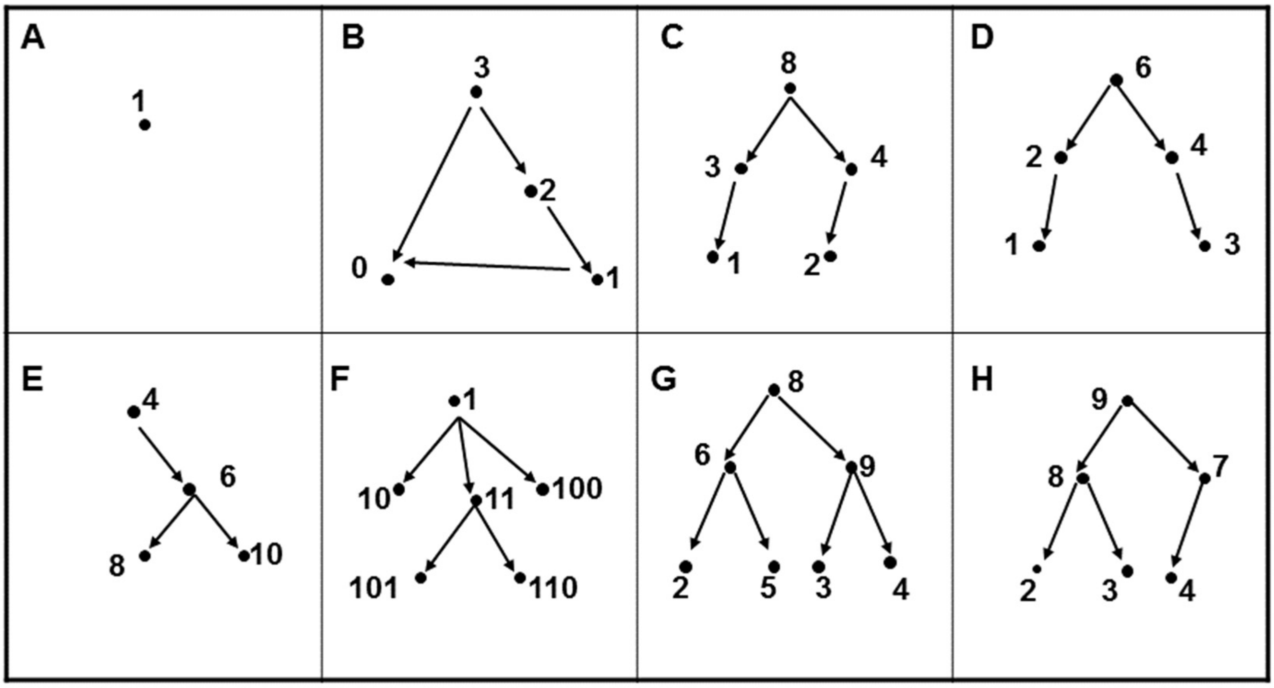
- Insertion sort
- Selection sort
- Merge sort
- Quicksort
- Bubble sort (the odd one out)

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 2015J Exam Q 5

- **Question 5** Consider the following diagrams A–H. Nodes are represented by black dots and edges by arrows. The numbers represent a node's key. **(4 marks)**



- Answer the following questions. Write your answer on the line that follows each question. In each case there is at least one diagram in the answer but there may be more than one. Explanations are **not** required.

(a) Which of A, B, C and D **do not** show trees ?

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

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

(d) Which of C, D, G and H are binary heaps ?

Go to Exam Soln 5

4.4 M269 2015J Exam Soln 5

(a) B is not a tree; it has more than one route from node 3 to node 0.

(b) E, G, and H are binary trees; (no more than 2 children per node).

(c) G, and H are complete binary trees.

(d) Only H is a heap; (complete binary tree, and parent nodes > children).

Go to Q 5

4.5 M269 2015J Exam Q 6

- **Question 6** Consider the following function, which takes an integer argument n. You can assume that n is positive. **(4 marks)**

```
def calculate(n):
    a = 5
    ans = 0
    for i in range(n):
        x = i * i
        for j in range(n):
```

```
            y = x + j * j
            for k in range(n):
                z = y + i * k
                ans = ans + z * a
    return ans
```

Go to Soln 6

- From the five options below, select the **one** that represents 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. $T(n) = n^3 + n^2 + n + 3$ and $O(n^3)$

B. $T(n) = 2n^3 + n^2 + 2$ and $O(2n^3)$

C. $T(n) = 2n^2 + n + 2$ and $O(n^2)$

D. $T(n) = 2n^3 + n^2 + n + 2$ and $O(n^3)$

E. $T(n) = 3n + 6$ and $O(n)$

- Now explain how you obtained T(n) and the Big-O complexity.

Go to Exam Soln 6

4.6 M269 2015J Exam Soln 6

D. $T(n) = 2n^3 + n^2 + n + 2$ and $O(n^3)$

- 2 assignment statements outside the loops
- 1 assignment statement in the outer loop
- 1 assignment statement in the middle loop
- 2 assignment statements in the inner loop
- n^3 is the dominat term

Go to Q 6

4.7 M269 2015J Exam Q 7

- **Question 7** In the KMP algorithm, for each character in turn, as it appears in the target string T, we identify the longest substring of T ending with that character which matches a prefix of T.
- These lengths are stored in what is known as a prefix table (which in Unit 4 we represented as a list).
- Consider the target string T:

  CDCECDCECE

- Below is an incomplete prefix table for the target string given above. Complete the prefix table by writing the missing numbers in the appropriate boxes. **(4 marks)**

C	D	C	E	C	D	C	E	C	E
0		1	0		2		4		0

Go to Soln 7

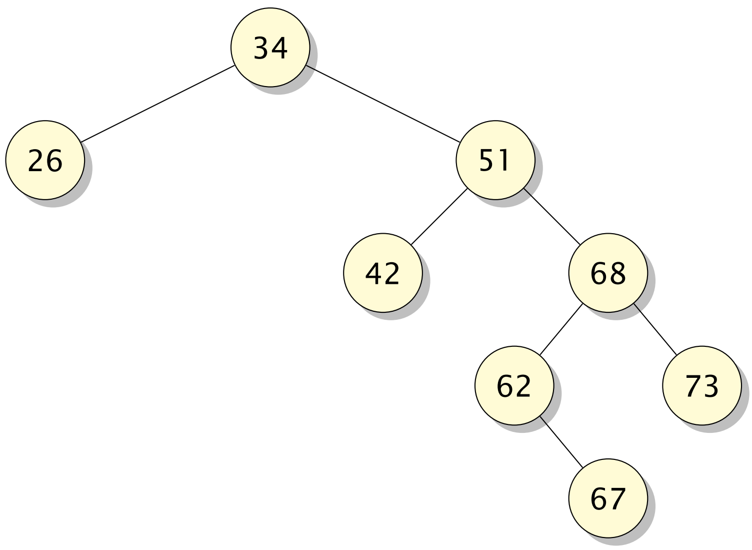
4.8 M269 2015J Exam Soln 7

C	D	C	E	C	D	C	E	C	E
0	0	1	0	1	2	3	4	1	0

Go to Q 7

4.9 M269 2015J Exam Q 8

- Consider the following Binary Search Tree. **(4 marks)**



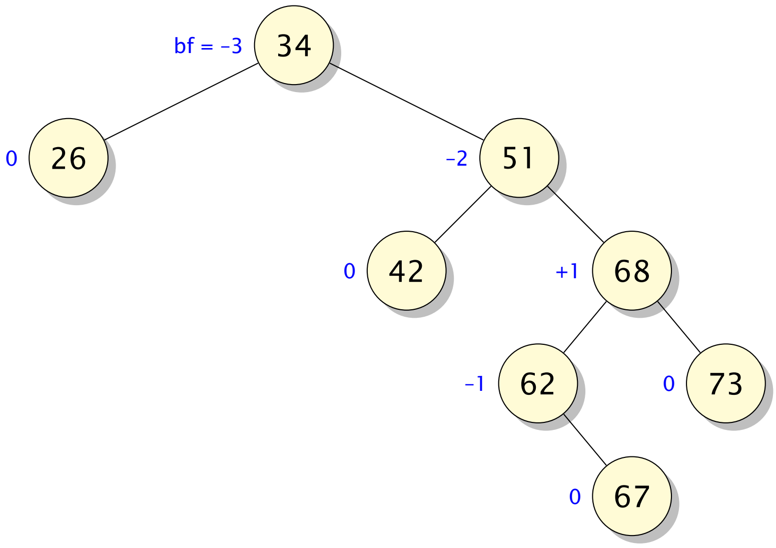
(a) Calculate the balance factors of each node in the above tree and annotate the above tree to show these balance factors.

(b) Redraw the tree after node 51 has been deleted.

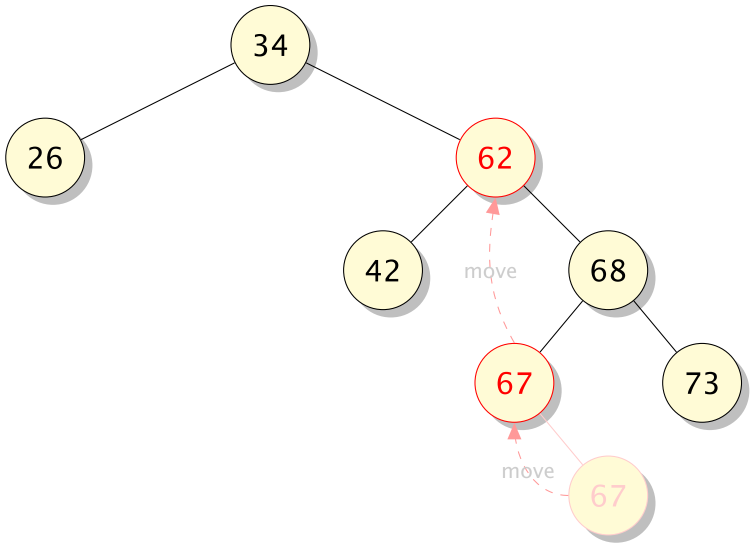
Go to Soln 8

4.10 M269 2015J Exam Soln 8

(a) Balance factors



(b) Delete 51



Go to Exam 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

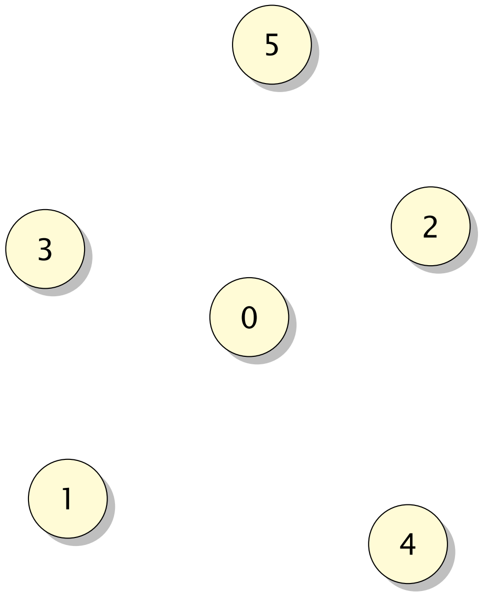
4.12 M269 2015J Exam Q 9

- **Question 9** In Python a dictionary of dictionaries can be used to represent a graph's adjacency list. Consider the following: **(4 marks)**

```
graph2 = {
  0:{'neighbours': [1,2,3,4]},
  1:{'neighbours': [0,3,4]},
  2:{'neighbours': [0,5]},
  3:{'neighbours': [0,1,5]},
  4:{'neighbours': [0,1]},
  5:{'neighbours': [2,3]}}
```

Go to Soln 9

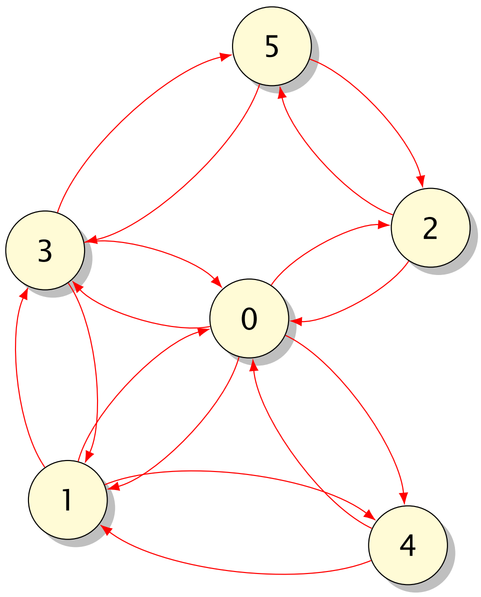
- In the space provided below, **complete** the graph corresponding to the adjacency list given above.



Go to Exam Soln 9

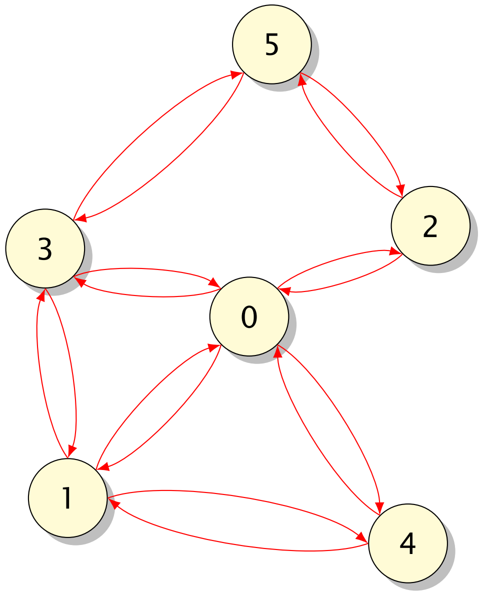
4.13 M269 2015J Exam Soln 9

- Here is a representation with unidirectional edges

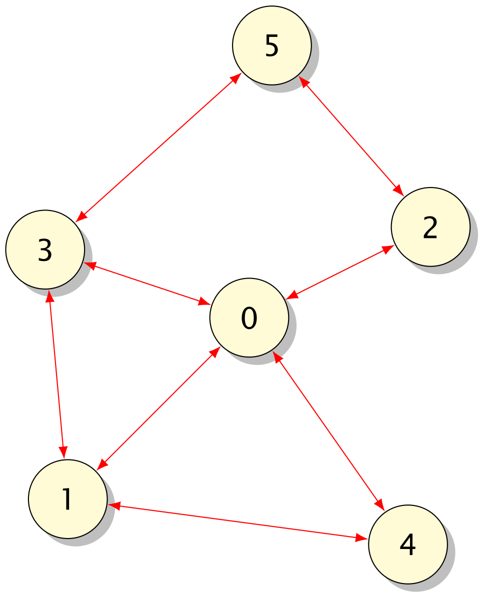


Go to Q 9

- Here is a representation with unidirectional edges



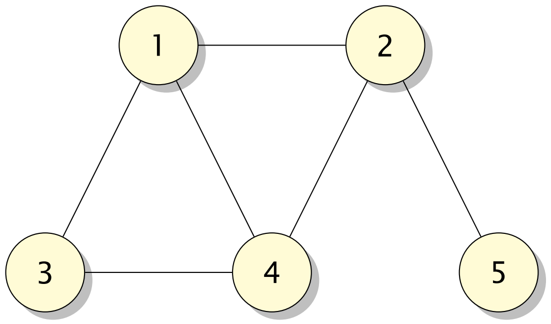
- Here is a representation with bidirectional edges (but we have not been told that every edge has a reverse edge and of the same weight or length)



Go to Exam Q 9

4.14 M269 2015J Exam Q 10

- **Question 10** Consider the following graph: **(4 marks)**

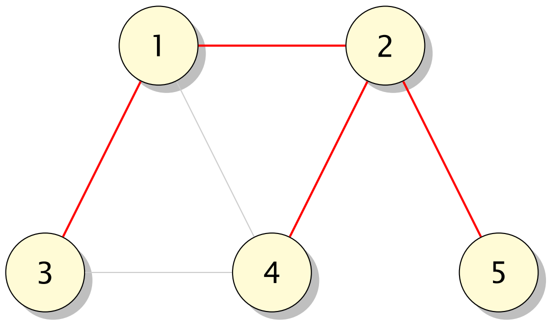


- In the space provided below, draw **one** spanning tree that could be generated from a **Breadth First Search** of the above graph starting at vertex 2.

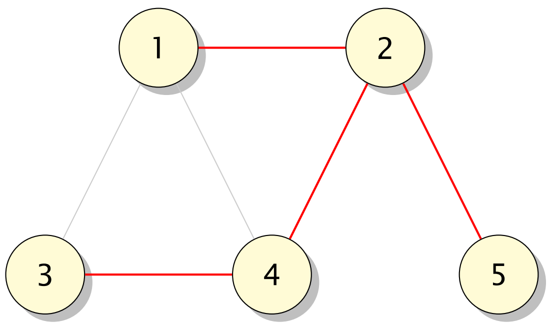
Go to Soln 10

4.15 M269 2015J Exam Soln 10

- *Spanning tree* from *breadth first search* from vertex 2 (1 of 2, in red)



- *Spanning tree* from *breadth first search* from vertex 2 (2 of 2, in red)



Go to Exam 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 2015J Exam Q 11

- **Question 11** **(4 marks)**

(a) What does it mean to say that two well-formed formulas (WFFs) are **logically equivalent** ? Use the space below for your answer.

(b) Is the following set of propositional WFFs **satisfiable** ?

$\{(P \to Q), (Q \to P)\}$

- Explain how you arrived at your answer in the space below:

Go to Soln 11

5.3 M269 2015J Exam Soln 11

(a) Two well-formed formulas (WFFs) **A** and **B** are logically equivalent if and only if **A** and **B** have the same value in all interpretations.

(b) The sets of WFFs is *satisfiable* if each member has the value *True* for some interpretation

P	Q	$P \to Q$	$Q \to P$
T	T	T	T
T	F	F	T
F	T	T	F
F	F	T	T

- The set is satisfiable

Go to Q 11

5.4 Predicate Logic

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

5.5 M269 2015J Exam Q 12

- **Question 12** Consider a domain with some board games and people. **(6 marks)**

  $\mathcal{D}$ = {Backgammon, Chess, Draughts, Joe, Mary, Sue}
- An interpretation assigns people to corresponding constants (you won't need the constants for games).

  $\mathcal{I}(joe)$ = Joe

  $\mathcal{I}(mary)$ = Mary

  $\mathcal{I}(sue)$ = Sue
- The predicates *owns* and *likes* are assigned to binary relations with the following comprehensions:

  $\mathcal{I}(owns)$ = {(P, G): the person P owns the game G}

  $\mathcal{I}(likes)$ = {(P, G): the person P likes the game G}

Go to Soln 12

- The enumerations of the relations are:
- $\mathcal{I}(owns)$ = {(Joe, Chess), (Mary, Backgammon), (Sue, Draughts)}
- $\mathcal{I}(likes)$ = {(Joe, Backgammon), (Mary, Backgammon), (Mary, Draughts), (Sue, Backgammon), (Sue, Chess)}
- You will find the questions on the next page.
- You are asked to translate a sentence of predicate logic to English or vice-versa.
- You also need to state whether the sentence is TRUE or FALSE in the interpretation that is provided on this page, and give an explanation of your answer.
- In your explanation you need to consider any relevant values for the variables, and *show, using the interpretation above, whether they make the quantified expression TRUE or FALSE*.
- When your explanation refers to the interpretation, make sure that you use formal notation.
- So instead of saying that *Joe likes Backgammon according to the interpretation*, write:

  (Joe, Backgammon) $\in \mathcal{I}(likes)$.
- Similarly, instead of *Joe doesn't like Backgammon* you would need to write:

  (Joe, Backgammon) $\notin \mathcal{I}(likes)$.

(a) $\forall X.(owns(joe, X) \rightarrow likes(joe, X))$

can be translated into English as:

- This sentence is ____ (choose from TRUE/FALSE), because:

(b) *There's something that both Mary and Sue like*

can be translated into predicate logic as:

- This sentence is ____ (choose from TRUE/FALSE), because:

Go to Exam Soln 12

5.6 M269 2015J Exam Soln 12

(a) Joe likes the games he owns

- False — Joe owns Chess but does not like it
- (Joe, Chess) $\in \mathcal{I}(owns)$
- but (Joe, Chess) $\notin \mathcal{I}(likes)$

(b) $\exists X.(likes(mary, X) \wedge likes(sue, X))$

- True — they both like Backgammon

Go to Q 12

5.7 SQL Queries

M269 Specimen Exam Q13 Topics

- Unit 6
- SQL queries

5.8 M269 2015J Exam Q 13

- **Question 13** The interpretation of the previous question can also be represented by a database with the following tables, *owns* and *likes*. **(6 marks)**

owns	
owner	*boardgame*
Joe	Chess
Mary	Backgammon
Sue	Draughts

likes	
person	*game*
Joe	Backgammon
Mary	Backgammon
Mary	Draughts
Sue	Backgammon
Sue	Draughts

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

```
SELECT person
FROM owns CROSS JOIN likes
WHERE game = boardgame AND person = owner;
```

- Write the question that the above query is answering.

(b) Write an SQL query that answers the question

Which games does Sue like?

- The answer should be the following table:

game
Backgammon
Chess

Go to Exam Soln 13

5.9 M269 2015J Exam Soln 13

(a) The table

Mary
Sue

- Who owns games they like ?

(b) The SQL query

```
SELECT game
FROM likes
WHERE person = 'Sue';
```

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 $\begin{array}{c} P_1 \\ \vdots \\ P_n \\ \hline C \end{array}$
- 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: $\emptyset \models C$
  - 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 $\{\mathbf{A}\}$ 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

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

Inference Rules — Implication

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

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

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

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

Inference Rules — Disjunction

- $\dfrac{\Gamma \vdash \mathbf{A}}{\Gamma \vdash \mathbf{A} \vee \mathbf{B}}$ ($\vee$-introduction left)
- $\dfrac{\Gamma \vdash \mathbf{B}}{\Gamma \vdash \mathbf{A} \vee \mathbf{B}}$ ($\vee$-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 $\mathbf{A}$ or $\mathbf{B}$ separately justifies $\mathbf{C}$, then Γ justifies $\mathbf{C}$

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:

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

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
- $\dfrac{\Gamma \vdash P \vee Q \qquad \dfrac{\begin{array}{c}\Gamma \cup \{P\} \\ \vdash P\end{array} \quad \begin{array}{c}\Gamma \cup \{P\} \\ \vdash P \to R\end{array}}{\Gamma \cup \{P\} \vdash R}\,(\to\text{-E}) \qquad \dfrac{\begin{array}{c}\Gamma \cup \{Q\} \\ \vdash Q\end{array} \quad \begin{array}{c}\Gamma \cup \{Q\} \\ \vdash Q \to R\end{array}}{\Gamma \cup \{Q\} \vdash R}\,(\to\text{-E})}{\Gamma \vdash R}\,(\vee\text{-E})$

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

5.11 M269 2015J Exam Q 14

- **Question 14** Consider the following axiom schema and rules: **(4 marks)**
- *Axiom schema* $\{\mathbf{A}, \mathbf{B}\} \vdash \mathbf{A}$

- *Rules*
- $\boxed{\dfrac{\Gamma \vdash \mathbf{A} \land \mathbf{B}}{\Gamma \vdash \mathbf{A}}\ (\land\text{-elimination left})}$
- $\boxed{\dfrac{\Gamma \vdash \mathbf{A} \land \mathbf{B}}{\Gamma \vdash \mathbf{B}}\ (\land\text{-elimination right})}$
- $\boxed{\dfrac{\Gamma \vdash \mathbf{A} \quad \Gamma \vdash \mathbf{B}}{\Gamma \vdash \mathbf{A} \land \mathbf{B}}\ (\land\text{-introduction})}$
- $\boxed{\dfrac{\Gamma \cup \{\mathbf{A}\} \vdash \mathbf{B}}{\Gamma \vdash \mathbf{A} \to \mathbf{B}}\ (\to\text{-introduction})}$
- $\boxed{\dfrac{\Gamma \vdash \mathbf{A} \quad \Gamma \vdash \mathbf{A} \to \mathbf{B}}{\Gamma \vdash \mathbf{B}}\ (\to\text{-elimination})}$

Go to Soln 14

- Complete the following proof by filling in the two boxes. You can use any of the above as appropriate.

1.	$\{V, W\} \vdash V$	[Axiom schema]
2.	$\boxed{??\qquad\qquad ??}$	[Axiom schema]
3.	$\{V, W\} \vdash V \land W$	$\boxed{??\qquad\qquad ??}$

Go to Exam Soln 14

5.12 M269 2015J Exam Soln 14

- Completed proof

1.	$\{V, W\} \vdash V$	[Axiom schema]
2.	$\boxed{\{V, W\} \vdash W}$	[Axiom schema]
3.	$\{V, W\} \vdash V \land W$	$\boxed{[\land\text{-introduction}]}$

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 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

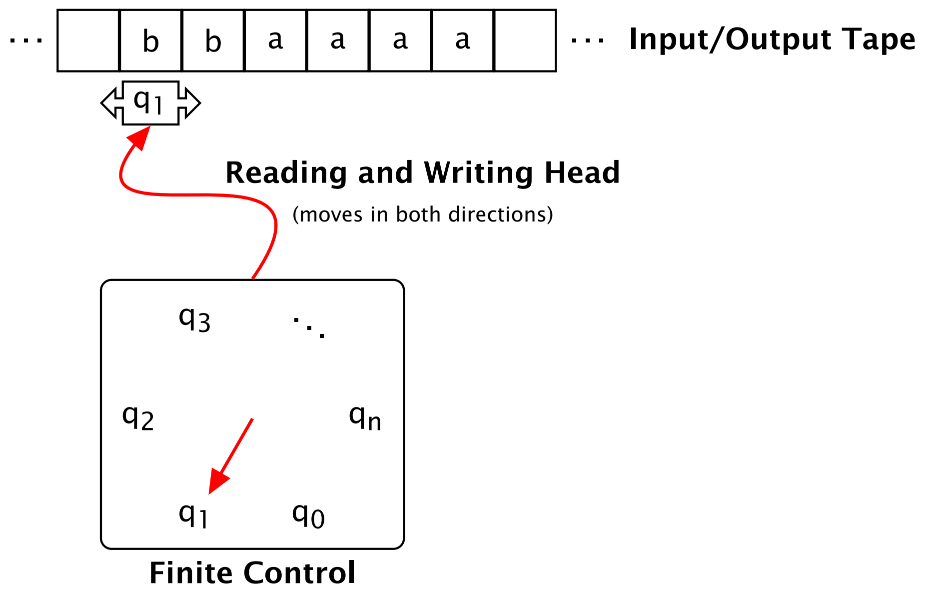
Reference: Section 4 of Unit 6 & 7 Reader

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)
  $\delta :: Q \times \Gamma \to Q \times \Gamma \times \{L, R, S\}$
  $\delta(q, X) \mapsto (p, Y, D)$
- $\delta(q, X)$ 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.

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)

5.14 M269 2015J Exam Q 15

- **Question 15** Which **two** of the following statements are true? (Tick **two** boxes.)

(a) The Halting Problem is semi-decidable.

(b) The Equivalence Problem is computable.

(c) The Church-Turing Thesis proves that all definitions of an algorithm are equivalent.

(d) A reduction from a non-computable problem A to a problem B proves that B is not computable.

- Note that the original exam did not have labels for the boxes

Go to Soln 15

5.15 M269 2015J Exam Soln 15

- **Question 15** Which **two** of the following statements are true? (Tick **two** boxes.)

(a) The Halting Problem is semi-decidable. **True**

(b) The Equivalence Problem is computable. **False**

(c) The Church-Turing Thesis proves that all definitions of an algorithm are equivalent. **False**

(d) A reduction from a non-computable problem A to a problem B proves that B is not computable. **True**

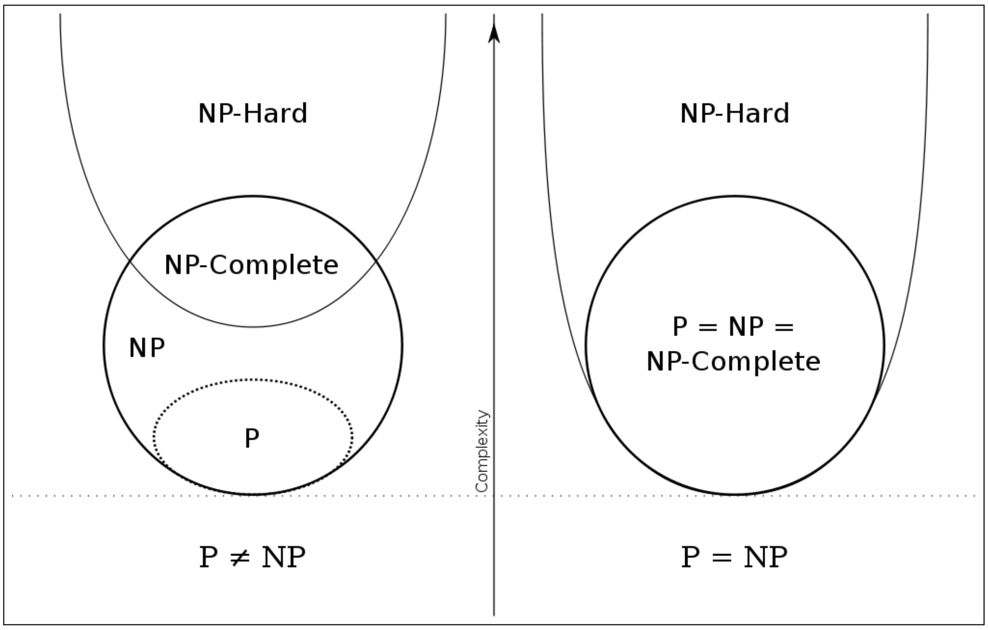
Go to Q 15

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

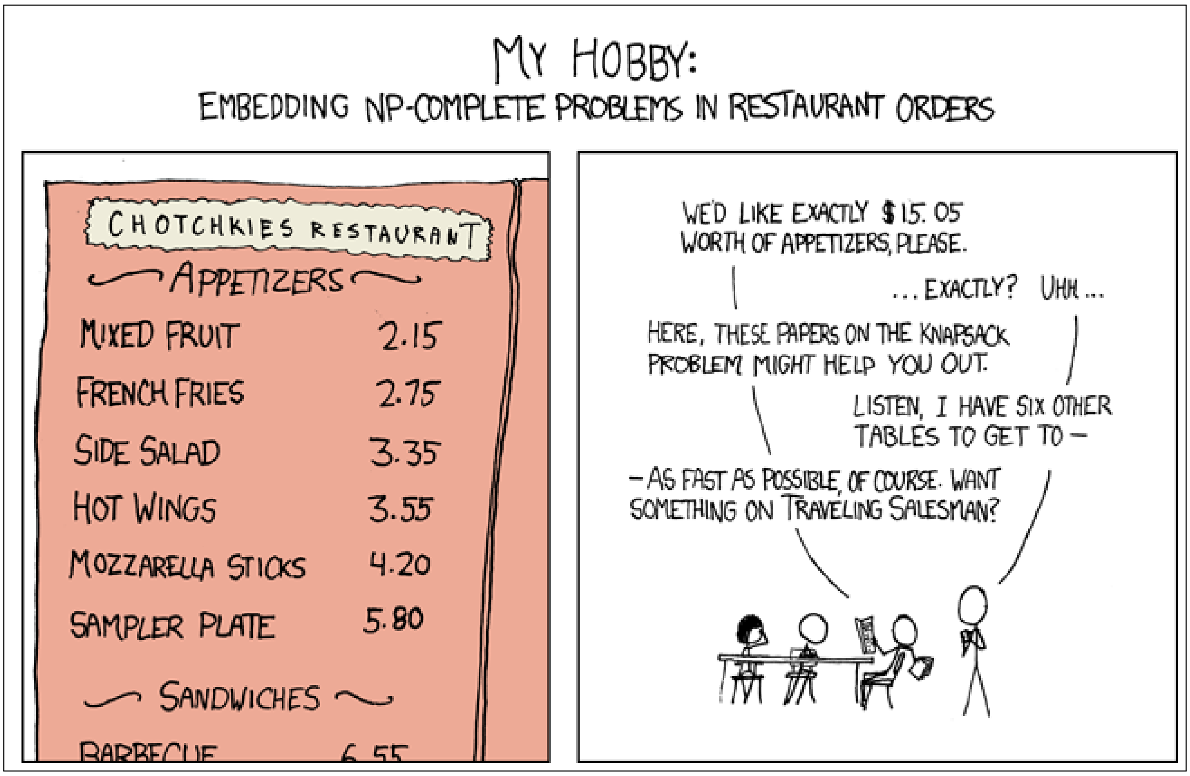


Source: Wikipedia NP-complete entry

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:
- 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 $\exists f : dp_{P_1} \to dp_{P_2}$ 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 $\exists f : \Sigma_1^* \to \Sigma_2^*$ 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 2015J 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 2015J Exam Q 16

- The Universal Product Corporation (UPC) keeps rather primitive computerised records of its sales of a range of world class products.
- These are contained in a sequence S of sales, where each sale records the number sold of a particular product, in the form of `[productCode, numberSold]`.
- The sequence S lists the sales as they were processed, from first to last.
- The sequence has at least one sale. Each product has a different `productCode`. There may be multiple sales for the same product.
- An example sequence S is, in Python notation:

  ```
  [['PR1', 5], ['B20', 10], ['PR1', 3]]
  ```

(a) The company requires a function that returns a sequence of how many sales were processed for each product. For example, the example sequence S given above would lead to an output of either:

`[['PR1', 2], ['B20', 1]]` or

`[['B20', 1], ['PR1', 2]]`

- showing that there are two sales for product `'PR1'` and one for product `'B20'`.
- Using the following template, formally state this as a computational problem, in the style adopted by M269. **(6 marks)**

Name: SalesSummary

Inputs:

Preconditions: *(indicate only **one**)*

Outputs:

Postconditions: *(indicate only **one**)*

(b) UPC want a function that returns the code of the product with the fewest sales processed, so that UPC can start promoting it.

- If the lowest number of sales is shared by several products, the function can return the product code of any one of them.
- A UPC employee has the following initial insight:
  - *Take the sales summary sequence (i.e. the output of SalesSummary) and use QuickSort to sort it in ascending order by the number of sales.*
  - *This will put one of the products with fewest sales in the first position, so then just return the product code of the first element of the sorted sequence.*

(i) What is the order of complexity, in Big-O notation, of the algorithm described by the employee's initial insight, in the best case?

- Assume that SalesSummary has already run.

(ii) Give the initial insight of a more efficient solution and state its order of complexity in Big-O notation. **(6 marks)**

(c) UPC introduce a further data sequence P, which is an unsorted sequence of product prices, such that each item in the sequence is in the form of `[productCode, price]` and each product is included exactly once.

(i) A function is required that will return the total value of all sales for each product.

- So given the sequence S of sales, each in the form `[productCode, numberSold]`:

`[['PR1', 5], ['B20', 10], ['PR1', 3]]`

- the output of the function would be:

`[['B20', 49.9], ['PR1', 28.0]]`

or `[['PR1', 27.5], ['B20', 49.9]]`

- This is because 10 items of product `'B20'` ($10 \times 4.99 = 49.9$) and 8 items of product `'PR1'` ($8 \times 3.50 = 28.0$) were sold.

- Write structured English or Python code for a computational solution of this problem.

(ii) Estimate the run time T() and the order of complexity in Big-O notation of your solution, in the worst case, taking assignment as the unit of computation.

- Explain your reasoning. If you make any assumptions, state them clearly. **(9 marks)**

(d) Storing complex data items (e.g. product/price combinations) in a list, as UPC have opted to do, can be problematic, in particular because retrieval may be slow when there are very large numbers of items. A more suitable means of storage is in a structure such as a *hash table*.

- Write roughly four to six sentences explaining: **(4 marks)**
  - the basic principles of *hash tables*
  - and hashing,
  - how these could work with the UPC price data, and
  - outline one problem that can arise from hashing.

Go to Soln 16

6.2 M269 2015J Exam Q 17

- Your local secondary school runs a computer club for sixth form students.
- You have been asked to give a talk on *greedy algorithms* and, in preparation, to prepare a report for the teachers summarising your talk.
- Assume that the students and teachers do not have a background in computer science, but have been writing programs in various computer languages and are IT literate.
- Your report **must** have the following structure:

1 A suitable title

2 A paragraph *setting the scene*: explain in layperson's terms what is meant by *a greedy algorithm* and give an example of where greed is not always good.

3 A paragraph in which you describe a minimum spanning tree (MST) and give an example of one. You don't need to explain what are trees and graphs.

4 A paragraph in which you briefly describe what is Prim's algorithm and some of its features. You do not need to describe Prim's algorithm completely.

5 A concluding paragraph, giving reasons, about the benefits or otherwise of a greedy algorithm.

- Note that a significant number of marks will be awarded for coherence and clarity, so avoid abrupt changes of topic and make sure your sentences fit together to tell an *overall* story.
- As a guide, you should aim to write roughly three to five sentences per paragraph.

Go to Soln 17

7 M269 Exam 2015J Soln Part2

- Part 2 solutions

Go to Q Part2

7.1 M269 2015J Exam Soln 16

(a) **Name:** SalesSummary

Inputs: An unsorted sequence of tuples $S = (s_1, s_2, \ldots, s_n)$ where $s_n = (p_n, q_n)$ and productCode, p_n is a string, and numberSold, q_n, is an integer.

Preconditions: length $S \geqslant 1$

Outputs: a list of tuples $O = (o_1, o_2, \ldots, o_m)$ where $o_p = (p_m, r_m)$ and p_m is a productCode and r_m is an integer.

Postconditions: Length O equals number of product codes

(b) **(i)** Complexity of Quicksort in the best case is $O(n \log n)$ and worst case is $O(n^2)$ (see Big-O Cheat Sheet)

- **(ii)** A linear search can find the smallest — $O(n)$

(c) **(i)** Sketch of programming strategy

- Sort the sequence of sales using Python's Timsort — worst case complexity $O(n \log n)$
- Group tuples for each product into sub-sequences — one traversal of the sequence $O(n)$
- For each sub-sequence calculate the value of a sale and sum the values — one traversal of each sub-sequence $O(n)$
- **(ii)** Overall complexity $O(n \log n)$

(d) Hash function and hash tables

- Hash function maps each input key to a hash value (or slot)
- Perfect hash function maps each key to a different hash value
- For UPC could translate productCode to an integer by using Unicode or ASCII values for each character
- Limited storage leads to hash functions having collisions — a hash function mapping two keys to the same slot
- Hash function collisions result in the need to either store multiple items in a single slot (closed table) or open addressing/open tables that use some mechanism to find a free slot

Go to Q 16

7.2 M269 2015J Exam Soln 17

- **Title** Greed is (sometimes) good
- Define **Graph** and **Tree** with example
- Define **Minimum Spanning Tree** of a graph is the spanning tree (includes every node but may not include every edge) that minimises total weight of edges
- Describe **Prim's algorithm** — repeatedly add the next *safe* edge — the only safe edge will be the one with the smallest edge from the tree so far
- Greed is (hardly ever) good — give an example where it does not work — knapsack problem.

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

Acknowledgements Toby Thurston for sample answers

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, Zoher (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.