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 2013J exam — Part 1 in reverse order
7. M269 2013J exam — Part 2 in notes version
8. Topics and discussion for each question
9. Exam technique

1.1 Revision strategies

- Organising your knowledge
- Each give one exam tip to the group
- TODO: add some more points

2 Units 6 & 7

2.1 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

2.1.1 M269 Exam 2013J Q 15

- Which *two* of the following statements are true?

A. A Turing Machine is a mathematical model of computational problems.

B. If the lower bound for a computational problem is $O(n^2)$, then there is an algorithm that solves the problem and which has complexity $O(n^2)$.

C. Searching a sorted list is not in the class NP.

D. The decision Travelling Salesperson Problem is NP-complete.

E. There is no known tractable quantum algorithm for solving a known NP-complete problem.

M269 Exam 2013J Q 15 Solution

- Only D and E are true.
- A Universal Turing Machine can compute any *computable* sequence but there are well defined problems that are not computable. (So not A)
- A lower bound may be lower than any actual algorithm. (So not B)
- Every problem in P is in NP — we just do not know if P == NP (So not C)

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

- $\forall x(\mathrm{dp}_{P_1}(x) = \mathrm{dp}_{P_2}(\mathrm{reduce}(x)))$
- 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

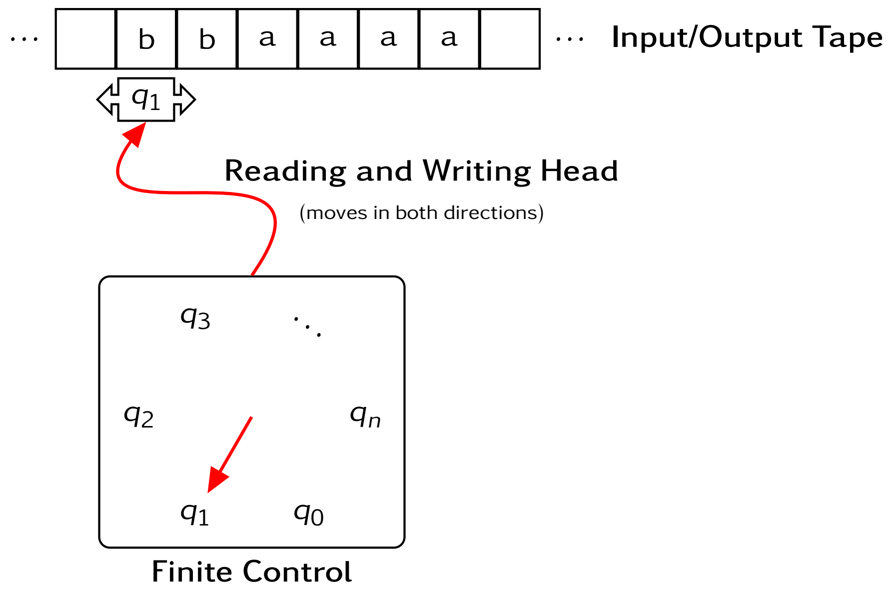
Reference: Section 4 of Unit 6 & 7 Reader

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

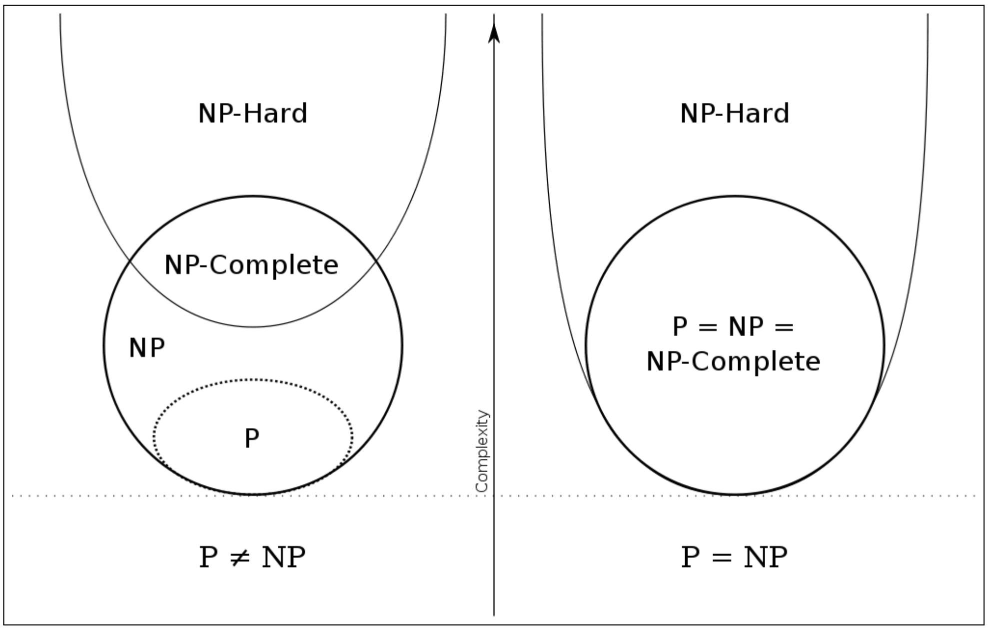
- Soare (1996) on the history of the terms *computable* and *recursive* meaning *calculable*
- See also Soare (2013, sections 9.9–9.15) in Copeland et al. (2013)

2.2 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

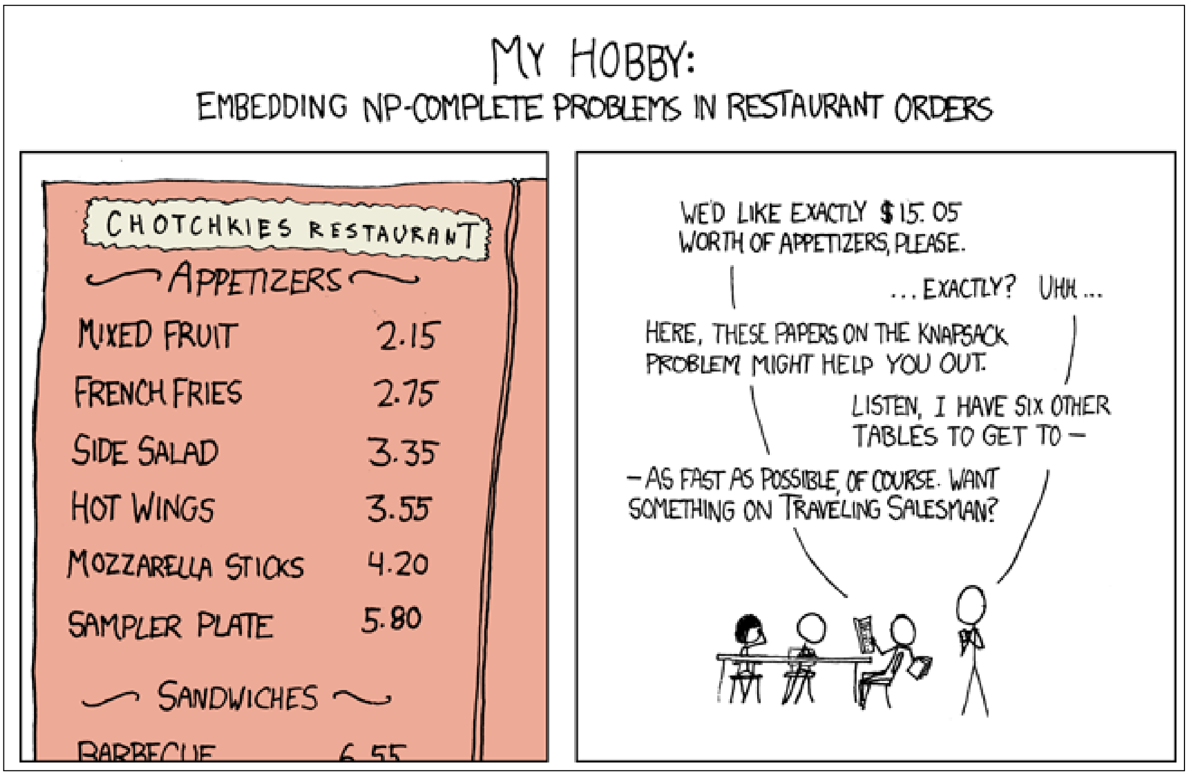


Source: Wikipedia NP-complete entry

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

2.2.1 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 $\mathrm{P} \neq \mathrm{NP}$, 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 $\exists f : \mathrm{dp}_{P_1} \to \mathrm{dp}_{P_2}$ such that
  - $\forall I \in \mathrm{dp}_{P_1} [I \in Y_{P_1} \Leftrightarrow f(I) \in Y_{P_2}]$
  - 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 $\forall L \in \mathrm{NP} : L \propto \mathrm{SAT}$ 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)

2.3 Logic

M269 Exam — Q14 topics

- Unit 7
- Proofs
- Natural deduction

Logicians, Logics, Notations

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

References: Davis (1995, page 289)

Logic and Programming Languages

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

Reference: Halpern et al. (2001)

Validity and Justification

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

Proof Systems, Soundness, Completeness

- Semantic validity and syntactic justification are different ways of modelling the same intuitive property: whether an argument is logically good.
- A proof system is *sound* if any statement we can prove (justify) is also valid (true)
- A proof system is *adequate* if any valid (true) statement has a proof (justification)
- A proof system that is sound and adequate is said to be *complete*
- Propositional and predicate logic are *complete* — arguments that are valid are also justifiable and vice versa
- Unit 7 section 2.4 describes another logic where there are valid arguments that are not justifiable (provable)

Reference: Chiswell and Hodges (2007, page 86)

Valid arguments

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

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

- This can be read as: any formula A can be derived from the assumption (premise) of $\{A\}$ itself
- The $\vdash$ symbol is called the *turnstile* and is often read as *proves*, denoting *syntactic entailment*
- In LaTeX $\vdash$ is produced from `\vdash`
- In Unicode $\vdash$ is called *RIGHT TACK* and is U+22A2, HTML ⊢

See (Thompson, 1991, Chp 1)

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

Inference Rules — Notation

- Inference rule notation:

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

Inference Rules — Conjunction

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

Inference Rules — Implication

- $\dfrac{\Gamma \cup \{A\} \vdash B}{\Gamma \vdash A \rightarrow B}$ ($\rightarrow$-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

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

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

Inference Rules — Disjunction

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

$$\dfrac{\Gamma \vdash A \lor B \quad \Gamma \cup \{A\} \vdash C \quad \Gamma \cup \{B\} \vdash C}{\Gamma \vdash C} \ (\lor\text{-elimination})$$

- The above should be read: if a set of premises Γ justifies the conclusion $A \lor B$ 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 $\{P \land (P \to Q)\} \vdash Q$ in *Question 4* (in the Logic tutorial notes) can be represented by the following diagram:

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

Self-Assessment activity 7.4 — tree layout

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

Self-assessment activity 7.4 — Linear Layout

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

2.3.1 M269 Exam 2013J Q 14

- Consider the following axiom schema and rules:
- *Axiom schema:* $\{\boldsymbol{A}\} \vdash \boldsymbol{A}$
- *Rules: (as Unit 7 for Natural Deduction)*
  - $\land$-elimination left, $\land$-elimination right
  - $\land$-introduction
  - $\to$-introduction, $\to$-elimination
- Complete the following proof:

1.	$\{P \land (Q \land R)\} \vdash P \land (Q \land R)$	[Axiom]
2.	....	[1,$\land$-elimination left]
3.	$\emptyset \vdash (P \land (Q \land R)) \to P$	....

M269 Exam 2013J Q 14 Solution

1.	$\{P \land (Q \land R)\} \vdash P \land (Q \land R)$	[Axiom]
2.	$\{P \land (Q \land R)\} \vdash P$	[1,$\land$-elimination left]
3.	$\emptyset \vdash (P \land (Q \land R)) \to P$	[2,$\to$-introduction]

2.4 SQL Queries

M269 Specimen Exam Q13 Topics

- Unit 6
- SQL queries

2.4.1 M269 Exam 2013J Q 13

- A database contains the following tables, *oilfield* and *operator*

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

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

- For each of the following SQL queries, give the table returned by the query

(a)

```
SELECT *
FROM operator;
```

(b)

```
SELECT name, production
FROM oilfield
WHERE production > 2;
```

(c)

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

M269 Exam 2013J Q 14 Solution

(a) This is simply the whole *operator* table.

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

(b) Retaining only the rows with *production* > 2

name	*production*
Warga	3
Lolli	5
Sugar	3

(c) Joining the tables

name	*production*	*company*
Warga	3	Amarco
Lolli	5	Bratape
Tolstoi	0.5	Rosbif
Dakhun	2	Taqar
Sugar	3	Bratape

2.5 Predicate Logic

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

2.5.1 M269 Exam 2013J Q 12

- A particular interpretation of predicate logic allows facts to be expressed about films that people have seen, and of which they own copies.
- Some of the assignments in the interpretation are given below (where the symbol $\mathcal{I}$ is used to show assignment).
- The interpretation assigns *Jane*, *John* and *Saira* to the constants *jane*, *john* and *saira*.

$$\begin{aligned} \mathcal{I}(\text{jane}) &= \text{Jane} \\ \mathcal{I}(\text{john}) &= \text{John} \\ \mathcal{I}(\text{saira}) &= \text{Saira} \end{aligned}$$

- The predicates *owns* and *has_seen* are assigned to binary relations. The comprehensions of the relations are:
  - $\mathcal{I}$(owns) = {(A,B): the person A owns a copy of film B}
  - $\mathcal{I}$(has_seen) = {(A,B): the person A has seen film B}
- The enumerations of the relations are:

$\mathcal{I}$(owns) = {(Jane, Django), (Jane, Casablanca), (John, Jaws), (John, The Omen), (John, El Topo), (Saira, El Topo), (Saira, Casablanca)}

$\mathcal{I}$(has_seen) = {(Jane, Django), (Jane, Candide), (Jane, Casablanca), (John, The Omen), (John, El Topo), (Saira, Django), (Saira, The Omen)}

- Parts (a) and (b) of this question are on the next page.
- In both parts, you are given a sentence of predicate logic and asked to provide an English translation of the sentence in the box immediately following it.
- 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 variable X, *and show, using the interpretation above, whether it makes the quantified expression TRUE*.

(a) $\forall$X. (owns(saira,X) $\rightarrow$ has_seen(saira,X)) can be translated in English as:

- This sentence is TRUE/FALSE because:

(b) $\exists$X.(has_seen(jane,X) $\land$ owns(jane,X)) can be translated in English as:

- This sentence is TRUE/FALSE because:

M269 Exam 2013J Q 12(a) Solution

(a) *For all films, if Saira owns a copy of the film, then Saira has seen the film.*

Or more idiomatically, *Saira has seen all of the films that she owns.*

- The sentence is *FALSE*, because the enumerations of the relations show that she owns a copy of *Casablanca*, but this is not one of the films that she has seen. She also owns a copy of *El Topo*, which she has not seen either, but we only need one counter-example to show that the sentence is false.

M269 Exam 2013J Q 12(b) Solution

(b) *There exists a film, such that Jane has seen it and Jane owns it.*

Or more idiomatically, *Jane has seen at least one of the films that she owns*

- This sentence is *TRUE*. The enumerations show that she owns *Casablanca* and that she has seen it. *Django* also provides a sufficient example to show that the sentence is true.

2.6 Propositional Logic

M269 Specimen Exam Q11 Topics

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

2.6.1 M269 Exam 2013J Q 11

(a) What does it mean to say that a well-formed formula (WFF) is *satisfiable* ?

(b) Is the following WFF satisfiable ?

$(P \to (Q \to P)) \vee \neg R$

Explain how you arrived at your answer

M269 Exam 2013J Q 11 Solution

(a) A WFF is *satisfiable* if it is possible to find an interpretation that makes the formula true.

(b) Truth table for the WFF

P	Q	R	$Q \to P$	$P \to (Q \to P)$	$\neg R$	$(P \to (Q \to P)) \vee \neg R$
F	F	F	T	T	T	T
F	F	T	T	T	F	T
F	T	F	F	T	T	T
F	T	T	F	T	F	T
T	F	F	T	T	T	T
T	F	T	T	T	F	T
T	T	F	T	T	T	T
T	T	T	T	T	F	T

- The truth table shows that the WFF $(P \to (Q \to P)) \vee \neg R$ is always true, so it satisfiable under any interpretation. But we don't need the whole truth table to prove this; the WFF is true for any interpretation in which R is false (for example).

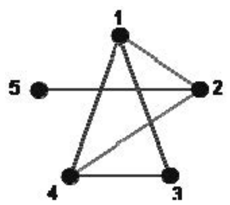
3 Units 3, 4 & 5

3.1 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

3.1.1 M269 Exam 2013J Q 10

- Consider the following graph:



- Complete the table below to show the order in which the vertices of the above graph could be visited in a Depth First Search (DFS) starting at vertex 3 and always choosing first the leftmost not yet visited vertex (as seen from the current vertex):

Vertex	3				

M269 Exam 2013J Q 10 Solution

- Depth First Search (DFS) starting at vertex 3 and always choosing first the leftmost not yet visited vertex (as seen from the current vertex):

Vertex	3	4	1	2	5

- Notice the ambiguity about the term *leftmost* — an alternative view could have been:

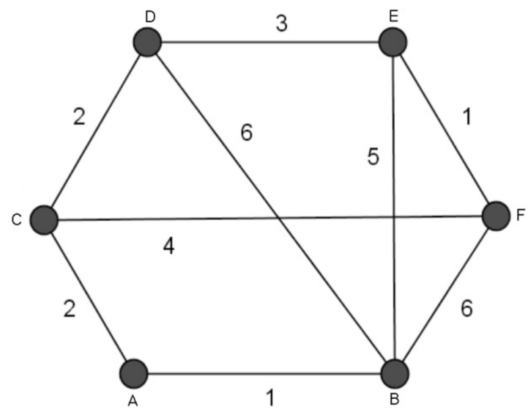
Vertex	3	1	4	2	5

3.1.2 M269 Exam 2013J Q 9

- Recall that the structured English for Dijkstra's algorithm is:

```
create priority~queue
set dist to 0 for v and dist to infinity
  for all other vertices
add all vertices to priority~queue
ITERATE while priority~queue is not empty
  remove u from the front of the queue
  ITERATE over w in the neighbours of u
    set new~distance to
      dist u + length of edge from u to w
    IF new~distance is less than dist w
      set dist w to new~distance
      change priority(w, new~distance)
```

- Now consider the following weighted graph:



- **Starting from vertex B**, the following table represents the distances after the second line of structured English is executed for the graph given above (using the convention that a blank cell represents infinity):

Vertex	A	B	C	D	E	F
Distance		0				

- *Note that neither the table above nor the subsequent tables represent the priority queue.*
- Now, complete the appropriate boxes in the next table to show the distances after the first and second iterations of the while loop of the algorithm.

Vertex	A	B	C	D	E	F	
Distance		0					First iteration
Distance		0					Second iteration

M269 Exam 2013J Q 9 Solution

- The completed table:

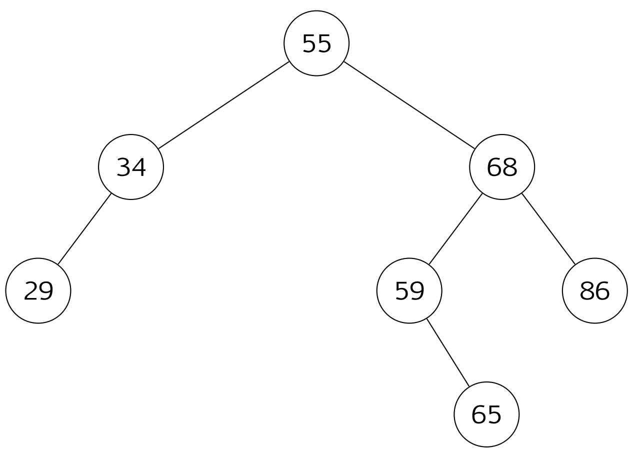
Vertex	A	B	C	D	E	F	
Distance	1	0		6	5	6	First iteration
Distance	1	0	3	6	5	6	Second iteration

3.2 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

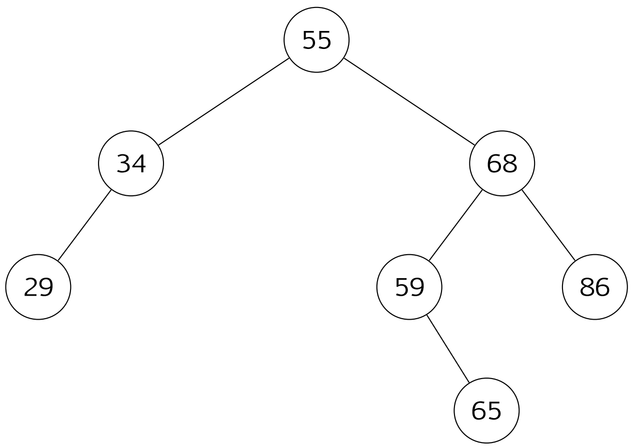
3.2.1 M269 Exam 2013J Q 8

(a) Consider the following Binary Search Tree.



- Modify (draw on) the above Binary Search Tree to insert a node with a key of 57.

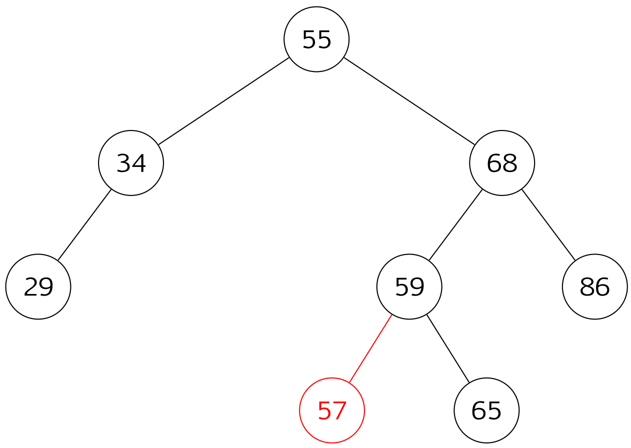
(b) Once again, consider the same Binary Search Tree.



- Calculate the balance factors of each node in the tree above and modify the diagram to show these balance factors.

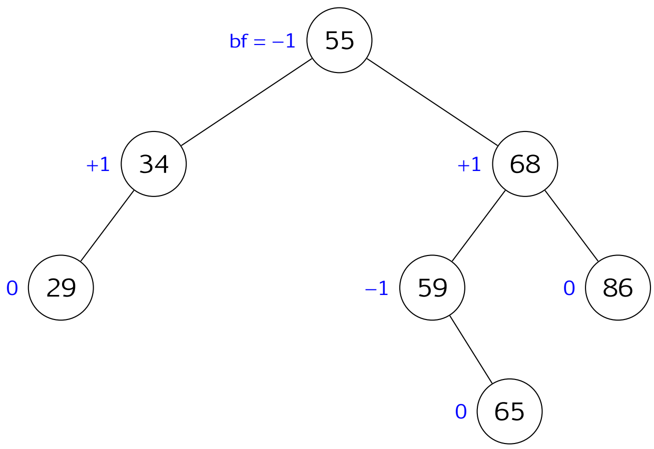
M269 Exam 2013J Q 8(a) Solution

(a) Answer, with inserted node shown in red



M269 Exam 2013J Q 8(b) Solution

- Answer, with balance factors shown in blue



3.2.2 M269 Exam 2013J Q 7

- In the KMP algorithm, for each character in the target string T we identify the longest substring of T ending with that character which matches a prefix of the target string.
- 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

A	B	A	C	A	B	A	C	A	C

- 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.

0		1	0		2		4		0

M269 Exam 2013J Q 7 Solution

- The complete prefix table, with new entries in red:

0	0	1	0	1	2	3	4	5	0

- Here is the target, prefix and shift:

	A	B	A	C	A	B	A	C	A	C	Target string, t
	0	1	2	3	4	5	6	7	8	9	Position (Index), p
0	1	2	3	4	5	6	7	8	9	10	Match, q
	0	0	1	0	1	2	3	4	5	0	$\text{prefixTable}(t, p)$
1	1	2	2	4	4	4	4	4	4	10	$\text{shift}(t, q)$

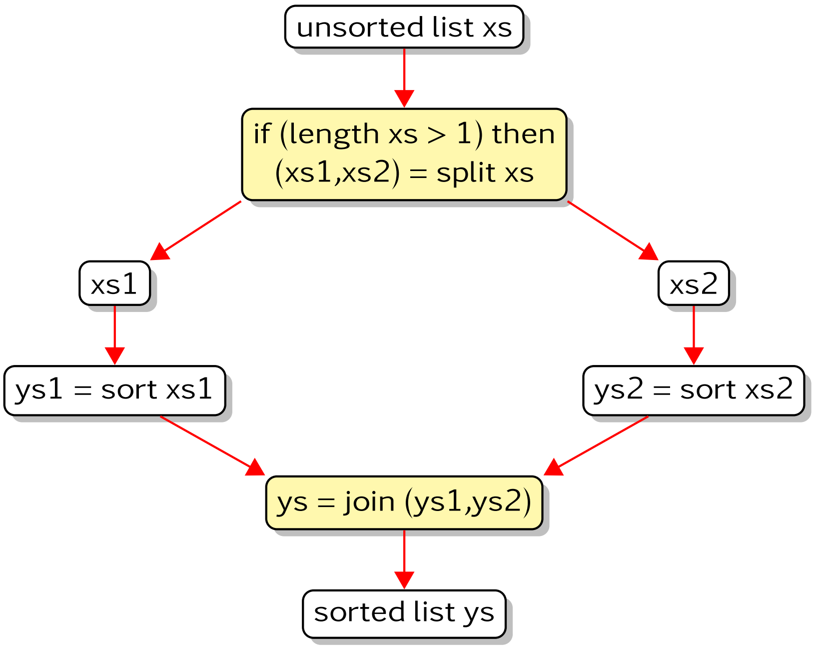
- The shift function takes the *target string, t,* and the number of characters matched, q.
- $\text{shift}(t, 0) = 1$
- $\text{shift}(t, q) = q - \text{prefixTable}(t, q - 1)$

3.3 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)

3.3.1 M269 Exam 2013J Q 6

- Consider the following function, which takes a list as an argument. You may assume that the list contains a number of integer values and is not empty.

```python
def average(aList):
    n = len(aList)
    total = 0
    for item in aList:
        total = total + item
    mean = total / n
return mean
```

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

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

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

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

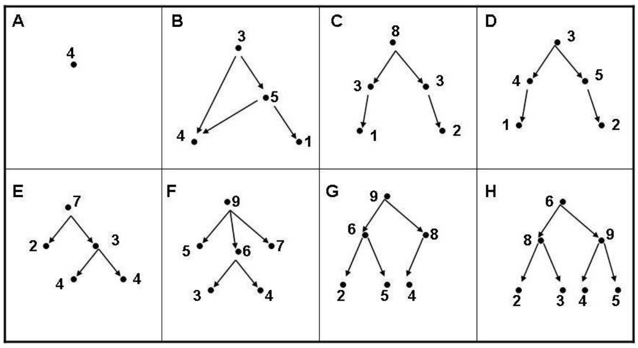
E. $T(n) = n + 3$ and $O(n)$

M269 Exam 2013J Q 6 Solution

- Option E is correct.
- The function does three assignments once per call, and one assignment for each of the n items in the argument, hence $T(n) = n + 3$.

3.3.2 M269 Exam 2013J Q 5

- Consider the following diagrams A–H. Nodes are represented by black dots and edges by arrows. The numbers represent a node's key.



- 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 heaps ?

M269 Exam 2013J Q 5 Solution

(a) B is not a tree; it has more than one route from node 3 to node 4.

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

4 Units 1 & 2

4.1 Unit 2 From Problems to Programs

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

4.1.1 M269 Exam 2013J Q 4

- Consider the guard in the following Python while loop header:

```
while (a < 6 and b > 8) or not(a >= 6 or b <= 8):
```

(a) Make the following substitutions:

P represents `a < 6`

Q represents `b > 8`

Then complete the following truth table:

P	Q	$\neg P$	$\neg Q$	$P \wedge Q$	$\neg P \vee \neg Q$	$\neg(\neg P \vee \neg Q)$	$(P \wedge Q) \vee \neg(\neg P \vee \neg Q)$
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. `(a < 6 and b > 8)`

B. `not(a < 6 and b > 8)`

C. `(a >= 6 or b <= 8)`

D. `(a >= 6 and b <= 8)`

E. `(a < 6 and b <= 8)`

M269 Exam 2013J Q 4 Solution

(a) The completed truth table:

P	Q	$\neg P$	$\neg Q$	$P \wedge Q$	$\neg P \vee \neg Q$	$\neg(\neg P \vee \neg Q)$	$(P \wedge Q) \vee \neg(\neg P \vee \neg Q)$
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) A is the simplest equivalent of the guard given.

4.1.2 M269 Exam 2013J Q 3

- A binary search is being carried out on the list shown below for item 67:

  [12,16,17,24,41,49,51,62,67,69,75,80,89,97,101]

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

(Pass 1) [$\boxed{\texttt{12,16,17,24,41,49,51,62,67,69,75,80,89,97,101}}$]
(Pass 2) [12,16,17,24,41,49,51,62,67,69,75,80,89,97,101]
(Pass 3) [12,16,17,24,41,49,51,62,67,69,75,80,89,97,101]

M269 Exam 2013J Q 3 Solution

- The complete binary search:

(Pass 1) [$\boxed{\texttt{12,16,17,24,41,49,51,62,67,69,75,80,89,97,101}}$]
(Pass 2) [12,16,17,24,41,49,51,62, $\boxed{\texttt{67,69,75,80,89,97,101}}$]
(Pass 3) [12,16,17,24,41,49,51,62, $\boxed{\texttt{67,69,75}}$,80,89,97,101]
(Pass 4) [12,16,17,24,41,49,51,62, $\boxed{\texttt{67}}$,69,75,80,89,97,101]

4.1.3 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

```python
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

```python
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

```
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)
```

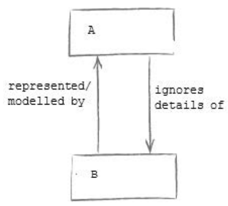
Miller and Ranum (2011, page 193)

4.2 Unit 1 Introduction

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

4.2.1 M269 Exam 2013J Q 2

- The general idea of abstraction as modelling can be shown with the following diagram.



- Complete the diagram above by adding an appropriate label (one of the numbers 1 to 4) in the space indicated by **A** and one in the space indicated by **B**. The possible answers are shown as 1 to 4 below. *The exam question had some pictures next to the texts*
    1. A car crash test dummy in the real world

2. An action man doll in the real world
3. A real car in the real world (after crashing)
4. A real driver in the real world

M269 Exam 2013J Q 2 Solution

- A real driver is modelled by a car crash test dummy, so A = 1 and B = 4

4.2.2 M269 Exam 2013J Q 1

- Which **two** of the following statements are **true**?

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

B. A computational problem is a problem that is expressed sufficiently precisely that it is possible to build an algorithm that will solve all instances of that problem.

C. An algorithm consists of a precisely stated, step-by-step list of instructions.

D. Computational thinking is the skill to formulate a problem as a computational problem, and then construct a good computational solution, in the form of an algorithm, to solve this problem, or explain why there is no such solution.

M269 Exam 2013J Q 1 Solution

- Options C and D are true.
- Option A is wrong because decision problems have to have a yes-no answer.
- Option B is wrong because there are computational problems that we can state and build algorithms for, but cannot always be solved.

5 M269 Exam Section 2

5.1 M269 Exam 2013J Q 16

- Multipart question
- Specification of program, data structures, pre and post conditions
- Write a small program
- Give the complexity of the small program
- Give insight into a sorting algorithm
- Give insight into insertion into a binary search tree
- See notes version for text

5.1.1 M269 Exam 2013J Q 16 Text

The Widget & Widget Widget Corporation (W&WWC) keeps records of every client that has purchased widgets from them, along with details of the value of every purchase. These records are stored on a computer in two sequences; the first of these, CLIENTS, is an (unsorted) sequence of the clients′ names; the second, SPENDS, contains a sequence of sequences, with each item representing the sequence of values of each of the purchases that a client has made. The index of a client in CLIENTS is the index of that client's sequence of purchases in SPENDS.

(a) W&WC requires a small computer program which will provide them with two facilities:

- one to return the average spend of a specified client, if that client is present in the data, and to return a suitable value if the client is not present;
- the other to return a sequence containing for each client in CLIENTS their average spend.

(i) Express both as a computational problem by completing the templates below (in your answer book). Make whatever decisions you think appropriate about the exact form of the input and output.

Name: `SpecifiedClientAverageSpend`

Inputs:

Outputs:

Name: `EachClientAverageSpend`

Inputs:

Inputs:

(ii) In addition, suggest one possible postcondition for the `SpecifiedClientAverageSpend` problem.

(iii) Provide the structured English for the `EachClientAverageSpend` algorithm. (If you wish you can write a Python function instead, but not both.)

(b) What will be the complexity, expressed in $T(n, q)$ and Big-O format, of your `EachClientAverageSpend` solution, assuming n clients with an average of q transactions each, and that the assignment statement is the unit of computation? Explain your reasoning.

(c) One of the drawbacks of the current way in which the data is stored is that the sequence of clients is not sorted. One of the best ways of sorting a sequence is the *Quicksort* algorithm. Express your understanding of this algorithm for in-place sorting in the form of an initial insight.

(d) Having developed the current program as far as they can using sequences, managers have now made the decision to store clients and transactions in a *Binary Search Tree* (BST).

M269 Exam 2013J Q 16 Sample Solution

(a)

(i)

Name: `SpecifiedClientAverageSpend`

Inputs: Client name as String

Outputs: If client is found: Average spend for client as Real else: None

Name: `EachClientAverageSpend`

Inputs: None

Inputs: List of average spends for each client as list of real

(ii) Returned average is a sensible size?

(iii)

```python
def EachClientAverageSpend():
    averages = list()
    for purchases in SPENDS:
        if len(purchases) > 0:
            average = sum(purchases)/len(purchases)
        else:
            average = None
        averages.append(average)
    return averages
```

(b) Assuming n clients with q transactions, and assuming (a) that the Python function `sum` does not create any extra assignments, and (b) that the Python `append` method counts as one assignment, we have one assignment to create the empty list of averages, one to create the average for each customer, and one more to append the average to the list, so $T(n,q) = 1 + 2n$.

On the other hand we might assume that `sum(purchases)` counted as q assignments, and in this case we would have $T(n,q) = 1 + 2qn$. In either case n remains the dominant term because we are doing something separate for each of the n customers, and not trying to do something that compares them to each other (for example), so the complexity is $O(n)$.

(c) *Quicksort* works by divide-and-conquer. The input is a list of values to sort. First we pick a pivot value; there are various ways to do this, but the simplest is just to pick the first value in the list. Then we divide the list into three parts: all those items less than the pivot value, the pivot item itself, and all those items greater than or equal to the pivot value. The output is a list composed of the lower part (sorted by calling ourselves recursively), followed by the pivot value, followed by the upper part (again sorted by calling ourselves recursively).

(d) The two defining features of a binary search tree are (1) that, for each node, all the keys in the left subtree are less than the key of the node, while all the keys in the right subtree are greater than the key of the node; and (2) that each key value is present only once.

An insertion algorithm based on these insights. Given the root node of a binary search tree, and a key value: if the node is undefined (ie it's a leaf node), then create a new node with the provided key value and a null left child and a null right child, and return this node; if the node is defined and the key value equals the key of the node then return a value to show the key is already present.

Otherwise if the key value is greater than the value of this node's key, then call ourselves recursively on this node's right child, with the same key, and return the result.

Finally if the key value is less than the value of this node's key, then call ourselves recursively on this node's left child, with the same key, and return the result.

5.2 M269 Exam 2013J Q 17

- Write short report on a computational topic
- Suitable title for the topic and audience
- Paragraph setting the scene — the context of the topic
- Paragraph describing the topic
- Paragraph on the role the topic plays in some area
- Conclusions justifying the importance of the topic
- See notes version for text.

5.2.1 M269 Exam 2013J Q 17 Text

Imagine that you are a potential speaker for your local University of the Third Age (U3A). Along with other potential speakers you've been asked to write a short report on a particular topic. The organising group will then look at these reports and choose which potential speaker to ask for a full evening presentation. Your topic is *The Turing Machine*. Write a short report. Your report must have the following structure:

1. A suitable title.
2. A paragraph *setting the scene* and explaining the historical importance of the deterministic Turing Machine *[about three sentences]*.
3. One paragraph in which you describe in layperson's terms what a deterministic Turing Machine is *[about three sentences]*.
4. One paragraph in which you describe the role that Turing Machines play in Turing's proof that there are computational problems that are not computable *[about three sentences]*.
5. A conclusion in which you give a reasoned conclusion about the importance of Turing Machines *[one sentence]*.

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. Allow up to four additional sentences to ensure this.

M269 Exam 2013J Q 17 Sample Solution

Alan Turing and his Marvellous Machine

Alan Turing who was one of the mathematical and computing heroes of Bletchley Park. Many U3A members may have visited the famous WW2 code-breaking centre at Bletchley Park — some of them may even have worked there — and will know that Turing was closely involved with building the first electronic computers and using them to decipher enemy radio messages. Members may also be interested to know that Turing had been working before the war in Cambridge on the fundamental ideas of computing and can be regarded as one of the founders of modern computer science.

Turing was interested in what humans do when they compute the answer to a problem, and whether this process could be done by a machine, and if it could, what would be be the limits of what such a machine could do. He imagined a simple, idealized machine that could read and write and erase symbols on an endless paper tape, and that could be set up to follow instructions by responding in a particular way to each different symbol. He showed that given enough time and a long enough tape, such a simplified machine could be set up to add up, or take away, or do arbitrarily complex mathematics. These imaginary machines became known as Turing Machines.

Turing's design was a purely theoretical one. But he used it to prove one of the most important practical results about computers: no matter how fast and how efficient we make our computers, there will always be problems that the computer cannot solve. His argument was based on the idea that if you made a long list of all possible Turing Machines that solve a particular type problem, it is always possible to construct another problem that cannot be solved by any possible machine. This fundamental result still shapes the way that computer scientists search for algorithms and develop programs to solve problems.

The real beauty of the Turing Machine is that it is so simple; this simplicity allows computer scientists to reason about computing and what can be computed without the distracting details of any particular real-world machine.

6 Exam Techniques

- Surviving in a time of great stress
- Each give one exam tip to the group
- TODO: add some more points

7 Web References

TODO: More Web links

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

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.