**Table of Contents**

1. **CA320 Intro**
   1. *Point of the Course*
   2. *Languages and Automata*
   3. *General Models of Computation*
   4. *Computability Questions*
   5. *Complexity Questions*
   6. *Problems*
   7. *Algorithms*
   8. *Correctness*
2. **Mathematical Prerequisites**
   1. *Sets*
   2. *Notation*
   3. *Common Universal Sets*
   4. *Sequences and Tuples*
   5. *Relations and Functions*
   6. *Injections, Bijections and Surjections*
   7. *Equivalence Relations*
   8. *Graphs*
   9. *Strings and Languages*
   10. *Operations on Languages*
   11. *Logic*
3. **Regular Languages**
   1. *Languages*
   2. *Regular Expressions*
   3. *Finite State Automata*
   4. *Deterministic Finite Automata*
   5. *Nondeterministic Finite Automata*
   6. *Equivalence of DFA and NDA*
   7. *Regular Grammars*
   8. *Grammar Theorem*
4. **Context-Free Languages**
   1. *Context Free Grammars*
   2. *Parse Trees*
   3. *Pushdown Automaton*
   4. *PDA Definition*
   5. *Deterministic PDA*
   6. *PDA Format*
   7. *Example PDA Functions*
5. **Context-Sensitive Languages**
   1. *Context-Sensitive Grammars*
   2. *Linear Bounded Automaton*
   3. *LBA Definition*

**CA320 Intro**

**Computability & Complexity**

*CA = 7/25*

*Exam =33/75 = 44% of written -> at least 5 questions fully correct*

Point of the course

* What problems can be *theoretically* solved by a computer
* What problems can be *efficiently* solved a computer

**Computability theory** explores the limitations of computing devices and problems the compute

**Complexity theory** classifies computable problems by their inherent difficulty

Languages and Automata

* **Computation** is regarded as a *language recognition*
* Recognition of a word in a language corresponds to acceptance of input by a program
* **Languages** can have increasing descriptive power
* **Automata** recognise languages and with increasing powers
* Some automata are as powerful as physical computers but also less powerful automata are still quite useful

General models of Computation

* + **Turing Machines**
    - *Mathematical model of computation that defines an abstract machine which manipulates symbols on a strip of tape according to a table of rules. Given any computer algorithm, a Turing machine can be constructed to interpret it.*
  + **Lambda Calculus**
    - *A formal system in mathematical logic for expressing computation based on function abstraction and application using variable binding and substitution. It is Turing complete so that it can simulate any Turing Machine.*
  + **Unrestricted Grammars**
    - *An unrestricted grammar is a formal grammar on which no restrictions are made on the left and right sides of its tree productions*
  + **Partial recursive functions**
    - *These are injective functions where the exact domain X is not known, of which many are in computability theory.*

Computability Questions

* Do algorithmic solutions to problems always exist?
* What are limitations of computational devices?
* Is there any insight to which problems are algorithmically solvable and which are not?
* Are the unsolvable problems somehow related?

Complexity Questions

* How do we compare the efficiency of different algorithms?
* How do we measure time/memory requirements?
* What problems are efficienly solvable?
* Are there solvable problems which do not have fficent algorithms?

Problems

* **Decision Problems:** Return *Yes/No*, *True/False*, *1/0*
* **Search Problems:** Find *x* that satisfies property *P*
* **Enumeration problems:** Find all *x* that satisfy property *P*
* **Counting Problems:** Count the number of *x*’s that satisfy property *P*
* **Optimisation Problems:** Find the *x* that best satisfies property *P*
* **Structuring Problems:** Transfom *x* to satisfy property P

Problem Specifications comprise:

* A characterisation of all legal inputs to the problem.
* A characterisation of the desired outputs as a function of the legal inputs.

Algorithms

* Algorithms solve problems
* A finite sequence of operations for solving some problem
* Each operation is chosen from a set of well-defined operations
* If the algorithm is executed by a suitable processor on any instance of the problem, it will give rise to a process that:
  + Halts in a finite time
  + Returns the answer for that problem instance

Correctness

* How can we be sure that an algorithm or program solves a problem?
* How can we be sure it terminates and produces the correct output for all its legal inputs?

**Mathematical Prerequisites**

Sets

A set is a collection of groups or objects or elements or members

* A set *contains* elements
* There must be a universal set *U* which contains all data including nulls.

Notation

* The elements are listed between braces: *S = {a, b, c, d}*
* Listing an object more tan onve does not change the set, order means nothing
* Predicate sets are listed as: *S = {x | P(x)}*
  + *S* contains all the elements from *U* which make *P* true

Common Universal Sets

* **R** -> Reals
* **N** -> Natural numbers
* **Z** -> Integers
* **Q** -> Rationals

Sequences and Tuples

* A sequence is a list of objects in some order
* Finite sequences are called tuples
* A sequence with *k* elements is a *k* tuple
* Tuples with 2 are called ordered pairs
* The cartesian product denoted by *A x B* is the set of ordered pairs

Relations and Functions

* A relation *R* on a set of ordered pairs *A x B* where A is the domain of R and B is the codomain of R
* If x is an element of A and y is an element of B then *xRy* is true.

Injections, Bijections, Surjections

* F is *one to one* or ***injective*** if every elements of A is related to an element in B
* F is *onto* or ***surjective*** if an element in A matches to one or more elements in B
* F is ***bijective*** if F is injective and surjective

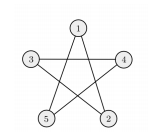
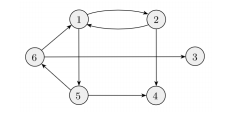
Equivalence Relations

A relation R on a set A is an equivalence relation I it satisfies the three properties

1. R is reflexive -> For all x in A each elements relates to itself
2. R is symmetric -> For all x in A each element is congruent/equal
3. R is transitive -> For all x ,y,z if *{x,y}* & {y,z} then {x,z} = true

Graphs

* A graph is a set of points with lines connecting points.
* The points are nodes/vertices and the lines are edges.
* The number of edges for a node is referred to as a degree
* A *directed graph* has arrows instead of lines



Strings and Languages

* An alphabet is a finite nonempty list of symbols.
* Strings are a finite sequence of symbols
* The length of a string |S| is the number of symbols in it.
* The empty string is of length zero
* Languages are a subset of strings that can contain only certain symbols

Operations on Languages

* Union of L and M is L U M
* Concatenation of L and M is LM
* Kleene Closure of L is the infinite set of possible lengths of symbols in L
* Positive Closure of L is the infinite set of possible length of multiple symbols in L

Logic

* Rules are as follows:
  + Conjunction P and Q
  + Disjunction P or Q
  + Negation Not P
  + Conditional If P then Q
  + Biconditional P if and only if Q
* Propositions can be formed from the logical operators
* A tautology is a compound proposition that is always true for every combination
* A contradiction is a compound proposition that is always false for every combination

**Regular Languages**

Languages

* Given an alphabet S, a language is a subset of *S\**
* A regular language is a formal language that can be expressed by a regular expression
* Regular Languages are recognisable by finite automata
* Empty languages are regular languages
* For each element a the singleton {a} is a regular language
* If A and B are regular languages then *A U B*, *AB* and *A\** are regular languages

Regular Expressions

* If R and S are regular expressions then *(R), R|S, RS, R\** are regular expressions
* The Order of precedence is
  1. *Parentheses*
  2. *Closure*
  3. *Concatenation*
  4. *Alternation*
* Each regular expression represents a regular language, and every language is represented by a regular expression
* Given a regex R, L(R) is the language representing R

Regex Properties

* R\* = R\*R\* = (R\*)\* = R|R\*
* R(SR)\* = (RS)\*R
* (R\*S)\* = null|(R|S)\*S
* (RS\*)\* = null|R(R|S)\*

Let Sigma = {a,b}

1. a|b denotes {a, b}
2. (a|b)(a|b) denotes {aa, ab, ba, bb}
3. a\* denotes {nul, a, aa, aaa, ….}
4. (a|b)\* denotes the set of all strings of a and b
5. a|a\*b denotes {a, b, ab, aab, aab, aaab, …}

Finite State Automata

* Regular languages and regular expressions describe a certain class of languages.
* FSA’s determine if a string is a member of a particular language
* Finite State Automata have a restricted model of computation:
  1. *No stored program*
  2. *No auxiliary memory*
  3. *Input is a string on a tape*
  4. *Read header moves over the string*
  5. *Output is ‘Accept’ or ‘Not Accept’*
* FSA operation is fully determined by input
* CPU is a finite collection of states
* ***Kleenes Theorem:*** A language is regular if and only if it is accepted by a finite state automaton

Deterministic Finite Automata

5 Tuple M = (Q, Sigma, Delta, q0, F)

* Q is a finite set of states
* Sigma is an alphabet, a finite set of symbols
* Delta is the transition function
* q0 is the initial state
* F is the set of final states

Nondeterministic Finite Automata

5 Tuple M = (Q, Sigma, Delta, q0, F)

* Q is a finite set of states
* Sigma is an alphabet
* Delta is the transition function
* q0 is the initial state
* F is the set of final states

***Nondeterminism*** allows a choice of more than one state for any given input symbol

DFA is a special NFA whereby

* No transition is labelled as null
* For each state q and input symbol a, there is at most one transition a leaving q

Equivalence of DFA and NFA

1. DFA is a subset of NFA
2. For each NFA there is an equivalent DFA
3. An NFA can be converted into a DFA by simulating a set of simultaneous states
4. Each DFA state corresponds to a set of NFA states

Regular Grammars

A grammar g is defined as a 4 tuple (S, N, T, P) where:

* S is the start symbol
* N is the set of non-terminal symbols
* T is the set of terminal symbols
* P is a set of productions or rewrite rules

*Terminal symbols* are basic symbols which belong the input alphabet

*Non-terminal symbols* are defined by rules as a concatenation of terminals and non terminals

*Start symbol* is a non-terminal which is used as a starting point for language definition

Grammar Theorem

A restriction can be placed on the form of a grammar to ensure that it describes a regular language.

For any regex *r*, there is a grammar g such that *L (r) = L (g)*

The grammars that generate regular sets are called regular grammars

In a regular grammars all productions have one of two forms:

1. A -> aB
2. A -> a

*A and B are any non-terminals and a is a terminal symbol*

Every regular grammar can be converted into a finite state automaton and vice-versa

To create an FSA to a regular grammar:

1. The input symbols are the terminal symbols of the grammar
2. The states in the FSA are the non-terminals of the grammar such that:
   * + The start state is the state corresponding to the start symbol
     + The only final state is F
3. The transitions of the FSA correspond to the production rules as follows:
   * + If the rule is of form A -> aB create a new transition of the form *Delta(A, a) = B*
     + If the rule is of the form A -> a create a new transition of the form *Delta(A, a) = F*

**Context-Free Languages**

Context-Free Grammars

Context-free languages are specified with a context-free grammar.

Context Free Grammars are a 4 tuple (T, N, S, P)

* T is the set of terminal symbols in the grammar
* N is the set of non-terminals, a set of syntactic variables that denote sets of strings occurring in the language
* S is a distinguished nonterminal denoting the entire set of strings in L(G) called the start symbol
* P is a finite set of productions specifying how terminals and non-terminals can be combined to form strings in the language.
* The set V = T UNION N is called the vocabulary of G

A ***derivation*** for a string is a series of productions that lead from the start symbol to the string

The process of discovering a ***derivation*** is called ***parsing***

At each step we choose a non-terminal to replace

Two of primary interest are:

* Leftmost derivation: The leftmost non-terminal is replaced
* Rightmost derivation: the rightmost non-terminal is replaced

Parse Trees

Derivation results can be expressed by using a parse tree:

* The *root* node is the start symbol of the grammar
* Each internal node is a non-terminal with the symbols from one production rule as it children
* The leaves of the tree are terminal symbols which when read from left to right produce the string derivation

A grammar is known as an ambiguous grammar if its language contains some string that has multiple parse trees

Pushdown Automaton

Not all context-free languages can be recognised by an FSA

We can extend the FSA concept to create FSA-like machines to recognise a context-free language

This can be done with stack concept

An NFA with stack memory is known as a ***Pushdown Automaton*** or PDA

For any CFG rule, a pushdown automaton can imitate it by reading the start state, putting the value on the stack and going the next state

The PDA can decide to pop values off the stack

A language is context-free if it can be recognised by pushdown automata

PDA Definition

A pushdown automaton is a 6 tuple M = (Q, Sigma, Gamma, q0, F, Delta)

* Q is the finite set of states
* Sigma is an alphabet of input symbols
* Gamma is an alphabet of stack symbols
* q0 is the initial state
* F is the set of final states
* Delta is the transition function

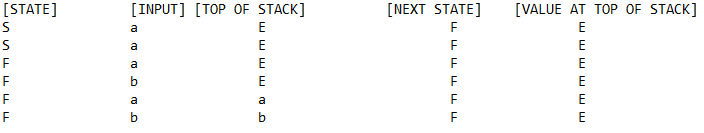
For an input of symbols, c is a special symbol to mark the bottom of the stack

Deterministic PDA

A PDA is deterministic if we can always decide which transition to use next

Not every non-deterministic PDA can be converted into an equivalent deterministic PDA

PDA Format



Example PDA Functions

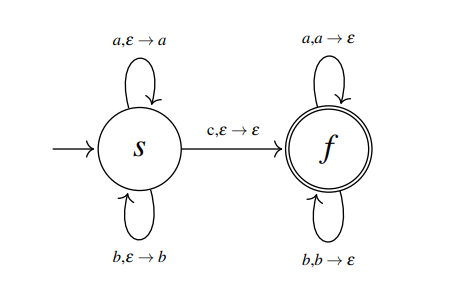
L = {wcwr: w IN {a, b}\*} **palindromes *of odd length***

Q = {s, f}

Sigma = {a, b, c}

Gamma = {a, b}

q0 = s

F = {f}

Delta = ((s, a, E), (s, a))

((s, a, E), (s, b))

((s, c, E), (s, b))

((f, a, a), (f, E))

((f, b, b), (f, E))

*The notation* ***a, E -> a*** *denotes reading symbol a, and replacing stack top E with a*

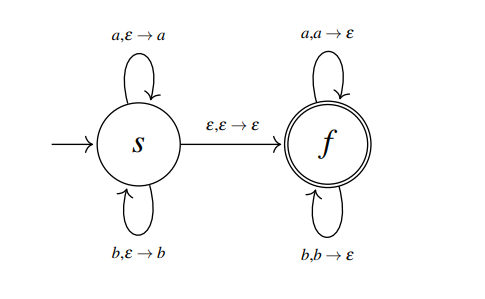
L = {wwr: w IN {a, b}\*} ***palindromes of even length***

Q = {s, f}

Sigma = {a, b}

Gamma = {a, b}

q0 = s

F = {f}

Delta = ((s, a, E), (s, a))

((s, b, E), (s, b))

((s, E, E), (f, E))

((f, a, a), (f, E))

((f, b, b), (f, E))

**Context Sensitive Languages**

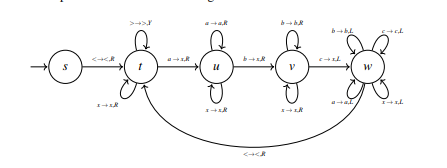
Context-Sensitive Grammars

Context-Sensitive Languages are specified with a context-sensitive grammar. Each product has the form alpha implies beta where |beta| >= |alpha|

Linear Bounded Automata

* An LBA is an automaton which has a finite length store called a tape. Characters are read and written at any position on this tape
* Its read-write head can shift left and right.
* The same tape is used for input and store. Special symbols < and > are used to mark the finite bounds of the tape beyond which the read-write head cannot move
* LBA will perform one of four actions *{Y, N, L, R}:*
  + Y denotes yes, accept the input string
  + N denotes no, don’t accept the input string
  + L denotes Left, move read-write head one space left
  + R denotes Right, move read-write head one space right
* A language is context-sensitive if it can be recognised by an LBA

LBA Definition

Linear Bounded Automata are a 5 tuple M = (Q, Sigma, Gamma, q0, Delta)

* Q is a finite set of states
* Sigma is an alphabet of input symbols
* Gamma is an alphabet of store symbols
* q0 is the initial state
* Delta is the transition function

*The notation* ***x -> y, &*** *denotes reading symbol x, writing symbol y and performing action &*

*The diagram above shows that on each pass, we match one a, one b and one c and replace them with an x until there are no a, b or c left.*

**General Models of Computation**

Models of Computation

* Turing Machines
* Unrestricted Grammars
* Partial Recursive Functions
* Lambda Calculus

1. Halting configures when an action ‘Y’ or ‘N’ is performed
2. Hanging configures when there is no transition for current state and current input symbol
3. Computation is a sequence of configurations of length n >= 0

Non-Context-Free Language

For a Turing Machine M,

* + - If start cell is empty, then halt
    - If the current cell contains a value, then write X and scan right
    - Look for b, replace with Y
    - Look for c replace with Z
    - Scan left to an a right of X
    - Make sure nothing remains of c

Computations

* M is said to halt on input w if (q0,w) yields a halting configuration
* M is said to hang on input w if (q0,w) yields a hanging configuration
* Turing machines compute functions from strings to strings

Functions on Natural Numbers

* Represent numbers on unary notation using only the symbol 1
* Zero is represented by the empty string

Recursive Languages

* A language is recursive if the characteristic function is Turing-computable where
  + In each XL(w) Y if w is an element in L, N otherwise
* M erases the marks from left to right, with the current parity encoded by the state. Once a blank is reached, mark Y/N as appropriate

Recursively Enumerable Languages

* M accepts a string w if M halts on input w
  + M accepts a language if M halts on w

Combining Turing Machines

* Two Turing Machines computations can be combined into larger machines
  + M1 prepares string as input into M2
  + M1 passes controls to M2 with I/O head at end of input
  + M1 retrieves control when M2 has completed
* Symbol writing machines and head moving machines R and L move the head appropriately
* Extensions
  + The following extensions do not increase the power of Turing Machines
    - 2 way infinite tape
    - Multiple tapes
    - Multiple heads on one tape
    - Two-dimensional tape
    - Non-determinism

Unrestricted Grammars

* The language recognised by a Turing Machines are those of unrestricted or free grammars
* These are also known as the rewriting system
* This is a grammar in which every production contains at least one non-terminal

Partial Recursive Functions

* We characterise functions from N to N which can be computed
* Firstly Define the followin functions
  + Zero(n1…nk) = 0
  + Succ(n) = n + 1 for all n is a natural number
  + P(n1…nk) = n for 1 <= i <= k
* We define a way of composing existing functions to define new ones:

*F(x) – h(g(X),…K(x)*

Primitive Recursion

Primitive recursive functions can be constructed:

*F(x,0) = h(x)*

*F(x,succ(y)) = g(x,y,f(x,y))*

Primitive Recursive functions are total recursive functions

Mu/Micro Recursion

***Unbounded minimalisation:*** for a predicate P, the unbounded minimalisation of P is the function f defined as follows:

*F(x) = min {y|P(x,y) is true}*

that is, the least value of y that satisfies P(x, y)

This is also denoted as follows:

*F(x) = muy.P(x, y) Is true*

Functions defined in this way are called micro-recursive functions

All partial recursive functions are micro-recursive functions

*Theorem:* A function is micro-recursive if it is computable

Godelization

* We can convert numbers to strings by using unary notation
* We can convert strings to unique numbers as well
* Assign each character in Sigma a unique number – the ith character has value i
* Denote the ith prime number as Pi
* We can use micro-recursive functions to compute functions from strings to strings

Lambda Calculus

Lambda Calculus is a universal model of computation that can be used to simulate any single-taped Turing Machine

* Primary Characteristics
  + - Computation is expressed using variable binding and substitution.
    - Lambda terms are constructed and reductions operations are performed on them
    - Binding is the association of entities with identifiers to reference the entity
* Syntax

Lambda terms are built up inductively in the following terms:

* + - Variables -> x, y, z
    - Applications, of the form s t, lambda terms
    - Abstractions, of the form Lambda.x.s, Lambda.y.t
  + Lambda terms can be described by:
    - *Term ::= Var | Term Term | Lambda Var.Term | (Term)*
  + Since the syntax is defined inductively we can define things by ***recursion*** and prove things by ***structural induction***
* Free Variables
  + Free variables in a lambda term is one which is not bound to any corresponding formal parameter
  + The free variables of a lambda term t, denoted by FV (t) can be defined as follows:

*FV (x) = {x}*

*FV(s t) = FV(s) UNION FV(t)*

*FV(Lambda.x.s) = FV(s)\{x}*

* A lambda term is closed if it does not contain any free variables
* Reduction
  + In order to express certain reduction rules we formalise the notion of substituting a term for a variable in another term
  + T[s\x] is substituting s for a variable x in another term t
  + We only substitute for free variables
* Reduction Rules

Lambda calculus is based on three fundamental reduction rules which transform one term into another:

* + - **Alpha conversion**
      * Lambda x,s -> Lambda y.s[y/x] provided y is not in FV(s)
      * Variables should not be already in a lambda functions in order to respect the interpretation
    - **Beta conversion**
      * (Lambda x.s) t -> s[t/x]
    - **Nu conversion**
      * Lambda x.t x -> provided x is not in FV(t)
* Lambda Equality
  + We say two terms s and t are equal (s = t) if there is a finite sequence of alpha, beta or nu reductions forwards and backwards at any depth
  + This equality is defined notion and not similar at the syntactic level
    - We call syntactic equality identity using the congruency symbol
  + Lambda x.x = Lambda y.y but they arent congruent
  + Equality relation is extensional i.e. if two functions f and g give equal results for all arguments, then they are equal
* Evaluation
  + The basic evaluation strategy in the Lambda calculus is reduction
  + Any sub expression of the form (Lambda x.s) t is a suitable candidate for reduction by beta-reduction
  + An evaluation is said to have completed when the expression contains no more redexes: it is in normal form
  + The leftmost innermost redex
    - This is called applicative-order reduction and corresponds to call-by-value evaluation
  + The leftmost-outermost redex
    - This is called normal-order reduction and corresponds to call-by-name evaluation
* Church-Rosser Theorem
  + This theorem states that if s -> t1 and s -> t2 then there is a term u such that t1 -> u and t2 ->u
  + By applying this theorem repeatedly, we get the stronger form that if t1 = t2 then there is a term u with t1 -> u and t2 -> u
  + So if a term reduces to, or is equal to a term in normal form, that normal form is unique up to alpha conversion
  + Also a consequence of the theorem is that lambda equality is non-trivia because two terms in normal form that are not alpha equivalent are not equal. ***Lambda x y.x*** *is* ***not equal*** *to* ***Lambda x y.y***
  + If any reduction sequence terminates then that one arrived at by systemically reducing the leftmost-outermost redexes will terminate

Constructions in Lamba-Calculus

As lambda calculus is a general model of computation, we need to show its expressive power of a programming language.

The following are some examples:

* Booleans
  + TRUE = Lambda t.Lambda f.t
  + FALSE = Lambda t.Lambda f.f
  + IF = Lambda b.Lambda x.Lambda y.b x y
* Natural Numbers
  + Coded as church numerals
    - n – lambda s.Lambdaz.sn z
  + This means
    - 0 = Lambda s.Lambdaz.z
    - 1 = Lambda s.Lambda z.s z
    - 2 = Lambda s.Lambda z.s (s z)
    - 3 = Lambda s.Lambda z.s (s(s z))
* Successor (succ)
  + The following operations add one to a church numeral
    - SUCC = Lambda n.Lambda s.Lambda z.s (n s z)
  + We can also test if a church numeral is zero:
    - ISZERO = Lambda n.n(Lambda x.FALSE) TRUE
* Pairs
  + Pairs are ordered as follows:
    - PAIR = Lambda f.Lambda s.Lambda p.p f s
  + Given this the destructors for pairs can be defined:
    - FST = Lambda p.p(Lambda x.Lambda y.x)
* Lists
  + We represent list as follows:
    - NIL = Lambda c.Lambda n n
    - CONS = Lambda h.Lambda t.Lambda c.Lambda n.c h t

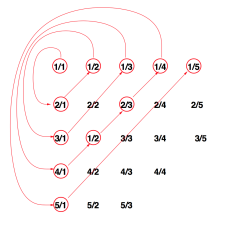
Church-Turing Thesis

* + We now have four general forms of computation: Turing Machines, unrestricted grammars, partial recursive functions and Lambda calculus.
  + Each one can transform into another. They are all equally powerful
  + Church-Turing thesis states that Turing machines are formal versions of algorithms and no computational procedure will be considered an algorithm unless it can be represented by a Turing Machine
  + It is a thesis since it unprovable, it asserts that a certain informal concept corresponds to a mathematical concept
  + To disprove the Church-Turing thesis, devise a model of computation that does finite work at each step and which computes functions not conputable by a Turing Machine

**Computability**

Countability

* If a set has the same cardinality as a subset of natural numbers N, then that set is countable
* If |A| = |N| then set A is countably infinite or denumerable
* If the set is not countable then it is uncountable
* Countability carries the implication of a listing or enumeration of the elements in the set

Countably Infinite sets

**Theorem:** The set of positive rational numbers Q is countably infinite

**Proof:** We enumerate Q along diagonals skipping duplicates

**Theorem:** The set of strings S over a finite alphabet A is countably infinite

**Proof:** List the strings in lexicographic order:

* All the strings of zero length
* All the strings of length 1
* All the strings of length 2 and so on

This implies a bijection from N to the list of strings and hence it is a countably infinite set

Uncountable sets

**Theorem:** The set of all real numbers between 0 and 1 is uncountable

**Proof:** We assume that it is countable and derive a contradiction

* If the set is countable, we can list the elements
* We can construct a real number between 0 and 1 which is not in the list
* Hence, there cannot exist a list and therefore the set is not countable

Countability of Turing Machines

**Theorem:** The set of all Turing machines is countable

**Proof:**

* Every Turing Machine ican be encoded as a string
* We have already seeen that the set of strings is countable
* Hence, all sets of Turing Machines are countable

Limits of Turing Machines

Uncountable Problems

* Any program that takes input is regarded as a function which does manipulation and produces some output
* Since all data is represented as some binary number, a program can be regarded from N to N
* For uncountable functions, no program exists for these types
* There does not exist a Turing Machine which can compute a number x between 0 and 1 since there are more numbers than Turing Machines to compute them

Universal Turing Machines

We can create a universal Turing machine:

* Input is a specification for a Turing Machine M and string w
* Output is the output from M on w
* The formula is ***U ((M,w)) = M(w)***

Halting Problem

Given a Turing machine M and string w, will M halt when run on w

If there is a Turing Machine H that solves this problem then this could be defined as

H((M,w) = {Y if M halts on input w, N otherwise}

This question is the halting problem

**Theorem:** There is no Turing Machine to solve the halting problem

**Proof:** By contradiction, there is a Turing machine H that solves this problem

We define a Turing Machine C that takes encoding of another Turing Machine M as its input and behaves as follows:

* Runs H on input(M, M)
* Loops forever if M halts on input M
* Halts otherwise

If the Halting problem is decidable then we construct a table for all possible Turing Machines and all possible inputs which indicate whether or not a given Turing machine halts for given input.

We construct a Turing machine T such that T(input) halts if T I (input i) does not halt and T(input) does not halt if T(input i) does halt

If the halting problem is decidable then T must be contained in enumeration

However T differs from the jth turing Machine for input j so we have a contradiction

Reducibility

In order to prove that other problems are undecidable, rather than perform more diagonalisation proofs, we show that these problems can be reduced to another problem which is known to be undecidable

**Theorem:** The problem of determining whether a program halts on the empty input is undecidable

**Proof:** Suppose that M’ decided the language L (M): accepts NULL

* Given arbitrary machine M and string w, we create a new machine M w that operates as follows on empty input:
  + Write w on the tape
  + Simulate the execution of M
* Now apply M’ to M w to solve the original halting problem
* Thus M’ does not exist

Other Undecidable Problems:

* Given a Turing Machine M, is there any string on which M halts?
* Given a Turing Machine M, does M halt on every input?
* Given a Turing Machine M, does M ever print tape symbol a?
* Given a Turing Machine M, does M ever enter state q?
* Given Turing Machines M1 and M2, do they halt on the same input strings?
* Given a Turing Machine M, is the language M accepts regular? Is it context-free? Or Turing-Undecidable?
* Does a particular line in a program get executed?
* Does a program contain a computer virus?

**Complexity**

Computational Complexity

A decidable problem is

* Computationally solvable in principle
* Not in practice

Problem resources are

* Time
* Space

Let M be a deterministic Turing Machine that halts on all inputs

The running time of M is function f; M -> M where f(n) is the max running time of M on input of length n

* ***Worst case:*** longest running time on input of given length
* ***Average case:*** average running time input of given length

Asymptotic notation

The exact running time of most algorithms is quite complex so it is better to estimate it

For functions f, g, we say that: *f (n) + O (g (n))*

If there are positive integers c1 and c2 such that *f(n) <= c1.g(n)* for *n >= c2*

Algorithm Classification

* ***O(1)*** Run time is independent of problem size, n
* ***O(log n)*** Occurs when a big problem is solved by transformation through some constant **logarithmic** style
* ***O(n)*** Occurs when each element of the problem requires a small amount of processing in **linear** fashion
* ***O(n log n)*** Occurs when a problem is broken into smaller subproblems, solving them independently and combining solutions i.e **linearithmic**
* ***O(n2)*** Occurs when an algorithm processses all pairs of elements which is **quadratic**
* ***O(2n)*** involves brute force approaches which is **exponential**

Polynomial and Exponential Complexities

* **DTIME** is a language decided by an O(f(n)) time deterministic TM
* If there is an upper bound of nk where k > 0 we say it runs in polynomial time.
* **P** is the set of languages decidable in polynomial time on deterministic TM’s which corresponds to tractable problems
* If there is an upper bound of 2nk where k > 0, then we say it runs in exponential time
* The class **EXPTIME** is the set of languages decidable in exponential time on deterministic TM’s. Problems in this class are intractable

PSPACE

* **SPACE(f(n))** is a language decided by a deterministic TM using O(f(n)) space
* If there is an upper bound of n to the power of k with k > 0 then it runs in polynomial time
* The class **PSPACE** is the set of languages decidable in polynomial space on deterministc TM’s

Complexity of Turing Machines

Deciding *{0n 1n }:*

* Single tape M1: O(n squared)
* Single tape M2: O(n log n)
* Two-tape M3: O(n)
* Computability: all reasonable models of computation are equivalent
* Complexity: Choice of model affects time-complexity, but all reasonable models of computation are polynomially equivalent

Thus class P is invariant for all models of computation polynomially equivalent to a deterministic single tape TM

Complexity Class NP

The running of non-deterministic TM M is a function *F:N -> N* where f(n) is the max number of steps that M uses on any branch of an input of size n

**NTIME(f(n))** is a language decided by an O(f(n)) time non deterministic TM

If the upper bound is n to the k and k > 0 it runs in polynomial time

NP is the set of languages decidable in polynomial time on non-deterministic TM’s

The complexity classes are **P** SUBSET **NP** SUBSET **PSPACE** SUBSET **EXPTIME**

Polynomial Verifiability

For a Hamiltonian path algorithm, we don’t know a fast way to find a path but we can check whether a path is in polynomial time.

Verifying a path is much easier than determining its existence

A verifier for a language L is an algorithm V where

*L = w|V accepts(w, c) for some string c}*

* Measure the verifier by the length of w
* A polynomial verifier runs in polynomial time
* A language L is polynomially verifiable if it has a polynomial verifier
* String c is a proof
* For PV’s c must have a length polynomial in |w|

NP Completeness

* In order to show that a problem belongs to a particular class w show that it can be reduced to another problem which is known for that complexity class. A language L1 is polynomial-time reducible to language L2 if there is a polynomial time computable function f where for every string w:

*W is in L1 if f(w) is in L2*

* A language is NP Complete if it satisfies the following conditions
  1. L is in NP
  2. Every L’ is in NP and is polynomial time reducible to L
* If L1 is NP-Complete an L1 is polynomial time reducible to L2 where L2 is in NP, then L2 is NP complete
* The class of NP-complete languages are
  1. Hardest languages in NP
  2. Least likely to be in P
  3. If any Np-complete L in P then NP = P

Cook-Levin Theorem

The Boolean satisfiability problem is NP complete. Any problem can be reduced in polynomial time by a deterministic TM to the problem of determining whether a Boolean formula is satisfiable

If there exists a deterministic polynomial time algorithm for solving Boolean satisfiability (which is the problem of determining if there exists a formula where the variables can be replaced by True and False), then every NP problem can be solved by a deterministic polynomial time algorithm. This is equivalent to the P v NP problem

TODO

Tutorials

* Lambda Calculus
* Computability

**Complexity**

**Pumping Lemma & Limits of Languages**

Limits of Regular Languages

Not al languages are regular

We cannot construct DFA to recognise the languages

* L = {pkqk}
* L = {w in {a,b}\* | w has an equal number of a and b}
* L = {wcwr | w in Sigma \*}, where wr is w reversed

We can construct DFA for:

* Alternating 0 and 1 = (NULL|1)(01)\*(NULL|0)
* Set of pairs of 0 and 1 = (01|10)+

Pumping Lemma for Regular Languages

The pumping lemma for regular languages is:

* Let L be an infinite regular language. There are strings x, y and z such that y is not null and xykz is subset of L for each k >= 0
* L is accepted by some DFA M
* L is infinite and M has a finite number of states so there is a cycle on M.
* The cycle corresponds to y.

Pumping Lemma for Context-Free Languages

The pumping lemma for context-free languages is:

* Let L be an infinite context-free language. There are strings v, w, x, y and z so that wy is not null and vwkxykz is a subset of L for each k >= 0
* The pumped variables w and y cannot contain two distinct letters
* At least one of w and y is non-empty
* Therefore not all letters can get pumped up equally

Regular Expressions Answers

