Disclaimer
This is a collection of **Computability and Complexity** (COCO) course content I produced for my own revision needs. This is by no means complete, may contain errors, and you should revise through all materials given in the module. I could not be held responsible for any deviations of this collection from the original course content. However, if you spot any problems in this collection, please do contact me through the email address on the top-right corner, so I can correct them for future readers. You should consult copyright holders and the department before using it for any purposes other than personal academic revisions. Source: 2015-2016 materials from http://www-module.cs.york.ac.uk/coco

Context Sensitive Grammar

A grammar is context-sensitive if each production $\alpha \rightarrow \beta$ satisfies $|\alpha| \leq |\beta|$, with the possible exception of $S \rightarrow \Lambda$.
If $S \rightarrow \Lambda$ is present, then S must not occur in any right-hand side. A language is context-sensitive if it is generated by some context-sensitive grammar.
Computed by a linear-bounded automaton (introduced later). Every context-free language is context-sensitive.

Unrestricted Grammar

Where all this module is about begins. Compared to **Context Sensitive Grammar**, there is no longer a restriction on length of β. Computed by a Turing machine.

Deterministic Turing Machine

A Turing machine (TM) is a 5-tuple $(Q, \Sigma, \Gamma, q_0, \delta)$ where Q is the finite set of states, including h_a and h_r , the halting states; Σ is the input alphabet; Γ is the tape alphabet, including the blank symbol ∆, where $\Sigma \subseteq \Gamma - \{\Delta\}$; $q_0 \in Q$ is the initial state; $\delta: (Q - \{h_a, h_r\}) \times \Gamma \rightarrow Q \times \Gamma \times \{L, R, S\}$ is the transition function.
A TM moves a tape head on a infinite tape of squares. A move $\delta(\mathbf{q}, \mathbf{a}) = (\mathbf{r}, \mathbf{b}, \mathbf{D})$ takes place if the machine is in state **q** and reads symbol **a**: the machine enters state **r**, overwrites **a** with **b**, and moves its head one square to the left (**D** = L), to the right (**D** = R), or leaves it stationary (**D** = S).
A configuration **uqv** represents the situation in which **uv** is the contents of an initial portion of the tape, the TM is in state **q**, the tape head reads the <u>first</u> symbol of **v**, and all squares to the right of **uv** are blank.
Initial configuration for input **w**: $\mathbf{q_0\Delta w}$.
<u>Halting configurations</u>: **accepting –** uh_av (unread v accepted altogether); **rejecting** – uh_rv.
A transition $uqv \vdash xry$ describes the effect of a single move of a TM. We write $uqv \vdash^* xry$ for a sequence of transitions.
Crash reject: define $qav \vdash h_rav$ if $\delta(q, a) = (r, b, L)$ for some $r \in Q$ and $b \in \Gamma$.
Transition reject: if there is no transition for some pair $(q, a) \in (Q - \{h_a, h_r\}) \times \Gamma$, then $\delta(q, a) = (h_r, a, S)$. Remember to count this transition when calculating **time complexity** (later).
The **language accepted by a TM** M is $L(M) = \{w \in \Sigma^* \mid q_0\Delta w \vdash^* uh_av \text{ for some } u, v \in \Gamma^* \}$.

Crash Avoidance

It is possible to construct TM M' for each TM M, such that L(M) = L(M′) and M′ does not crash on any input, by extending Γ with a new symbol #. M' writes # on the first square, inserts ∆ on the 2nd square by shifting the input string one position to the right. It then moves the tape head to the 2nd square, and then executes the transitions of M. For each $q \in Q' - \{h_a, h_r\}$, $\delta'(q, \#) = (h_r, \#, S)$.

Nondeterministic Turing Machine
A nondeterministic Turing machine (NTM) is defined like a Turing machine, except that state h_r is dropped and that δ has the form $\delta : (Q - \{h_a\}) \times \Gamma \rightarrow 2^{Q \times \Gamma \times \{L, R, S\}}$.
Compared to the transition reject of deterministic Turing machine, if δ(q, a) = empty (no a-labelled arcs leave q in the transition diagram), an NTM halts without acceptance or rejection: the input string may or may not be in the machine's language.

Multitape Turing Machine
This kind of Turing machine has n tapes on which it works simultaneously. A configuration is an n-tuple $(u_1qv_1, u_2qv_2,..., u_nqv_n)$, where $q \in Q$ and $u_i , v_i \in \Gamma^*$. The initial configuration for an input w is $(q_0\Delta w, q_0\Delta,..., q_0\Delta)$.
For every multitape Turing machine N there exists a Turing machine M such that L(N) = L(M).

Semidecidable Language
A language L is **semidecidable** (or recursively enumerable) if there exists a Turing machine that accepts L. A language L is semidecidable if and only if L is generated by some (unrestricted) grammar.
A language L is enumerated by some multitape Turing machine if and only if L is semidecidable.

Linear-bounded Automata
A linear-bounded automaton (LBA) is defined like a nondeterministic Turing machine M, except that the initial configuration for an input w is q_0<w>, where '<', '>' ∈ Γ − Σ; the tape head cannot move to the left of '<' and not to the right of '>', and '<' and '>' cannot be overwritten.

Context Sensitive Language and Decidability
A language L is is accepted by some linear-bounded automaton if and only if L is context-sensitive. Every context-sensitive language is decidable. The converse does not hold.

Decidable Language
A language L is **decidable** (or recursive) if it is accepted by some Turing machine M that on every input eventually reaches a halting configuration. We say that M decides L.
If a language L is decidable, then its complement $L' = \Sigma^* - L$ is also decidable. To prove, modify M to a machine M' by swapping the accept and reject states, then the language accepted by M is the complement of the language accepted by M'. M' also halts on every input because M does. Hence L' is decidable.
If both a language L and its complement L' are semidecidable, then L is decidable. To prove, construct a 2-tape TM which copies its input to Tape 2 and then simulates M and M' in parallel. The TM will always reach h_a' or h_r' depending on whether the input is in the language or in its complement. Thus L(M′) = L and M′ halts on every input.

Encoding Turing machines
To compute languages that are not semidecidable, Turing machines taking Turing machines as input are required. Input can be provided by encoding a Turing machine.
First start with encoding the alphabet, $01^{n1}01^{n2}...0$, with number of 1's n_1, n_2 ...representing each element of the alphabet. Then for each transition, encode as followed:

$01^{initial\_state}01^{symbol\_read}01^{end\_state}01^{symbol\_written}01^{tape\_head\_action}0$, note that the initial and trailing zeros present for each transition, so there will be in total two zeros between encodings of different transitions. Finally, a final '0' terminates the encoding of transitions.
There is an algorithm which for every string $w \in \{0, 1\}^*$ decides whether w is the code of some Turing machine and, if this is the case, constructs a machine M such that e(M) = w.
If M and M′ are Turing machines such that e(M) = e(M′), then M and M′ are equal up to renaming of states. Hence L(M) = L(M′) and M and M′ halt on the same inputs.

Self-acceptance Properties
Self Accepting Language (SA) is defined by SA = {e(M) | M is a TM and e(M) ∈ L(M)}, and NSA = SA'. SA consists of the codes of all Turing machines that accept their own codes, while **Non-self Accepting** Language (NSA) consists of the codes of all TMs that do not.
NSA is not semidecidable. Proved by contradiction: suppose NSA is, then: if the code of the TM accepting the NSA language is in NSA, the definition of the NSA requires the code to not be self-accepted – not in NSA, a contradiction; if the code of the TM accepting the NSA language is not in NSA, the definition of NSA would require the code to be in NSA, again a contradiction.
SA is not decidable (since its complement NSA is not semidecidable), but semidecidable. A Turing machine T_{SA} can be built to first reject input if it's not code of a Turing machine, then construct a Turing machine M such that e(M) = w. After verifying that {0, 1} is a subset of M's input alphabet, T_{SA} runs M on w, accept if and only if M accepts w.

Universal Turing Machine
A universal Turing machine U takes inputs of the form e(M)e(w), where M is a TM and w is a string over M's input alphabet. Machine U simulates M on input w, where it accepts, rejects or diverges on w as M would. U is an interpreter for Turing machines.

Countable Sets
A countable set is a set with the same number of elements as some subset of the natural number set. Whether countably finite or infinite, it can always be counted "one at a time", finish-able or not.
A set S is countably infinite if there exists a bijective function {0,1,2,...}→S, and countable if it is countably infinite or finite.
Union of countable sets are countable; powersets of infinite sets are uncountable; result of deduction of a countable subset from an uncountable set is uncountable.
For every non-empty alphabet Σ, there are uncountably many languages over Σ that are not semidecidable.

Partial Functions
A partial function is a function that may not map every element in the domain into a result.
A Turing machine M = (Q, Σ, Γ, q0, δ) computes the partial function $f_M : \Sigma^* \rightarrow \Sigma^*$ defined by $f_M(x)$ = (y if $q_0\Delta x \vdash^* h_a\Delta y$ for some $y \in \Sigma^*$), $f_M(x)$ = undefined otherwise. It may be that M diverges, halts in rejecting state, halts with tape head not on square 1; with a non-blank symbol on square 1; with a symbol not in tape alphabet or with non-blank symbols after result.
<u>At the end of partial function computation, the tape head must return to square 1 with blank symbol, leaving the result of function on tape, and halt in accepting state.</u>

A partial function $f : (\Sigma^*)^k \to \Sigma^*$ is **Turing-computable** (or simply computable) if there exists a Turing machine M such that $f_M = f$. A partial function $f : N^k \to N$ on natural numbers is computable if the corresponding function $(\{1\}^*)^k \to \{1\}^*$ is computable, where each $n \in N$ is represented by 1^n.

Characteristic Function of a Language
The characteristic function of a language $L \subseteq \Sigma^*$ is the (total) function $\chi_L : \Sigma^* \to \Sigma^*$ defined by $\chi_L(x) = (w_1$ if $x \in L)$, $\chi_L(x) = w_2$ otherwise; for fix distinct strings $w_1, w_2 \in \Sigma^*$. A language L is decidable if and only if its characteristic function χ_L is computable.

Graph of a Function
The **graph** of a partial function $f : \Sigma^* \to \Sigma^*$ is the set $\text{Graph}(f) = \{ (v, w) \in \Sigma^* \times \Sigma^* \mid f(v) = w \}$, which can be turned into a **language** over $\Sigma \cup \{\#\}$: $\text{Graph}_\#(f) = \{ v\#w \mid (v, w) \in \text{Graph}(f) \}$.
A **partial** function $f : \Sigma^* \to \Sigma^*$ is computable if and only if $\text{Graph}_\#(f)$ is semidecidable.
A **total** function $f : \Sigma^* \to \Sigma^*$ is computable if and only if $\text{Graph}_\#(f)$ is decidable.

Decision Problems
A decision problem is a set of questions each of which has the answer 'yes' or 'no', with each question as a yes-instance or a no-instance. For the problem to be reasonable, the language of yes-instances (Y(*P*)) and the language of no-instances (N(*P*)) must be disjoint (the problem must be partionable); the problem must be decidable and the encoding of the problem must be decodable.
To decide a decision problem, we consider the yes-instances. A decision problem P is decidable (solvable) if Y(*P*) is decidable, otherwise P is undecidable (unsolvable). P is semidecidable if Y(*P*) is semidecidable. A decision problem can be *undecidable but semidecidable*, which is not the case for semidecidable languages.
The complement of a decision problem is obtained by swapping the yes and no instances. The complement is decidable if the original is decidable.

Language and Problem Reduction
Let $L_1, L_2 \subseteq \Sigma^*$ be languages. **L_1 is reducible to L_2** if there is a computable total function $f : \Sigma^* \to \Sigma^*$ such that for all $x \in \Sigma^*$, **$x \in L_1$ iff $f(x) \in L_2$**. If L_2 is decidable, then L_1 is decidable. If L_2 is semidecidable, then L_1 is semidecidable.
Given decision problems P_1, P_2, we say that P_1 is reducible to P_2 if $Y(P_1)$ is reducible to $Y(P_2)$. If P_2 is decidable, then P_1 is decidable. If P_2 is semidecidable, then P_1 is semidecidable.
Problem reduction is the process of reducing a decision problem A to a decision problem B, showing that the solvability of B implies solvability of A. The solver for the problem after reduction – B, may be used as part of the solver for the problem before reduction – A. A is usually "simpler" than B.

Undecidable (but may be Semidecidable – UBS) Problems
SAP: does a Turing machine accept its own code? It is UBS, with the Turing machine constructed earlier.
MP: does a Turing machine accept a specific input? It is UBS. Proved through problem reduction from SAP to MP. Construct a TM T_{SA} which solves SAP. T_{SA} transforms input e(M) into e(M)e(e(M)), and run T_{MP} on this string, T_{SA} accepts e(M) if and only if T_{MP}

accepts e(M)e(e(M)), which is when e(M) ∈ L(M), and halts on all inputs. This would imply that T_{SA} solves SAP, which is undecidable. Therefore MP cannot be decidable, as MP is reduced from SAP. Y(MP) is however, equivalent to the language represented by a universal Turing machine, so MP is semidecidable.
HP: does a Turing machine reaches a halting configuration on a specific input? It is UBS. Proof-by-contradiction shows that if the halting problem is decidable, then MP reducing to HP can solve the membership problem for semidecidable languages by running T_{halt} on e(M)e(w) (the input of the universal Turing machine), then running M on w. But MP is undecidable, so HP is undecidable. To show HP is semidecidable, modify the universal Turing machine so that it accepts input e(M)e(w) if and only if it halts on e(M)e(w).
Accept-Λ: does a Turing machine accepts the empty string? It is UBS. Undecidability is shown by reducing MP to A-Λ. A Turing machine can be constructed that it first replaces input e(M)e(w) into wiping tape and always running M on w, and then feed the replaced input into T_Λ which solves A-Λ. It can then be shown that MP is solvable by reducing to A-Λ, which is a contradiction. Semidecidability can be shown by constructing a Turing machine that runs M on Λ, on input e(M), and accept e(M) if and only if M accepts Λ.
Other examples including Accept-no-input, Accept-same-inputs, Write-non-blank.

Language Property
A property *P* of Turing machines is a language property of language L, if every Turing machine accepting the same language L have property P.
Examples: L(M) is finite; $L(M) = \Sigma^*$.
Counter-examples: HP and SAP are properties of Turing machines, and not language properties.
A language property *P* of Turing machines is trivial if either: every TM has property *P*; or: no TM has property P. Example: L(M) is semidecidable. Counter-example: $L(M) = \Sigma^*$.
Rice's Theorem: Every non-trivial language property of Turing machines is undecidable.
Some decidable problems for context-free languages: MP (for context-free), Emptiness, Finiteness. Undecidable problems: Empty-intersection, Ambiguity, Inherent-ambiguity.
Church-Turing Thesis: Every effective procedure (algorithm, computable partial function, solvable decision problem) can be carried out by a Turing machine.

Complexity
We consider worse case complexity for Turing machines that terminate on all inputs.
The **time complexity** of a (possibly nondeterministic) Turing machine M is the function $\tau_M : N \to N$, where $\tau_M(n)$ is the maximum number of transitions M can make on any input of length *n*.
The **space complexity** of a (possibly nondeterministic) Turing machine M is the function $s_M : N \to N$, where $s_M(n)$ is the maximum number of tape squares M can visit on any input of length *n*.
Space is bounded by time, for every TM M, $s_M(n) \leq \tau_M(n) + 1$.
Big-O hierarchy: $O(1) < O(\log(n)) < O(n) < O(n\log(n)) < O(n^2) < O(n^3) < O(n^r) < O(b^n) < O(n!)$. Function f is of order g if there are positive integers c and n_0 such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.
Properties: There exist a single tape TM that can compute the same language as its multitape equivalent, in $O(\tau^2_M)$ time. There is always a TM which computes a decidable language with higher complexity than any total function – there are arbitrarily complex languages. There is always a TM that computes a same decidable language in $O(\log_2 \tau_M)$

time – there need not exist a fastest TM.

P and PSpace

A language is **decidable in polynomial time** if there is a Turing machine accepting the language in $O(n^r)$ time for some $r \in N$. The class of this kind of languages is denoted by **P**.
A language that is **decidable in polynomial space** is defined in the same manner, collectively denoted as **PSpace**.
A decision problem is said to be in P if the language Y(*Problem*) is in P, assuming polynomial-time encoding and decoding (e.g. binary notation).
Example: Path problem (determining is a graph has a path from one specific node to another) is in P. Proved by showing a polynomial algorithm checking each connected node from the initial node, and check if the target node is directly or indirectly connected.

NP and NPSpace

A language is **accepted in nondeterministic polynomial time** if there is a nondeterministic Turing machine accepting the language in $O(n^r)$ time for some $r \in N$. The class of this kind of languages is denoted by **NP**.
A language that is **accepted in nondeterministic polynomial space** is defined in the same manner, collectively denoted as **NPSpace**.
A decision problem is said to be in NP if the language Y(*Problem*) is in NP, assuming polynomial time encoding and decoding.
P?=NP: All problems in P are also in NP, as deterministic Turing machines are also nondeterministic ones. But unless P=NP can be proven, not all problems in NP are in P.
A problem in NP can be solved by a nondeterministic Turing machine repeatedly "guess and check" solutions (as long as the checking can be done in polynomial time), and hope that one such solution will be valid.
Examples: Complete Subgraph problem (Clique problem) and the Hamiltonian Circuit problem are in NP.

Relations between Complexity Classes

A language L is **tractable** if $L \in P$, otherwise L is **intractable**. A decision problem P is tractable if the language Y(P) is in P.
The relationship between classes are $P \subseteq NP \subseteq PSpace = NPSpace \subseteq ExpTime$.
Proof of PSpace = NPSpace: **Savitch's Theorem** states that for every NTM N there is a TM M such that L(M) = L(N) and $s_M \in O(s^2_N)$. Hence $s_N \in O(n^r)$ implies $s_M \in O(n^{2r})$.
Polynomial Time Reduction: If a Turing machine that can reduce a language L_1 to another L_2 in polynomial time, the L_1 is polynomial-time reducible to L_2. L_1 is no harder than L_2, and if L_2 is in P (or just NP), so does L_1.
A language L is **NP-hard** if every language in NP is **polynomial-time reducible** to L – no language in NP is harder than L.
There exist NP-hard problems more complex than the ExpTime class.
A language is **NP-complete** if it is **NP-hard** and belongs to **NP** – they are the hardest languages in NP. **NP-complete** ⊆ **NP-hard**.
For every NP-complete language L, $L \in P$ if and only if P = NP. Otherwise, P and NP-complete classes are disjoint. A few known problems, like Graph Isomorphism problem, are neither in P nor NP-complete.
Example: The CNF Sat problem is polynomial-time reducible to the Clique problem, and it is NP-complete.