CS301 Complexity of Algorithms Notes

Based on lectures by Dr. Matthias Englert
Notes by Alex J. Best

May 30, 2014

Introduction These are some rough notes put together for CS301 in 2014 to make it a little easier to revise. The headings correspond roughly to the contents of the module that is on the module webpage so hopefully these are fairly complete, however they are not guaranteed to be.

What is a problem?

Definition 1. A *problem* is a function

$$f\colon \{0,1\}^* \to \{0,1\}^*.$$

A *decision problem* is a function

$$f\colon \{0,1\}^* \to \{0,1\}.$$

We identify a decision problem f with the language

$$L_f = \{x : f(x) = 1\}.$$

and call the problem of computing f the problem of deciding the language L_f.

What is a computation? A set of fixed mechanical rules for computing a function for any input.

Definition of a Turing machine.

Definition 2. A *Turing machine* consists of an infinite tape with letters of the alphabet Γ (usually $= \{0,1,\Box\}$) written on it. At the start the input is written on the tape beginning at the head and the state is q_{start}. The transition function

$$\delta\colon Q \times \Gamma \to Q \times \Gamma \times \{L, S, R\}$$

dictates what to read, which state to switch to and which direction to move to after reading a symbol while in a given state. So the machine is given by (Γ, Q, δ). When the machine reaches the state q_{end} the computation halts and the output is the section of tape starting at the head until the first blank $\Box$.

Definition 3. A Turing machine M *computes a function* $f\colon \{0,1\}^* \to \{0,1\}^*$ *in time* $T\colon \mathbb{N} \to \mathbb{N}$ if for every $x \in \{0,1\}^*$ the output of M when given x initially is $f(x)$, this computation must finish after at most $T(|x|)$ steps (this implies M halts on every input).

We say M *computes* f if it computes f in $T(n)$ time for any function T.

What happens if we increase the alphabet?

Claim 1. Let $f\colon \{0,1\}^* \to \{0,1\}^*$ and $T\colon \mathbb{N} \to \mathbb{N}$ be some functions. If f is computable in time $T(n)$ by a Turing machine M using alphabet Γ then f is computable in time

$$3\lceil \log |\Gamma| \rceil T(n)$$

by a Turing machine $\tilde{M}$ that uses only the alphabet $\{0, 1, \Box\}$.

To see this we can encode all the old symbols in terms of only $\{0, 1, \Box\}$ and create a new Turing machine that does what the old one would do, we have to move back an fourth along a strip of size $\log |\Gamma|$ at most 3 times to do this however.

k-tape Turing machines. We can also using Turing machines that have k tapes and k heads rather than simply one however this doesn't make much difference either.

Claim 2. Let $f\colon \{0,1\}^* \to \{0,1\}^*$ and $T\colon \mathbb{N} \to \mathbb{N}$ be some functions. If f is computable in time $T(n)$ by a k-tape Turing machine M using then f is computable in time

$$7kT(n)^2$$

by a single tape Turing machine $\tilde{M}$.

Church-Turing thesis. The Church-Turing thesis is the statement that

> Every physically realisable computation device (be it silicon-based, DNA-based, neuron based, ...) can be simulated by a Turing machine.

This is generally believed to be true.

Universal Turing machines. As a Turing machine is given by some finite amount of data we can represent it by some string encoding. We assume that we have picked an encoding so that every string represents a Turing machine and every Turing machine can be encoded by infinitely many strings. The Turing machine encoded by a string α is denoted M_α.

Theorem 1. There exists a Turing machine (a universal Turing machine) that when given a pair $\langle \alpha, x \rangle$ as input outputs the result of running the Turing machine encoded by α with input x. Moreover if the machine encoded by α halts within $T(|x|)$ steps on input x then the universal Turing machine halts within $C\,\mathrm{poly}(T(|x|))$ steps, where C is independent of $|x|$.

We assume we have fixed some encoding for Turing machines and a corresponding universal Turing machine, denoted $\mathcal{U}$.

Uncomputable functions. The set of all functions

$$f\colon \{0,1\}^* \to \{0,1\}$$

is uncountable. So as any Turing machine can be encoded as a finite string of bits under some fixed encoding, there are countably many Turing machines.

Corollary 1. There are functions

$$f\colon \{0,1\}^* \to \{0,1\}$$

that are not computable.

Example. The function

$$f(\alpha) = \begin{cases} 0 & M_\alpha(\alpha) = 1, \\ 1 & \text{otherwise} \end{cases}$$

is not computable.

This is as if we had a Turing machine M that computed f then M halts for all x, with $M(x) = f(x)$. So if we let x be a string representing the Turing machine M then $M(x) = f(x)$, but this is a contradiction by the definition of f.

HALT is not computable.

Claim 3. Another function that is not computable is

$$\text{HALT}(\langle \alpha, x \rangle) = \begin{cases} 1 & M_\alpha \text{ halts on input } x, \\ 0 & \text{otherwise.} \end{cases}$$

Proof. Assume that M_{HALT} is a Turing machine that computes HALT, then we can design a Turing machine to compute the function f in example , thus deriving a contradiction.

The Turing machine for f would first run $M_{\text{HALT}}(\langle \alpha, \alpha \rangle)$ outputting 0 if this returned 1. Otherwise it would then use the universal Turing machine $\mathcal{U}$ to compute $M_\alpha(\alpha)$, outputting 1 if this returned 0 and 0 otherwise. □

There are more functions that are practically useful that are not computable. For example deciding if a Diophantine equation (a possibly multivariate polynomial with integer coefficients) has a solution in the integers is impossible in general.

The language

$$\{\alpha : M_\alpha \text{ halts on all inputs}\}$$

is also undecidable. This is as if we could compute this language we could compute HALT by taking a pair $\langle \alpha, x \rangle$ and forming a Turing machine the always computes $M_\alpha(x)$ no matter what input it is given. If we could decide if the new Turing machine halted on all inputs we could decide if M_α halts on input x.

Rice's Theorem. All Turing machines correspond to a function

$$f \colon \{0,1\}^* \to \{0,1\}^* \cup \{\bot\}$$

where $f(x) = \bot$ means that the Turing machine does not halt on input x. Not all of these functions correspond to Turing machines, but we can take $\mathcal{R}$ to be the set of all such functions that do correspond to Turing machines.

Theorem 2 (Rice's Theorem)**.** Let $\mathcal{C}$ be a non-empty proper subset of $\mathcal{R}$, then the language

$$\{\alpha : M_\alpha \text{ corresponds to a function } f \in \mathcal{C}\}$$

is undecidable.

Proof. □

The complexity class P. We now define some *complexity classes*, sets of functions that can be computed with some given resources.

Definition 4 (The class DTIME)**.** Let $T\colon \mathbb{N} \to \mathbb{N}$ be a function, then we let DTIME($T(n)$) be the set of boolean functions computable in $O(T(n))$ time.

Definition 5 (The class P)**.**

$$\mathrm{P} = \bigcup_{k \geq 1} \mathrm{DTIME}(n^k).$$

This class does not depend on the exact definition of Turing machine used. Problems in P are thought of as efficiently solvable and are a very natural model for this concept.

Strong Church-Turing thesis. The strong Church-Turing thesis is the statement that

> Every physically realisable computation device (be it silicon-based, DNA-based, neuron based, ...) can be simulated by a Turing machine *with only a polynomial overhead.*

This is more controversial than the normal Church-Turing thesis as either

a) quantum mechanics does not behave as we currently understand it to,

b) a classical computer can factor integer's in polynomial time or

c) the strong Church-Turing thesis is wrong.

Reductions.

Definition 6 (Karp reduction)**.** A language $L \subseteq \{0,1\}^*$ is *Karp reducible* to a language $L' \subseteq \{0,1\}^*$ if there exists a computable function

$$f\colon \{0,1\}^* \to \{0,1\}^*,$$

such that for all x

$$x \in L \iff f(x) \in L'.$$

Definition 7 (Polynomial time Karp reduction)**.** A language $L \subseteq \{0,1\}^*$ is *polynomial time* Karp reducible to a language $L' \subseteq \{0,1\}^*$ if there exists a *polynomial time* computable function

$$f\colon \{0,1\}^* \to \{0,1\}^*,$$

such that for all x

$$x \in L \iff f(x) \in L'.$$

We denote this relationship by

$$L \leq_p L'.$$

If $L \leq_p L'$ and $L' \leq_p L$ then we write $L \equiv_p L'$.

Vertex Cover and Independent Set are equivalent. INDEPENDENT SET: Given a graph $G = (V, E)$ and an integer k is there set $S \subset V$ of size at least k where each edge of G has at most one endpoint in S.
VERTEX COVER: Given a graph $G = (V, E)$ and an integer k is there set $S \subset V$ of size at most k where each edge of G has at least one endpoint in S.

Claim 4. VERTEX COVER $\equiv_p$ INDEPENDENT SET.

Proof. S is an independent set if and only if $V \setminus S$ is a vertex cover. □

Vertex Cover reduces to Set Cover. SET COVER: Given a set U, a collection $S_1, \ldots, S_m$ of subsets of U and an integer k, does there exist a collection of $\leq k$ of the subsets whose union is all of U.

Claim 5. VERTEX COVER $\leq_p$ SET COVER.

Proof. We create a set cover instance for a graph $G = (E, V)$ by letting $U = E$ and $S_v = \{e \in E : e \text{ is incident to } v\}$ be our collection of subsets. Then there exists a set cover of size at most k if and only if there exists a vertex cover of size at most k in the graph G. □

3-SAT reduces to Independent Set.

Definition 8. A *literal* is a boolean variable or its negation.
A *clause* is a disjunction (OR) of literals.
A propositional formula Φ is in *conjunctive normal form* if it is a conjunction (AND) of clauses.
The problem SAT asks if a given propositional formula Φ that is in CNF has a satisfying assignment of variables.
A special case of this is 3-SAT where we require that each clause be a disjunction of exactly three literals.

Claim 6. 3-SAT $\leq_p$ INDEPENDENT SET.

Proof. We use the given propositional formula to construct an instance of INDEPENDENT SET as follows. For each clause we add a triangle of vertices to a graph G, labelled by each literal. We then connect all literals appearing to all of their negations. Then G contains an independent set of size k if and only if Φ is satisfiable. □

Transitivity for the reducibility-relation. Reduction is transitive, i.e. if $X \leq_p Y$ and $Y \leq_p Z$ then $X \leq_p Z$. To see this we can think of composing the functions that provide the reductions from X to Y and Y to Z.

Example.

$$\text{3-SAT} \leq_p \text{INDEPENDENT-SET} \leq_p \text{VERTEX-COVER} \leq_p \text{SET-COVER}$$

Definition of NP. For comparison we give an equivalent definition of the class P to the one given above.

Definition 9 (Complexity class P)**.** A language L is in the class P if there exists a Turing machine M and polynomial T so that M terminates on input x after at most $T(|x|)$ steps and M accepts x if and only if $x \in L$.

Definition 10 (Complexity class NP)**.** A language L is in the class NP if there exists a Turing machine M and polynomials T and p so that M terminates on any input x after at most $T(|x|)$ steps. We also require that if $x \in L$ then there exists a certificate $t \in \{0,1\}^{p(|x|)}$ so that M accepts $\langle x, t \rangle$, conversely if $x \notin L$ we require that M rejects any pair $\langle x, t \rangle$ where $t \in \{0,1\}^{p(|x|)}$.

In the definition of NP above M is called a certifier or verifier and t a certificate or proof for x.

Claim 7. P $\subseteq$ NP

Proof. Take the certifier M to be the decider for the problem from P, ignoring the second element of the input pair. □

Example. The language

$$\text{COMPOSITES} = \{s \in \mathbb{N} : s \text{ is composite}\}$$

is in NP. This is as we can use a non-trivial proper factor as a certificate, and then the certifier just needs to check the factor is as claimed by checking bounds and dividing. Here $|t| \leq |s|$.

Example. The problem SAT is in NP as we can take a satisfying assignment of the n boolean variables in the formula as the certificate. The verifier then needs only run through the formula checking that each clause has at least one true literal.

Example. Another problem in NP is HAM-CYCLE which asks if a given graph $G = (V, E)$ has a simple cycle that visits every node. The certificate for this problem is a permutation of the nodes of G and so the certifier need only check that all nodes are in this permutation exactly once and that there are edges between consecutive nodes in the permutation.

Definition of NP-completeness.

Definition 11 (NP-completeness)**.** A (decision) problem Y is called *NP-complete* if it is in NP and has the property that every other problem X in NP reduces to it in polynomial time.

Theorem 3. Let Y be an NP-complete problem, then Y is solvable in polynomial time if and only if P = NP.

Proof. ($\Leftarrow$) If P = NP then Y can be solved in polynomial time as Y is in NP.
($\Rightarrow$) If Y can be solved in polynomial time then as any problem X in NP reduces to Y in polynomial time we can solve X is polynomial time. Hence NP $\subseteq$ P and so P = NP. $\square$

An easy artificial NP-complete problem: TMSAT.

Alternative definition of NP using reductions and a representative (i.e. complete) problem of the class. Given a natural NP-complete problem we can equivalently define the complexity class NP to be all problems polynomial time reducible to that problem.

Cook-Levin theorem.

Theorem 4 (Cook, Levin)**.** SAT is NP-complete.

Proof. We know SAT is in NP, in order to show that all other problems in NP reduce to it we need some more machinery! $\square$

Oblivious Turing machines.

Definition 12. A Turing machine is called *oblivious* is its head movements do not depend on the input x but only on the length $|x|$ of the input.

Theorem 5. Given any Turing machine M that decides a language in time $T_M(n)$ there exists an oblivious Turing machine that decides the same language in time $O(T_m(n)^2)$.

Proof. Exercise! $\square$

Proof of the Cook-Levin theorem. We will solve the problem of certificate ex

Now we have established a natural NP-complete problem others are easier to do. To establish the NP-completeness of a given problem Y we can perform the following

Step 1 Show Y is in NP.

Step 2 Choose an appropriate NP-complete problem X.

Step 3 Prove $X \leq_p Y$.

A major problem of complexity theory is determining whether P = NP or not, if they are equal then there are efficient algorithms for all NP-complete problems. If not, no such algorithms are possible, and most computer scientists believe this to be the case. Most NP problems are known to be in P or NP-complete, factoring and graph isomorphism are exceptions.

3-SAT is NP-complete.

Theorem 6. 3-SAT is NP-complete.

Proof. It suffices to show that SAT $\leq_p$ 3-SAT since we know that 3-SAT is in NP.Let Φ be any SAT-formula and let $C = (L_1 \vee L_2 \vee \cdots \vee L_k)$ be a clause of size $k > 3$ ($L_i = x_i$ or $L_i = \overline{x_i}$). Introduce a new variable z and replace C with

$$C' = (L_1 \vee L_2 \vee \cdots \vee L_{k-2} \vee z) \text{ and } C'' = (L_{k-1} \vee L_k \vee \overline{z}).$$

Doing this will transform all clauses into clauses of at most 3 literals. Finally we turn the clauses of length < 3 into ones of length 3. □

Subset-Sum is NP-complete. SUBSET-SUM: Given natural numbers $w_1, \ldots, w_n$ and an integer W, is there a subset that adds up exactly to W?

Example.

$$\{1, 4, 16, 64, 256, 1040, 1041, 1093, 1284, 1344\}, \ W = 3754.$$

is a yes instance as

$$1 + 16 + 64 + 256 + 1040 + 1093 + 1284 = 3754.$$

Remark. With arithmetic problems, input integers are encoded in binary. Polynomial reduction must be polynomial in the size of the *binary* encoding.

Claim 8. 3-SAT $\leq_p$ SUBSET-SUM.

Proof. Given an instance Φ of 3-SAT, we construct an instance of SUBSET-SUM that has a solution if and only if Φ is satisfiable.

Given a 3-SAT instance Φ with n variables and k clauses, form $2n+2k$ decimal integers, each of $n+k$ digits. For each variable x_i we include two decimal integers with 1 in the $(n+k-1-i)$th digit, the first with 1s in the last decimal digits corresponding to the clauses x_i appears in and the second with 1s for the clauses where $\overline{x_i}$ appears. We then add in $2k$ slack variables with either 1 or 2 for the digit corresponding to each clause and 0s elsewhere, this will allow us to always reach the target value with 4s in the last k digits and 1s in the first n as long as there is an assignment of each variable to a true or false value such that each clause has at least one literal true. □

Scheduling With Release Times is NP-complete. SCHEDULE-RELEASE-TIMES: Given a set of n jobs with processing time t_i, release time r_i and deadlines d_i, is it possible to schedule all jobs on a single machine such that job i is processed with a contiguous time slot of t_i time units in the interval $[r_i, d_i]$?

Claim 9. SUBSET-SUM $\leq_p$ SCHEDULE-RELEASE-TIMES.

Proof. Given an instance of subset sum with set $\{w_1, \ldots, w_n\}$ and target sum W we create n jobs with $t_i = w_i$, release time 0 and deadline $d_i = 1 + \sum_j w_j$. Now create another job taking 1 unit of time and being released at time W with deadline $W + 1$. □

Hamiltonian cycle is NP-complete. HAM-CYCLE: Given an undirected graph $G = (V, E)$ does there exist a simple cycle Γ that contains every node in V?

The answer is no for a bipartite graph with an odd number of nodes.

DIR-HAM-CYCLE: Given a directed graph $G = (V, E)$ does there exist a simple directed cycle Γ that contains every node in V?

Claim 10. DIR-HAM-CYCLE $\leq_p$ HAM-CYCLE.

Proof. Given a directed graph $G = (V, E)$ construct an undirected graph G' with $3n$ nodes by replacing each node v with 3 nodes v_i, v and v_o such that all edges (v, w) now go from v_o to w_i and so that v_i, v and v_o are connected in order by edges.

If there were a directed Hamiltonian cycle in G then traversing the vertices in the same order gives a Hamiltonian cycle in G' by visiting the vertices in the same order.

Conversely if we have a Hamiltonian cycle in G' we can see that the cycle must either go from a o vertex to a normal one to an i vertex and repeat this, or do the reverse, in either case the sequence of nodes without subscripts is either a Hamiltonian cycle in G or the reverse of one. □

Claim 11. 3-SAT $\leq_p$ DIR-HAM-CYCLE.

Proof. We want to create a graph that has 2^n Hamiltonian cycles that correspond to truth assignments of variables for the 3-SAT instance. If the 3-SAT instance has k clauses then we create n rows of $3k + 3$ nodes each, connected bidirectionally, we then create a source node and end node connecting the source the each end of the first row and then two edges going to both ends of the next row and the same to the third. Next we connect both end nodes of the last row to the sink node and connect this to the source allowing for 2^n Hamiltonian cycles going from the start and then picking the direction to traverse each row. □

TSP is NP-complete. TSP: Given a set of n cities and a pairwise distance function $d(u, v)$, is there a tour of length $\leq D$.

Claim 12. HAM-CYCLE $\leq_p$ TSP.

Proof. Given an instance $G = (V, E)$ of HAM-CYCLE create n cities with distance function defined by

$$d(u, v) = \begin{cases} 1, & \text{if } (u, v) \in E, \\ 2, & \text{if } (u, v) \notin E. \end{cases}$$

This TSP instance has a tour of length $\leq n$ if and only if G is Hamiltonian. □

3-Colouring is NP-complete. 3-COLOUR: Given an undirected graph G, does there exist a colouring of the vertices with three colours such that no two adjacent vertices have the same colour?

REGISTER-ALLOCATION: Is there an assignment of program variables to no more than k machine registers such that no two program variables that are used at the same time are assigned to the same register?

We can form a graph to study this problem by making nodes for variables and edges if there is an operation that uses both variables at the same time. We can then observe that we can solve the register allocation problem if and only if this graph is k-colourable.

Fact. 3-COLOUR $\leq_p$ k-REGISTER-ALLOCATION.

Claim 13. 3-SAT $\leq_p$ 3-COLOUR.

Proof. Given 3-SAT instance we create a node for each literal. Then we create 3 new nodes T, F and B connected in a triangle and connect each literal to B. Next we connect each literal to its negation. $\square$

Planar 3-Coloring is NP-complete. PLANAR-3-COLOUR: Given a planar map, can it be coloured using 3 colours such that no two adjacent regions have the same colour?

Claim 14. 3-COLOUR $\leq_p$ PLANAR-3-COLOUR.

Proof. Given an instance of 3-COLOUR draw the graph in the plane (with edges crossing if necessary), then replace the edge crossings with a gadget made of edges and vertices with a gadget such that the colours the vertices are preserved. $\square$

Planar k-Coloring. PLANAR-2-COLOUR is solvable in linear time, whereas PLANAR-3-COLOUR is NP complete note that PLANAR-4-COLOUR is solvable in $O(1)$ time as the answer is always yes.

Oracle Turing Machines.

Definition 13. An *Oracle machine* OM is a Turing machine with a special extra tape called the *oracle tape* and three extra states, q_{ask}, q_{yes} and q_{no}. OM computes as normal but with access to an oracle function $f\colon \{0,1\}^* \to \{0,1\}$. When the machine OM enters the state q_{ask} the state changes to q_{yes} if f of the oracle tape is 1 and q_{no} otherwise. The output of the machine OM on input x when given access to the oracle f is denoted $\text{OM}^f(x)$.

Cook-reductions.

Definition 14. A problem f is *Cook reducible* to a problem g if there exists an oracle machine M which with access to the oracle g computes f.

Definition 15. A problem f is *polynomial time* Cook reducible to a problem g if there exists a polynomial $p(n)$ and an oracle machine M which with access to the oracle g computes f in time $p(n)$.

We write this as

$$f \leq_p^C g.$$

Self-Reducibility. Note the difference between search and decision problems, e.g. determining if there exists a vertex cover of size $\leq k$ is different to actually finding one of smallest size.

A problem is said to be self reducible if the search problem $\leq_p^C$ the decision problem. This concept applies to all the problems we consider and thus justifies our focus on decision problems.

Example. To find a minimum cardinality vertex cover we perform a (binary) search for the cardinality of the minimum cover k. We then find a vertex v such that $G \setminus v$ has a vertex cover of size $\leq k-1$ any vertex in the minimum vertex cover will have this property so we remove v from the graph and recursively find a cover for the smaller graph.

Asymmetry of NP. NP is asymmetric, we only need short proofs of yes instances.

Example. SAT vs. TAUTOLOGY. In TAUTOLOGY we ask if a given boolean formula is true for any truth assignment. We can prove a boolean formula is satisfiable by giving a truth assignment, but proving that any assignment satisfies is different.

Example. HAM-CYCLE vs. NO-HAM-CYCLE: We can prove a graph is Hamiltonian by describing a Hamiltonian cycle, but how can we prove that there isn't one?

SAT is NP-complete, how do we classify TAUTOLOGY.

NP versus coNP.

Definition 16. Given a decision problem X its *complement* $\overline{X}$ is the same problem with yes and no answers reversed.

So we define the complexity class coNP to be the set of problems whose complements are in NP.

Example. TAUTOLOGY, NO-HAM-CYCLE and PRIMES are all in coNP.

It is a fundamental question whether NP = coNP, if so all yes instances of a problem have polynomial time certificates iff all no instances do. The consensus opinion is that the answer to this question is no.

Some properties of NP and coNP.

Claim 15. NP is not a proper subset of coNP.

Proof. Assume there is a problem X in coNP that is not in NP. Its complement $\overline{X}$ is in NP by definition. Now $\overline{X}$ is in coNP as we are assuming NP $\subset$ coNP and hence X must be in NP, but this is a contradiction. □

Claim 16. P $\subseteq$ coNP.

Proof. If a problem X is in P, then so is its complement as we can decide the problem in polynomial time and then invert the output. As $\overline{X}$ is then in P too it must be in NP and hence X is in coNP. □

As P is closed under complement if we had P = NP then P = NP = coNP.

Similarly to the definition for NP we say a problem X is coNP-complete if $X \in$ coNP and any other problem in coNP can be reduced to X with a polynomial time reduction.

Claim 17. If an NP-complete problem lies in NP $\cap$ coNP then NP = coNP.

Proof. Suppose X is NP-complete and in NP $\cap$ coNP. Then take a problem Y from NP and as $Y \leq_p X$ hence Y is in both NP and coNP too and so NP $\subseteq$ NP $\cap$ coNP. □

Well characterized problems. We notice that if a problem X is in both NP and coNP then there should be short certificates for yes instances and short disqualifiers for no instances. We call this a good characterisation or say that the problem is well characterised.

Example. When deciding if there exists a perfect matching in a bipartite graph we can exhibit such a matching if one exists otherwise we could give a set of nodes whose neighbourhood is less than the size of the set, showing that no such matching can exist.

Observe that $\text{P} \subset \text{NP} \cap \text{coNP}$, sometimes finding a good characterisation seems easier than finding an efficient algorithm.

Another fundamental question is therefore whether $\text{P} = \text{NP} \cap \text{coNP}$? There are mixed opinions on this, many problems were found to have good characterisations and many years later found to actually be in P, e.g. linear programming and primality testing. There are still problems such as FACTOR which are in $\text{NP} \cap \text{coNP}$ but are not known to be in P.

PRIMES is in NP.

Theorem 7. PRIMES is in $\text{NP} \cap \text{coNP}$.

Proof. We already know that PRIMES is in coNP, so it remains to show that PRIMES is in NP. For this we use Pratt's theorem which states that an odd integer s is prime iff there exists an integer $1 < t < s$ such that

$$t^{s-1} \equiv 1 \pmod{s},$$
$$t^{(s-1)/p} \not\equiv 1 \pmod{s} \text{ for all primes } p \mid (s-1).$$

We can therefore use the prime factorisation of $s-1$ as a certificate that s is prime, note that this certificate needs to be recursive, all primes in the factorisation of $s-1$ need to be proved prime too, nevertheless this is still small enough! To certificate quickly a good method of powering is needed such as powering by repeated squaring. □

FACTOR is well characterized. FACTORISE: Given an integer x, find its prime factorisation.

FACTOR: Given two integers x and y does x have a non-trivial factor of size at most y.

Theorem 8. FACTORISE $\leq_p^C$ FACTOR.

Theorem 9. FACTOR is in $\text{NP} \cap \text{coNP}$.

Proof. We can use a factor p of x that is less than y as a certificate. And for a disqualifier we can take the prime factorisation of x as each prime factor of x must be larger than y in a no instance. □

We have now established that PRIMES $\equiv_p^C$ COMPOSITES $\leq_p$ FACTOR. Does FACTOR $\leq_p^C$ PRIMES? The current state of the art is that PRIMES is in P, but FACTOR is not believed to be in P.

PSPACE.

Definition 17. The complexity class PSPACE is the set of decision problems that are solvable using at most polynomial *space*.

Note that $\text{P} \subseteq \text{PSPACE}$ as a polynomial time algorithm can use at most polynomial space.

Claim 18. 3-SAT is in PSPACE.

Proof. We can enumerate all possible truth assignments by counting in binary from 0 to $2^n - 1$ using at most n bits. Then we simply check each assignment in turn to see if it satisfies all clauses. □

Theorem 10. NP $\subseteq$ PSPACE.

Proof. Consider some Y in NP, as $Y \leq_p 3-SAT$ we can decide whether $w \in Y$ by computing a function f in polynomial time along with deciding if $f(w) \in$ 3-SAT. We can decide if $f(w) \in$ 3-SAT in polynomial space. □

QSAT is in PSPACE. QSAT: Let $\Phi(x_1, \dots, x_n)$ be a boolean formula in CNF, is the propositional formula

$$\exists x_1 \forall x_2 \exists x_3 \cdots \forall x_{n-1} \exists x_n \Phi(x_1, \dots, x_n)$$

true?

We can think about this question by imagining two players playing a game, one picking truth values for x_i when i is odd and the other doing even is and asking if the first player can force the formula to be satisfied.

Example. For

$$(x_1 \vee x_2) \wedge (x_2 \vee \overline{x_3}) \wedge (\overline{x_1} \vee \overline{x_2} \vee x_3)$$

the answer is yes as the first player can set x_1 true and then setting x_3 to be the same as whatever x_2 was set to.

Example. For

$$(x_1 \vee x_2) \wedge (\overline{x_2} \vee \overline{x_3}) \wedge (\overline{x_1} \vee \overline{x_2} \vee x_3)$$

the answer is no as the second player can set x_2 to be whatever x_1 was set to and then the formula is not satisfiable.

Theorem 11. QSAT is in PSPACE.

Proof. We can use an algorithm that recursively tries all possibilities to answer the question, we only need one bit of information from each subproblem and so the amount of space used is proportional to the depth of the function call stack, which is the same as the number of variables. The algorithm for the for the subproblems where we have a for all first should return true if and only if both subproblems are true, whereas for the subproblems with a there exists first true should be returned when at least one subproblem returned true. □

Definition 18. A problem Y is called *PSPACE-complete* if it is in PSPACE and every other problem X in PSPACE we have $X \leq_p Y$.

Theorem 12 (Stockmeyer-Meyer 1973)**.** QSAT is PSPACE-complete.

PSPACE is a subset of EXPTIME.

Theorem 13. PSPACE $\subseteq$ EXPTIME.

Proof. The algorithm described for QSAT above runs in exponential time and QSAT is PSPACE-complete. □

To summarise, we now have

$$\text{P} \subseteq \text{NP} \subseteq \text{PSPACE} \subseteq \text{EXPTIME}.$$

It is known that P $\neq$ EXPTIME but not known which of the above inclusions is strict. It is conjectured that all are.

Competitive Facility Location is PSPACE-complete. For the competitive facility location problem we are given a graph with positive node weights and a target weight B. Two competing players then alternate selecting nodes and they are not allowed to select a node if any of its neighbours has already been selected by either player. The problem then asks if the first player can prevent the second from selecting nodes weighing in total at least B.

Claim 19. COMPETITIVE-FACILITY is PSPACE-complete.

Proof. To solve the problem in polynomial space we can use a recursive algorithm as in QSAT above, but this time we have at most n choices at each stage rather than just 2.

To see completeness we show that QSAT reduces to it in polynomial time we construct a COMPETITIVE-FACILITY instance that is a yes instance if and only if a given QSAT formula is true. Let $\Phi(x_1, \dots, x_n) = C_1 \wedge C_2 \wedge \cdots \wedge C_k$ be the formula for a QSAT instance with n odd. Include a node in the COMPETITIVE-FACILITY graph for each literal and its negation and connect them so at most one of each x_i and its negation can be selected by a player. Let $c = k + 2$ an give variable x_i the weight c^i and the same weight for its negation. We then set the target weight B to be $c^{n-1} + \cdots + c^4 + c^2 + 1$. This ensures that the variables are selected in the order $x_n, x_{n-1}, \dots, x_1$. If we stopped here player 2 will always lose by one unit, so we add in nodes for each clause that have weight 1 and are connected to each of the literals in the clause. The second player can now make a final move if and only if the truth assignment defined by the players turns fails to satisfy some clause. □

RP.

Definition 19. A language L is in the class RP (*randomised polynomial time*) if there exists a Turing machine M and polynomials T and p such that:

- For each input x the machine M terminates after at most $T(|x|)$ steps.
- If $x \in L$ then
$$\Pr_{t \in \{0,1\}^{p(|x|)}} [M \text{ accepts } \langle x, t \rangle] \geq \frac{1}{2}.$$
- If $x \notin L$ then
$$\Pr_{t \in \{0,1\}^{p(|x|)}} [M \text{ rejects } \langle x, t \rangle] = 1.$$

This is very similar to NP but at least half of all possible certificates must make the verifier accept. We never get false positives with this set up but we may get false negatives.

We can define this class in an alternate but equivalent manner by using a *probabilistic* Turing machine that can generate a random string itself. A probabilistic Turing machine can (in addition to writing 0, 1 or $\square$) write a symbol that is either 0 or 1 with probability 1/2.

Claim 20. $\mathrm{P} \subseteq \mathrm{RP}$.

Proof. Consider a problem X in P. As we have that there is a polynomial time Turing machine that decides X we can use make a verifier that ignores the string t in the input and just decides if x is in the language. This satisfies the definition for RP. □

Claim 21. $\mathrm{RP} \subseteq \mathrm{NP}$.

Proof. The definition for NP can be seen to be the same as that of RP but where we have the line

If $x \in L$ then

$$\Pr_{t \in \{0,1\}^{p(|x|)}}[M \text{ accepts } \langle x, t\rangle] \geq \frac{1}{2}.$$

replaced with

If $x \in L$ then

$$\Pr_{t \in \{0,1\}^{p(|x|)}}[M \text{ accepts } \langle x, t\rangle] > 0.$$

So a machine satisfying the first condition satisfies the latter too. □

Probability Amplification. Using the technique of probability amplification we can see that the 1/2 term in

If $x \in L$ then

$$\Pr_{t \in \{0,1\}^{p(|x|)}}[M \text{ accepts } \langle x, t\rangle] \geq \frac{1}{2}.$$

can be replaced with any other constant, or even a term of the form $1/|x|^c$ for a constant c. We can dramatically decrease the chance of false negatives without changing the complexity class.

To achieve this we define a new Turing machine M' that takes in ts that are $|x|$ times longer by splitting the input t into $|x|$ chunks and running M on x along with each of these chunks in turn. We run M a total of $|x|$ times and accept if M accepts at least once (as we have no false positives). Now our probabilities are

$$\text{If } x \in L \text{ then } \Pr_{t \in \{0,1\}^{p(|x|)}}[M \text{ accepts } \langle x, t\rangle] \geq \frac{1}{2}.$$

$$\text{If } x \notin L \text{ then } \Pr_{t \in \{0,1\}^{p(|x|)}}[M \text{ rejects } \langle x, t\rangle] = 1.$$

coRP.

Definition 20. The complexity class coRP $= \{X \mid \overline{X} \in \text{RP}\}$.

Alternatively we can say a language L is in coRP if there exists a Turing machine M and polynomials T and p such that:

- For each input x the machine M terminates after at most $T(|x|)$ steps.
- If $x \in L$ then
$$\Pr_{t \in \{0,1\}^{p(|x|)}}[M \text{ accepts } \langle x, t\rangle] = 1.$$
- If $x \notin L$ then
$$\Pr_{t \in \{0,1\}^{p(|x|)}}[M \text{ rejects } \langle x, t\rangle] \geq \frac{1}{2}.$$

This is completely analogous to RP and so we have P $\subseteq$ coRP $\subseteq$ coNP. It is unknown whether RP = coRP.

Polynomial equality. The polynomial equality problem asks if two polynomials P and Q are the same, that is, whether they agree on all inputs.

Example. Does

$$P(x,y) = x^6 + y^6$$

equal

$$Q(x,y) = (x^2+y^2)(x^4+y^4-(xy)^2)?$$

We might try and answer this by expanding both polynomials and comparing terms, but there could be exponentially many terms in the expansion compared to the input size. This problem is not known to be in P, however we can reduce it to another problem by observing that it is enough to determine whether $P - Q$ is the zero polynomial.

Polynomial identity testing. POLY-ID: Given a polynomial Q in some encoding that has degree d is this polynomial identically zero.

Claim 22. POLY-ID $\in$ coRP.

Proof. To keep things simple we only consider univariate polynomials over the real numbers. If Q is non-zero there are at most d distinct values x for which $Q(x)$ is zero. So if Q is not identically zero there and we pick some x at random from $\{1,\dots,2d\}$ then $\Pr[Q(x)=0] \leq \frac{1}{2}$. We need $\log d$ random bits to pick this x randomly. Evaluating the polynomial $Q(x)$ could actually take exponential time, however as our x is now an integer we can fix this by doing computations modulo some sufficiently large prime p. We then accept if and only if $Q(x) = 0$, so if Q is identically zero we always accept and if it is not we reject with probability $\geq \frac{1}{2}$. $\square$

BPP. For some problems we might need two sided errors (both false positives and false negatives).

Definition 21. A language L is in BPP (*bounded error probabilistic polynomial time*) if there exists a Turing machine M and polynomials T and p such that:

- For each input x the machine M terminates after at most $T(|x|)$ steps.
- If $x \in L$ then
$$\Pr_{t\in\{0,1\}^{p(|x|)}}[M \text{ accepts } \langle x,t\rangle] \geq \frac{2}{3}.$$
- If $x \notin L$ then
$$\Pr_{t\in\{0,1\}^{p(|x|)}}[M \text{ rejects } \langle x,t\rangle] \geq \frac{2}{3}.$$

So we can decide L with probability of error at most $\frac{1}{3}$

We can see that P $\subseteq$ RP $\subseteq$ BPP and that P $\subseteq$ coRP $\subseteq$ BPP. Since this definition is symmetric we have that BPP = coBPP, where coBPP is defined in the usual way. It is not known whether BPP $\subseteq$ NP or NP $\subseteq$ BPP or neither!

Probability Amplification for BPP.

Lemma 1 (Amplification lemma)**.** Let $0 < \epsilon_1 < \epsilon_2 < \frac{1}{2}$ then there is a polynomial time probabilistic Turing machine M_1 which decides L with error probability ϵ_1 if and only if there is a poly-time PTM M_2 which decides L with error probability ϵ_2.

Proof. ($\Rightarrow$) Since $\epsilon_1 < \epsilon_2$, M_1s error probability is also bounded by ϵ_2.
($\Leftarrow$) M_1 runs M_2 a total of $2k+1$ times and chooses the majority result. M_1 is correct if M_2 is correct at least $k+1$ times. Each of the runs is an independent Bernoulli trial in which M_2 is correct with probability at least $1-\epsilon_2 > \frac{1}{2}$. The Chernoff bound then says that

$$\Pr[M_1 \text{ is incorrect}] \leq e^{-k\left(\frac{1}{2}-\epsilon_2\right)^2}$$

so if we take

$$k > \frac{\log_e\left(\frac{1}{\epsilon_1}\right)}{\left(\frac{1}{2}-\epsilon_2\right)^2}$$

the error probability is less than ϵ_1. □

ZPP. What about if we didn't want any errors? We could allow the Turing machine to either accept, reject or output that it doesn't know.

Definition 22. A language L is in ZPP (*zero error probabilistic polynomial time*) if there exists a Turing machine M and polynomials T and p such that:

- For each input x the machine M terminates after at most $T(|x|)$ steps.
- If $x \in L$ then
$$\Pr_{t\in\{0,1\}^{p(|x|)}}[M \text{ accepts } \langle x,t\rangle \text{ or returns dunno}] = 1.$$
- If $x \notin L$ then
$$\Pr_{t\in\{0,1\}^{p(|x|)}}[M \text{ rejects } \langle x,t\rangle \text{ or returns dunno}] = 1.$$
- $$\Pr_{t\in\{0,1\}^{p(|x|)}}[M \text{ returns dunno on } \langle x,t\rangle] \leq \frac{1}{2}.$$

As above for RP the choice of constant 1/2 here is not significant. We have that P $\subseteq$ ZPP by the same arguments as before and as this definition is symmetric we have ZPP = coZPP.

ZPP = RP $\cap$ coRP.

Claim 23. ZPP = RP $\cap$ coRP.

Proof. We prove both inclusions individually, starting with ZPP $\subseteq$ RP $\cap$ coRP.

Looking at the definitions of ZPP and RP we note that we can turn a machine that satisfies the requirements for ZPP into one for RP by replacing the output dunno with reject. Therefore any language in ZPP is also in RP. We can make the same argument for coRP by replacing dunno with accept and so we can conclude that ZPP $\subseteq$ RP $\cap$ coRP.

To see the reverse inclusion we make a machine M_1 satisfying the definition for ZPP out of two other machines M_2 and M_3 that are machines for a language that satisfy the definitions of RP and coRP respectively. We run M_2 first accepting if it does, we then run M_3, rejecting if it does. If neither of these things happen we output dunno and so as M_2 accepted with probability $\geq \frac{1}{2}$ and M_3 rejected with probability $\geq \frac{1}{2}$ our new machine M_1 outputs dunno with probability $\leq \frac{1}{2}$ and so satisfies the requirements of ZPP as it never gives an incorrect result. So we have ZPP $\supseteq$ RP $\cap$ coRP and can conclude that ZPP = RP $\cap$ coRP. □

ZPP as expected poly-time. We can define ZPP in an alternative way.

Definition 23. A language L is in ZPP if there exists a zero error probabilistic Turing machine that computes L in *expected* polynomial time.

To transform a machine satisfying the old definition to one satisfying the new one we can rerun it until it either accepts or rejects. The new machine either accepts or rejects without error and never returns dunno. In expectation the machine will run twice so as the original machine ran in polynomial time the new one runs in expected polynomial time.

To go from a new machine to the old we run the new one for at most some polynomial number of steps, cutting it off and returning dunno if it takes too long.

Summary:

- RP - bounded error, false negatives but no false positives.
- coRP - bounded error, false positives but no false negatives.
- BPP - bounded error, false negatives and false positives.
- ZPP - no errors, but may answer dunno with bounded probability.

IP. In an interactive proof the prover writes down a sequence of symbols which the verifier can check, only true statements can be proved.

Definition 24. A *k-round interaction* between two functions f and g on an input x is a sequence of strings (called *the transcript*) $a_1, \ldots, a_k$ defined by

$$\begin{aligned}
a_1 &= f(\langle x, r\rangle) \\
a_2 &= g(\langle x, a_1\rangle) \\
a_3 &= f(\langle x, r, a_1, a_2\rangle) \\
&\vdots \\
a_{2n+1} &= f(\langle x, r, a_1, \ldots, a_{2i}\rangle) \\
a_{2n+2} &= g(\langle x, a_1, \ldots, a_{2i+1}\rangle) \\
&\vdots
\end{aligned}$$

where r is a random bitstring of length $\mathrm{poly}(|x|)$. The output is then $\mathrm{out}_{f,g}(x) = f(\langle x, r, a_1, \ldots, a_k\rangle)$. We assume this output is either 0 or 1. Since r is random, the transcript and output can be random too.

Definition 25. A language L is in the class IP is there exists a $\mathrm{poly}(|x|)$-round interaction between a function V (verifier) and P (prover) such that

- If $x \in L$ then there exists P such that $\Pr[\mathrm{out}_{V,P}(x) = 1] \geq \frac{2}{3}$ (completeness).
- If $x \notin L$ then for all P $\Pr[\mathrm{out}_{V,P}(x) = 0] \geq \frac{2}{3}$ (soundness).

Additionally we require that V be computable in time $\mathrm{poly}(|x|)$.

As with BPP we can use probability amplification to reduce the error probability. Changing the constant $2/3$ in completeness to 1 does not change the class IP. Changing the constant in soundness to 1 rather than $2/3$ results in the class NP (exercise). As only the verifier is computationally bounded the prover can use far more computational resources and this is crucial to the class.

IP protocol for Graph-Non-Isomorphism. GRAPH-NON-ISOMORPHISM: Are two given graphs G_1 and G_2 *not* isomorphic.

Claim 24. GRAPH-NON-ISOMORPHISM is in IP.

Proof. First the verifier generates another graph H by permuting node and edge labels of one of G_1 or G_2. The verifier then asks the prover which of G_1 or G_2 the graph H was generated from, if the graphs are isomorphic the prover has no way to tell other than guessing and gets it wrong with probability 1/2. If however the graphs are not isomorphic the prover can work out which of G_1 and G_2 the graph H came from. So if the prover gets it right the verifier answers yes otherwise the verifier answers no. $\square$

Here we use the fact probability amplification can be used to decrease the error probability. Graph isomorphism is not known to be in NP. Here the prover needs to be able to solve graph isomorphism which is not known to be in P, but the verifier does run in polynomial time.

dIP = NP. If in the definition of IP we do not allow randomization the resulting class, dIP, is actually equal to NP. This is as the prover can anticipate all the questions a deterministic verifier could ask and give all the answers in a single certificate after one step.

coNP is contained in IP, arithmetization, and the sumcheck protocol. We now show that coNP $\subseteq$ IP by showing NON-SATISFIABILITY $\in$ IP, this works as NON-SATISFIABILITY is coNP-complete.

To make an interactive proof that a formula Φ is not satisfiable we could instead make an interactive proof that the *number* of satisfying assignments is exactly K, the naive way of implementing this method does not work so well however as the verifier could only catch the prover lying with probability $1/2^n$. So we arithmetize the problem by turning the boolean formula into a polynomial over $\mathbb{F}_p$ for some prime p between 2^n and 2^{2n}. We take variables X to $1 - X$ and $\overline{X}$ to X, then a clause of literals $V_1 \vee V_2 \vee V_3$ goes to $1 - V_1V_2V_3$ and we multiply all clauses together. The resulting polynomial P_Φ is of degree $3m$, where m is the original number of clauses. We then have that the number of satisfying assignments is

$$\sum_{b_1\in\{0,1\}} \sum_{b_2\in\{0,1\}} \cdots \sum_{b_n\in\{0,1\}} P_\Phi(b_1, \ldots, b_n).$$

We have now reduced to a different problem for which there is a good interactive proof.

Theorem 14. Given a degree $d = \text{poly}(n)$ polynomial $g(X_1, \ldots, x_n)$ over $\mathbb{F}_p$ for some prime $p \in [2^n, 2^{2n}]$ and an integer K there is an interactive proof for

$$K = \sum_{b_1\in\{0,1\}} \sum_{b_2\in\{0,1\}} \cdots \sum_{b_n\in\{0,1\}} g(b_1, \ldots, b_n) \pmod{p}.$$

Proof. We first define a univariate degree d polynomial by

$$h(X_1) = \sum_{b_2\in\{0,1\}} \cdots \sum_{b_n\in\{0,1\}} g(X_1, b_2, \ldots, b_n).$$

If $n = 1$ the verifier checks that $g(0) + g(1) = K$, if so the verifier accepts, otherwise it rejects. If however $n > 1$ the verifier asks the prover to send $h(X_1)$. $\square$

IP = PSPACE.

Theorem 15 (Shamir 1990)**.** IP = PSPACE.

Proof idea. IP $\subseteq$ PSPACE: show an appropriate prover can be found in polynomial space. PSPACE $\subseteq$ IP: Show QSAT $\in$ IP. $\square$

Program checking. One application of interactive proofs is on the fly error checking for programs, in this scenario the program is the prover and the verifier checks in real time that the output is correct. As the verifier normally uses less resources than the prover this can typically be done with small overhead.

Zero knowledge proofs. Zero knowledge proofs are special interactive proofs, ones that can give the verifier no additional information, other than the fact the statement is true.

Definition 26. An interactive proof for L is *zero-knowledge* if for any verifier V^* there is an expected polynomial time Turing machine S, called the simulator, that if $x \in L$ can produce the entire transcript of the interaction between P and V^* *without* access to P.

As the transcripts are randomised we want the transcripts produced by the simulator to have the same distribution as the actual transcripts.

Zero knowledge proof protocol for graph isomorphism.

Communication Complexity. In communication complexity we have two people (Alice and Bob) who each know some bits of data (x and y) and wish to compute some function of both pieces of data ($f(x, y)$). They must communicate following a prearranged protocol which determines who communicates when and what they send, the cost of a given protocol is the *worst case* number of bits that they exchange.

Definition 27. The *communication complexity* of a given function f is then the cost of the *best* protocol for f.

Computing OR. Say Alice has $x = x_1x_2 \cdots x_n$, Bob has $y = y_1y_2 \cdots y_n$ and they wish to compute

$$f(x, y) = x_1 \vee x_2 \vee \cdots \vee x_n \vee y_1 \vee y_2 \vee \cdots \vee y_n.$$

One protocol is for Alice to send x to Bob then for Bob to compute $z = f(x, y)$ and send the result to Alice. The cost of this protocol is therefore $n + 1$ bits.

In fact the natural extension of this approach shows that for any $f\colon X \times Y \to Z$ we have $D(f) \leq \lceil \log |X| \rceil + \lceil \log |Z| \rceil$.

However for this function we can do better by having Alice compute $g(x) = x_1 \vee \cdots \vee x_n$ and send the result to Bob who can then compute the result $f(x, y) = g(x) \vee y_1 \vee \cdots \vee y_n$ and sends it to Alice. The cost of this protocol is 2 bits.

Computing MEDIAN. Suppose Alice has $x \subseteq \{2, 4, \ldots, 2n\}$ and Bob has $y \subseteq \{1, 3, \ldots, 2n-1\}$ together they wish to compute the median of $x \cup y$.One protocol is for Alice to send her subset to Bob (n bits) and for Bob to find the median and send it back to Alice ($\lceil \log 2n \rceil$ bits).

This however can be improved by having both Alice and Bob maintain an interval $[i, j] \subset [1, 2n]$ which they are sure contains the median $f(x, y)$. First we let $k = (i + j)/2$, Alice then computes $L_x = |[i, k] \cap x|$ and $R_x = |[k + 1, j] \cap x|$, similarly Bob computes $L_y = |[i, k] \cap y|$ and

$R_y = |[k+1, j] \cap y|$. Alice and Bob then exchange these numbers and so Alice and Bob can each decide whether $f(x, y)$ is in $[i, k]$ or in $[k+1, j]$ and then update the interval they are looking at because of this. They repeat this until the interval they have contains only one element, the median.

Each iteration of this involves transmitting $O(\log n)$ bits cuts the interval in half so the total cost of this protocol is $O((\log n)^2)$.

Protocol Trees.

EQUALITY. For equality Alice and Bob have $x = x_1 \cdots x_n$ and $y = y_1 \cdots y_n$ respectively and they wish to compute $f(x, y)$ which is 1 if and only if $x = y$. We can do this by transferring $n+1$ bits, Alice sends x to Bob who computes $f(x, y)$ and sends it back. How close is this to the communication complexity of this protocol?

Claim 25. *$D(f) \geq n$.*

Proof. Suppose there is a protocol for f that uses at most $n-1$ bits. There are then at most 2^{n-1} different transcripts. Hence there must be two inputs (a, a) and (b, b) with the same transcript. We then must have that (a, b) and (b, a) have the same transcript again, but this is a contradiction as the result of (a, a) is different to that of (a, b). □

Deciding Palindromes requires quadratic time. Let PAL be the language of all $x \in \{0, 1\}^*$ that are palindromes. We can decide this language in time $O(n^2)$ using a simple Turing machine.

Claim 26. PAL cannot be decided in under time $O(n^2)$ by a Turing machine.

Proof. Assume we have a Turing machine for PAL that leaves some cell i in the range $n/3$ to $2n/3$ (where cells of the tape are indexed starting from the far left) at most k times, we can use this Turing machine to find a $O(k)$ bit communication protocol for EQUALITY. To do this we write x in the first $n/3$ cells and y in reverse in the cells from $2n/3$ to n and fill the gap in the middle with zeroes. For the protocol Alice will simulate the Turing machine on cells 0 to i and Bob simulates the Turing machine on $i+1$ to n. When the head of the machine crosses between cell i and cell $i+1$ Alice and Bob communicate to hand over the simulation, this takes $O(1)$ bits each time.

We know that EQUALITY requires $\Omega(n)$ communication and so k must be $\Omega(n)$. Hence every Turing machine for PAL must leave each cell in the middle range at least $\Omega(n)$ times and so these Turing machines must take $\Omega(n^2)$ steps. □

Protocol trees and combinatorial rectangles.

EQUALITY (again).

DISJOINTNESS.

INNER PRODUCT and the rank technique.

Area-Time tradeoffs for VLSI chips.

NP-hard. We now look at how we can express the complexity of non-decision problems.

Definition 28. A problem A is NP-*hard* if for every $L \in \text{NP}$, L reduces (via a cook reduction) to A in polynomial time.

Example. Finding a minimum vertex cover, finding a Hamiltonian cycle and the halting problem are all NP-hard.

Note that an NP-hard problem can be much harder than anything in NP.

Approximation Algorithms. If we want to solve NP-hard problems we probably have to sacrifice one of three things:

1. Solving the problem optimally.
2. Solving the problem in polynomial time.
3. Solving arbitrary instances of the problem.

We now look at what we can do if we sacrifice 1, but not 2 or 3.

Definition 29. An α-approximation algorithm for an optimization problem is a polynomial time algorithm and will find a solution that is within α ratio of the optimal solution for an arbitrary instance of the problem.

2-approximation for MAX-SAT. MAX-SAT: Find an assignment that maximises the number of satisfied clauses for a formula Φ. We denote this maximum by $M(\Phi)$.

The decision version of this problem (is $M(\Phi) > k$) is NP-complete.

Claim 27. There is a 2-approximation algorithm for MAX-SAT that finds the number of clauses satisfied by setting all variables to true and the number satisfied when all variables are false and returns the maximum of these two numbers.

Proof. Say Φ has c clauses in total. Any clause not satisfied when all variables are true must be satisfied when all variables are false. Therefore summing the number of satisfied clauses in each of these assignments must result in a number larger than c. Hence the maximum of the two numbers must be at least $c/2$ and so the algorithm given is a 2-approximation. □

2-approximation for Load Balancing (List Scheduling). In the load balancing problem we are given m identical machines and n jobs to run on them, each job j has a processing time t_j. A job must be run from start to finish on one machine and a machine can do at most one job at a time. We define the *load* of a machine to be the sum of the processing times of the jobs assigned to it. We also say that the *makespan* is the maximum load on any machine. The task is then to minimise the makespan.

There is a greedy algorithm for this problem that fixes some ordering on the jobs and then goes through the jobs one by one assigning each job to the machine that has the least load so far.

Theorem 16 (Graham, 1966)**.** The greedy algorithm is a 2-approximation.

Proof. First observe that the optimal makespan L^* must be greater than the maximum processing time for any job as this job must be processed by some machine.

We also have that the optimal makespan is at least

$$\frac{1}{m}\sum_j t_j$$

as the one machine must do at least $1/m$ times the total work.

Now consider the machine i with the largest load after the greedy algorithm has run and let j be the problem last assigned to it. When job j was assigned to machine i it must have had the least load and so we have that its load before $L_i - t_j \leq L_k$ for all $1 \leq k \leq m$. So summing these inequalities and dividing by m gives that

$$L_i - t_j = \frac{1}{m}\sum_k L_k = \frac{1}{m}\sum_{k'} t_{k'} \leq L^*.$$

And so

$$L_i = (L_i - t_j) + t_j \leq 2L^*.$$

□

This analysis is tight as using the right jobs we can make the greedy algorithm arbitrarily bad up to this limit.

3/2-approximation for Load Balancing (LPT). One improvement we can make to this greedy algorithm is called the *longest processing time* (LPT) rule, we first sort the jobs in descending order of processing time and then we run the greedy list scheduling algorithm. First note that if there are at most m jobs then list scheduling is optimal as each job can go on its own machine.

Lemma 2. If there are more than m jobs then after ordering the t_i in descending order we have $L^* \geq 2t_{m+1}$.

Proof. Consider the first $m+1$ jobs $t_1, \ldots, t_{m+1}$. As we have ordered the jobs each of these takes at least time t_{m+1}. The pidgeonhole principle then gives us that some machine has two jobs assigned to it and hence has load at least $2t_{m+1}$. □

Theorem 17. The LPT rule gives us a 3/2-approximation algorithm for load balancing.

Proof. The approach is the same as for the above approximation

$$L_i = (L_i - t_j) + t_j \leq \frac{3}{2}L^*.$$

□

However this is not tight and we have the following theorem.

Theorem 18 (Graham, 1969)**.** The LPT rule gives us a 4/3-approximation algorithm for load balancing.

This is essentially tight as we can get arbitrarily bad examples up to this limit.

$O(\log n)$-approximation for Set Cover. SET-COVER: Given a set U of elements, a collection $S_1, S_2, \ldots, S_m$ of subsets of U, find the smallest collection of sets whose union is equal to U.

A natural greedy algorithm is to repeatedly include the set containing the most uncovered elements until all are covered.

Theorem 19. This greedy algorithm is a $O(\log n)$-approximation for SET-COVER.

Proof. □

2-approximation for Vertex Cover.

PTAS and FPTAS.

Definition 30. A *polynomial time approximation scheme* (PTAS) is a family of algorithms such that for any $\epsilon > 0$ there is an algorithm A_ϵ in the family with A_ϵ a $(1+\epsilon)$-approximation algorithm.

A PTAS can give us an arbitrarily good solution but trades off accuracy for time as it does not have to be polynomial in $1/\epsilon$.

Definition 31. A *fully polynomial time approximation scheme* (FPTAS) is a PTAS where the algorithms are also required to be polynomial in $1/\epsilon$.

An FPTAS for Knapsack. In the knapsack problem we have n objects and a knapsack. Each item has some value $v_i > 0$ and weight $w_i > 0$ and we can carry up to weight W. The goal is to fill the knapsack to maximise the total value.

We can define the decision problem KNAPSACK by: Given a finite set X, nonnegative weights w_i, nonnegative values v_i, a weight limit W and a target value V is there a subset $S \subseteq X$ such that $\sum_{i \in S} w_i \leq W$ and $\sum_{i \in S} v_i \geq V$.

Claim 28. SUBSET-SUM $\leq_p$ KNAPSACK.

Proof. Given an instance $(u_1, \ldots, u_n, U)$ of SUBSET-SUM, we create a KNAPSACK instance by letting $v_i = w_i = u_i$ and $V = W = U$. □

This along with the observation that KNAPSACK is in NP gives that KNAPSACK is NP-complete.

There is a dynamic programming approach to the knapsack problem which is obtained by letting $\mathrm{OPT}(i, w)$ be the maximum value subset of items $1, \ldots, i$ with weight limit w, this gives the recurrence

$$\mathrm{OPT}(i,w) = \begin{cases} 0 & \text{if } i = 0, \\ \mathrm{OPT}(i-1,w) & \text{if } w_i > w, \\ \max\{\mathrm{OPT}(i-1,w), v_i + \mathrm{OPT}(i-1, w-w_i)\} & \text{otherwise.} \end{cases}$$

The running time of the algorithm that uses this approach is $O(n \cdot W)$ where W is the weight limit, this is not polynomial in input size!

We can try another dynamic programming method by letting $\mathrm{OPT}(i,v)$ be the minimum weight subset of items $1,\dots,i$ with value *exactly* v, this gives the recurrence

$$\mathrm{OPT}(i,v)=\begin{cases}0 & \text{if } v=0,\\ \infty & \text{if } i=0,\ v>0,\\ \mathrm{OPT}(i-1,v) & \text{if } v_i>v,\\ \min\{\mathrm{OPT}(i-1,v), w_i+\mathrm{OPT}(i-1,v-v_i)\} & \text{otherwise.}\end{cases}$$

The running time of this approach is $O(n\cdot V^*)=O(n^2 v_{\max})$ where V^* is the optimal value which is the maximum v such that $\mathrm{OPT}(n,v)\le W$. This is still not polynomial in the input size.

Linear Programming based approximation algorithm for Weighted Set Cover.