7. Complexity: P & NP

Goals of this chapter: Given a model of computation and a measure of complexity of computations, it is possible to define the **inherent complexity of a class of problems**. This is a **lower bound on the complexity of any algorithm** that solves instances of the given problem class. The model of computation considered are Turing machines, the complexity measure is time as measured by the number of transitions executed sequentially (other complexity measures, e.g. space, are only mentioned). Understand the drastic difference between deterministic and non-deterministic computation, and concepts such as P, NP, NP-complete, NP-hard. Satisfiability and other NP-complete problems.

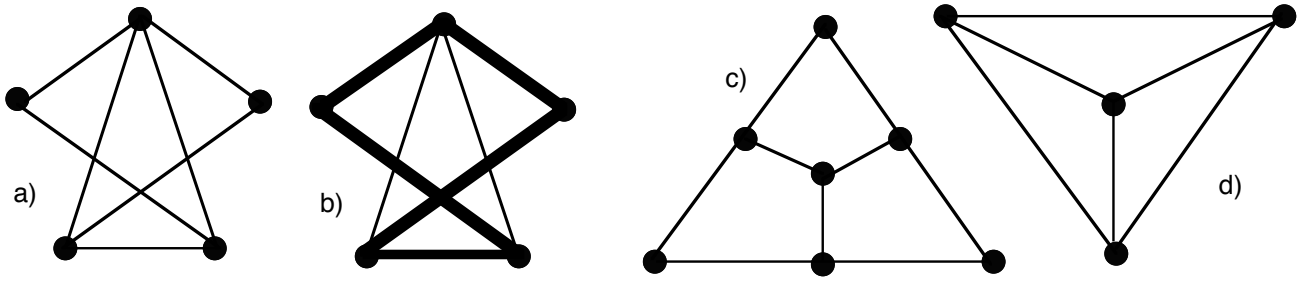
7.1 Decision problems, optimization problems: examples

Df: **Hamiltonian cycle** in a graph G = (V, E): a cycle that contains each of the vertices in V (exactly once)

Df: **Traveling salesman problem (TSP):**
Weighted graph G = (V, E, w: E -> Reals), V = {1, .., n}, E = { (i, j) | i < j } (often the complete graph Kn).
Weight (or length) of a path or cycle = sum of the weights of its edges.
Given G= (V, E, w) find a shortest Hamiltonian cycles, if any exist.

Df: A **clique** in a graph G = (V, E): a subset V' $\subseteq$ V such that for all u, v $\in$ V', (u, v) $\in$ E.
|V'| is the size of the clique. A clique of size k is called a k-clique.

Df: **Clique problems**: Given G= (V, E), answer questions about the existence of cliques, find maximum clique, enumerate all cliques.



Ex: the graph a) shown at left has exactly 1 Hamiltonian cycle, highlighted in b). The graph c) has none. The complete graph K4 has 3 Hamiltonian cycles. d) The complete graph Kn has (n-1)! / 2 Hamiltonian cycles
Ex: Graph a) has three maximum 3-cliques. The only cliques in graph c) are the vertices and the edges, i.e. the 1-cliques and the 2-cliques. Graph d) is a 4-clique, and every subset of its vertices is a clique.

As the examples above show, "basically the same" combinatorial problem often comes in different versions, some of which are optimization problems, others decision problems. The three problems: 1) "find a maximum clique", 2) "what is the size of a maximum clique", 3) "is there a k-clique for a given value k?" clearly call for similar computational techniques. Combinatorial problems in practice are usually concerned with optimization, i.e. we search for an object which is best according to a given objective function. The complexity theory of this chapter, on the other hand, is mostly concerned with decision problems. To appreciate its practical relevance it is important to understand that optimization problems and decision problems are often related: any information about one type of problem provides useful knowledge about the other. We distinguish 3 types of problems of seemingly increasing difficulty:

Decision problems:
• Given G, does G have a Hamiltonian cycle? Given G and k, does G have a k-clique?
Given G = (V, E, w) and a real number B, does G have a Hamiltonian cycle of length $\leq$ B?
Finding the answer to a decision problem is often hard, whereas verifying a positive answer is often easy: we are shown an object and merely have to verify that it meets the specifications (e.g. trace the cycle shown in b). Decision problems are naturally formalized in terms of machines accepting languages, as follows: problem instances (e.g. graphs) are coded as strings, and the code words of all instances that have the answer YES (e.g. have a Hamiltonian cycle) form the language to be accepted.

Optimization problems:
• Given G, construct a maximum clique.

• TSP: Given Kn = (V, E, w) find a Hamiltonian cycle of minimal total length.
Both problems, of finding the answer and verifying it, are usually hard. If I claim to show you a maximum clique, and it contains k vertices, how do you verify that I haven't missed a bigger one? Do you have to enumerate all the subsets of k+1 vertices to be sure that there is no (k+1)-clique? Nevertheless, verifying is usually easier than finding a solution, because the claimed solution provides a bound that eliminates many suboptimal candidates.

Enumeration problems:
• Given G, construct all Hamiltonian cycles, or all cliques, or all maximum cliques.
Enumeration problems are solved by exhaustive search techniques such as backtrack. They are time consuming but often conceptually simple, except when the objects must be enumerated in a prescribed order. Enumeration is the technique of last resort for solving decision problems or optimization problems that admit no efficient algorithms, or for which no efficient algorithm is known. It is an expensive technique, since the number of objects to be examined often grows exponentially with the length of their description.

The complexity theory of this chapter is formulated in terms of decision problems,
Whereas in practice, most problems of interest are optimization problems!

In practice, hardly any problem calls for a simple yes/no answer - almost all computations call for constructing a solution object of greater complexity than a single bit! It is therefore important to realize that the complexity theory of decision problems can be made to apply to optimization problems also. The way to do this is to associate with an optimization problem a related decision problem such that the complexity of the latter has implications for the complexity of the former. We distinguish 3 types of optimization problems:
a) **optimization problem** (in the strict sense): find an optimal solution
b) **evaluation problem**: determine the value of an optimal solution
c) **bound problem**: given a bound B, determine whether the value of an optimal solution is above or below B.

Intuitively, a solution to problem a) gives more information than b), which in turn contains more information than c). Indeed, for all "reasonable problems", an efficient solution to an optimization problem in the strict sense a) implies an efficient solution to the corresponding evaluation problem b) by simply computing the cost of this solution. Of course, one can construct pathological examples where the evaluation function is very complex, or even non-computable. In that case, given a solution s, we might not be able to compute its value v(s). But such cases don't seem to occur in practice.
In turn, an efficient solution to an evaluation problem b) implies an efficient solution to the corresponding bound problem c) - just by comparing 2 numbers.

The opposite direction is less obvious. Nevertheless, we show that a solution to the bound problem c) helps in finding a solution to the evaluation problem b), which in turn helps in finding a solution to the optimization problem a). There is a considerable cost involved, but this cost is "polynomial", and in the generous complexity measure of this chapter we ignore polynomial costs. Although c) contains less information than b), and b) usually less than a), it is not surprising that even a small amount of information can be exploited to speed up the solution to a harder problem, as follows.

To exploit c) in solving b), we use the fact that the possible values of a combinatorial problem are usually discrete and can be taken to be integers. Assume we can solve the bound problem c) within time T. For the corresponding evaluation problem b) one usually knows a priori that the value lies within a certain range [L, U] of integers. Using binary search, we solve the evaluation problem with log | U - L | calls to the bound problem c), and hence in time T log | U - L |.

Ex: TSP on weighted graph Kn = (V, E, w: E -> Reals), |V| = n, |E| = n-choose-2.
Use c) to solve b). A tour or Hamiltonian cycle in a graph of n vertices has exactly n edges. Thus, the sum S of the n longest edges is an upper bound for the length of the optimal tour. On the other hand, the sums of all m-choose-n subsets of n edges is a finite set of numbers, and the minimal non-zero difference d among two of these numbers defines the granularity of the tour lengths. Two distinct tours either have the same value, or their lengths differ by at least d. Thus, a binary search that computes log(S / d) bound problems determines the length (value) of an optimal tour.

To exploit b) in solving a), we use the fact that the solution of a combinatorial problem is a discrete structure D that consists of predefined elementary parts, such as vertices or edges in a graph, say P1, .., Pn. And that the evaluation function v(D) is often monotonic with respect to subsets of D = P1, .., Pn - if we drop some parts in a maximization problem, the value v(D') of the subset D' may be less than v(D). Assume that we can solve the evaluation problem b) within time T, and that there are n elementary parts P1, .., Pn, each of which is a potential component of a solution. First, solve the evaluation problem for the given data D consisting of n elementary

parts to obtain the target value v(D). Thereafter, for each part Pi, obtain the value v(D - Pi). If v(D - Pi) = v(D), part Pi is not an essential component of an opti,mal solution, and can be omitted. Whereas if v(D - Pi) < v(D), part Pi is an essential component of any solution object and must e retained. By applying this test of essentiality to each of the n parts Pi, we construct a solution object within time (n+1) T.

Ex: TSP on weighted graph Kn = (V, E, w: E -> Reals), |V| = n, |E| = n-choose-2.
Use b) to solve a). First, find the length L of an optimal tour. Thereafter, for each edge ej find the length Lj of an optimal tour when ej is prohibited; this is achieved by temporarily assigning to ej a huge weight, eg making ej longer than the sum of all other edges. In a non-degenerate configuration, when the optimal tour is unique, the set of n edges j for which Lj < L form exactly this unique optimal tour. For a degenerate configuration such as the equilateral triangle with its center shown in Fig d) above, which has 3 isomorphic optimal tours, the rule is a bit more complicated (no single edge is essential).

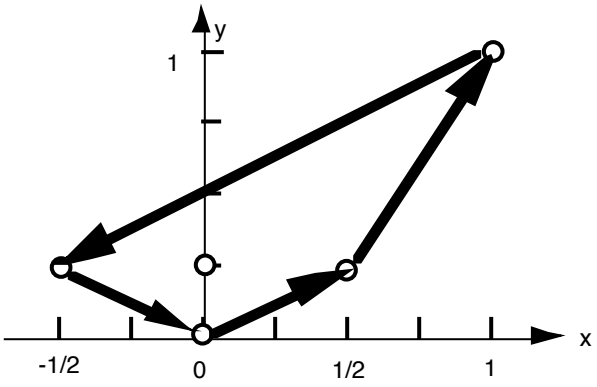
7.2 Problem reduction, classes of equivalent problems

Scientific advances often follow from a successful attempt to establish similarities and relationships among seemingly different problems. In mathematics, such relationships are formalized in two ways:
- problem A can be reduced to problem B, $A \leq B$, implies that if we can solve B, we can also solve A
- problems A and B are equivalent, $A \leq B$ and $B \leq A$, implies that if we can solve either problem, we can also solve the other, roughly with an equivalent investment of resources. Whereas here we use "≤" in an intuitive sense, section 7.5 presents a precise definition of polynomial reducibility, denoted by "≤p".
This chapter is all about reducing some problems to others, and to characterizing some prominent classes of equivalent problems. Consider some examples.

Sorting as a key data management operation. Answering almost any query about an unstructured set D of data items requires, at least, looking at each data item. If the set D is organized according to some useful structure, on the other hand, many important operations can be done faster than in linear time. Sorting D according to some useful total order, for example, enables binary search to find, in logarithmic time, any query item given by its key. As a data management operation of major importance, sorting has been studied extensively. In the RAM model of computation, the worst case complexity of sorting is $\Theta(n \log n)$. This fact can be used in two ways to bound the complexity of many other problems: 1) to show that some other problem P can be solved in time $O(n \log n)$, because P reduces to sorting, and 2) to show that some other problem Q requires time $\Omega(n \log n)$, because sorting reduces to Q. Two examples:

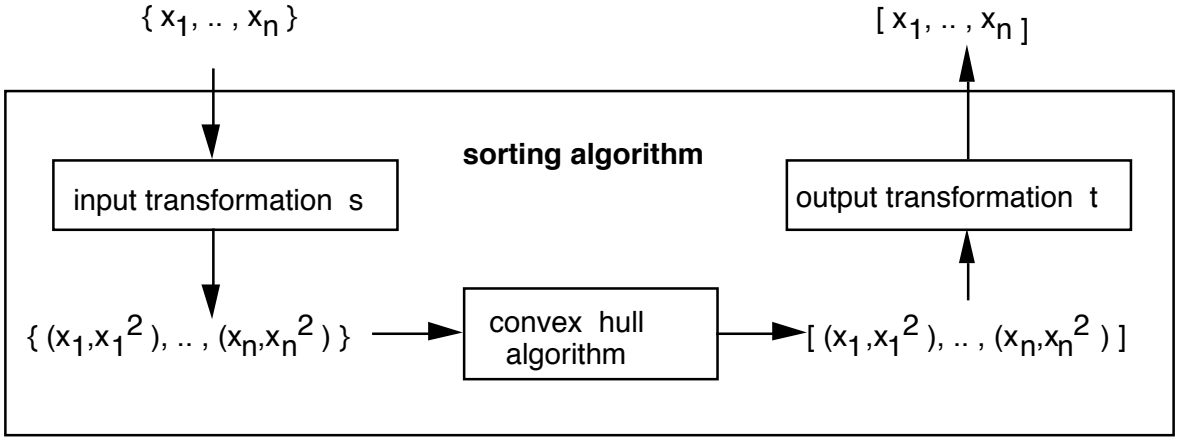
1) Finding the median of n numbers x1, .., xn can be done in time $O(n \log n)$. After sorting x1, .., xn in an array, we find the median in constant time as x [n div 2]. A more sophisticated analysis shows that the median can be determined in linear time, but this does not detract from the fact that we can quickly and easily establish an upper bound of $O(n \log n)$.

2) Finding the convex hull of n points in the plane requires time $\Omega(n \log n)$. An algorithm for computing the convex hull of n points in the plane takes as input a set $\{ (x_1, y_1), .. , (x_n, y_n) \}$ of coordinates and delivers as ouput a **sequence** $[(x_{i1}, y_{i1}), .. , (x_{ik}, y_{ik})]$ of extremal points, i.e. those on the convex hull, ordered clockwise or counter-clockwise. The figure shows 5 unordered input points and the ordered convex hull.

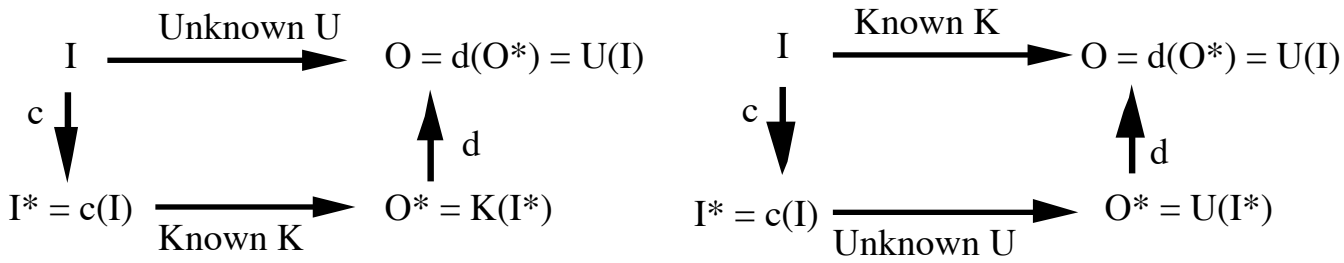


We now reduce the sorting problem of known complexity $\Omega(n \log n)$ to the convex hull problem as shown in the next figure. Given a set $\{ x_1, .. , x_n \}$ of reals, we generate the set $\{ (x_i, x_i^2), .. , (x_n, y_n^2) \}$ of point coordinates. Due to the convexity of the functiuon x^2, all points lie on the convex hull. Thus, a convex hull algorithm returns the **sequence** of points ordered by increasing x-coordinates. After dropping the irrelevant y-coordinates, we obtain the sorted sequence $[x_1, .. , x_n]$ .Thus, a convex hull algorithm can be used to obtain a

ridiculously complicated sorting algorithm, but that is not our aim. The aim is to show that the convex hull problem is at least as complex as sorting - if it was significantly easier, then sorting would be easier, too.



The following figure explains problem reduction in general. Given a problem U of unknown complexity, and a problem and algorithm K of known complexity. A coding function c transforms a given input I for one problem to input I* for the other, and the decoding function d transforms output O* into output O. This type of problem reduction is of interest only when the complexity of c and of d is no greater than the complexity of K, and when the input transformation I* = c(I) leaves the size n = | I | of the input data asymptotically unchanged: | I*| ∈ Θ (n). | This is what we assume in the following.



Deriving an **upper bound**. At left we construct a solution to the problem U of unknown complexity by the detour U(I) = d(K(c(I))). Letting | c |, | K | and | d | stand for the time required for c, K and d, respectively, this detour proves the upper bound | U | ≤ | c | + | K | + | d | . Under the usual assumption that | c | ≤ | K | and | d | ≤ | K | , this bound asymptotically reduces to | U | ≤ | K | . More explicitly: if we know that K(n), c(n), d(n) are all in O(f(n)), then also U(n) ∈ O(f(n)).

Deriving a **lower bound**. At right we argue that a problem U of unknown complexity | U | is asymptotically at least as complex as a problem K for which we know a lower bound, K(n) ∈ Ω(f(n)). Counter-intuitively, we reduce the known problem K to the unknown problem U, obtaining the inequality | K | ≤ | c | + | U | + | d | and | U | ≥ | K | - | c | - | d | . If K(n) ∈ Ω(f(n)) and c and d are strictly less complex than K, i.e. c(n) ∈ o(f(n)), d(n) ∈ o(f(n)), then we obtain the lower bound U(n) ∈ Ω(f(n)).

7.3 The class P of problems solvable in polynomial time

Practically all standard combinatorial algorithms presented in a course on Algorithms and Data Structures run in polynomial time. They are sequential algorithms that terminate after a number of computational steps that is bounded by some polynomial p(n) in the size n of the input data, as measured by the number of data items that define the problem. A computational step is any operation that takes constant time, i.e. time independent of n.

In practical algorithm analysis there is a fair amount of leeway in the definition of “computational step” and “data item”. For example, an integer may be considered a single data item, regardless of its magnitude, and any arithmetic operation on integers as a single step. This is reasonable when we know a priori that all numbers generated are bounded by some integer “maxint”, and is unreasonable for computations that generate numbers of unbounded magnitude.

In complexity theory based on Turing machines the definition is clear: a computational step is a transition executed, a data item is a character of the alphabet, read or written on a square of tape. The alphabet is usually chosen to be {0, 1} and the size of data is measured in bits. When studying the class **P** of problems solvable in

polynomial time, we only consider **deterministic TMs that halt on all inputs**.

Let t_M: A* -> Integers be the number of steps executed by M on input $x \in A^*$.

This chapter deals with TMs whose running time is bounded by some polynomial in the length of the input.

Df: TM M is or runs in polynomial time iff $\exists$ polynomial p such $\forall x \in A^*$: $t_M(x) \leq p(|x|)$

Df: $\mathbf{P} = \{ L \subseteq A^* \mid \exists$ TM M, $\exists$ polynomial p such that L = L(M) and $\forall x \in A^*$: $t_M(x) \leq p(|x|) \}$

Notice that we do **not** specify the precise version of TM to be used, in particular the number of tapes of M is left open. This may be surprising in view of the fact that a multi-tape TM is much faster than a single-tape TM. A detailed analysis shows that "much faster" is polynomially bounded: a single-tape TM S can simulate any multi-tape TM M with at most a polynomial slow-down: for any multi-tape TM M there is a single-tape TM S and a polynomial p such that for all $x \in A^*$, $t_S(x) \leq p(t_M(x))$. This simulation property, and the fact that a polynomial of a polynomial is again a polynomial, makes the definition of the class P extremely robust.

The question arises whether the generous accounting that ignores polynomial speed-ups or slow-downs is of practical relevance. After all, these are much greater differences than ignoring constant factors as one does routinely in asymptotics. The answer is a definite YES, based on several considerations:

1) Practical computing uses random access memory, not sequential storage such as tapes. Thus, the issue of how many tapes are used does not even arise. The theory is formulated in terms of TMs, rather than more realistic models of computation such as conventional programming languages, for the sake of mathematical precision. And it turns out that the slowness of tape manipulation gets absorbed, in comparison with a RAM model, by the polynomial transformations we ignore so generously.

2) Most practical algorithms are of low degree, such as $O(n)$, $O(n \log n)$, $O(n^2)$, or $O(n^3)$. Low-degree polynomials grow slowly enough that the corresponding algorithms are computationally feasible for many values of n that occur in practice. E.g. for $n = 1000$, $n^3 = 10^9$ is a number of moderate size when compared to processor clock rates of 1 GHz and memories of 1 GByte. Polynomial growth rates are exceedingly slow compared to exponential growth (consider 2^{1000}).

3) This complexity theory, like any theory at all, is a model that mirrors some aspects of reality well, and others poorly. It is the responsibility of the programmer or algorithm designer to determine in each specific case, whether or not an algoritm "in **P**" is practical or not.

Examples of problems (perhaps?) in **P**:

1) Every context-free language is in **P**. In Ch5 we saw an $O(n^3)$ parsing algorithm that solves the word problem for CFLs, where n is the length of the input string.

2) The complexity of problems that involve integers depends on the representation. Fortunately, with the usual radix representation, the choice of radix $r > 1$ is immaterial (why?). But if we choose an exotic notation, everything might change. For example, if integers were given as a list of their prime factors, many arithmetic problems would become easier. Paradoxically, if integers were given in the unwieldy unary notation, some things might also become "easier" according to the measure of this chapter. This is because the length of the unary representation $< k >_1$ of k is exponential in the length of the radix $r \geq 2$ representation $< k >_r$. Given an exponentially longer input, a polynomial-time TM is allowed to use an exponentially longer computation time as compared to the "same" problem given in the form of a concise input.

The following example illustrates the fact that the complexity of arithmetic problems depends on the number representation chosen. The assumed difficulty of factoring an integer lies at the core of modern cryptography. Factoring algorithms known today require, in the worst case, time exponential in the length, i.e. number of bits, of the radix representation of the number to be factored. But there is no proof that factoring is NP-hard - according to today's knowledge, there might exist polynomial-time factoring algorithms. This possibility gained plausibility when it was proven that primality, i.e. the problem of determining whether a natural number (represented in radix notation) is prime or composite, is in P (M. Agrawal, N. Kayal, N. Saxena: PRIMES is in P, Indian Institute of Technology Kanpur, August 2002). If we represent numbers in unary notation, on the other hand, then it is straightforward to design a polynomial-time factoring algorithm (why?).

7.4 The class NP of problems solvable in non-deterministic polynomial time

"NP" stands for "non-deterministic polynomial". It is instructive to introduce two different but equivalent definitions of NP, because each definition highlights a different key aspect of NP. The original definition explicitly introduces non-deterministic TMs:

Df 1: **NP** = { $L \subseteq A^*$ | $\exists$ **N**TM N, $\exists$ polynomial p such that L = L(N) and $\forall x \in A^*$: $t_N(x) \leq p(|x|)$ }

Notice that this differs from the definition of **P** only in the single letter "**N**" in the phrase **"** $\exists$ **N**TM N ..", indicating that we mean non-deterministic Turing machines. We had seen that deterministic and non-deterministic TMs are equally powerful in the presence of unbounded resources of time and memory. But there is a huge difference in terms of the **time** they take for certain computations. A NTM pursues simultaneously a number of computation paths that can grow exponentially with the length of the computation.

Because of many computation paths pursued simultaneously we must redefine the function t_N: A* -> Integers that measures the number of steps executed. An input $x \in A^*$ may be accepted along a short path as well as along a long path, and some other paths may not terminate. Therefore we define **$t_N(x)$ as the minimum number of steps executed along any accepting path for x**.

Whereas the original definition of NP in terms of non-deterministic TMs has the intuitive interpretation via parallel computation, an equivalent definition based on deterministic TMs is perhaps technically simpler to handle. The fact that these two definitions are equivalent provides two different ways of looking at **NP**.

The motivation for this second definition of **NP** comes from the observation that it may be difficult to decide whether a string meeting certain specifications **exists**; but that it is often easier to decide whether or not a **given string** meets the specifications. In other words, finding a solution, or merely determining whether a solution exists, is harder than checking a proposed solution's correctness. The fact that this intuitive argument leads to a rigorous definition of **NP** is surprising and useful!

In the following definition, the language L describes a problem class, e.g. all graphs with a desired property ; the string w describes a problem instance, e.g. a specific graph G; the string c = c(w), called a "certificate for w" or a witness, plays the role of a key that "unlocks w": the pair w, c is easier to check than w alone!

Example: for the problems of Hamiltonian cycles and cliques introduced in 7.1, w = <G> is the representation of graph G, and the certificate c is the representation of a cycle or a clique, respectively. Given c, it is easy to verify that c represents a cycle or a clique in G.

Df: a verifier for $L \subseteq A^*$ is a **deterministic** TM V with the property that
$L = \{ w \mid \exists\ c(w) \in A^*$ such that V accepts $< w, c > \}$

The string c = c(w) is called a certificate for w's membership in L. <w, c> denotes a representation of the pair (w, c) as a single string, e.g. w#c, where a reserved symbol # separates w from its certificate. The idea behind this concept of certificate is that it is easy to verify $w \in L$ if you are given w's certificate c. If not, you would have to try all strings in the hope of finding the right vertificate, a process that may not terminate. We formalize the phrase "easy to verify" by requiring that a verifier V is (or runs in) polynomial-time.

Df 2: NP is the class of languages that have deterministic polynomial-time verifiers.

Definitions Df1 and Df2 provide two different interpretations of the same class NP: Df1 in terms of **parallel computation**, Df2 in terms of **sequential verification**. A technical advantage of Df2 is that the theory of NP can be developed using only deterministic TMs, which are simpler than NTMs. We now present an intuitive argument that the two definitions are equivalent.

Df1 -> Df2 ("simulate and check"): If L is in **NP** according to Df1, there is a NDTM N with the following property: every $w \in L$ has an accepting path p of length polynomial in |w|. It is tempting to merely remove the transitions of N that are not executed along the path p, but this does not turn N into a deterministic TM, because different w's may use different transitions. Instead, we construct a DTM M that reads as input a string <w, N, P> that represents the following information: the word w to be accepted, the NTM N, the accepting path p. M is a universal TM that interprets the description of N: at every step along the path p, M checks that this step is one of the possible transitions of N on input w. In this construction, the pair <N, p> is w's certificate.

Df2 -> Df1 ("guess and check": We present the idea using as an example the problem of Hamiltonian cycle. According to Df2, there is a DTM M that verifies <G, c> in polynomial time, where the certificate c is a

Hamiltonian cycle of G. Construct a NTM N that first generates in parallel lists of n numbers v1, v2, .., vn, vi ∈ 1 .. n. Each vi can be generated one bit at a time in logarithmic time, the entire list in time n log n. Each list is checked for syntactic correctness in the sense that there are no repetitions of vertex numbers, and that any 2 consecutive vertices, including vn v1, are edges of G. This checking is identical to the verification done by the deterministic TM M, but the non-deterministic N does it on all lists simultaneously. If any one list is verified as being a Hamiltonian cycle, then N accepts <G>.

From both definitions Df1 and Df2 it is evident that **P ⊆ NP**. From Df1 because a DTM is a special case of a NTM. From Df2 because P is the class of languages L that require no certificate, so we can take the nullstring as a trivial certificate for all words of L.

P = NP ? is one of the outstanding questions of the theory of computations, so far unanswered. All experience and intuition suggests that **P ≠ NP.** Non-determinism provides computing power that grows exponentially with the length of the sequential computation. In the time that a DTM performs t steps, a NTM can perform at least $\mathbf{2^t}$ steps. Whereas a DTM must backtrack to explore alternatives, a NTM explores all alternatives at the same time. The belief that P ≠ NP is so strong that today many researchers prove theorems that end with the caveat "..unless P = NP". By this they mean to imply that the theorem is to be considered empirically true, i.e. as true as the conjecture P ≠ NP. But mathematics has its own strict rules as to what constitutes a theorem and a proof, and today there is no proof in sight to give the conjecture P ≠ NP the status of a theorem.

7.5. Polynomial time reducibility, NP-hard, NP-complete

We add some formal definitions to the general concept of problem reduction of 7.2.

Df: A function f: A* -> A* is polynomial-time computable iff there is a polynomial p and a DTM M which, when started with w on its tape, halts with f(w) on its tape, and $t_M(w) \le p(|w|)$.

Df: L is polynomial-time reducible to L', denoted by $L \le_p L'$ iff
there is polynomial-time computable f: A* -> A* such that $\forall w \in A^*$, $w \in L$ iff $f(w) \in L'$.

In other words, the question w ∈ L? can be answered by deciding f(w) ∈ L'. Thus, the complexity of deciding membership in L is at most the complexity of evaluating f plus the complexity of deciding membership in L'. Since we generously ignore polynomial times, this justifies the notation $L \le_p L'$.

A remarkable fact makes the theory of NP interesting and rich. With respect to the class **NP** and the notion of polynomial-time reducibility, there are "hardest problems" in the sense that all decision problems in **NP** are reducible to any one of these"hardest problems".

Df: L' is **NP-hard** iff $\forall L \in NP$, $L \le_p L'$
Df: L' is **NP-complete** iff $L' \in NP$ and L' is NP-hard

7.6 Satisfiability of Boolean expressions (SAT) is NP-complete

SAT = satisfiability of Boolean expressions: given an arbitrary Boolean expression E over variables x1, x2, .. , xd, is there an assignment of truth values to x1, x2, .. that makes E true?

It does not matter what Boolean operators occur, conventionally one considers And ∧, Or ∨, Not ¬.

SAT is the prototypical "hard problem". I.e. the problem that is generally proven to be NP-complete "from scratch", whereafter all other problems to be proven NP-complete are reduced to SAT. The theory of NP-completeness began with the key theorem **(Cook 1971): SAT is NP-complete.**

Given this central role of SAT it is useful to develop an intuitive understanding of the nature of the problem, including details of measurement and "easy" versions of SAT.

1) It does not matter what Boolean operators occur, conventionally one considers And ∧, Or ∨, Not ¬. Special forms, eg CNF

2) The length n of E can be measured by the number of characters of E. It is more convenient, however, to first eliminate "Not-chains" ¬ ¬ ¬ .. using the identity ¬ ¬ x = x, and to measure the length n of E by the number n of **occurences** of variables (d denotes the number of **distinct** variables x1, x2, .. xd in E, $d \le n$). It is

convenient to avoid mentioning the unary operator ¬ explicitly by introducing, for each variable x, its negation ¬ x as a dependent variable. A variable and its negation are called **literals**. Thus, an expression E with d variables has 2d distinct literals that may occur in E.

3) Any Boolean expression E can be evaluated in linear time. Given truth values for x1, x2, .. x_d, the n occurrences of literals are the leaves of a binary tree with n -1 internal nodes, each of which represents a binary Boolean operator. The bottom-up evaluation of this tree requires n-1 operations.

4) Satisfiability of expressions over a constant number d of distinct variables can be decided in linear time. By trying all 2^d assignments of truth values to the variables, SAT can be decided in time $O(2^d \ n)$, a bound that is linear in n and exponential in d. If we consider d constant, 2^d is also a constant - hence this version of the satisfiability problem can be solved in linear time! This argument shows that, if SAT turns out to be a difficult problem, this is due to an unbounded growth of the number of distinct variables. Indeed, if d grows proportionately to n, the argument above yields an exponential upper bound of $O(2^n)$.

5) In order to express SAT as a language, choose a suitable alphabet A and a coding scheme that assigns to any expression E a word code(E) ∈ A*, and define: SAT = { code(E) | E is a satisfiable Boolean expression }

Thm (Cook 1971): SAT is NP-complete - the key theorem at the origin of all other NP-complete results.

1) SAT is in **NP**. Given E and an assignment of truth values as a certificate of satisfiability, the truth value of E can be computed and verified in linear time.

2) Idea of proof that SAT is NP-hard. Let L be any language ∈ **NP** , we have to prove L ≤p SAT. L ∈ **NP** means there is a deterministic polynomial-time verifier M with L = L(M). We construct in polynomial time a Boolean expression E = E(M, w) with the property that w ∈ L iff E is satisfiable. The intuition is that E(M, w) simulates, i.e. describes, the computation of M on w step by step. Although E is "huge", a detailed analysis shows that the length |E| of E is polynomially bounded in the length |w|, and moreover, that E can be computed in time polynomial in |w|.

The following idea guides the construction of E(M, w). The computation of M on any word w is a sequence of configurations, where a configuration contains the state of the tape, the state of M, and the location of M's read/write head on the tape. This sequence can be described, or modeled, by a (huge) set of Boolean variables. For example, we introduce a variable s(99, 1) which we want to be true iff after 99 steps M is in state 1. The transition from step 99 to step 100 can be modeled by some expression that relates all the variables involved, such as t(99, 0), which we want to be true iff after 99 steps M is reading symbol 0 on its tape. Thus, the behavior of any TM M on w can be modeled by a huge expression which is true iff M accepts w. QED.

Satisfiability remains NP-complete even if we restrict the class of Boolean expressions in various ways. A standard and convenient normal form, that still allows us to express any given Boolean function, is the **conjunctive normal form, CNF** .

Df: a CNF expression E is the conjunction of clauses Ci, where each clause is the disjunction of literals Lj, and a literal is a single variable or the negation of a single variable, e.g. x:
E = C1 ∧ C2 ∧ C3 ∧ ... where Ci = (L1 ∨ L2 ∨ L3 ∨ ..) and Lk = x or Lk = ¬x.

Any Boolean expression E can be transformed into an equivalent expression in CNF using the identities:
- de Morgan's law: ¬ (x ∧ y) = ¬ x ∨ ¬ y , ¬ (x ∨ y) = ¬ x ∧ ¬ y
- distributive law: x ∧ (y ∨ z) = (x ∧ y) ∨ (x ∧ z) , x ∨ (y ∧ z) = (x ∨ y) ∧ (x ∨ z)

Let CNF be the problem of deciding the satisfiability of CNF expressions. Since SAT is NP-complete and any Boolean expression E can be transformed into an equivalent expression F in CNF, it is obvious that SAT is reducible to CNF, and it is plausible to conjecture that SAT ≤p CNF, and hence that CNF is NP-complete. The conjecture is correct, but the plausibility is too simple minded. What is wrong with the following "pseudo-proof"? In order to prove SAT ≤p CNF, transform any given Boolean expression E in arbitrary form into an equivalent CNF expression F; E is satisfiable iff F is satisfiable. "QED"

The error in the argument above is hidden in the letter 'p' in '≤p', which we defined as a **polynomial-time reduction**. The transformation of an arbitrary Boolean expression E into an equivalent CNF expression F may cause F to be **exponentially longer** than E, and thus to require time exponential in the length of E merely to write down the result! Witness the following formula E in disjunctive normal form (DNF):

$E = x11 \wedge x12 \vee x21 \wedge x22 \vee x31 \wedge x32 \vee \ldots \vee xn1 \wedge xn2$

It consists of 2n literals each of which appears only once, thus it has length 2n as measured by the number of occurrences of literals. repeated application of the distributive laws yields

$F = \bigwedge(j1 \ldots jn = 1,2)\ (x1j1 \vee x2j2 \vee x3j3 \vee \ldots \vee x njn)$

F is an AND of 2n clauses, where each clause has exactly one variable xj for j = 1 .. n, with the second index chosen independently of the choice of all other second indices. Thus, the length of F is n 2n , a transformation that cannot be done in polynomial time. (Exercise: prove that E and F are equivalent).

We will now provide a correct proof that CNF is NP-complete, as a consequence of the stronger assertion that 3-CNF is NP-complete.

Df: A 3-CNF expression is the special case of a CNF expression where each clause has 3 literals ("exactly 3" or "at most 3" is the same - why?).

Because of its restricted form, 3-CNF is conceptually simpler than SAT, and so it is easier to reduce it to new problems that we aim to prove NP-complete. We aim to show that 3-CNF is NP-complete by reducing SAT to 3-CNF, i.e. SAT $\leq_p$ 3-CNF. In contrast to the argument used in the wrong proof above, which failed because it ignores exponential lengthening, the following construction keeps the length of F short by using the trick that E and F need not be equivalent as Boolean expressions!

Thm: 3-CNF SAT is NP-complete

Pf idea: reduce SAT to 3-CNF SAT, SAT $\leq_p$ 3-CNF SAT. To any Boolean expression E we assign in polynomial time a 3-CNF expression F that is **equivalent in the weak sense** that either both E and F are satisfiable, or neither is. Notice that E and F need not be equivalent as Boolean expressions, i.e. they need not represent the same function! They merely behave the same w.r.t. satisfiability.

Given E, we construct F in 4 steps, illustrated using the example $E = \neg (\neg x \wedge (y \vee z))$

1) Use de Morgan's law to push negations to the leaves of the expression tree:
   $E1 = x \vee \neg (y \vee z) = x \vee (\neg y \wedge \neg z)$

2) Assign a new Boolean variable to each internal node of the expression tree, i.e. to each occurrence of an operator, and use the Boolean operator 'equivalence' $\Leftrightarrow$ to state the fact that this variable must be the result of the corresponding operation: $u \Leftrightarrow (\neg y \wedge \neg z)$, $w \Leftrightarrow x \vee u$

3) Construct an expression E2 that states that the root of the expression tree must be true, traces the evaluation of the entire tree, node by node, and combines all these assertions using ANDs:
   $E2 = w \wedge (w \Leftrightarrow (x \vee u)) \wedge (u \Leftrightarrow (\neg y \wedge \neg z))$.

   E2 and E are equivalent in the weak sense of simultaneous satisfiability. If E is satisfiable, then E2 is also, by simply assigning to the new variables u and w the result of the corresponding operation. Conversely, if E2 is satisfiable, then E is also, using the same values of the original variables x, y, z as appear in E2.
   Notice that E2 is in conjunctive form at the outermost level, but its subexpressions are not, so we need a last transformation step.

4) Recall the Boolean identity for implication: $a \Rightarrow b = \neg a \vee b$ to derive the identities:
   $a \Leftrightarrow (b \vee c) = (a \vee \neg b) \wedge (a \vee \neg c) \wedge (\neg a \vee b \vee c)$
   $a \Leftrightarrow (b \wedge c) = (\neg a \vee b) \wedge (\neg a \vee c) \wedge (a \vee \neg b \vee \neg c)$
   Using these identities on the subexpressions, E2 gets transformed into F in 3-CNF:
   $F = w \wedge (w \vee \neg x) \wedge (w \vee \neg u) \wedge (\neg w \vee x \vee u) \wedge (\neg u \vee x) \wedge (\neg u \vee z) \wedge (u \vee y \vee z)$

Each of the four transformation steps can be done in linear time and lengthens the expression by at most a constant factor. Thus, the reduction of a Boolean expression E in general form to one, F, in 3-CNF can be done in polynomial time. QED

Notice the critical role of the integer '3'in 3-CNF: we need to express the result of a binary operator, such as $w \Leftrightarrow x \vee u$, which naturally involves three literals. Thus, it is no surprise that the technique used in the proof above fails to work for 2-CNF. Indeed, **2-CNF is in P** (Exercise: look up the proof in some textbook).

In analogy to CNF we define the **disjunctive normal form DNF** as an OR of terms, each of which is an

AND of literals: E = T1 ∨ T2 ∨ T3 ∨ ... where Ti = (L1 ∧ L2 ∧ L3 ∧ ..)

Notice: **DNF-SAT** is in **P**, in fact DNF-SAT can be decided in linear time (consider xy'z ∨ x x' ∨ ...).
Does this mean that the NP-completeness of SAT is merely a matter of representation? No!
Exercise: Show that the **Falsifiability of DNF-SAT is NP-complete**.
This relationship between CNF and DNF implies that the transformation DNF <-> CNF must be hard.

7.7 Hundreds of well-known problems are NP-complete

3-CNF SAT is the starting point of chains of problem reduction theorems to show that hundreds of other well known decision problems are NP complete. The problem CLIQUE is an example: Given a graph G and an integer k, does G contain a clique of size ≥ k?

Thm: CLIQUE is NP-complete

Pf: Show that 3-CNF ≤p CLIQUE. Given a 3-CNF expression F, construct a graph G = (V, E) and an integer k such that F is satisfiable iff G has a k-clique.

Let F = (z11 ∨ z12 ∨ z13) ∧ (z21 ∨ z22 ∨ z23) ∧ ... ∧ (zm1 ∨ zm2 ∨ zm3), where each zij is a literal.
To each occurrence of a literal we assign a vertex, i.e. V = { (1,1), (1,2), (1,3), ... , (m, 1), (m, 2), (m, 3) }
We introduce an edge ((i, j) (p, q)) iff i ≠ p (the two literals are in different clauses)
and zij ≠ ¬zpq (the 2 literals do not clash, i.e. both can be made true under the same assignment).
Finally, let k, the desired clique size, be = m, the number of clauses.
With this construction of G we observe that F is satisfiable via an assignment A
iff 1) each clause contains a literal that is true under A, say $z_{1, j1}$, $z_{2, j2}$, ... , $z_{m, jm}$
iff 2) there are literals $z_{1, j1}$, $z_{2, j2}$, ... , $z_{m, jm}$ no 2 of which are negations of each other
iff 3) there are vertices (1, j1), (2, j2), ... , (m, jm) that are pairwise connected by an edge
iff 4) G has a k-clique.

7.8 The P versus NP Problem

The Clay Mathematics Institute of Cambridge, Massachusetts (CMI, www.claymath.org, www.claymath.org/Millennium_Prize_Problems/P_vs_NP) has named seven "Millennium Prize Problems." The Scientific Advisory Board of CMI selected these problems, focusing on important classic questions that have resisted solution over the years. The Board of Directors of CMI designated a $7 million prize fund for the solution to these problems, with $1 million allocated to each. The first in the list is the "P versus NP Problem", deescribed as follows:

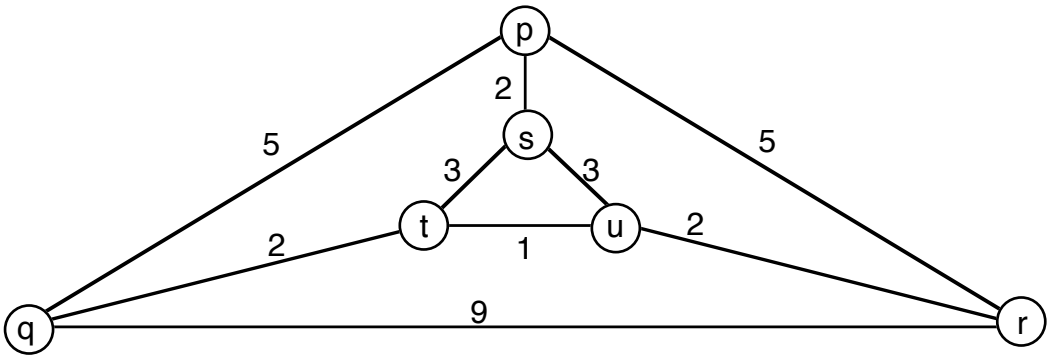
"Suppose that you are organizing housing accommodations for a group of four hundred university students. Space is limited and only one hundred of the students will receive places in the dormitory. To complicate matters, the Dean has provided you with a list of pairs of incompatible students, and requested that no pair from this list appear in your final choice. This is an example of what computer scientists call an NP-problem, since it is easy to check if a given choice of one hundred students proposed by a coworker is satisfactory (i.e., no pair from taken from your coworker's list also appears on the list from the Dean's office), however the task of generating such a list from scratch seems to be so hard as to be completely impractical. Indeed, the total number of ways of choosing one hundred students from the four hundred applicants is greater than the number of atoms in the known universe! Thus no future civilization could ever hope to build a supercomputer capable of solving the problem by brute force; that is, by checking every possible combination of 100 students. However, this apparent difficulty may only reflect the lack of ingenuity of your programmer. In fact, one of the outstanding problems in computer science is determining whether questions exist whose answer can be quickly checked, but which require an impossibly long time to solve by any direct procedure. Problems like the one listed above certainly seem to be of this kind, but so far no one has managed to prove that any of them really are so hard as they appear, i.e., that there really is no feasible way to generate an answer with the help of a computer. Stephen Cook and Leonid Levin formulated the P (i.e., easy to find) versus NP (i.e., easy to check) problem independently in 1971. "

Hw 7.1: Polynomal-time problem reduction: HC ≤p TSP

A Hamiltonian cycle in a graph G(V, E) is a simple closed path that touches every vertex exactly once. The Hamiltonian Cycle Problem HC asks you to decide, given a Graph G, whether or not G has a Hamiltonian cycle.We consider the Traveling Salesman Problem TSP in the form of a decision problem. Given a weighted

graph G(V, E, w), where w: E -> Reals defines edge weights, and given a real number B: is there a traveling salesman tour (i.e. a Hamiltonian cycle in G) of total weight ≤ B?

a) Decide whether the weighted graph shown here has a tour of length ≤ 21. If yes, show it, if not, say why.



Complexity theory represents a class D of decision problems by coding each problem instance d by a string c(d) over some alphabet A. c(d) is a code that uniquely defines d. The class D and the code c partition A* into 3 subsets: 1) the set of strings that are not codewords c(d) for any problem instance d ("syntax error"), 2) the set of codewords c(d) for instances d with a negative answer, and 3) the set L(D) of codewords c(d) for instances d with a positive answer. Solving the decision problem means constructing a Turing machine that always halts and accepts the language L(D).

b) Define a coding scheme to represent weighted graphs and show the codeword that your scheme assigns to the example of TSP-problem b).

c) Recall the definition of "L1 ≤p L2", i.e. language L1 is polynomial-time reducible to language L2. Show in general how HC is polynomial-time reducible to TSP, i.e. HC ≤p TSP, or more precisely, L(HC) ≤p L(TSP). Hint: Given a graph G as input to HC, construct an appropriate weighted graph G'and an appropriate bound B as input to TSP.

Hw 7.2: CNF ≤p 3-CNF

Show that the satisfiability problem CNF can be reduced in polynomial time to its special case 3-CNF.
Hint: Given a CNF expression E that contains long clauses, any clause C with > 3 literals,

$C = (x1 \vee x2 \vee x3 \vee .. \vee xk)$

can be broken into a number of shorter clauses by repeated application of the following transformation that introduces additional variables, z:

$C' = (x1 \vee x2 \vee z) \wedge (x3 \vee .. \vee \neg \subseteq z)$

Show 1) that E and the 3-CNF expression F that results from repeated application of this transformation are equivalent in the weak sense that E is satisfiable iff F is satisfiable; 2) that F can be constructed from E in polynomial time; and 3), work out an interesting example.

Hw 7.2: Set cover is NP-complete

Set cover problem: Given a finite set U ("universe"), n subsets S1, S2, .., Sn ⊆ U, and an integer k ≤ n, does there exist a selection of k subsets Sj1, Sj2, .., Sjk (among the given subsets) whose union is U?
Your task: search the library or the web to find a proof of the theorem 'Set cover is NP-complete', understand it, an be prepared to present this proof to your assistant. If you don't succeed with 'Set cover', do the same for some other NP-complete problem of your choice.

End of chapter