Lecture notes for the course Complexity IBC028

H. Zantema

Version March 28, 2019

All references are to the book:
Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein: Introduction to Algorithms
MIT Press, 2009, ISBN 978-0-262-53305-8, Third Edition

Goals of the course

- Techniques to compute time complexity of recursive algorithms, in particular master theorem. This is roughly Chapter 4 of the book.
- Some examples of algorithms and their complexity, in particular some geometrical algorithms from Chapter 33 of the book.
- NP-completeness: analysis of a wide class of decision problems for which the existence of a polynomial algorithm is very unlikely. This is roughly Chapter 34 of the book. Additionally, some remarks on PSPACE-completeness are made: an even higher class in which not only time complexity but also space complexity is likely to be superpolynomial.

Some basics

See Chapter 3 of the book.

Time complexity is the number of steps that an algorithm takes, depending on the size of the input. Note that this depends on the definition of steps. For instance, sometimes the addition of two numbers is considered to be a single step, but sometimes the basic steps are the bit-wise basic operations, by which the addition of two numbers is logarithmic in the size of the numbers.

In space complexity the size of the required memory is considered instead of the number of steps.

In order to ignore effects of issues like the choice of the programming languages or details of the implementation, complexity is always considered up to a constant, for large input sizes. Important notations are:

- $f(n) = O(g(n))$: $\exists c, N > 0 : \forall n > N : f(n) \leq c \cdot g(n)$.
- $f(n) = \Omega(g(n))$: $\exists c, N > 0 : \forall n > N : f(n) \geq c \cdot g(n)$.
- $f(n) = \Theta(g(n))$: $f(n) = O(g(n)) \wedge f(n) = \Omega(g(n))$.

We typically use these notations for f being the complexity of an algorithm, where n is the size of the input, and g is a well-known function, like $g(n) = \log n$, $g(n) = n$, $g(n) = n \log n$, $g(n) = n^2$ or $g(n) = 2^n$.

In proving properties we often use the *principle of induction*. In its basic version it states:

> Let P be a property depending on natural numbers, for which $P(0)$ holds, and for every n we can conclude $P(n+1)$ from the induction hypothesis $P(n)$. Then $P(n)$ holds for every natural number n.

A fruitful variant, sometimes called *strong induction*, is the following:

> Let P be a property depending on natural numbers, for which for every n we can conclude $P(n)$ from the induction hypothesis $\forall k < n : P(k)$. Then $P(n)$ holds for every natural number n.

Here k, n run over the natural numbers.

So the difference is as follows. In the basic version we prove $P(n+1)$ from the assumption that $P(n)$ holds, that is, for n being the direct predecessor of the value $n+1$ for which we want to prove the property. In contrast, in the strong version we try to prove $P(n)$, assuming that $P(k)$ holds for *all* predecessors k of n, not only the direct predecessor.

Surprisingly, we can prove validity of the strong version by only using the basic version, as follows.

Assume that we can conclude $P(n)$ from the (strong) induction hypothesis $\forall k < n : P(k)$. We have to prove that $P(n)$ holds for every natural number n. We do this by proving that $Q(n) \equiv \forall k < n : P(k)$ holds for every n, by (the basic version of) induction on n. First we observe that $Q(0)$ holds, which is trivially true since no natural number $k < 0$ exists. Next we assume the (basic) induction hypothesis $Q(n)$ and try to prove $Q(n+1) \equiv \forall k < n+1 : P(k)$. So let $k < n+1$. If $k < n$ then we conclude $P(k)$ by the (basic) induction hypothesis $Q(n)$. If not, then $k = n$, and by our start assumption we obtain $P(n)$. So we conclude $Q(n+1) \equiv \forall k < n+1 : P(k)$. So by the basic version of induction we conclude that $Q(n)$ holds for every n. But we had to conclude that $P(n)$ holds for every n. So let n be arbitrary. Since $n < n+1$ and we know that $Q(n+1)$ holds, we conclude that $P(n)$ holds, concluding the proof.

Recurrence relations

Consider the following recursive program $P(n)$ to compute 2^n for $n \geq 0$:

if $n = 0$ then return 1 else return $P(n-1) + P(n-1)$.

Let $T(n)$ be the number of steps to be done when running $P(n)$, in which computing $+$ is considered as a single step. If $n = 0$, then immediately 1 is returned, so $T(0) = 1$. Otherwise, we have two recursive calls $P(n-1)$ and one addition, so $T(n) = 1 + 2T(n-1)$ if $n > 0$. Observe that T is fully defined by the *recurrence relation*, shortly called recurrence

$$T(0) = 1, \;\; T(n) = 1 + 2T(n-1) \;\; \text{if } n > 0.$$

So for computing T we may forget the whole program and only exploit this recurrence. In this case one easily proves by induction that $T(n) = 2^{n+1} - 1$ for all $n \geq 0$, so $T(n) = \Theta(2^n)$.

An interesting recurrence relation is

$$f(0) = 1, \; f(1) = 1, \; f(n) = f(n-1) + f(n-2) \;\; \text{if } n > 1.$$

The resulting function is called the *Fibonacci function*, named after Fibonacci, also called Leonardo from Pisa, who lived from around 1170 to around 1250.

This Fibonacci function appears on many places, like in matrix exponentiation, and has many remarkable properties, as appears in the next exercises.

Exercise 1.

Let $f = \text{fib}$ be defined by $f(i) = i$ for $i = 0, 1$ and $f(i) = f(i-1) + f(i-2)$ for $i > 1$. Prove by induction on n that

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n = \begin{pmatrix} f(n-1) & f(n) \\ f(n) & f(n+1) \end{pmatrix}$$

for all $n \geq 1$.

Exercise 2.

Let $f = \text{fib}$ be defined by $f(i) = i$ for $i = 0, 1$ and $f(i) = f(i-1) + f(i-2)$ for $i > 1$. Prove by induction on n that

$$f(n-1)f(n+1) = f(n)^2 + 1$$

if n is even, and

$$f(n-1)f(n+1) = f(n)^2 - 1$$

if n is odd, for all $n \geq 1$.

Next we will find a closed expression for the Fibonacci function f from which we will conclude that f grows exponentially, just as we did above for T. For the time being we forget the initial values $f(0) = 1,\ f(1) = 1$ and only look for solutions of $f(n) = f(n-1) + f(n-2)$. As a first trick we try for a solution $f(n) = \alpha^n$ for all n, for a value α to be established. Then from $f(n) = f(n-1) + f(n-2)$ we conclude

$$\alpha^n = \alpha^{n-1} + \alpha^{n-2}$$

for all $n > 1$. This is equivalent to $\alpha^{n-2}(\alpha^2 - \alpha - 1) = 0$. Assuming $\alpha \neq 0$, this is equivalent to $\alpha^2 - \alpha - 1 = 0$. This equation has two solutions: $\alpha = \phi = (1 + \sqrt{5})/2 \approx 1.618$ and $\alpha = \hat{\phi} = (1 - \sqrt{5})/2 \approx -0.618$.

Next observe that not only ϕ^n and $\hat{\phi}^n$ satisfy $f(n) = f(n-1) + f(n-2)$, also $c\phi^n + d\hat{\phi}^n$ for any two numbers c, d. Next we remember the requirement $f(0) = 1,\ f(1) = 1$, which is satisfied if $c + d = c\phi^0 + d\hat{\phi}^0 = 0$ and $c\phi + d\hat{\phi} = 1$. Solving c, d from these requirements yields the closed expression

$$f(n) \ = \ (\phi^n - \hat{\phi}^n)/\sqrt{5}.$$

This may look tricky, but by construction this closed expression $f(n) \ = \ (\phi^n - \hat{\phi}^n)/\sqrt{5}$ satisfies the requirements

$$f(0) = 1,\ f(1) = 1,\ f(n) = f(n-1) + f(n-2) \ \text{ if } n > 1.$$

and since these requirements uniquely define f, we conclude that this closed expression is correct. As a side remark, it is a nice example of how a richer number system (in this case with $\sqrt{5}$) is helpful to study a problem that originally is stated only in the natural numbers.

From this closed expression we easily derive the desired exponential behavior. Since $\hat{\phi} \approx -0.618$ we conclude that $\hat{\phi}^n$ very quickly converges to 0, so $f(n) \approx \phi^n/\sqrt{5} = \Theta(\phi^n)$ for $\phi \approx 1.618$, proving that f is exponential. A similar argument is given in Section 19.4 of the book (page 523).

This observation has many important applications to the complexity of algorithms. As an example we consider the following reasoning showing that AVL trees have logarithmic depth. An AVL tree is a binary search tree with the extra requirement that for every node the depth of the left branch and the depth of the right branch differ by at most 1. The motivation for binary search trees is that they can be used as a data structure for storing ordered data, in such a way that searching, insertion and deletion all can be done in logarithmic time. But this only works well if we add an extra requirement to the binary search tree to force that it is balanced: the depth of the tree should be logarithmic in the total number of nodes. One option to force this is to use red-black trees, as is exploited in the book in Chapter 13, but an alternative is to use AVL trees. Here we elaborate Problem 13-3 on page 333 of the book.

First we prove by induction on n that the number of nodes in an AVL tree of depth n is at least $f(n)$ for f being the Fibonacci function. For $n = 0, 1$ this holds: an AVL tree of depth 0 has no nodes, and an AVL tree of depth 1 has at least one node. For an AVL tree of depth $n > 1$ we consider the left and right branch of the root. By definition of depth at least one them has depth $n - 1$, and by definition of AVL tree the other has depth at least $n - 2$. So by induction hypothesis the numbers of nodes below the root is at least $f(n-1)+f(n-2) = f(n)$, concluding the proof. Using our earlier observation $f(n) = \Theta(\phi^n)$ for $\phi \approx 1.618$, we conclude that there exists $c > 0$ such that for every AVL tree of depth $n > 0$ the number of nodes is at least $c \cdot \phi^n$. Next we prove our main claim: the depth of an AVL tree with k nodes is $O(\log k)$. Define $d(k)$ to be the highest depth of any AVL tree with k nodes. According to the above observation (since there is a AVL tree with k nodes and depth $d(k)$) we obtain

$$k \geq c \cdot \phi^{d(k)}.$$

This yields $\log k \geq \log(c \cdot \phi^{d(k)}) = \log c + d(k) \log \phi$. Using $\log \phi > 0$ we obtain

$$d(k) \leq \frac{-\log c + \log k}{\log \phi} = O(\log k),$$

concluding the proof.

Divide and Conquer

Divide and conquer is the strategy to develop an algorithm by first doing a number of recursive calls to smaller instances of the problem to be solved, and then exploit these results to obtain the desired result. As a basic example we consider Merge-sort, that sorts an array.

It is convenient to use the notation Merge-sort(A, p, q) for sorting the part of the array A starting in $A[p]$ and ending in $A[q]$, and leaving the rest of the array unchanged. So full sorting of an array A of length n is then done by calling Merge-sort$(A, 1, n)$. For a positive real number x we write $\lfloor x \rfloor$ for x rounded down to a natural number. Assume we have a linear algorithm Merge for which Merge(A, p, q, r) merges the sub-arrays $A[p..q]$ and $A[q + 1..r]$. Here we assume that $p \leq q < r$ and that these sub-arrays are already sorted. Details can be found in Section 2.3.1 of the book, page 30-34.

Now Merge-sort is defined as follows:

```
Merge-sort$(A, p, r)$:
if $p < r$ then
$q := \lfloor (p + r)/2 \rfloor$;
Merge-sort$(A, p, q)$;
Merge-sort$(A, q + 1, r)$;
```

Merge(A, p, q, r);

For correctness of the algorithm we refer to Section 2.3.1 of the book, here we focus on analyzing the complexity. Let $T(n)$ be the number of steps of the execution of Merge-sort of an array of length n. Since Merge is assumed to be linear and Merge-sort does two recursive calls for arrays of length $n/2$, we obtain

$$T(1) = 1;$$
$$T(n) = 2T(n/2) + \Theta(n).$$

Here $T(1) = 1$ is usually left implicit, and $n/2$ should be integer, so it is $n/2$ rounded, either up or down. In the worst case it is rounded up, denoted by $\lceil n/2 \rceil$. Our main goal now is to find upper bounds for $T(n)$ for T given by such a recurrence relation. In this particular case we can conclude an $n \log n$ bound.

More precisely, we prove that if $T(1) = 1$ and

$$T(n) \leq 2T(\lfloor n/2 \rfloor) + Cn$$

for all $n > 1$ for some $C \geq 1$, then $T(n) \leq 2Cn \log n$ for all $n \geq 2$. Here log is the logarithm in base 2. We do this by induction on n.

For $n = 2$ we have $T(2) \leq 2T(\lfloor 1 \rfloor) + 2C = 2 + 2C \leq 4C = 2C2 \log 2$ since $C \geq 1$, so the claim holds for $n = 2$.

For $n > 2$ we observe that $\lfloor n/2 \rfloor < n$ and we want to apply the induction hypothesis, which is only allowed if $\lfloor n/2 \rfloor \geq 2$, so $n > 3$. For $n = 3$ the property holds by $T(3) \leq 2T(1) + 3C = 2 + 3C \leq 2C3 \log 3$. For $n > 3$ we obtain

$$\begin{array}{rcll} T(n) & \leq & 2T(\lfloor n/2 \rfloor) + Cn & \text{(given)} \\ & \leq & 2(2C(\lfloor n/2 \rfloor) \log(\lfloor n/2 \rfloor) + Cn & \text{(induction hypothesis)} \\ & \leq & 2(2C(n/2) \log(n/2) + Cn & \\ & = & 2Cn((\log n) - 1) + Cn & \\ & < & 2Cn \log n, & \end{array}$$

concluding the proof.

In the merge sort example $n/2$ is also rounded up, by which exactly this result does not apply. But if instead the weaker property

$$T(n) \leq 2T(\lceil n/2 \rceil) + Cn$$

holds, or even more general:

$$T(n) \leq 2T(\lfloor n/2 \rfloor + D) + Cn$$

for some constant D, then still $T(n) = O(n \log n)$ can be proved as follows. Let $U(n) = T(n + 2D)$. Then

$$\begin{array}{rcl} U(n) & = & T(n + 2D) \\ & \leq & 2T(\lfloor (n + 2D)/2 \rfloor + D) + C(n + 2D) \\ & = & 2T(\lfloor n/2 \rfloor + 2D) + C(n + 2D) \\ & = & 2U(\lfloor n/2 \rfloor) + C(n + 2D) \\ & \leq & 2U(\lfloor n/2 \rfloor) + 2Cn \end{array}$$

for $n \geq 2D$. So we can choose C' such that

$$U(n) \leq 2U(\lfloor n/2 \rfloor) + C'n$$

for all $n > 1$. Then from the above argument we obtain $U(n) = O(n \log n)$, and then by using $U(n) = T(n + 2D)$ we obtain $T(n) = O(n \log n)$.

This approach is already an example of the *substitution method*: if you expect that $T(n) = O(f(n))$ follows from some recurrence for T, then try to prove $T(n) \leq Cf(n) + D$ by induction on n for suitable C, D.

Warning: When doing induction proofs, always first make $O(f(n))$ explicit by introducing a constant C, and then do the induction proof of a claim not involving O.

As an illustration we give a wrong proof. Let $T(n) = T(n-1) + O(n)$. Then one can prove $T(n) = O(n^2)$, but not $T(n) = O(n)$. A wrong proof of $T(n) = O(n)$ would be

$$T(n) = T(n-1) + O(n) = O(n) + O(n) = O(n),$$

using the induction hypothesis $T(n-1) = O(n)$. The reason why this is wrong is that $O(n)$ means that is below Cn for some fixed constant, while in the example by $O(n) + O(n) = O(n)$ the constant increases every step, so is not constant.

Computing the median in linear time

As a next example we consider computing the median in linear time. More precisely, for an array of n elements on which an order exists, but the elements of the array are not ordered, we want to establish which of the elements would be the $\lceil n/2 \rceil$-th in case the array was ordered. One way to do this is first to order the array, and then pick the right element. However, we want the algorithm of complexity $O(n)$, and if we first do ordering, we need $\Theta(n \log n)$, exceeding $O(n)$.

We will describe an algorithm $M(A, k)$ that gets an array A for some length n as input and a number k satisfying $1 \leq k \leq n$, and returns the k-th element of the array in case it would have been ordered. So $M(A, \lceil n/2 \rceil)$ returns the median of A. For simplicity we assume that n is divisible by 5; if not then a minor adjustment can be made not disturbing the complexity argument, as is given in Section 9.3 (pp 220-222) of the book. If $n = 5$ then we directly compute the median in constant time. Otherwise we proceed as follows:

- Split up A in $n/5$ groups of 5 element each.
- For each of these groups of 5 elements compute the median in constant time.
- Compute by recursion the median M of all these $n/5$ medians, so $n/10$ groups of 5 have a median $\leq M$, and $n/10$ groups of 5 have a median $> M$ (up to rounding).

- Compute c to be the number of elements of A that are $\leq M$.
- If $c > k$ then we know that the result of $M(A,k)$ should be $\leq M$. The algorithm returns $M(A',k)$ in which A' is obtained from A by removing the 3 largest elements of the $n/10$ groups of 5 having a median $> M$. This is correct since all these elements are $> M$.
- If $c \leq k$ then we know that the result of $M(A,k)$ should be $> M$. The algorithm returns $M(A', k - \frac{3n}{10})$ in which A' is obtained from A by removing the 3 smallest elements of the $n/10$ groups of 5 having a median $< M$. This is correct since we removed $\frac{3n}{10}$ elements all being $\leq M$.

We already gave an argument that the algorithm returns the right value, ignoring rounding effects. Now we will investigate the complexity. Let $T(n)$ be the number of steps of the algorithm $M(A,k)$ for A being an array of n elements. The algorithm is recursive on two positions: first it is applied recursively on $n/5$ medians of groups of 5, and next it is applied recursively on an array A' having size $\frac{7n}{10}$ in both cases. For the rest the algorithm does a linear number of steps, to compute the medians of $n/5$ groups of 5, and count the number of elements of A that are $\leq M$ and remove elements from A to obtain A'. So we obtain

$$T(n) = T(\frac{n}{5}) + T(\frac{7n}{10}) + O(n).$$

Now we will prove that $T(n) = O(n)$. Choose C such that $T(n) \leq 10Cn$ for small values of n, and

$$T(n) \leq T(\frac{n}{5}) + T(\frac{7n}{10}) + Cn$$

for large values of n. We will prove by induction that $T(n) \leq 10Cn$ for all values of n. For small values of n this holds by assumption, for greater values we obtain

$$\begin{array}{rcll} T(n) & \leq & T(\frac{n}{5}) + T(\frac{7n}{10}) + Cn & \text{(by assumption)} \\ & \leq & 10C\frac{n}{5} + 10C\frac{7n}{10} + Cn & \text{(by induction hypothesis } 2\times\text{)} \\ & = & 2Cn + 7Cn + Cn & \\ & = & 10Cn, & \end{array}$$

concluding the proof.

Again this is an example of the *substitution method.* An obvious question is how to find the magic right value of the constant, in this case 10, that does the job. A way to do this is build a *recursion tree* and see what happens. For details about the recursion tree method we refer to the book.

Finally, we often meet the pattern

$$T(n) = aT(n/b) + f(n).$$

Complexity bounds for most recurrences of this shape are given by the *Master Theorem*. For details about the Master Theorem method we refer to the book.

As an application, we consider Karatsuba's algorithm for multiplication of big numbers X and Y of n binary digits, in which n is a large number. Assume n is even, otherwise add a leading 0. Let a be the number represented by the first $n/2$ bits of X and b be the number represented by the remaining $n/2$ bits of X, so $X = a2^{n/2} + b$, and similarly $Y = c2^{n/2} + d$. We want to compute

$$XY = ac2^n + (ad + bc)2^{n/2} + bd.$$

A possible way to do this is by computing all four products ac, ad, bc and bd recursively. Since the rest consists of addition and operations that are linear in n, we then would obtain

$$T(n) = 4T(n/2) + O(n),$$

for which case 1 of the Master Theorem applies for $a = 4$, $b = 2$, yielding

$$T(n) = \Theta(n^{\log_b a}) = \Theta(n^{\log_2 4}) = \Theta(n^2),$$

being the same complexity of the multiplication algorithm as we learned in school. In the recurrence for odd n the $n/2$ should be interpreted as $\lceil n/2 \rceil$, which is allowed by the Master Theorem.

However, it can be done better, and that's by the following algorithm found by Karatsuba in 1960. Instead of computing the four products ac, ad, bc and bd recursively, we compute

$$\begin{array}{rcl} X_1 & = & a + b, \\ X_2 & = & c + d, \\ X_3 & = & X_1 X_2, \\ X_4 & = & ac, \\ X_5 & = & bd, \\ X_6 & = & X_3 - X_4 - X_5, \end{array}$$

in which only three recursive product computations are required for computing X_3, X_4 and X_5, while the other computations only consist of addition and subtraction and can be done in $O(n)$. Since $X_3 = X_1 X_2 = ac + ad + bc + bd$, we obtain $X_6 = ad + bc$. So by returning

$$XY = X_4 2^n + X_6 2^{n/2} + X_5$$

the algorithm returns the right answer, but now has a complexity determined by

$$T(n) = 3T(n/2) + O(n).$$

Now case 1 of the Master Theorem applies for $a = 3$, $b = 2$, yielding

$$T(n) = \Theta(n^{\log_b a}) = \Theta(n^{\log_2 3}) = \Theta(n^{1.584}).$$

For large n this is much better than $\Theta(n^2)$.

By a similar trick the standard $\Theta(n^3)$ algorithm for multiplying two $n \times n$ matrices can be improved to $\Theta(n^{\log_2 7}) = \Theta(n^{2.8074})$. For details about this algorithm by Strassen we refer to the book (pp 79-82). By combining this result with case 3 of the Master Theorem, one shows that also matrix inversion can be done in $\Theta(n^{\log_2 7}) = \Theta(n^{2.8074})$. For details we refer to the book: pp 827-831.

Computational Geometry

As a series of interesting algorithms for geometrical problems having complexity $O(n \log n)$, Chapter 33 of the book is considered. In all cases the geometrical problems are in the two-dimensional plane, and every point is given by two numbers: the x-coordinate and the y-coordinate. In particular it is shown how

- from two segments each given by its two end points, we can establish in constant time whether they intersect or not;
- as an example of a *sweep-line* algorithm, how for n segments given by their end points, it is established in $O(n \log n)$ whether there exist two of them that intersect;
- how for a set of n points its *convex hull* is computed in $O(n \log n)$, by the *Graham scan*;
- how for a set of n points the smallest distance of any two of them can be determined in $O(n \log n)$, by a *divide and conquer* algorithm.

In earlier years (until 2016) all these algorithms were presented. In order to have a better focus on core complexity issues, since 2017 only the last *divide and conquer* algorithm were presented, for which the resulting complexity is an instance of the Master Theorem. Since we exactly follow the book, the details are not elaborated in these notes.

NP-completeness

Again we mainly follow the book (Chapter 34), so here we only present some highlights and topics that differ from the book.

The main issue is the question whether for particular problems efficient algorithms exist or not, so it is not about algorithms but about problems for which an algorithm is desired.

A next main issue is that the focus is on *decision problems*, that is, problems of which the desired answer is only 'yes' or 'no'. So finding the shortest path from one node to another in a graph is not a decision problem, but determine whether a path of length n from one node to another exists in a graph is a decision problem.

How to represent decisions problems? The input for an algorithm can be anything, but if it is finite, then it can always be represented in a finite Boolean string. So a decision problem can be described as a set of finite Boolean strings, namely the set of strings that are a correct encoding of a possible input for the algorithm, and on which the intended answer of the decision problem is 'yes'.

In theoretical computer science a set of strings over some alphabet (in this case only the booleans) is called a *language* over the alphabet. So decision problems can be identified with languages. So for our shortest path example, we choose an encoding in booleans of a finite graph, two nodes p, q in this graph and a number n, and then the language consists of all strings that are a correct encoding of the combination of a graph, two nodes p, q and a number n, for which the graph contains a path of length n from p to q. Clearly this depends on the chosen representation, but as different choices for representations can be transformed to each other in polynomial time, for complexity analysis the particular choice of the representation does not matter.

Since we know that this path decision problem admits a polynomial algorithm, we conclude that the corresponding language is in the class P, defined as follows:

$$\begin{aligned} \mathrm{P} \;=\; \{L \subseteq \{0,1\}^* \mid &\text{ there is } k \text{ and an algorithm that decides} \\ &\qquad\qquad \text{for } x \in \{0,1\}^* \text{ whether } x \in L, \text{ in } O(|x|^k)\}. \end{aligned}$$

The notion of such a decision algorithm can be made more precise by defining it to be a (deterministic) Turing machine that is halting for every input, and when it runs with the input $x \in \{0,1\}^*$ on the initial tape, then it will end in a special accepting state if and only if $x \in L$. The requirement of being polynomial then means that the number of Turing machine steps of this computation is $O(|x|^k)$ for some k.

For many other problems it is unlikely that they are in P. But often they are in NP: then there is a notion of a *certificate* such that for every input x for the decision problem the answer is 'yes' if and only if for x a corresponding certificate y exists, and the length of y and the algorithm for checking the certificate should be polynomial in the length of x. More precisely:

$$\begin{aligned} \mathrm{NP} \;=\; \{L \subseteq \{0,1\}^* \mid &\text{ there is } k, c \text{ and a polynomial algorithm } A \text{ such that} \\ &\quad L = \{x \in \{0,1\}^* \mid \exists y \in \{0,1\}^* : |y| < c|x|^k \wedge A(x,y) = 1\}\}. \end{aligned}$$

The abbreviation NP stands for *non-deterministic polynomial*. It can be shown that this class NP coincides with the class of languages accepted by non-deterministic Turing machines, that is, x is in the language if and only if a Turing machine computation exists ending in the accepting state. To get a feeling about this equivalence, you may think of a non-deterministic Turing machine that first creates the certificate y nondeterministically, and then runs the deterministic algorithm A. These remarks are mainly for motivating the name and abbreviation; for really dealing with NP it is most convenient to use the above definition.

The classes P and NP are about time complexity. One may also define PSPACE for the class of problems that can be described by algorithms with polynomial space complexity, NL for non-deterministic logarithmic complexity, and EXPTIME and EXPSPACE for exponential complexity. One obtains

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

It is likely that all inclusions are strict, so none of them are equalities, but for none of them this has been proved. Our focus is on P and NP; at the end also some remarks on PSPACE will be made. Apart from NP there is also the class co-NP: the class of languages for which the complement is in NP. There are also problems in NP $\cap$ co-NP for which no polynomial algorithm is known, the most important one being *primality*: given a positive number, decide whether it is a prime number or not.

A key notion in analyzing P and NP (and also the other classes) is *reduce to*: a decision problem L_1 reduces to a decision problem L_2, notation $L_1 \leq_P L_2$, if a polynomially computable function $f : \{0,1\}^* \to \{0,1\}^*$ exists such that

$$x \in L_1 \iff f(x) \in L_2$$

for all $x \in \{0,1\}^*$. This means that if we can decide in polynomial time whether $x \in L_2$, then we can do the same for L_1, namely by applying f and then check whether the result is in L_2. So roughly speaking: L_2 is at least as hard as L_1. The function f is called a *reduction function*.

The relation $\leq_P$ is *transitive*: if $L_1 \leq_P L_2$ and $L_2 \leq_P L_3$ then $L_1 \leq_P L_3$. This is proved as follows: if $x \in L_1 \iff f(x) \in L_2$ and $x \in L_2 \iff g(x) \in L_3$ for polynomially computable f, g, then

$$x \in L_1 \iff f(x) \in L_2 \iff g(f(x)) \in L_3,$$

which proves $L_1 \leq_P L_3$ since the function mapping x to $g(f(x))$ is the composition of two polynomially computable functions, hence polynomially computable itself.

A decision problem L is called *NP-hard* if $L' \leq_P L$ for every $L' \in$ NP. Roughly speaking, this means that every problem in NP is at least as hard as L.

The following is the key theorem to prove that a decision problem L is NP-hard.

Theorem:
If $L' \leq_P L$ and L' is NP-hard, then L is NP-hard.

Proof: Let L'' be an arbitrary NP-hard problem. Since L' is NP-hard, we have $L'' \leq_P L'$. Since $L' \leq_P L$ and $\leq_P$ is transitive, we conclude $L'' \leq_P L$. This proves that L is NP-hard. □

This theorem provides the main way to prove that a decision problem L is NP-hard:

> Choose a decision problem L' for which NP-hardness is already known, and then prove $L' \leq_P L$, more precisely, find a polynomially computable function $f : \{0,1\}^* \to \{0,1\}^*$ such that
>
> $$x \in L' \iff f(x) \in L$$
>
> for all $x \in \{0,1\}^*$.

A decision problem L is called *NP-complete* if it is NP-hard and it is in NP. So roughly speaking, all NP-complete problems are equally hard, up to polynomial complexity. For proving NP-completeness one has to prove two things: it is in NP, for which a notion of a certificate has to be introduced that can be checked in polynomial time, and it is NP-hard, to be done along the lines sketched above.

But the whole machinery only works if we can start by a particular problem for which NP-hardness is known. The particular problem that serves for this is *satisfiability.*

Satisfiability

The mother of all NP-complete problems is SAT: satisfiability, in particular for CNF, as we will explain now.

A formula composed from Boolean variables and operations $\neg$, $\vee$, $\wedge$ is called a *propositional formula.* Such a propositional formula is called *satisfiable* if it is possible to give Boolean values to the variables in such a way that the formula yields true.

So $p \wedge (\neg p \vee \neg q)$ is satisfiable, since if p is true and q is false the formula yields true.

The formula $p \wedge (\neg p \vee \neg q) \wedge q$ is unsatisfiable, since for all four ways to give values to p and q it yields false.

A *conjunctive normal form (CNF)* is a conjunction of clauses.

A *clause* is a disjunction of literals.

A *literal* is either a variable or the negation of a variable.

Hence a CNF is of the shape

$$\bigwedge_i (\bigvee_j \ell_{ij})$$

where ℓ_{ij} are literals. A CNF is satisfiable if and only if it is possible to give values to the variables such that in every clause at least one of the literals yields true.

Now we arrive at the main result: CNF-SAT, that is, the decision problem whether a given CNF is satisfiable, is NP-complete. This originates from Stephen

Cook and Leonid Levin in 1970. Our proof differs from the one given in the book in two ways: in the book it is done for circuit satisfiability, and the argument in the book is based on a machine model that is only sketched, while we use the more standard and well-known Turing machines.

Theorem: (Cook-Levin)
CNF-SAT is NP-complete.

Proof:
First we prove that CNF-SAT is in NP. In order to do so we have to define the corresponding notion of certificate. As a certificate for a given CNF we choose an assignment to the boolean variables for which the CNF yields true. Indeed the CNF is satisfiable if and only if such a certificate exists, by definition. Next we have to check that verifying whether a candidate for a certificate is indeed a certificate can be done in polynomial time. This can be done even in linear time: for every clause check whether the assignment yields true. So CNF-SAT is in NP.

It remains to prove that CNF-SAT is NP-hard, that is, for every $L \in$ NP we have $L \leq_P$ CNF-SAT. So we take an arbitrary L in NP, and we have to find a polynomial algorithm f yielding a CNF such that

$$x \in L \iff f(x) \in \text{CNF-SAT}.$$

Since L is in NP, there is a polynomial algorithm A such that

$$x \in L \iff \exists y \in \{0,1\}^*, |y| = O(|x|^c), A(x,y) = 1.$$

We describe A by a Turing machine (Q, Σ, δ), that is, if the Turing machine starts in state $q_0 \in Q$, and the tape contains x, y, right from the tape head, separated by marking symbols, then the Turing machine computation will reach state $q_F \in Q$ if and only if $A(x,y) = 1$. Due to the polynomiality assumption, we assume that both the size of the part of the tape that is used in this computation and the total number of steps in this computation is bounded by some number n, being polynomial in $|x|$.

Next we encode the Turing machine configuration consisting of the tape content $a_1, a_2, \ldots, a_n$, where the machine head points to a_k, and the machine is in state $q \in Q$ by the string

$$a_1 a_2 \cdots a_k q a_{k+1} \cdots a_n.$$

For every $i = 0, \ldots, n$, $j = 1, \ldots, n+1$, $a \in \Sigma \cup Q$ we introduce a boolean variable $p_{i,j,a}$, with the intended meaning that this variable is true if and only if after i step of computation, the symbol a is on position j in the above string encoding for Turing machine configurations. This is quite a big number of variables, but polynomial in $|x|$.

Next we build a formula in CNF expressing this intended meaning. The CNF is the conjunction of many parts that we now describe subsequently. Everywhere i runs from 0 to n and j runs from 1 to $n+1$.

The first part is

$$\bigwedge_{i,j}\Big(\big(\bigvee_{a\in\Sigma\cup Q} p_{i,j,a}\big) \wedge \bigwedge_{a,b\in\Sigma\cup Q, a\neq b} (\neg p_{i,j,a} \vee \neg p_{i,j,b})\Big),$$

expressing that for every i, j exactly one of the variables $p_{i,j,a}$ has the value true.

The second part describes the initial configuration, containing many clauses, mostly consisting of a single variable:

- $p_{0,1,q_0}$,
- $p_{0,j+1,0}$ for all x-positions j for which $x_j = 0$,
- $p_{0,j+1,1}$ for all x-positions j for which $x_j = 1$,
- $p_{0,j,0} \vee p_{0,j,1}$ for all positions j for y, denoting that this should be either 0 or 1, but both are allowed due to '$\exists y$' in the requirement for A,
- $p_{0,j,m}$ for the marked positions j marking the ends of x and y, for the marking symbol $m \in \Sigma$,
- $p_{0,j,\square}$ for all other positions j, for the blank symbol $\square$.

The next part describes the Turing machine computation. If $\delta(q,a) = (q', a', R)$, then the part aqb in the string encoding should be replaced by $a'bq'$: the head points to a, this a is replaced by a', the state q is replaced by q', and this q' is shifted one position to the right, while all other elements in the string remain unchanged. If the three positions in aqb are j, $j+1$ and $j+2$, his is expressed by

$$(p_{i,j,a} \wedge p_{i,j+1,q} \wedge p_{i,j+2,b}) \rightarrow (p_{i+1,j,a'} \wedge p_{i+1,j+1,b} \wedge p_{i+1,j+2,q'})$$

being expressed in CNF by the three clauses

$$\neg p_{i,j,a} \vee \neg p_{i,j+1,q} \vee \neg p_{i,j+2,b} \vee p_{i+1,j,a'},$$

$$\neg p_{i,j,a} \vee \neg p_{i,j+1,q} \vee \neg p_{i,j+2,b} \vee p_{i+1,j+1,b},$$

$$\neg p_{i,j,a} \vee \neg p_{i,j+1,q} \vee \neg p_{i,j+2,b} \vee p_{i+1,j+2,q'},$$

and for every $c \in Q \cup \Sigma$ and for every $k < j$ and every $k > j+2$ the requirement

$$(p_{i,j,a} \wedge p_{i,j+1,q} \wedge p_{i,j+2,b}) \rightarrow (p_{i+1,k,c} \leftrightarrow p_{i,k,c})$$

being expressed in CNF by the two clauses

$$\neg p_{i,j,a} \vee \neg p_{i,j+1,q} \vee \neg p_{i,j+2,b} \vee p_{i+1,k,c} \vee \neg p_{i,k,c},$$

$$\neg p_{i,j,a} \vee \neg p_{i,j+1,q} \vee \neg p_{i,j+2,b} \vee \neg p_{i+1,k,c} \vee p_{i,k,c}.$$

This is done for all i, j, q, a, b satisfying $\delta(q, a) = (q', a', R)$. A similar construction is given for for $\delta(q, a) = (q', a', L)$, yielding a great number of clauses, but still polynomial in $|x|$.

Finally, we express the requirement that somewhere the Turing machine computation will reach state $q_F \in Q$ by a single clause

$$\bigvee_{i,j} p_{i,j,q_F}.$$

The resulting formula is $f(x)$, and is composed in polynomial time. It has been composed in such a way that in any satisfying assignment the values of $p_{0,j,a}$ describe an initial configuration in which the positions of x are given by the value of x, and the positions of y may be any values in $\{0, 1\}$. Furthermore, the formula forces that for $i = 1, \ldots, n$, the values of $p_{i,j,a}$ exactly describe the deterministic Turing machine computation. Since we added $\bigvee_{i,j} p_{i,j,q_F}$, we conclude by the assumption on the algorithm A that $f(x)$ is satisfiable if and only if $\exists y : A(x, y) = 1$, concluding the proof.

Variants of SAT

As we have proved that CNF-SAT is NP-hard, we have a method to prove NP-hardness of another decision problem L: prove that CNF-SAT $\leq_P L$, that is, give a polynomial transformation from an arbitrary CNF to an instance of the new problem L, such that the CNF is satisfiable if and only if the property described by L holds for the result of the transformation.

The first examples for which we apply this are variants of CNF-SAT, in particular

- $\leq$3-SAT: satisfiability of CNFs in which every clause contains at most 3 literals, and
- 3-SAT: satisfiability of CNFs in which every clause contains exactly 3 literals.

For $\leq$3-SAT the main trick is to replace a a big clause $\bigvee_{i=1}^{n} \ell_i$ in a CNF by two smaller clauses, using a fresh variable A:

Lemma.

Let $n \geq 4$, $\ell_1, \ldots, \ell_n$ be literals and F be a CNF. Let A be a fresh variable, then $F \wedge \bigvee_{i=1}^{n} \ell_i$ is satisfiable if and only if

$$F \wedge (\ell_1 \vee \ell_2 \vee A) \wedge (\neg A \vee \bigvee_{i=3}^{n} \ell_i)$$

is satisfiable.

Proof.

Take a satisfying assignment of $F \wedge \bigvee_{i=1}^{n} \ell_i$. If in this assignment $\ell_1 \vee \ell_2$ is true then give A the value false, otherwise give A the value true. Then in both cases $F \wedge (\ell_1 \vee \ell_2 \vee A) \wedge (\neg A \vee \bigvee_{i=3}^{n} \ell_i)$ has the value true, so is satisfiable.

Conversely, take a satisfying assignment of $F \wedge (\ell_1 \vee \ell_2 \vee A) \wedge (\neg A \vee \bigvee_{i=3}^{n} \ell_i)$. If here A has the value true, then $\bigvee_{i=3}^{n} \ell_i)$, holds, otherwise $\ell_1 \vee \ell_2$ holds. So in both cases $\bigvee_{i=1}^{n} \ell_i$ holds. As also F yields true, in both case we have a satisfying assignment for $F \wedge \bigvee_{i=1}^{n} \ell_i$.

End of proof.

For proving that ≤3-SAT is NP-hard we have to transform an arbitrary CNF to a CNF in which every clause has at most 3 literals, such that the original CNF is satisfiable if and only if the transformed CNF is satisfiable. Exploiting the above lemma this is easily done: as long as a clause with ≥ 4 literals exist, introduce a fresh variable and apply the construction of the lemma. As in the construction a clause of $n \geq 4$ literals is replaced by a clause with 3 literal and a clause with $n-1$ literals, after repeating this $O(k)$ times for k being the size of the formula, we end up in a CNF in which every clause has at most 3 literals, and the original CNF is satisfiable if and only if the transformed CNF is satisfiable, since this hold for every step separately. As this transformation is polynomial, we have proved CNF-SAT $\leq_P$ ≤3-SAT, and we conclude:

≤3-SAT is NP-hard.

The next step is to prove ≤3-SAT $\leq_P$ 3-SAT. To this end, we have to extend clauses with less then three literals to exactly three literals, without disturbing satisfiability. This is done by adding fresh variables a, p, q, a' and eight clauses

$$\begin{array}{ll}
a \vee p \vee q, & a' \vee p \vee q, \\
a \vee p \vee \neg q, & a' \vee p \vee \neg q, \\
a \vee \neg p \vee q, & a' \vee \neg p \vee q, \\
a \vee \neg p \vee \neg q, & a' \vee \neg p \vee \neg q.
\end{array}$$

Al these clauses have exactly three literals, but force that both a and a' have to be true: for all four possible values of p and q this can be concluded. Hence adding $\neg a$ or $\neg a'$ to other clauses has no influence on satisfiability. So apart from adding these eight clauses, to every clause with one literal we add the literals $\neg a$ and $\neg a'$, and to every clause with two literals we add the literal $\neg a$. As this transformation is polynomial and the original CNF (which may contain clauses with < 3 literals) is satisfiable if and only if the resulting CNF (in which every clause has exactly three literals) is satisfiable, this proves ≤3-SAT $\leq_P$ 3-SAT, and we conclude:

3-SAT is NP-hard.

One may wonder whether one may go further. For instance: only allowing clauses with exactly two literals. But this is not the case: it can be proved that 2-SAT, that satisfiability of CNF's in which all clauses consist of two literals, can be solved in polynomial time.

One minor further step is possible: we may consider CNFs in which all clauses have exactly three literals and for every boolean variable p the literals p and $\neg p$ both occur in the CNF. Let's temporarily write 3o-SAT for satisfiability of this kind of CNFs. We will show that $\leq$3-SAT $\leq_P$ 3o-SAT. The transformation is the same as the one above for $\leq$3-SAT $\leq_P$ 3-SAT, with the following modification. First for every boolean variable p for which p does not occur in any clause of the CNF we remove every clause containing $\neg p$. This has no influence on satisfiability: if we give p the value false, then the original CNF yields true if and only if after this removal it yields true. Similarly, for every p for which $\neg p$ does not occur in any clause of the CNF we remove every clause containing p. Next we do the same trick as above by adding a and a' until all clauses have size 3. Since the original CNF is satisfiable if and only if the transformed one is satisfiable, we have proved $\leq$3-SAT $\leq_P$ 3o-SAT, and conclude that also 3o-SAT is NP-hard, and since checking validity of a satisfying assignment is still polynomial, it is NP-complete. In the future we do not distinguish between 3-SAT and 3o-SAT, and simply assume for a 3-SAT instance that all variables occur both positive and negative if desired.

Other NP-complete problems

For thousands of decision problems NP-completeness has been proved. For many of them we exactly follow the book and therefore they are not treated here. In particular this holds for the subset sum problem and for the graph problems clique, vertex cover and 3-colorability. Here we treat integer linear programming and the Mahjong game.

Integer linear programming (ILP) refers to the problem: given a system of linear inequalities with coefficients in the integers, does it have a solution in the integers?

For instance, consider the three inequalities

$$x + 3y \geq 5, 3x + y \leq 6, 3x - 2y \geq 0$$

has real valued solutions, like $x = 1, y = \frac{3}{2}$, but it has no integer solutions.

Here we prove that ILP is NP-complete. First we show it is in NP: as the notion of certificate we choose the solution: by definition an ILP problem has a solution if and only if such a certificate exists, and checking whether a solution is indeed a solution can be done in polynomial time.

It remains to show that ILP is NP-hard. We do this by proving CNF-SAT $\leq_P$ ILP. So we have to transform an arbitrary CNF to an ILP problem in such a way that the CNF is satisfiable, if and only if the resulting ILP problem has a solution. Let $x_1, \ldots, x_n$ the boolean variables in the CNF; we will use the same symbols for the integer valued variables in the ILP problem. First we create the two inequalities

$$x_i \geq 0, \quad x_i \leq 1$$

for every $i = 1, \ldots, n$. Hence all solutions of the inequalities are solutions in $\{0, 1\}$, which are identified with booleans, where 0 corresponds to false and 1 with true. Next for every clause an inequality is generated, by replacing every '$\vee$' by $+$, every x_i without a negation by x_i, and every $\neg x_i$ by $(1-x_i)$. Finally, the resulting expression is concluded by '≥ 1'. As an example, the clause $x_1 \vee x_3 \vee \neg x_5$ is transformed to $x_1 + x_3 + (1 - x_5) \geq 1$. The key observation is that for any boolean assignment to $x_1, \ldots, x_n$, a clause yields true if and only if the resulting inequality holds. Indeed, $1 - x$ represents $\neg x$ for $x \in \{0, 1\}$, and a clause yields true, if and only at least one of its literals yields true. As all these literals have value 0 or 1, this is equivalent to stating that their sum is ≥ 1. Hence the resulting set of inequalities has a solution in the integers, if and only if the CNF is satisfiable, concluding the proof.

Mahjong

Not only problems from graph theory or mathematics are NP-complete, also several puzzles and games give rise to NP-complete problems. Here consider the Mahjong game, but several other candidates are available. A perfect topic for a bachelor thesis is elaborating an NP-completeness proof for some puzzle or game of your own choice, see for instance
`http://www.ics.uci.edu/~eppstein/cgt/hard.html#shang`. The following is inspired by a proof presented by Michiel de Bondt.

Mahjong is a Chinese game, and can be played with more people, but also individually, the latter sometimes called Shanghai. The game consists of 144 tiles with 36 pictures, each occurring 4 times (some pictures belonging together are not completely identical, but this has no influence on the game). A tile is free if there is no tile on top of it, and either left or right (or both) it is not directly connected to another tile.

At the start of the game the 144 tiles are put in some configuration, and the goal of (the individual version of) the game is to repeat removing two equal free tiles until all tiles have been removed. When playing this game (on the Internet several free versions are available) one observes that it is far from clear whether this is possible or not for a given initial configuration. One can think of establishing this by backtracking, but then the algorithm will be quite inefficient. When keeping the number of 144 tiles fixed it remains a finite problem, and it is hard to speak about complexity. But a natural generalization is to play the game for $4n$ tiles consisting of n groups of 4 with equal pictures. So we consider the decision problem Mahjong for which the input is a configuration of $4n$ tiles, and it has to be decided whether it is possible to repeat removing two equal free tiles until all tiles have been removed. We will argue that the existence of an efficient algorithm is unlikely by proving that the problem is NP-hard, even for the simple version in which never tiles are on top of other tiles.

We will do this by proving 3-SAT $\leq_P$ Mahjong. We do this in two steps: first we consider a variant of Mahjong that we call Stack Mahjong, and we will prove Stack Mahjong $\leq_P$ Mahjong and 3-SAT $\leq_P$ Stack Mahjong.

In Stack Mahjong there are also n different tiles each occurring in 4 copies, but now they are arranged in stacks of which only the top element is free. Moreover, there are two types of stacks: normal stacks for which after removal of the top a stack remains for which only the top becomes free, and flat stacks for which after removal of the top all underlying tiles become free. The decision problem is whether pairs of equal free tiles can be removed until all tiles are removed. As an instance consider two normal stacks and a flat stack, denoting a normal stack simply by the elements on top of each other, and a flat stack by a horizontal line with the top element on top and the other elements below:

$$\begin{array}{ccc} a & a & \\ b & c & \dfrac{a}{abbcc} \\ c & b & \end{array}$$

There are three types of tiles: a, b and c, each occurring four times. Initially only the three a's on top are free. One may play by choosing the two a's on top of the two leftmost (normal) stacks. Then we obtain three stacks: two normal ones with b and c on top, and one flat stack with a on top. However, now no further move is possible. Instead we could start by choosing the top of the leftmost stack and the top of the flat stack, after which two normal stacks with top b and a remain and five singletons a, b, b, c, c that are all free. By consecutively choosing the top of a normal stack and one of these singletons, all tiles can be removed now.

Now we show that Stack Mahjong $\leq_P$ Mahjong, that is, we transform any Stack Mahjong instance to a normal Mahjong instance in a polynomial way, such that the original instance has a solution if and only if the transformed one has a solution. In his transformation we introduce several fresh tiles, each occurring twice rather than four times, but at the end they are doubled by which a valid Mahjong instance is obtained.

A normal stack is transformed to a row, starting from the left by the top, and at the end followed by two copies of a fresh tile p. For these normal stacks the steps that are allowed in Stack Mahjong are the same as steps allowed in Mahjong: the top can be played since it is free at the left, and the p at the right end can not be played as the other occurrence of p is hidden as long as the stack is not empty. As soon as the stack is empty, a row of two ps remains that can be fully removed in one step by playing the outside free two ps.

A flat stack with n tiles $a_1, \ldots, a_n$ below the top a is first transformed to $n+1$ normal stacks introducing n fresh tiles $p_1, \ldots, p_n$: one of size $n+1$ with a on top and $p_1, \ldots, p_n$ below, and for every $i = 1, \ldots, n$ a stack of size 2 with p_i on top and a_i below. This is chosen in such a way that the only way to play the new stacks is to mimic playing the flat stack. After this the above transformation from normal stacks to Mahjong rows applied to all these $n+1$ normal stacks.

Finally, in order to meet the requirement that every tile occurs exactly 4 times, every occurrence of a fresh tile is replaced by two copies next to each other, without changing the possible moves.

The full transformation applied to the above Stack Mahjong example yields the following Mahjong instance:

$$abcq_1q_1q_1q_1$$

$$acbq_2q_2q_2q_2$$

$$ap_1p_1p_2p_2p_3p_3p_4p_4p_5p_5q_3q_3q_3q_3$$

$$p_1p_1aq_4q_4q_4q_4$$

$$p_2p_2bq_5q_5q_5q_5$$

$$p_3p_3bq_6q_6q_6q_6$$

$$p_4p_4cq_7q_7q_7q_7$$

$$p_5p_5cq_8q_8q_8q_8$$

in which $p_1, \ldots, p_5$ are the fresh variables for transforming flat stacks to normal stacks, and $q_1, \ldots, p_8$ are the fresh variables for transforming normal stacks to Mahjong rows, indeed all occurring exactly four times. Since this construction is polynomial and every successful Stack Mahjong play can be mimicked by a successful Mahjong play, and conversely, this proves that Stack Mahjong $\leq_P$ Mahjong.

So it remains to show 3-SAT $\leq_P$ Stack Mahjong. To this end we start by an arbitrary CNF in which every clause consists of exactly 3 clauses. Let the clauses be $C_1, \ldots, C_k$ and the variables be $p_1, \ldots, p_n$. These $C_1, \ldots, C_k$, $p_1, \ldots, p_n$ also serve as the names of the tiles in the Stack Mahjong instance we will construct now. It consists of one big normal stack with p_1 on top, followed by $p_2, \ldots, p_n$, next followed by $C_1, \ldots, C_k$, and finally again followed by $p_1, \ldots, p_n$. Moreover, there are $2n$ flat stacks. For every $i = 1, \ldots, n$ there is a flat stack with p_i on top, and below all names of clauses in which the literal p occurs. Finally, for every $i = 1, \ldots, n$ there is a flat stack with p_i on top, and below all names of clauses in which the literal $\neg p$ occurs. Note that in this way all tiles $C_1, \ldots, C_k$, $p_1, \ldots, p_n$ occur exactly four times. Now the claim is that this resulting Stack Mahjong game admits a solution, if and only if the original CNF is satisfiable.

Assume the CNF is satisfiable. Then it admits a satisfying assignment, and we play the following Stack Mahjong game. First we do the following for $i = 1, \ldots, n$. If p_i is true then we play the top of the normal stack with the top of the flat stack with p_i on top and the clauses below in which p_i occurs. If p_i is false then we play the top of the normal stack with the top of the flat stack with p_i on top and the clauses below in which $\neg p_i$ occurs. Since every clause contains a literal that yields true, this process yields an occurrence of a free singleton C_i for every $i = 1, \ldots, k$. So next we can play $C_1, \ldots, C_k$ consecutively, combining the top of the normal stack and a corresponding free singleton C_i. We continue by consecutively playing the remaining $p_1, \ldots, p_n$ of the normal stack with the tops of remaining flat stacks. In this way all p-tiles have been removed. Now for every $i = 1, \ldots, k$ exactly two free singletons C_i remain that can be played away, by which all tiles have been removed, solving the game.

Conversely assume we have a solution of the Stack Mahjong game. Then before the C-s in the normal stack have been played, every p_i on top of it has been played with one of the flat stacks with p_i on top. If it is the one for p_i, then give p_i the value true, otherwise the value false. When C_i is played, it is played with a free singleton, and that is only made free if C_i contains a literal that is true in the above defined assignment. This holds for all $i = 1, \dots, k$, by which every clause $C_1, \dots, C_k$ yields true, so the CNF is satisfiable. This concludes the proof that Stack Mahjong, and hence Mahjong, is NP-hard. Since checking whether a played game is valid is done in polynomial time, it is also NP-complete.

As an example, consider the CNF consisting of the following three clauses:

$$\begin{array}{lr} C_1: & p_1 \vee p_2 \vee p_3 \\ C_2: & p_1 \vee p_2 \vee p_4 \\ C_3: & \neg p_1 \vee \neg p_3 \vee p_4 \end{array}$$

Then the above construction yields the following Stack Mahjong instance:

$$\begin{array}{c} p_1 \\ p_2 \\ p_3 \\ p_4 \\ C_1 \\ C_2 \\ C_3 \\ p_1 \\ p_2 \\ p_3 \\ p_4 \end{array} \qquad \frac{p_1}{C_1C_2} \qquad \frac{p_2}{C_1C_2} \qquad \frac{p_3}{C_1} \qquad \frac{p_4}{C_2C_3} \qquad \frac{p_1}{C_3} \qquad p_2 \qquad \frac{p_3}{C_3} \qquad p_4,$$

consisting of one normal stack of size 11 and 8 flat stacks, of which two are already singletons since $\neg p_2$ and $\neg p_4$ do not occur in the CNF. The CNF is satisfiable, for instance, a satisfying assignment is $p_1 = p_2 =$ true, $p_3 = p_4 =$ false. Playing the four ps on top of the normal stack with the corresponding flat stacks yields

$$\begin{array}{c} C_1 \\ C_2 \\ C_3 \\ p_1 \\ p_2 \\ p_3 \\ p_4 \end{array} \qquad C_1 \qquad C_2 \qquad C_1 \qquad C_2 \qquad \frac{p_3}{C_1} \qquad \frac{p_4}{C_2C_3} \qquad \frac{p_1}{C_3} \qquad p_2 \qquad C_3.$$

Next we see free singletons C_1, C_2, C_3, so playing these with the Cs on the normal stack yields

$$\begin{array}{c} p_1 \\ p_2 \\ p_3 \\ p_4 \end{array} \qquad C_1 \qquad C_2 \qquad \frac{p_3}{C_1} \qquad \frac{p_4}{C_2C_3} \qquad \frac{p_1}{C_3} \qquad p_2.$$

Next, playing the remaining p_1, p_2, p_3, p_4 in the normal stack with the tops of the remaining flat stacks yields 6 free singletons $C_1, C_2, C_1, C_2, C_3, C_3$ that are removed in three steps concluding the solution of the game.

1 PSPACE-completeness

By definition, for decision problems in NP there is a notion of certificate that can be checked in polynomial time. But there are also natural classes of problems for which this does not hold. For instance, in a two-player game, for the question whether white can win in a limited number of steps, the certificate would be a winning strategy, but this winning strategy is of exponential size since for every play of black an answer by white has to be defined, typically consisting of a number of cases that is exponential in the number of steps. Such decision problems tend to be in the class PSPACE, and are often PSPACE-complete, as will be defined now.

Recall that P is the class of decision problems L for which a polynomial decision algorithm exists, more precisely, a (deterministic) Turing machine that is halting in at most a polynomial number of steps for every input, and when it runs with the input $x \in \{0,1\}^*$ on the initial tape, then it will end in a special accepting state if and only if $x \in L$. Now PSPACE is defined in the same way, in which the requirement of a polynomial number of steps is replaced by a polynomial use of space = memory. More precisely:

> PSPACE is the class of decision problems L for which a deterministic Turing machine M exists for which
>
> - M is halting for every input, and
> - when M runs with the input $x \in \{0,1\}^*$ on the initial tape, then it will end in a special accepting state if and only if $x \in L$, and
> - the size of the part of tape that is used in this run of M, is polynomial in $|x|$.

So PSPACE is the class of decision problems that can be decided by polynomial space decision algorithms. From the definition it is obvious that P $\subseteq$ PSPACE. But it also holds that NP $\subseteq$ PSPACE: by definition for every language L in NP there is k, c and a polynomial algorithm A such that

$$L = \{x \in \{0,1\}^* \mid \exists y \in \{0,1\}^* : |y| < c|x|^k \wedge A(x,y) = 1\}.$$

Now a polynomial space algorithm for L can be made by consecutively generate all (exponentially many) $y \in \{0,1\}^*$ with $|y| < c|x|^k$, and for all of them run $A(x,y)$. The use of space is the same as is required for A, which is polynomial, plus the administration for generating all $y \in \{0,1\}^*$ with $|y| < c|x|^k$, which is polynomial too.

One can also define NPSPACE by taking the same definition in which the Turing machine is allowed to be non-deterministic. Surprisingly, it can be shown that PSPACE and NPSPACE coincide. This is called Savitch's Theorem.

Now we are ready to define PSPACE-hard and PSPACE-complete, very similar to NP-hard and NP-complete.

A decision problem L is called *PSPACE-hard* if $L' \leq_P L$ for every $L' \in$ PSPACE.

A decision problem L is called *PSPACE-complete* if $L \in$ PSPACE and L is PSPACE-hard.

Roughly speaking, L is PSPACE-hard means that every problem in PSPACE is at least as hard as L. Note that, just like in the definition of NP, 'at least as hard' refers to time-complexity, and not to space-complexity.

Very similar to NP-hardness we have the following key theorem to prove that a decision problem L is PSPACE-hard.

Theorem:
If $L' \leq_P L$ and L' is PSPACE-hard, then L is PSPACE-hard.

Proof: Let L'' be an arbitrary PSPACE-hard problem. Since L' is PSPACE-hard, we have $L'' \leq_P L'$. Since $L' \leq_P L$ and $\leq_P$ is transitive, we conclude $L'' \leq_P L$. This proves that L is PSPACE-hard. $\square$

In order to prove that a particular decision problem is PSPACE-hard using this theorem, we need to have a PSPACE-hardness proof for one particular decision problem, which may serve then as the mother of all PSPACE-hard problems. Just like SAT is the mother of all NP-hard problems, the mother of all PSPACE-hard problems is QBF: Quantified Boolean Formulas. The corresponding formulas (also referred to by QBF) are defined by the following syntax:

- Every propositional formula is a QBF, in which by definition all variables occur free.
- If ϕ is a QBF, and x is a boolean variable occurring free in ϕ, then $\forall x : \phi$ is a QBF in which x does not occur free.
- If ϕ is a QBF, and x is a boolean variable occurring free in ϕ, then $\exists x : \phi$ is a QBF in which x does not occur free.

For a variable $y \neq x$, y occurs free in $\forall x : \phi$ if and only if y occurs free in ϕ, and similar for $\exists x : \phi$. A QBF without free variables has a truth value 1 = true or 0 = false by simply interpreting the suggested meaning: the propositional operations are interpreted by their usual meaning, and

- $\forall x : \phi$ yields 1 if and only if
  - ϕ yields 1 if every occurrence of x is replaced by 0, and
  - ϕ yields 1 if every occurrence of x is replaced by 1.
- $\exists x : \phi$ yields 1 if and only if
  - ϕ yields 1 if every occurrence of x is replaced by 0, or

– ϕ yields 1 if every occurrence of x is replaced by 1.

For example, the QBF $\forall x \exists y ((x \wedge y) \vee (\neg x \wedge \neg y))$ yields true, since by replacing every x by 0 the formula $\exists y ((0 \wedge y) \vee (\neg 0 \wedge \neg y))$ yields true by choosing $y = 0$, and by replacing every x by 1 the formula $\exists y ((1 \wedge y) \vee (\neg 1 \wedge \neg y))$ yields true by choosing $y = 1$.

Now the decision problem QBF is:

given a QBF without free variables, does it have truth value 1?

Note that a propositional formula ϕ over variables $x_1, x_2, \ldots, x_n$ is satisfiable if and only if the QBF $\exists x_1 \exists x_2 \cdots \exists x_n \phi$ yields true. So QBF can be seen as an extension of SAT. An immediately consequence is that QBF is NP-hard, which follows from SAT $\leq_P$ QBF: simply let the transformation function put $\exists x_1 \exists x_2 \cdots \exists x_n$ in front of the SAT formula.

Probably QBF is not in NP. In particular, we do not have the notion of a certificate any more that can be checked in polynomial time: if a QBF yields true then a certificate for this claim consists of a choice for existentially quantified ($\exists$) variables for every combination of values of universally quantified ($\forall$) variables, and this yields 2^n choices if n is the number of universally quantified variables.

Nevertheless, it is not hard to see that QBF $\in$ PSPACE: to check whether a QBF yields 1 in the obvious way yields exponential time complexity, but is done in polynomial space complexity. Much harder to prove is that QBF is PSPACE-hard. We do not give the proof here; it exploits details of Turing machine computations in a similar way as we did in proving Cook-Levin Theorem.

In our definition of QBF we only allow a sequence of quantifications in front of a propositional formula, and no quantifications inside propositional operators. Some texts allow this too in QBF. But this more relaxed format is easily transformed to our format in polynomial time, so this has no influence on complexity measures.

Typically, hard problems tend to be NP-complete if they admit an obvious notion of certificate that can be checked in polynomial time, including many example we have seen. In case such a notion does not exist, typically if the size of the intended certificate is worst case exponential, then the problem may be PSPACE-complete. An example of this is the decision problem: given a regular expression r and an alphabet, does r generated all strings over the alphabet?

Another class of problems that tend to be PSPACE-complete are two-player games: given an instance in a two-player game, in which white is on turn, is it the case that white can always win? There is a relationship with QBF: such a winning strategy means that there is a play for white, for every play of black, there is a play for white, and so on, so it has to be checked that a formula of the shape $\exists \forall \exists \forall \cdots$ yields true.

An instance of such a game is GG: generalized geography. Geography is a children's game, which is good for a long car trip (according to wikipedia), where players take turns naming cities from anywhere in the world. Each city chosen must begin with the same letter that ended the previous city name. Repetition is not allowed. The game begins with an arbitrary starting city and ends when a player loses because he or she is unable to continue. If you define the directed graph in which the nodes are the known cities in the world, and draw an edge from A to B if and only the name of A ends in the initial letter of B, you can see this game as an instance of GG: given a direct graph in which one node is marked, white starts by choosing the marked node, next black chooses a node that can be reached by an edge from this marked node, and then alternatingly white and black choose a node that can be reached by an edge from the last chosen node. A player looses if no such node can be chosen; for an example see `https://en.wikipedia.org/wiki/Generalized_geography`. So GG is the decision problem: given a directed graph and a marked node, decide whether white may win this game. It turns out that this game is PSPACE-complete. For proving GG $\in$ PSPACE one analyzes that the exponential time backtracking algorithm investigating all possible choices, can be implemented in a way using only polynomial space. For proving that GG is PSPACE-hard one has to prove that QBF $\leq_P$ GG. We do not give the technical details, but the key idea is that an arbitrary QBF formula has to be transformed in polynomial time to a GG instance, in such a way that the formula yields true if and only if the resulting GG instance has a winning strategy for white.

In its turn, many other two-player games are proved to be PSPACE-hard by reducing GG to the new game. An example of this is Go-Bang: white puts a white stone on an empty node in an $n \times n$-grid, next black puts a black stone on an empty node, and so on, and a player wins as soon as he has five consecutive stones of his color: horizontally, vertically or diagonally.

So both for proving NP-hardness and PSPACE-hardness of a decision problem, the problem boils down to proving $L_1 \leq_P L_2$ for particular languages L_1, L_2, for which a polynomial transformation f has to be designed for which $x \in L_1 \iff f(x) \in L_2$.