1 Complexity Classes

1.1 Resources

What kind of resources do we have in computation? There are many choices:

- Number of states of a TM
- Number of letters in the alphabet
- Number of tapes used
- Number of transitions made
- Number of tape cells used
- Number of non-deterministic choices made

Not all of these choices are good choices for us to count the resources we have. We care about how the resource usage *scales* with input size. For example, the number of states of a TM is usually constant no matter the size of the input, so the number of states is not a good **complexity measure**.

There are two widely used measures for complexity:

- **TIME** - The number of transitions made by the TM. Time is a resource which cannot be reused, similar to a resource like power consumed
- **SPACE** - The number of tape cells touched by the TM. Unlike time, space can be reused, similar to a resource like network bandwidth.

1.2 Language Recognition

The language recognition problem from a language L over an alphabet A is:

Input: a word $w \in A^*$
Output: "yes" if $w \in L$, "no" if $w \notin L$

An *instance* of the problem is a particular word w and the *size* of the instance is $|w|$.

1.2.1 Basic Definitions

Let f be a function $f : N \to N$, let n be the length of the input, and let L denote a language:

Definition 1.1. DTIME($f(n)$) = $\{L | L$ is decided by some deterministic TM in at most $f(n)$ steps$\}$

Definition 1.2. DSPACE($f(n)$) = $\{L | L$ is decided by some deterministic TM via at most $f(n)$ cells$\}$

The functions were are all positive non-decreasing functions such as $log(n)$, n^2, etc. For example the language $\{0^k1^k\}$ is in DTIME(n^2) *and* in DTIME($nlog(n)$). Any regular language is in DSPACE(1).

One obvious question we can ask is if we allow more resources, does the set of language that are recognisable go up? In other words if we allowed for more transitions or more tape cells used, can we recognise more languages?

1.2.2 Time Speedup Lemma

The **linear speedup theorem** states that given any real $c > 0$ and any TM that can solve a problem in $f(n)$ time, there is another machine that can solve the same problem in time *at most* $c \cdot f(n) + O(n)$.

Theorem 1.1. Let L ∈DTIME(f(n)). Then, for any $c > 0$, L ∈DTIME(f'(n)) where

$$f'(n) = cf(n) + O(n)$$

Proof. Simulate by a new TM and **increase** the number of states and alphabet (by a constant factor depending on c) so that the new TM can make $\geq c$ transitions for each transition of the original TM.
Then in $O(n)$ time, we can encode the original input. □

1.2.3 Tape Compression Lemma

Similarly, DSPACE($f(n)$) = DSPACE($c \cdot f(n)$).

Theorem 1.2. Let L ∈DSPACE($f(n)$). Then, for any $c > 0$, L ∈DSPACE($f'(n)$) where

$$f'(n) = cf(n) + O(1)$$

Proof. Again, **increase** the number of states and alphabet (by a constant factor of c) so that the new TM can use $\geq c$ cells for each tape cell used in the original TM. □

This shows that constant factors do not matter. If we allow 10 times more time, we do not gain any more power and the same goes for space. The set of decision problems we can solve remain the same.

1.3 Big-O Notation

1.3.1 Asymptotic Upper Bound

Definition 1.3. Big-O. $T(n)$ is $O(f(n))$
if ∃ a c and a $n_0 > 0$
s.t. $\forall n > n_0$

$$T(n) \leq c \cdot f(n)$$

This means that for $O(n)$, $T(n)$ grows **no more than** $f(n)$ for large n. For example $T(n) = 2n^2$ is $O(n^2)$. We can choose $c = 2$ and $n_0 = 1$, then $2n^2 \leq cn^2$ for all $n > n_0$. This is called Big-O and is an **inclusive** upper bound.

The small-o is the **exclusive** upper bound where $T(n)$ grows **strictly slower** than $f(n)$ and is defined as follows:

Definition 1.4. Little-O. $T(n)$ is $O(f(n))$
if *for every* $c > 0$ there exists a $n_0 > 0$
s.t. for all $n > n_0$

$$T(n) < c \cdot f(n)$$

For example, n is *not* $o(2n)$ because if $c < 0.5$, then $n > 2cn$ for all n. However, $2n$ *is* $o(n^2)$ because if we choose $n_0 = \frac{2}{c}$. Then for all $n > n_0$, we get $cn^2 = (\frac{2}{n_0})n^2 > 2n$.

1.3.2 Asymptotic Lower Bound

As we have asymptotic upper bounds, we also have asymptotic lower bounds, denoted using omega (Ω and ω). Again we have an inclusive lower bound and exclusive lower bound as defined below.

Definition 1.5. Big-Omega. $T(n)$ is $\Omega(f(n))$
if there exists constants c and a n_0 ¿ 0
s.t. for all $n > n_0$

$$T(n) \geq c \cdot f(n)$$

Definition 1.6. Little-Omega. $T(n)$ is $\omega(f(n))$
if for every $c > 0$, there exists a $n_0 > 0$
s.t. for all $n > n_0$

$$T(n) > c \cdot f(n)$$

In a similar way to the asymptotic upper bounds, the inclusive lower bound $\Omega(n)$ describes that $T(n)$ grows **no less** than $f(n)$ for large n. For the exclusive lower bound $\omega 0$, $T(n)$ grows **strictly faster** than $f(n)$ for large n. So for example if $T(n) = 2n$, it is $\Omega(n)$ but not $\omega(n)$. This is because $2n$ grows at least as much as n, but not strictly more than n, since if $c > 2$, $2n < cn$ for all n.

1.3.3 Asymptotic Tight Bound

Given we now have both the upper and lower bound, there is also a **tight** bound.

Definition 1.7. Big-Theta. $T(n)$ is $\Theta(f(n))$
if there exists constants c_1, c_2 and a $n_0 > 0$
s.t. for all $n > n_0$

$$c_1 \cdot f(n) \leq T(n) \leq c_2 \cdot f(n)$$

This just means that $T(n)$ grows **like** $f(n)$ for large n.

1.3.4 Time Hierarchy Theorem

The **time hierarchy theorem** says that given more time, a TM can solve more problems. For example, there are problems that can be solved with n^2 time but not n time.

Theorem 1.3. If $f(n)$ is a time-constructible function, then there exists a decision problem (decidable) which cannot be solved in worst-case deterministic time $f(n)$, but can be solved in worst-case deterministic time $f(n)^2$.

$$\text{DTIME}(f(n)) \not\subseteq \text{DTIME}(f(n)^2)$$

More generally, it can be shown that if $f(n)$ is time-constructible, then

$$\text{DTIME}(\text{o}(\tfrac{f(n)}{log(f(n))})) \not\subseteq \text{DTIME}(f(n))$$

Proof. Construct a universal TM. □

1.3.5 Space Hierarchy Theorem

Just as there is a time hierarchy theorem, there is also a space hierarchy theorem. We need *much more* time to increase our computational power. The deterministic space hierarchy states that for all space-constructible functions $f(n)$:

$$\text{DSPACE}(o(f(n))) \not\subseteq \text{DSPACE}(f(n)) \tag{1}$$

In other words, for every positive non-decreasing function, there is a language that is decidable in $O(f(n))$, but not decidable in $o(f(n))$. As a consequence of the hierarchy theorems, we now know that much more time or space means more problems are solvable. For example:

- $\text{DTIME}(n^2)$ is strictly more powerful than $\text{DTIME}(n)$
- $\text{DSPACE}(nlog(n))$ is strictly more powerful than $\text{DTIME}(n)$
- $\text{DTIME}(n^1042)$ is strictly more powerful than $\text{DTIME}(n^1041)$

1.4 Non-determinism

Deterministic machines have **exactly one entry** in their transition tables. Non-determinism lets machines have **zero or more** entries. This is like having parallel computations at the same time and branches can vanish as needed. It can be seen as used for "free guesses".

Another way of describing this problem is we guess the answer (the **certificate** and the non-deterministic time is just time to check the certificate. A certificate is a string that certifies the answer to the computation, or the decidability/membership of some string in a language.

1.4.1 Non-deterministic complexity

Similar to deterministic complexity, we can define NTIME and NSPACE as languages which are decided by some non-deterministic TM in at most $f(n)$ steps or tape cells.

All deterministic space and time belong to the set of NSPACE and NTIME as we can think of a deterministic TM as a special case of a non-deterministic TM.

$$\text{DSPACE}(f(n)) \subseteq \text{NSPACE}(f(n)) \tag{2}$$

$$\text{DTIME}(f(n)) \subseteq \text{DTIME}(f(n)) \tag{3}$$

We can actually also relate time and space complexity

$$\text{DTIME}(f(n)) \subseteq \text{DSPACE}(f(n)) \tag{4}$$

$$\text{NTIME}(f(n)) \subseteq \text{NSPACE}(f(n)) \tag{5}$$

This is true because you can not use more than $f(n)$ space in only $f(n)$ time. By concentrating on exponential style increases, we can union some classes into larger classes.

The class P is the class of problems decidable in *polynomial* time. That is to say any $f(n) = n^k$ where $k \geq 0$. From this we get the following:

$$\text{P} = \cup_{k\geq 1}\text{DTIME}(n^k) \tag{6}$$

$$\text{PSPACE} = \cup_{k\geq 1}\text{DSPACE}(n^k) \tag{7}$$

There is also a class of exponential time and space in which we can union the deterministic time and space functions.

$$\text{EXP} = \cup_{k\geq 1}\text{DTIME}(2^{n^k}) \tag{8}$$

$$\text{EXPSPACE} = \cup_{k\geq 1}\text{EXPSPACE}(2^{n^k}) \tag{9}$$

Similarly the non-deterministic versions can be defined in terms of the corresponding NTIME and NSPACE classes:

$$\text{NP} = \cup_{k\geq 1}\text{NTIME}(n^k) \tag{10}$$

$$\text{NPSPACE} = \cup_{k\geq 1}\text{NSPACE}(n^k) \tag{11}$$

$$\text{NEXP} = \cup_{k\geq 1}\text{NTIME}(2^{n^k}) \tag{12}$$

By simply applying the hierarchy theorems we can relate the larger complexity classes and see which classes belong in which class:

$$\text{P} \subset \text{EXP}, \text{NP} \subset \text{NEXP}, \text{L} \subset \text{PSPACE} \subset \text{EXPSPACE} \tag{13}$$

All deterministic classes also belong to their corresponding non-deterministic classes. For example $\text{P} \subseteq \text{NP}$

1.5 Relating Time and Space

We can relate time and space as follows:

$$\text{NTIME}(f(n)) \subseteq \text{DSPACE}(f(n)) \tag{14}$$

Proof. Suppose we have a non-deterministic TM that generates choices of length of at most $f(n)$, each in time $f(n)$ or less. Then we can construct a deterministic TM that simulates these choices one-by-one, *re-using* the same space every time. □

This shows we can relate space and time, determinism and non-determinism:

$$\text{DTIME}(f(n)) \subseteq \text{NTIME}(f(n)) \subseteq \text{DSPACE}(f(n)) \tag{15}$$

On the other hand, we can also relate non-deterministic space with deterministic time.

$$\text{NSPACE}(f(n)) \subseteq \cup_c \text{DTIME}(c^{f(n)}) \tag{16}$$

This shows that we can get non-deterministic space if we use an exponential amount of deterministic time.

Proof. Suppose that we have a non-deterministic TM that uses at most $f(n)$ cells. There are $c^{f(n)}$ different possible states of those cells, so a graph can be created with nodes of those states, and edges going from one of those states to another whenever there is a one-step transition in the non-deterministic TM. Accepting a language means reachability in this graph, which can be done in time polynomial in the graph size $c^f(n)$. □

Now we can further relate space and time, determinism and non-determinism.

$$\text{DTIME}(f(n)) \subseteq \text{NTIME}(f(n)) \subseteq \text{DSPACE}(f(n)) \tag{17}$$

1.6 Non-deterministic Hierarchy Theorems

1.6.1 Non-deterministic Time Hierarchy

For every positive non-decreasing function $f(n)$, there is a language that is decidable by a non-deterministic TM in $f(n+1)logf(n+1)$, but not decidable in $o(f(n))$.

Theorem 1.4. Let $f(n+1)logf(n+1) = o(f(n))$. Then

$$\text{NTIME}(g(n)) \setminus \text{NTIME}(f(n)) \neq \emptyset$$

1.6.2 Non-deterministic Space Hierarchy

Just like the non-deterministic version of the time hierarchy theorem, there is the space hierarchy:

Theorem 1.5. Let $f(n) = o(g(n))$. Then

$$\text{NSPACE}(g(n)) \setminus \text{NSPACE}(f(n)) \neq \emptyset$$

However, for non-deterministic vs deterministic space, we can know more. **Savitch's Theorem** states that if a non-deterministic TM can decide in $f(n)$ space, then a deterministic TM can decide in $f(n)^2$ space, so the speed up is much less than exponential.

Theorem 1.6. Savitch's Theorem Let $f(n) \geq log(n)$. Then,

$$\text{NSPACE}(f(n)) \subseteq \text{DSPACE}(f(n)^2)$$

Proof. The proof for this theorem is a problem of directed graph reachability. Suppose our decision problem is: "Given a directed graph G and vertices s and t, is there a path from s to t? This problem is called **reachability** and is as hard as any problem in NL.

Algorithm 1: Proof Sketch of Savitch's Theorem

```
function N-GCON(G = (V, E), s, t)
  Start at s
  for vertex in V
    Choose a neighbour $u$ non-deterministically
    Jump to $u$
    Accept if $u = t$
  end for
  Reject
end function
```

Counting up to $|V|$ takes $log|V|$ space. Storing the current position takes $log|V|$ space, so we're in NL. Now, we need a deterministic TM that only needs the square of this space ($(log|V|)^2$). The trick is to see if vertex u is a midpoint between s and t by recursively testing for paths from u to s and t that are of length at most $\frac{|V|}{2}$. These recursive checks can all use the same space.

Algorithm 2: Finding midpoint path

```
function PATH(s, t, |V|)
  for vertex u in V
    if PATH(s, u, |V|/2) and PATH(u, t, |V|/2) then
      return true
    end if
  end for
  return false
end function
```

This search calls itself to a recursion depth of $O(log(n))$ levels, each of which require $O(log(n))$ bits to store the function arguments and local variables at that level. So the total space used by the algorithm is $O((log(n))^2)$. This shows that

$$\text{NL} \subseteq \text{DSPACE}((log(n))^2) \tag{18}$$

□

By using non-deterministic space which gives a quadratic reduction in complexity, a quadratic different is a polynomial difference, so

$$\text{PSPACE} = \cup_{k \geq 1} \text{DSPACE}(n^k) = \text{NPSPACE} \tag{19}$$

Although this shows that PSPACE is equal to NPSPACE, it seems that space complexity is different from time complexity, since we have no proof that P = NP.

1.7 SAT Problem

The **satisfiability problem** is a classic logic problem where we must resolve propositional logic formulas in CNF. Let us look at the special case of 2-SAT, which is the SAT problem but only with CNF clauses that contain 2 literals.

By using resolution, we know that 2-SAT can be decided in polynomial time. However, since we know that NL ⊆ P, if we show that 2-SAT is in NL, then we show that the intersection of the two sets is non-empty. This would hold if it is the case that the 2-SAT problem is as hard as any problem in NL.

1.7.1 Showing 2-SAT is in NL

Given an instance of the 2-SAT problem with k clauses and n variables, we construct a directed graph with $2n$ vertices, which we call:

$$\{x_1, ..., x_n\}\ \{\neg x_1, ..., \neg x_n\}$$

There is an edge between two vertices (x_i, x_j) *iff* $(x_i \vee \neg x_j)$ or $(\neg x_i \vee x_j)$.

Proof. "if" - Suppose such paths exist and that the 2-SAT instance can be satisfied by assignment T.
Suppose that $T(x_i) = \text{true}$ and then $T(\neg x_i) = \text{false}$. Since there is a path from x_i to $\neg x_i$ there is an edge on this path such that $T(\alpha) = \text{true}$ and $T(\beta) = false$. But (α, β) corresponds to the clause $(\neg\alpha, \beta)$ in the 2-SAT instance. Hence it is not satisfied by T, a contradiction. □

Proof. "only if" - Suppose that there is no variable with such paths. Repeat until each node has a truth value:

1. Pick any unassigned node $\alpha$with no path from α to $\neg\alpha$
2. Assign true to all nodes reachable from α

3. Assign false to all node from which $\neg\alpha$is reachable

This assignment is satisfying, since no edge can lead between true and false. We can then use function `N-GCON(G, `α`, `$\neg\alpha$`)` to decide reachability. This takes non-deterministic $log(n)$ space, so 2-SAT is in NL. □

2 P vs NP: non-determinism

2.1 Why do we care about non-determinism

The equivalent characterisation of non-determinism is that the machine **guesses**. We only care about successful guesses, but it seems not very practical for a machine to implement. However, if we suppose that all choices were made non-deterministically, a correct answer is called a **certificate**. Non-deterministic algorithms are ones that can **verify** that the certificate is correct.

For lots of problems, verifying the proposed solution is easy in the "obvious" sense, not necessarily in the technical time complexity. For example, given a number N, which are its prime factors? This is a hard problem to find a solution for large N, however given a solution it is easy to verify it. Intuitively, verification takes much more resources than solution. Formally, nobody knows.

With the factorisation problem, no good algorithms are known and because of this difficulty, factorisation forms the basis of RSA encryption so if a fast algorithm was known, then many existing crypto-systems would be easily broken. Other problems such as the SAT problem and graph colouring are similarly easy to verify given a solution, but hard to find a solution from nothing.

2.2 Determinism vs Non-determinism

2.2.1 Space

By **Savitch's Theorem**, we know that PSPACE = NPSPACE. However, this result is not known for time, where P may not be equal to NP. Most suspect it will be different.

2.2.2 Time

For time, we known that:

$$\text{DTIME}(\text{O}(\text{N})) \subset \text{NTIME}(O(n)) \tag{20}$$

"For multi-tape TMs, non-deterministic linear time is more powerful than deterministic linear time." It may or may not be possible to extend this to other classes of functions.

2.2.3 The P = NP question

Is P = NP, or is it P ≠NP?

In essence, this question is asking are there any non-deterministic polynomial algorithms that are more than polynomial deterministically? Most have described this problem as the most important question in Computer Science and some say that even that is an understatement.

2.3 Reductions

Reductions are also known as transformations. The idea is that given an algorithm for a problem p, can we use it as a subroutine to solve a problem q? And if we can, can we do so efficiently or at least in polynomial time?

Algorithm designs in many different fields have struggled to find efficient algorithms for different problems:

- Graphs: is there a clique of size k?
- Graphs: can a graph be coloured by k colours?
- Linear programming: can a linear program with only integer answers acceptable be solved, for k variables?
- Scheduling: is there a schedule with precedence constraints?
- Graphs: Is there a Hamiltonian Cycle (path visiting all node)?
- Logic: Is a formula satisfiable?

There are polynomial reductions between *all* the problems above, and several hundred others as well. In other words, if **any one** of them has a polynomial time algorithm, that means that **all** do.

This is one of the reason why the P = NP question is of great significance. By Cook's Theorem, if **any** one of those problems have a polynomial algorithm, **every NP problem** has one. This property is formally called NP-completeness. The opposite is also true. If P ≠NP, then **none** of those problems have a polynomial algorithm.

2.3.1 3-SAT reduction

Let us suppose for this example that we can devise a polynomial time algorithm for SAT if we have a polynomial time algorithm for 3-SAT. This is said to be a **reduction** of SAT to 3SAT, or

$$\text{SAT} \leq \text{3SAT} \tag{21}$$

If we are given an instance i of a SAT problem, we need to find an instance of 3SAT which is satisfiable *iff* i is satisfiable. To do this, we can take any clause in i that has 4 or more literals and split the clauses.

A clause

$$\{l_1, l_2, l_3, l_4, ...\} \tag{22}$$

becomes split into the following clauses:

$$\{l_1, l_2, y_1\},\ \{\neg y_1, l_3, l_4, ...\} \tag{23}$$

where y_1 is a new variable than did not occur in i. Now the top clause is satisfiable *iff* the bottom two clauses are. If we do this repeatedly, we will split any clause in i that has more than 3 literals into clauses with only 3 literals. At worst, we treble the length of i, so this is a polynomial transformation of SAT to 3SAT.

2.3.2 Polynomial Karp Reductions

A more general case of reductions instead of just reducing SAT to 3SAT is reducing any problem A to a problem B.

Setup: Have a problem A and a problem B
Note: Usually, we know that A and B are in NP
Given: A particular instance of problem A. Suppose that we can transform in **polynomial time** into an instance of problem B such that the problem is solved for A if it is solved for B.
Then: Problem A is said to transform into problem B (or reduce to problem B)

$$A \leq B \tag{24}$$

The consequence of this, is if we have a polynomial deterministic time algorithm for B, then we also have one for A. This helps us define if problems are *hard* or *complete* (i.e, **NP-Hard**, **NP-Complete**).

2.3.3 Polynomial *Cook/Turing* Reductions

A Cook/Turing reduction is slightly different from that of the previous polynomial reduction. Instead of transforming an instance of a problem to the instance of another problem, a Turing reduction is one where from problem A to problem B, there is an algorithm that solves problem A using a polynomial number of calls to a subroutine for problem B. In other words:

Given: A problem A, suppose we can create an algorithm that is polynomial time, *except* it calls (at most polynomial time) a **subroutine** for solving problem B.
Then: Problem A is said to Cook/Turing transform into problem B.

$$A \leq_T B \tag{25}$$

A Karp reduction implies a Cook/Turing reduction as it is just one call to the subroutine.

2.4 Hard and Complete Problems

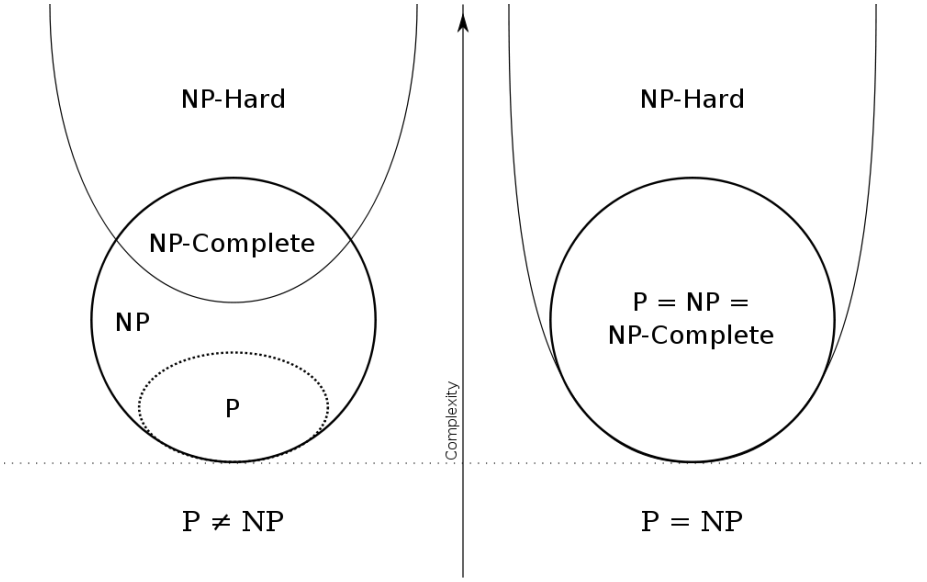
Definition 2.1. NP-Hard Problems A problem A is said to be NP-Hard, if, for any problem B in NP,

$$B \leq A \tag{26}$$

In other words, a problem A is NP-Hard, if there is a NP-Complete problem B, such that B is reducible to A in polynomial time. Intuitively, these are problems that are *at least as hard* as NP-Complete problems. Note that NP-Hard problems do not have to be in NP, and they also do not have to be decision problems.

Definition 2.2. NP-Complete A problem A is said to be NP-Complete if it is both in NP and is NP-Hard.

NP-Complete is a complexity class which represents the set of all problems A in NP for which it is possible to reduce any other NP problem B to A in polynomial time.



It can be shown that every NP problem can be reduced to another, for example 3SAT. This is known as **Cook's theorem**. What makes NP-Complete problems important is that if a deterministic polynomial time algorithm can be found to solve one of them, every NP problem is solvable in polynomial time.

2.5 P, NP, or NP-Complete

Let us once again go through the complexity classes so we can see if we have a problem p, is it in the class P, NP, or NP-Complete.

2.5.1 P

Definition 2.3. P is a complexity class that represents the set of all decision problems that can be solved in polynomial time.

That is, given an instance of the problem, the answer yes or no can be decided in polynomial time. *A full proof of whether a problem is in P requires an exhibition of a polynomial time algorithm.* Note that the algorithm can use other polynomial time algorithms as subroutines, called at most a polynomial number of times. Thus reductions from a polynomial problem. For example, given a connected graph G, can its vertices be coloured using two colours so that no edge is monochromatic?

The algorithm to solve this problem can be done in polynomial time. We start with an arbitrary vertex and colour it red, then all of its neighbours blue and continue to another arbitrary uncoloured vertex. Stop when we run out of vertices or we are forced to make an edge have both of its endpoints be the same colour.

2.5.2 NP

Definition 2.4. NP is a complexity class that represents the set of all decision problems for which the instances where the answer is "yes" or "no" have proofs that can be verified in polynomial time.

This means that if someone gives us an instance of the problem and a certificate to the answer being yes, we can check the certificate is polynomial in size and is correct in polynomial time. For example integer factorisation is in NP. For large numbers in polynomial time, we cannot find its two factors, however, given the two factors, checking if they are correct is trivial.

2.5.3 NP-Complete

Definition 2.5. NP-Complete is a complexity class which represents the set of all problems A in NP, for which it is possible to reduce any other NP problem B to A in polynomial time.

Intuitively, this means that we can solve B quickly if we know how to solve A quickly. More precisely, B is reducible to A if there exists a polynomial time algorithm f which can transform instance of B to instances of $A = f(B)$ in polynomial time. The answer to B is yes is true if and only if the answer to $f(B)$ is true.

To show that a problem is NP-Complete, we must show the problem is in NP and *any problem* in NP can be transformed to it.

2.5.4 Transformations

Suppose there is a known NP-Complete problem Π. Then to show a new problem p is NP-Complete, all we have to do it show that Πcan be transformed to p. Now we just have to worry about one problem.

Because of this, it is really useful to have just **one** NP-Complete program that all other problems can reduce to. This was what Stephen Cook and Leonid Levin (independently) discovered after proving that SAT was NP-Complete.

2.6 Graph Colouring

A graph colouring problem is a classical problem from map making. Maps can show regions more clearly if adjacent regions have different colours. The **4-Colouring Conjecture** is that any map can be coloured with 4 distinct colours.

This problem also has implications and importance in computers. Abstractly we can think of two adjacent nodes of some structured problem cannot be done together. This has importance in scheduling algorithms, as well as for example register-allocation for compilers. Two variables cannot be put into the same register if their value is used in the same part of the program. Graph colouring was shown to be **NP-Complete** by Karp.

2.7 Cook-Levin Theorem

Theorem 2.1. Cook-Levin Theorem SAT is NP-Complete.

The importance of Cook's theorem is that we now know *if we have one NP-Complete problem, we can show several to be NP-Complete.* To show that SAT is NP-Complete we must first show that SAT is in NP, and then show that any other problem in NP can be reduced in polynomial time to a instance of SAT. In other words it requires showing

- If p is a problem with an NP algorithm
- ...then $p \leq SAT$

Proof. To show that SAT is in NP is easy. Simply non-deterministically guess an assignment to the variables, then we can check in polynomial time if each clause it true for that assignment. Now we have to prove for an arbitrary NP problem, that it can be reduced to SAT.

If we have an arbitrary problem $A \in$ NP, this **implies:**

- We have a non-deterministic TM M_A for A
- For any word $w \in A$, M_A on w takes $p(|w|)$ time

We have to create a SAT instance that is satisfiable if and only if M_A accepts w in time $p(|w|)$. We have to ensure that the formula is polynomial in size. □

2.8 Why Polynomials?

One question might be why we care so much about polynomials compared to the other complexity classes. Notice that both P and NP are an infinite union of polynomials. Each polynomial degree has more power than the next,

$$\text{DTIME}(n^{120}) \subset \text{DTIME}(n^{121}) \tag{27}$$

We know that many natural algorithms have polynomial time, such as merge sort and bubble sort. We also know that some natural algorithms are exponential and it is relatively clear that these exponential algorithms are much more expensive.

However, though it may seem that quadratic or even cubic time is much better than exponential, it is sometimes hard to reason that a degree one million polynomial ($n^{1000000}$) is much better than exponential. In the real world, not many algorithms with higher than the sixth power are known. In some cases such as linear programming, we prefer the worst case exponential algorithm compared to the worst case n^5 algorithm.

Ultimately, polynomials is a class of algorithms with good theoretical properties and arguable good practical properties. Answers to the P = NP question also imply answers to separation of analogous deterministic vs non-deterministic questions.

2.8.1 Extended Church-Turing Thesis

The Church-Turing Thesis states that *all reasonable models of computation have the same power in terms of recognising languages.* An extended version of the thesis is that *all reasonable models of computation when restricted to polynomials have the same power.* This applies to all **deterministic** models and another way of stating it is any model can be simulated by any other model with at most polynomial slowdown.

We have seen that polynomially related changes involve:

- TM steps needed
- Number of tapes
- Number of states

- Number of symbols needed

However, it is unknown whether it applies to non-deterministic computers and unknown whether it applies to quantum computers. Remember that this is just a **hypothesis** and cannot be proved, but can be disproved.

3 Analysis of Algorithms

4 Analysing Recursive Programs

Analysing algorithms is quite straightforward for iterative algorithms but more difficult for recursive algorithms. A key technical tool for this kind of analysis is **recurrence relations**.

$$T(n) \text{ is the time taken for input size } n \tag{28}$$

5 Reductions and Games