Handout 11A : Complexity review

William Pires

This handout has proofs to claims that we sometimes gave in class without proof. You are only responsible for what was taught in class, but it may be useful to see the proofs (or better, come up with them yourself) as exercise. Also some of the later material here will be taught in the next lecture.

1 Time Complexity

Say we have a TM M that decides a language L. Given some input x, we want to understand how long it takes M to accept or reject x. To do so, we consider the running time of M as a function of the length of x (how many alphabet symbols it takes to write x as input).

Definition 1. Let M be a Turing machine that halts on every input. The running time of M denoted $t(n)$ is the maximum number of steps M takes on any input of length n. That is :

$$t(n) := \max_{x \in \Sigma^*, |x|=n} \text{number of steps } M \text{ takes before it halts on } x$$

In the above, the number of **steps** of M on a input x is the number of transitions M takes before it halts on x.

Counting exactly how many steps a TM takes an input is quite cumbersome, so we use Big O notation to hide away constants. Recall that for two functions $f, g : \mathbb{N} \to \mathbb{R}^+$, we say $f(n) = O(g(n))$ if there exists constants $c > 0$ and $n_0 \in \mathbb{N}$ such that for all $n \geq n_0$:

$$f(n) \leq g(n).$$

Definition 2 (Time($t(n)$))**.** Let $t : \mathbb{N} \to \mathbb{N}$, we define the class Time($t(n)$) as the following set of languages :

$$\text{Time}(t(n)) := \{\ L \mid L \text{ is a language decided by an } O(t(n)) \text{ time Turing machine } \}$$

Unpacking the definition, this means there is some constant c and n_0, such that for all $n \geq n_0$, M halts in time $\leq c * t(n)$ on all inputs of length n and M decides L.

We note that the definition of Time($t(n)$) is sensitive to the model of computation we consider: One tape vs multi-tape TMs, algorithm (like Python) etc. But the definition of P, which we will focus on, doesn't depend on these. So you can safely ignore such issues, and use (deterministic) pseudo-Code to describe algorithms.

Definition 3. We say $t : \mathbb{N} \to \mathbb{N}$ is polynomial if there exists some constant k such that $f(n) = O(n^k)$.

Examples : $n^5 + 1000$, $\log(n)$ are both polynomial. $2^n + n^3$ isn't polynomial.

Good to know: If $f : \mathbb{N} \to \mathbb{N}$ and $g : \mathbb{N} \to \mathbb{N}$ are both polynomials then the following are also polynomials:

- $f + g$, where $(f + g)(x) = f(x) + g(x)$.
- $f * g$, where $(f * g)(x) = f(x) * g(x)$.
- $f \circ g$, where $(f \circ g)(x) = f(g(x))$.

You can use this without proof, although it is not hard to prove, and we give a detailed proof of the last one within the proof of theorem 6.

2 P and NP

2.1 P

Definition 4 (P)**.**

$$\mathrm{P} = \{\ L \mid L \text{ is a language and there exists a decider } M \text{ for } L \text{ running polynomial time } \}$$

Equivalently,

$$\mathrm{P} := \bigcup_{k \in N} \mathrm{Time}(n^k)$$

Proof Template 1 (Show $L \in \mathrm{P}$)**.**

1. Give pseudo-Code of a deterministic decider for L
2. Show that if $x \in L$, your algorithm accepts.
3. Show that if $x \notin L$, your algorithm rejects.
4. Show that M runs in polynomial time, namely in time $O(n^k)$ for some constant k. (No need to prove specifically what k is).

We will do a proof using this template in Section 4.

2.2 NP

A non-deterministic Turing machine (NTM) is a Turing machine that is allowed to take non-deterministic transitions. That is, given the current state q and the symbols the heads of the TM

see on the different tapes, there could be multiple transitions possible. For instance, with a one tape TM with states Q and alphabet Σ, the transition function is now of the form $\delta : Q \times \Sigma \to \mathcal{P}(Q \times \Sigma \times \{L, R\})$.

Recall that we say that a non-deterministic TM M decides a languages L if:

- M always halts on an input x, no matter what choice of transitions it made.
- If $x \in L$, there is some choices of the transitions such that M accepts x.
- If $x \notin L$, for all choices of the transitions, M rejects x.

We often think of such a TM as being able to "take a guess" and then checking if this guess is a solution. If there exists any lucky guess, the string is in the language.

Definition 5. Let M be a non-deterministic TM that halts on every input. We say M has running time $t(n)$ if for every input x of length n, every possible computation of M on x terminates within at most $t(n)$ steps.

In particular, for M to run in time $t(n)$, after at most $t(n)$ transitions, M must have halted on any input of length n. And this is no mater what transitions we non deterministically picked at each step. Thus, the notion of running time of an NTM is a worst-case notion across every possible computation path. The notion of acceptance is a best case: an input is in the language if there exists some guess (computation path) that accepts.

As in the deterministic case, we have the following two definitions :

Definition 6 (NTime($t(n)$))**.** Let $t : \mathbb{N} \to \mathbb{N}$, we define the class NTime($t(n)$) as the following set of languages :

NTime($t(n)$) := { L | L is a language decided by an $O(t(n))$ time **non-deterministic** Turing machine }

And the class NP is defined as :

Definition 7 (NP)**.**

NP = { L | L is a language decided by a polynomial time **non-deterministic** Turing machine }

Equivalently,

$$\text{NP} := \bigcup_{k \in N} \text{NTime}(n^k)$$

Proof Template 2 (Show $L \in$ NP using a non-deterministic TM).

1. Give pseudo-Code of a non-deterministic decider M for L. Here M takes as input x only.
2. Show that if $x \in L$, there is some non-deterministic choices that makes M accept.
3. Show that if $x \notin L$, M always reject x, no matter what the non-deterministic steps are.
4. Show that M runs in polynomial time, namely in time $O(n^k)$ for some constant k. (No need to prove specifically what k is).

Example 1. Consider the problem 3-Coloring. In this problem you are given as input as input a graph G. G is given by a list V of n nodes, and a list E of m edges. The goal is to know if there exists a 3-Coloring of G. That if for each node we can assign one of the colors $\{r, b, g\}$ and if (u, v) is an edge in G then u and v must have different colors.

We want to show that 3-Coloring:=$\{\langle V, E\rangle \mid \langle V, E\rangle$ represents a 3-Colorable graph$\}$ is in NP.

This can be solves by a Non-deterministic TM as follow.

Algorithm 1 3-Coloring NTM

Input: $\langle V, E\rangle$
▷ Here V a list of n nodes. E is a list of edges between the nodes in V.

---

for each vertex v in V **do** ▷ Non-deterministically pick a coloring of the graph.
  Non-deterministically assign a color from {r,b,g} to v.
end for

for Every edge $(u, v) \in E$ **do** ▷ Verify this is a good coloring.
  If u, v have the same color : Reject $\langle V, E\rangle$.
end for
Accept $\langle V, E\rangle$.

---

[1]

The above algorithm runs in polynomial time, namely : Assigning a color to every vertex takes $O(|V|)$ time as it only involves going over all vertices in V and picking a color for each. Then checking every edge (u, v) to make sure (u, v) have different colors takes time polynomial in $|V| * |E|$ as we go iterate through every vertex in E and look up the colors of two vertices.

If a 3-Coloring C of the graph exists, then there is some non-deterministic branch where the TM assigns this coloring to the vertex and thus accepts.

[1] How can a TM assign different color to every vertex ? Imagine the input is on the first tape, M looks at the list of node in G one by one. Each time we read a node u, we have three possible transitions, which makes us write one of r, b, g on the second tape. This corresponds to the color we assign to u.

If the input is a graph but no 3-Coloring exists, then the TM can never accept. Indeed for every possible coloring picked in the first for loop, since no 3-Coloring exists, we'll find an edge with two vertices of the same color and reject.

For problems in NP, it's often hard to deterministically see if $x \in L$ unless you use an exponential-time brute force algorithm checking all computations, but if $x \in L$, I can give you a "proof" y such that by looking at y you can be convinced $x \in L$ (in polynomial time).

This motivates an alternative definition for NP based on verifiers :

Definition 8 (Verifier)**.** A verifier V for a language L, is a **deterministic** algorithm such that V takes as as input x and some string c and

$$x \in L \leftrightarrow \exists\, c \text{ such that } V(x, c) \text{ accepts.}$$

V is said to be a polytime verifier if it runs in time $O(|x|^k)$ for some constant k.

In the above we can think of c as the proof (sometimes called "certificate" or "witness") that $x \in L$. If $x \in L$, some proof must make the verifier accept, if $x \notin L$, no proof can make the verifier accept.

If V is a polytime verifier we must always have $|c| = O(|x|^k)$, that is the proof must have polynomial length in $|x|$ (otherwise, the TM wouldn't even be able to read c in time $O(|x|^k)$) .

Proof Template 3 (Show $L \in$ NP using a Verifier)**.**

1. Give pseudo-Code of a deterministic verifier V for L. Here V takes as input x and c (the certificate).[a]
2. Show that if $x \in L$, there is some c such that V accepts (x, c).
3. Show that if $x \notin L$, for all c, V rejects (x, c).
4. Show that V runs in polynomial time, namely in time $O(n^k)$ for some constant k. (No need to prove specifically what k is).

[a] You can choose what kind of certificate you want your algorithm to use, as long as it is of length polynomial in $|x|$.

Example 2. Going back to the 3-Coloring example, a verifier for the problem is the following :

This algorithm works, because it runs in time $\mathcal{O}(|E| * |V|)$, since we look at all the edges in the graph, and each time check the color of the nodes on the edge (Everytime, we need to go through C to find the colors of u and v. And this takes time $O(|V|)$). So this runs in polynomial time in $|E|, |V|$ (and hence polynomial in the input length).

Algorithm 2 3-Coloring verifier

Input: $\langle V, E\rangle, C$

▷ Here V a list of n nodes. E is a list of edges between the nodes in V. C is an assignment of $\{r, g, b\}$ to each node in V

---

for each edge (u, v) in E **do**
  if u, v have the same colors in C **then**
    Reject. ▷ C isn't a valid coloring
  end if
end for
Accept.

It is clear that if the input graph $\langle V, E\rangle$ has a valid 3-Coloring, then there exists a C – that coloring – such that the above algorithm accepts $(\langle V, E\rangle, C)$. On the other hand, if no coloring exists, then for any potential coloring C there will be some edge (u, v), where both u, v have the same color (otherwise this would contradict no coloring exists). So, the above algorithm will reject $(\langle V, E\rangle, C)$.

As you saw in class we have the following equivalent definition of NP :

Theorem 1. NP $= \{ L \mid L$ is a language and there exists a polytime verifier for $L \}$

We won't give the full formal proof of the above here. You can see the book if you're interested. The main idea is that given a NTM, we can construct a verifier, where c is simply which choice to make at each non-deterministic step of the TM so that it accepts. The verifier then runs the NTM using c to pick what transition to follow. In the other direction, given a verifier, the NTM will start by non-deterministically guessing the certificate c, and then running the verifier using that guessed certificate.

Thus, when asked to show $L \in$ NP, you have two options, give a polytime verifier for L or a polytime non-deterministic TM for L (we showed both these options for the 3-Colroing example).

Since a deterministic TM is a special case of a non-determinisitc TM we have the following :

Theorem 2. P $\subseteq$ NP

A major open question is whether NP $\subseteq$ P.

3 NP Hardness and Completeness

There are many problems in NP, including many for which we do not know any polynomial time algorithm to solve them. This motivates the notion of NP hardness and NP completeness.

Definition 9 (Poly-time mapping reducible)**.** A language $A \subseteq \Sigma^*$ is polynomial-time (mapping) reducible to $B \subseteq \Sigma^*$ (written A $\leq_{\mathrm{P}}$ B) if there is a polynomial-time computable function $f : \Sigma^* \to \Sigma^*$ such that :

$$x \in A \iff f(x) \in B$$

In particular, we must have $|f(x)|$ is polynomial in $|x|$.

In particular this means the following:

Theorem 3. If $A \leq_{\mathrm{P}} B$ and $B \in \mathrm{P}$, then $A \in \mathrm{P}$.

Proof. Say $A \leq_{\mathrm{P}} B$ where $B \in \mathrm{P}$. Let M be a poly-time algorithm for B, and f be a poly-time function such that $f(x) \in B \iff x \in A$. Then I get a polynomial time algorithm M' for A as follow :

Algorithm 3 A polytime algorithm for A given a polytime decider M for B

Input: x

Compute $f(x)$ ▷ This takes polynomial time since f is poly-time computable

Run M on $f(x)$ ▷ M runs in polynomial time in $|x|$
▷ Since $|f(x)|$ is polynomial in $|x|$, running M takes polynomial time in $|x|$

if M accepts $f(x)$ **then**
  Accept x
else if M rejects $f(x)$ **then**
  Reject x
end if

▷ This clearly runs in polynomial time in $|x|$

We want to prove the above is a polynomial-time decider for A.

By definition $f(x) \in B \iff x \in A$. Algorithm 3 accepts if and only if M accepts $f(x)$. Since M is a decider for B, Algorithm 3 accepts if and only $f(x) \in B$. So we can conclude Algorithm 3 accepts if and only if $x \in A$.

We now need to prove this runs in polynomial time $O(|x|^k)$ for some k. This follows because every step in the algorithm runs in polynomial time, as explained in the comments to each step in the algorithm above.

A more detailed argument (not necessary, but may be helpful for understanding) is the following. Computing $f(x)$ runs in polynomial time in $|x|$, since we are given that f is poly-time computable.

In particular, we have $|f(x)| \leq |x|^{k_1}$ for some constant k_1. Then running M on $f(x)$ will take time polynomial in $|f(x)|$, so this runs in time $|f(x)|^{k_2}$ for constant k_2, so running M takes time $|x|^{k_1+k_2}$. Overall, this runs in time $O(|x|^{k_1+k_2})$ which is polynomial in x. □

Similarly, we have the following:

Theorem 4. If $A \leq_{\mathrm{P}} B$ and $B \in \mathrm{NP}$, then $A \in \mathrm{NP}$.

Proof. The proof is similar to the case where $P \in B$, but by replacing M with a polytime NTM that decides B. □

Poly-time mapping reductions motivate the following definition :

Definition 10 (NP-Hardness)**.** A language B is NP hard, if **for all** languages $A \in \mathrm{NP}$ we have that there is a polynomial time reduction from A to B (A $\leq_{\mathrm{P}}$ B).

In particular, if we were to find a polynomial time for an NP-hard problem, we would have that NP = P. Another important definition is that of NP completeness :

Definition 11 (NP-Complete)**.** A language B is NP complete iff $B \in \mathrm{NP}$ **and** B is NP-hard.

Obviously, you might wonder, how we can show a problem is NP hard in the first place. How would we be able to show that **all** languages in NP are polytime reducible to some language ? This is the seminal result of Cook and Levin who independently proved the following :

Theorem 5. 3-SAT is NP-Complete.

Following this breakthrough results, many languages were shown to also be NP-Complete / NP-Hard. In particular, the following theorem is key in proving NP-hardness of other languages.

Theorem 6. If $A \leq_{\mathrm{P}} B$ and A is NP-hard, then B is NP-hard.

Proof. Let A, B be such that $A \leq_{\mathrm{P}} B$ and A is NP-hard. Let L be any language in NP (we want to prove that $L \leq_{\mathrm{P}} B$). By definition of NP-Hardness we have $L \leq_{\mathrm{P}} A$. Thus, there exists a polynomial

time computable function g such that $x \in L \iff g(x) \in A$. Furthermore since $A \leq_{\mathrm{P}} B$, we have that there exists a poly-time computable function f such that $x \in A \iff f(x) \in B$.

So we claim $f \circ g$ is a polynomial time function and that $x \in L \iff f(g(x)) \in A$. Assuming these two claims are true, we have $L \leq_{\mathrm{P}} B$ (by definition), and thus any language $L \in \mathrm{NP}$ is polynomial time reducible to B, so B is $\mathrm{NP} - hard$.

First, let's show $f \circ g$ is computable in polynomial time. Since g is polynomial time computable, given x as input, we can compute $g(x)$ in polynomial time. So we know $|g(x)|$ is polynomial in $|x|$ (otherwise, we wouldn't even have the time to write $g(x)$ in polynomial time). Since f is also computable in polynomial time, we can compute $f(g(x))$ in polynomial time in $|g(x)|$. But this is also polynomial in $|x|$ (since $|g(x)|$ is polynomial in $|x|$). [2] So this shows that $f \circ g$ is computable in polynomial time.

Now we want to show this $x \in L \iff f(g(x)) \in B$.

- For the first direction, if $x \in L$, then we must have $g(x) \in A$. But we know $y \in A \Rightarrow f(y) \in B$, so $g(x) \in A \Rightarrow f(g(x)) \in B$, so putting it all together, $x \in L \Rightarrow g(x) \in A \Rightarrow f(g(x)) \in B$.
- For the other direction, assume $f(g(x)) \in B$, then it must be that $g(x) \in A$ by the property of f. But also, by property of g, if $g(x) \in A$ we must have $x \in L$. So $f(g(x)) \in B \Rightarrow x \in L$.

So we proved both direction, so the claim is true.

□

When asked to prove a language L is NP hard or NP complete, you should proceed as follow :

Proof Template 4 (Show that B is NP Hard)**.**

1. Pick NP-hard problem A for which you want to show that $A \leq_{\mathrm{P}} B$.
2. Give pseudo-Code for a function f.
3. Show that given x as input, $f(x)$ can be computed in polynomial time, namely in time $O(n^k)$ for some constant k. (No need to prove specifically what k is).
4. Show that if $x \in A$, then $f(x) \in B$
5. Show that if $x \notin A$, then $f(x) \notin B$

Proof Template 5 (Show that B is NP Complete)**.**

1. Show $B \in \mathrm{NP}$
2. Show B is NP hard

[2] More details: $g(x)$ must have length $|x|^{k_1}$ for some constant k_1. $f(g(x))$ can be computed in time $|g(x)|^{k_2}$ for some constant k_2. So $f(g(x))$ can be computed in time $O(|x|^{k_1+k_2})$ which is polynomial in $|x|$.

Here are some examples of NP complete problems: :

- SAT: Given a CNF formula ϕ, is ϕ satisfiable ? And 3-SAT : Given a CNF formula ϕ where every clause has 3 literals, is the ϕ satisfiable ?
- Hampath. Given a graph G and two nodes s, t is there an Hamiltonian path from s to t in G ?
- Clique. Given a graph G and an integer k, does G contain a k-clique (G contains as a subgraph the complete graph on k vertices) ?
- Sudoku. Given a $n^2 \times n^2$ sudoku puzzle, does it have a solution?
- Many, many more...

4 Good to know : How to deal with subsets

Often a problem will require to look at all the subsets of size k of some set. If you have a set with n elements, the number of subsets of size k is denoted $\binom{n}{k}$. **It's important to know that :**

$$\binom{n}{k} = O(n^k)$$

Here's an example of a problem in P.

Example 3. The problem 3-Sum is defined as follow. You are given as input a set S of n numbers in $\mathbb{Z}$. The goal is to know whether there is any way to pick 3 distinct element a, b, c in S such that $a + b + c = 0$.

We will follow the template given earlier to prove this is in P.

Algorithm 4 3-Sum Algorithm

Input: $\langle S \rangle$
▷ Here S is a set of n numbers in $\mathbb{Z}$.

for every subset $\{a, b, c\}$ of S **do**
  If $a + b + c = 0$: Accept.
end for
Reject.

The above runs in polynomial time: We look all the subsets of 3 elements in S and check if the 3 elements sum to 0. Thus we need to look at $\binom{n}{3}$ subsets of S and for each subset, we check if $a + b + c = 0$ (you can assume this check takes polynomial time). So the running time is $O\left(\binom{n}{k}\right) = O(n^3)$, which is polynomial.

This algorithm is clearly correct:

- If $\langle S \rangle$ isn't in 3-Sum, we will never find $a, b, c \in S$ that sum to 0 during the for loop, so we'll reject.
- If $\langle S \rangle$ in in 3-Sum, then there are $a, b, c \in S$ that sum to 0. So when we look at these three elements, the algorithm will accept.

5 Good to know: Representing numbers

Given a number N, there are many ways to give it as input to a Turing Machine. The most common encodings are the following:

- Unary: Give as input 1^N (so N times the number 1).
- Binary: Give as input the binary representation of N.

In the first case, this takes N bits. In the second case, this takes $O(\log(N))$ bits. So there is an exponential gap between how many bits are used. This is a very important difference when thinking about poly-time algorithm.

In particular, unless explicitly mentioned, any number received as input is given in binary. So given N, your algorithm is polytime if it runs in time $O(\log(N)^k)$, if it runs in time $O(N^k)$ this is not polytime.

Consider following language:

$$\mathsf{Composite} := \{\langle N \rangle \mid N \text{ is the binary representation of a composite integer} \geq 2\}.$$

Consider the following decider for Composite.

Algorithm 5 A decider for Composite

Input: $\langle N \rangle$ the binary representation of a number ≥ 2.

---

for $2 \leq i < N$ **do**
  If i divides N, accept $\langle N \rangle$.
end for
Reject $\langle N \rangle$.

Clearly, the above accepts if and only if there exists i that divides N. So this is indeed a decider for Composite. However, this runs in time $O(N)$, since the for loop might be executed $O(N)$ times. So this isn't polynomial time! And even if we changed the loop to only go through $\sqrt{N}$ it still won't be polynomial time, as $\sqrt{N} = 2^{\frac{\log N}{2}}$ so it runs it time $2^{n/2}$ where n is the length of the input.

However, here is a simple NTM to decide Composite.

Clearly, this algorithm can accept if and only if N is composite. If N is composite, choosing i that divides N will lead to accept. If N isn't composite, no i can divide N, so we never accept.

Algorithm 6 A polytime NTM for Composite

Input: $\langle N \rangle$the binary representation of a number ≥ 2.

Non-deterministically guess $2 \leq i < N$.
if i divides N **then**
    Accept $\langle N \rangle$.
else
    Reject $\langle N \rangle$.
end if

Why is this polytime ? Guessing $i < N$ is polytime[3]. Checking if i divides N can also be done in polytime in $\log(N)$ (you can assume it). So everything runs in time $O(\log(N)^k)$, thus polynomial.

Composite is actually in P, but the algorithm for this (the AKS primality test) is extremely complicated.

6 Notes about SAT

When dealing with a boolean formula we have **variables** $x_1, \ldots, x_n$. An assignment to the variables $x_1, \ldots, x_n$ means we give each x_i value 0 (false) or the value 1 (true). In particular, if we have n variables there is 2^n possible assignments.

A **literal** is a boolean variable x or it's negation $\bar{x}$ (I.e. if we assign $x = 0$ then $\bar{x} = 1$ and if $x = 1$ then $\bar{x} = 0$).

A clause is a $\vee$ (logical OR) of literals. For example $C_1 := x_1$, $C_2 := \bar{x}_1 \vee x_3 \vee \bar{x}_4$ are example of clauses.

Given an assignment of 0/1 to each variable x_i, we say the assignment satisfies a clause C, if **at least one literal** of C evaluate to 1.

Consider the clause C_2 above. To satisfy C_2 we need at least one of $\bar{x}_1, x_3$ or $\bar{x}_4$ to be set to 1. Thus is we assign $x_1 = 0$ or $x_3 = 1$ or $x_4 = 0$ then C_2 is satisfied. An assignment falsifies C_2 if it sets $x_1 = 1, x_3 = 0$ and $x_4 = 1$, since then all the literals of C_2 evaluate to 0.

A CNF formula ϕ is the $\wedge$ of a bunch of clauses. Here are some examples :

- $\phi := (x_1) \wedge (x_2 \vee \bar{x}_3)$
- $\phi := (x_1) \wedge (x_2 \vee \bar{x}_4 \vee x_5) \wedge (\bar{x}_1 \vee \bar{x}_4)$

[3] i has at most $O(\log(N))$ bits since $i < N$. For each of this bits, the TM can guess 0/1.

What does it to satisfy a CNF formula ϕ ? If ϕ contains variables $x_1, \ldots, x_n$, then we must assign each x_i to 0 or 1 so that **all** the clauses of ϕ are satisfied. Sometimes, it's impossible to satisfy a CNF formula.

Definition 12. A k-CNF formula, is a CNF formula where all clauses have at most k literals

Here's an example of a 3-CNF :

$$\phi := (x_1 \vee x_2 \vee \bar{x}_3) \wedge (\bar{x_2} \vee \bar{x_3} \vee \bar{x}_4) \wedge (x_1 \vee x_2 \vee x_5)$$

A k CNF formula has at most $\binom{2n}{1} + \binom{2n}{2} + \ldots + \binom{2n}{k}$ clauses, since for each clause we pick at most k literals from $x_1, \bar{x}_1, \ldots, x_n, \bar{x}_n$.

To check a k-CNF formula ϕ is satisfiable, we can use a brute force algorithm that goes over all possible assignments to the variables. In particular, we have n variables, where each can take 0/1 value. There is 1 assignment will all variables set to 0, $\binom{n}{1}$ assignments with 1 variable set to 1, $\binom{n}{2}$ assignments with 2 variable set to 1, etc... So in total there is :

$$\sum_{k=0}^{n} \binom{n}{k} = 2^n \text{ many possible assignments .}$$

[4] Checking an assignment satisfies ϕ means checking that each clause has at least one literal evaluating to 1, so that takes time $O(|\phi|)$. So the run time of this brute force algorithm is $O(2^n * |\phi|)$. Note that this runtime is exponential whenever $|\phi|$ is polynomial in n (when the number of clauses is polynomial in the number of variables), which is always the case for a k-CNF where k is constant. While the brute force algorithm is not efficient, finding an efficient algorithm is extremely hard, and in fact believed to be impossible. Indeed, a polynomial time algorithm for checking satisfiability exists if and only if P = NP (since this is an NP-complete problem).

Here's an example of the brute force algorithm on a CNF. We go over all assignments to the variables, and for each clause check if it's satisfied. If they are all satisfied then ϕ is satisfied :

[4] Another way to see this is that every variable of ϕ can be assigned 0 or 1 so there are 2^n options to assign the variables.

Example 4.

$$\phi := (x_1 \vee x_2) \wedge (x_2 \vee x_3) \wedge (\bar{x}_3 \vee \bar{x}_1) \wedge (\bar{x}_4 \vee \bar{x}_2) \wedge x_4$$

x_1	x_2	x_3	x_4	$x_1 \vee x_2$	$x_2 \vee x_3$	$\bar{x}_3 \vee \bar{x}_1$	$\bar{x}_4 \vee \bar{x}_2$	x_4	ϕ
0	0	0	0	0	0	1	1	0	0
0	0	0	1	0	0	1	1	1	0
0	0	1	0	0	1	1	1	0	0
0	0	1	1	0	1	1	1	1	0
0	1	0	0	1	1	1	1	0	0
0	1	0	1	1	1	1	0	1	0
0	1	1	0	1	1	1	1	0	0
0	1	1	1	1	1	1	0	1	0
1	0	0	0	1	0	1	1	0	0
1	0	0	1	1	0	1	1	1	0
1	1	0	0	1	1	1	1	0	0
1	0	1	0	1	1	0	1	0	0
1	0	1	1	1	1	0	1	1	0
1	1	0	1	1	1	1	0	1	0
1	1	1	0	1	1	0	1	0	0
1	1	1	1	1	1	0	0	1	0

You can see that for all truth assignment, we always have one clause that evaluates to 0 (no literal is set to true), thus ϕ isn't satisfiable.

7 Exercises

Exercise 1 (True False)**.**

1. All of $n * \log(n)$, $n^{100} + 3n^2$, $n^{0.001n}$, $2^{\sqrt{n}}$ are polynomial.
2. If you show a verifier $V(x, c)$ for a language L, where V runs in time polynomial in $|x| + |c|$, then L must be in NP.
3. All NP Complete problems are polytime reducible to each other.
4. Say the input of a TM is an integer N given in binary, and M runs in time $O(N^2)$, then M runs in polynomial time.

Exercise 2. Show that NP is closed under union. That is if $L_1, L_2 \in$ NP then $L_1 \cup L_2 \in$ NP.

Exercise 3.

- Fix some constant k. The problem k-Clique is defined as follow : You are given as input a graph G, G is a graph on n nodes. You can think of G as being described by a $n \times n$ binary matrix A where $A_{i,j} = 1$ iff there's an edge (i, j) in G (so the input size is n^2). You must accept G if and only there is a subset of k vertices V^* of G, such that these vertices form a complete graph (any two nodes have an edge between them). Why is this problem in P ?
- The NP complete problem Clique is defined as follow : You are given as input a graph G, G is a graph on n nodes. **But now k is given to you as input in decimal**. (So the input size is roughly $(n^2 + \log_{10}(k))$). Again, You must accept G if and only there is a subset of k vertices V^* of G, such that these vertices form a complete graph. Why doesn't the previous proof work to show this problem is in P ?

Exercise 4. Problem 7.18 in the book. Show that if P = NP,then every language $A \in$ P, except $A = \emptyset$ and $A = \Sigma^*$, is NP-Complete. (Hint : think about the definition of $L \leq_{\mathrm{P}} A$, knowing that $L \in$ NP implies $L \in$ P.)

Exercise 5. Problem 7.21 in the book. Let G represent an undirected graph. Also let

- SPATH = $\{\langle G, a, b, k\rangle$— G contains a simple path of and length **at most** k from a to $b\}$
- LPATH = $\{\langle G, a, b, k\rangle$— G contains a simple path of and length **at least** k from a to $b\}$

Show that SPATH $\in$ P and LPATH is NP-Complete. (A simple path is a path that doesn't visit the same node twice.)