1 Recall NP, NP-completeness

Recall from last time

NP: class of decision (yes/No answer) problems $\Pi$ for which there exists a ‘verification procedure’ $V_\Pi$ and a $c$ such that for $x$ an input of $\Pi$ where $n = |x|$,  

1. $V_\Pi(x, y)$ outputs True or False.
2. $\Pi(x) = Yes$ if and only if there exists a $y$ such that $|y| < n^c$, and $V_\Pi(x, y) = True$
3. $V_\Pi(x, y)$ runs in polynomial time in $n$.

A *poly-time reduction* from problem $A$ to problem $B$ satisfies the following conditions:

1. if $x$ is a YES-instance of $A$ then $f(x)$ is a YES-instance of $B$.
2. if $x$ is a NO-instance of $A$ then $f(x)$ is a NO-instance of $B$.
3. $f$ can be computed in polynomial time.

An *NP-complete* if a decision problem has these two properties:

- it is in $\textbf{NP}$;
- all other problems in $\textbf{NP}$ reduce to it (All problems can be solved if NP complete problem can be solved)
If a problem is not in NP but we can show that all other problems in NP reduce to it, we say that it is **NP-hard**.

Why do we care about classifying problems as NP-complete (i.e. the hardest in NP)?

There are two reasons to study NP-complete problems. The practical one is that if you recognize that your problem is NP-complete, then you have three choices:

1. you can use a known slow algorithm for it, instead of wasting your time trying to find a polynomial time algorithm for it;
2. you can settle for approximating the solution, e.g. finding a nearly best solution rather than the optimum (this applies to the search and optimization version of the decision problem which was shown NP complete); or
3. you can change your problem formulation so that it is in P rather than being NP complete. Sometimes special cases of NP complete problems are surprisingly easy.

From a theoretical point of view, as we discussed above, similarly to cSAT any of the NP-complete have the interesting property that if you can solve it in polynomial time, then you can solve every problem in NP in polynomial time. If you find a polynomial-time algorithm for just one NP-complete problem, you will have shown that $P = NP$. Conversely, if you could show that any one of the NP-complete problems that we will study cannot be solved in polynomial time, then you will have not only shown that $P \neq NP$, but also that none of the NP-complete problems can be solved in polynomial time.

## 2 Cooks Theorem

Recall the problem of **circuit satisfiability** (**cSAT**) that we defined last time.
Given a description of a Boolean circuit $C$ specified by a sequence of gates (AND, OR, and NOT) and variables is there a way to set 1/0 values for the input variables $x_1...x_n$ so that the the value of the output gate is 1 (true)? In this case we write $C(x) = 1(True)$ and say that the circuit $C$ is satisfiable.

**Cook’s Theorem:** cSAT is NP-complete

This is one of the most important results in Computer Science. Let us argue its correctness.

First of all, obviously cSAT is in NP. For an input $C$ to cSAT, given a set of values for the inputs $x$ of $C$, it is easy to check if $C(x) = 1$.

Second, we have to show that for every other problem $A$ in NP; there is a reduction from $A$ to cSAT, namely a polynomial time function $R$ such that $A(x) = \text{Yes}$ iff $R(x)$ is satisfiable.

Let us illustrate how to do this for one specific problem in NP (the factoring problem) and then show the same idea can work for any problem in NP. Consider for an example the decision version of the factoring problem. Recall: on input $(N,B)$, a certificate that factoring’$(N,B)=\text{Yes}$, would be $d$ such that $d$ divides $N$ and $1<d<N$.

Let us illustrate how to do this for one specific problem in NP (the factoring problem) and then show the same idea can work for any problem in NP. Consider for an example the decision version of the factoring problem. Recall: on input $(N,B)$, a certificate that factoring’$(N,B)=\text{Yes}$, would be $d$ such that $d$ divides $N$ and $1<d<N$. $V_{\text{factoring'}}((N,B),d)$ checks that

- $d$ divides $N$
- $1<d<B$

The idea is that we will design a circuit with input $d$: that will check if an input $d$ is a factor of $N$ which is less than $B$. Satisfying that circuit will be possible only if such $d$ exists.

The polynomial time reduction from factoring to cSAT works as follows: On input $(N,B)$:

- Design two circuits $C_1$ and $C_2$ such that (1) $C_1(d,N) = 1$ if $N \mod d = 0$ (this checks that $d$ divides $N$ without remainder) and (2) $C_2(d,N,B) = 1$ if $1<d<B$ (comparison).
• The circuit $C'(N, B, d) = C1 \text{ AND } C2$

• Now fix those input variables corresponding to $N$ and $B$ (to the values of $N$ and $B$) but do not fix the input variables corresponding to $d$ (since you do not know $d$!!). Let $C_{N,B}(d)$ denote the circuit $C'$ with variables $N$ and $B$ fixed.

• output $C_{N,B}$.

Namely the reduction function $R(N, B) = C_{N,B}$. It is easy to see that as required factoring $(N, B) = \text{Yes}$ if and only if $C_{N,B}$ (in variables $d$) is satisfiable. What you need to check for yourselves is how to design a polynomial in $n$ size circuits $C1$ and $C$ where $n$ is the number of bits in $N$ and $B$.

We are ready to do the general proof. Lets take any problem $A$ in $\text{NP}$. The fact that $A$ is in $\text{NP}$ means that there is a polynomial algorithm $V_A$ such that for each yes input $x$, there exists a certificate $y$ s.t. $V_A(x, y) = \text{true}$.

A reduction from $A$ to $\text{cSAT}$ works as follows: on input $x$ build a circuit that is satisfiable iff there exists an input $y$ that makes $V_A(x, y) = \text{true}$. The circuit is essentially an implementation of $V_A(x, )$.

The technical part is to show how can a polynomial time ($p(n)$ time say on input of size $n$) algorithm with input $y$ be implemented by a polynomial size circuit (of size $p^2(n)$) which is satisfiable with $y$ exactly when the algorithm on $y$ outputs $\text{true}$.

We will show this with a lot of hand waiving. Any verification algorithm $V_A$ computes in steps according to a set of fixed instructions: starting from initial start state, and going on to the next state possibly writing some output and reading some inputs bit to compute what is the next state – this process can be described by what is called a tableau of length $p(n)$ rows in which each row is at most $n + p(n) + \log p(n)$ long, summarizing the input, the contents of the memory, and the program counter. The first row contains only the input (nothing is written yet, and the program counter is at 0). How to go from row to row is based on a a computation of a very simple combinatorial circuit
(made of AND, OR, and NOT gates and wires connecting between them) on \(p(n) + n + \log p(n)\) input variables originating from the previous state and \(p(n) + n + \log p(n)\) output variables. If we superpose enough (polynomially many) such circuits, we get a large circuit that describes the full run of the validation algorithm. The input to the input gates of the circuit \(C\) are the bits of the input to \(A\) and the bits of the certificate. Furthermore, it is possible given the description of \(V\), come up with the description of the circuit in polynomial time.

This can be technically formulated in

**Lemma:** Suppose algorithm \(A\) computes a one bit output, in \(p(n)\) time, where \(n\) is the length of the input. Assume that the input is represented as a sequence of bits. Then, for every fixed \(n\), there is a circuit \(C\) of size about \(O(p(n)^2 \log p(n)^{O(1)})\) such that for every input \(x\) of length \(n\), we have \(A(x) = C(x)\). Furthermore, there exists an efficient algorithm (running in time polynomial in \(p(n)\)) that on input \(n\) and the description of \(A\) produces \(C\).

Putting all of this together. Suppose now that we are given an input \(x\) of \(A\). Plug in \(x\) to \(C\) in the appropriate input wires to get circuit \(C_x\), and keep the input wires that correspond to the certificate unspecified, we get an instance of \(c\text{SAT}\) such that \(x\) is a “yes” instance of \(A\) if and only if \(C_x\) can be satisfied by some value for the certificate. The construction of the circuit we described is the reduction from \(A\) to **circuit SAT!**  QED

Proving the existence of NP complete problems seems on the face of it an enormous task. But it will be easier now that we know of at least one: **circuit SAT.**

### 3 How to show that more problems are NP-complete

Note that it is easy to see that polynomial-time reducibility is *transitive*. Namely, if \(A \leq_p B\) and \(B \leq_p C\) then \(A \leq_p C\). This means that to prove
that a problem $C$ is $\text{NP}$-complete, we only need to find a problem $B$ that is already known to be $\text{NP}$-complete and reduce $B$ to $C$. It will follow that all problems $A$ in $\text{NP}$ which were reducible to $B$ are also reducible to $C$.

**Theorem:** Let $C$ and $B$ be two decision problems and $B$ is $\text{NP}$-complete. If $C$ is in $\text{NP}$ and $B \leq_p C$, then $C$ is $\text{NP}$-complete.

**Proof:** If $A$ reduces to $B$, there is a polynomial time computable function $R_1$ such that $A(x) = B(R_1(x))$; if $B$ reduces to $C$ it means that there is a polynomial time computable function $R_2$ such that $B(y) = C(R_2(y))$. Set $R(x) = R_2(R_1(x))$. We conclude that $R$ is a polynomial time reduction from $A$ to $C$ since: (1) $A(x) = C(R(x))$; and (2) since both $R_1$ and $R_2$ run in polynomial time, so does $R$. Since $B$ is $\text{NP}$-complete, it has been shown that for all $A$ in $\text{NP}$, $A \leq_p B$ and thus $A \leq_p C$ which established $C$ as $\text{NP}$ complete.

QED

Now that circuit SAT has provided a place to start, we shall prove many problems $\text{NP}$-complete by the reductions.

If two problems $A$ and $B$ are reducible to each other, we say that they are polynomial time equivalent (upto the cost of the reduction).

**Definition:** Let $a$ and $B$ be two decision problems. We say that $A \equiv B$ if $A \leq_p B$ and $B \leq_p A$.

We can (and do) draw graphs of reductions

$cSAT \leq_p 3SAT \leq_p CLIQUE \leq_p IndependentSet \leq_p VertexCover \leq_p SetCover$ (we will go down this path in these notes).

or going down another path $cSAT \leq_p 3SAT \leq_p 3DimensionalMatching$ and so forth...

Note: most of these problems were defined in the previous lecture notes IndependentSet, VertexCover, and SetCover are defined below.

### 3.1 From cSAT to 3SAT
Recall that a Boolean formula in Conjunctive Normal Form (CNF) is the conjunction of clauses each of which is made of a disjunction of literals. A literal is a variable which is either negated or not negated. In the case of 3CNF formulas, each clause is restricted to consist of at most three literals.

An input to 3SAT is a 3CNF formula \( \phi \). We ask if there exists a setting of the variables of \( \phi \) to True and False (1/0 correspondingly) which make \( \phi \) evaluate to True. If such a setting exists, we say that \( \phi \) is satisfiable.

For example, \( \phi = C_1 \text{ AND } C_2 \) for clauses \( C_1 = (x_1 \text{ OR } \neg(x_2)) \) and \( C_2 = (\neg(x_1) \text{ OR } x_2) \) is satisfied whenever \( x_1 = \neg(x_2) \).

**Fact:** 3SAT is NP-complete

The easy part is to show that 3SAT is in NP. The assignment of 1/0 to the variables which makes a formula evaluate to T is a certificate to the satisfiability of the formula.

We want to prove that 3SAT it is NP-hard, and we will do so by reducing from cSAT which we already have shown to be NP-complete. Suppose that we are given a Boolean circuit \( C \) with variables \( x \), we need to construct from it an input to 3SAT, that is, a formula \( \phi \) that is satisfiable if and only if there is a satisfying setting of the unknown inputs of \( C \).

**Attempt 1:**

Given a circuit, transform it into a Boolean CNF formula that computes the same Boolean function. Unfortunately, this approach cannot lead to a polynomial time reduction. Consider the Boolean function that is 1 iff an odd number of inputs is 1. There is a circuit of size \( O(n) \) that computes this function for inputs of length \( n \): it simply adds input bits and checks if the number is odd (design an \( O(n) \) size circuit for it!). But it can be proven the smallest CNF for this function has size more than \( 2^n \).

This means we cannot translate a circuit into a CNF formula of comparable size that computes the same function, but we may still be able to transform a circuit into a CNF formula such that the circuit is satisfiable iff the formula is satisfiable (although the circuit and the formula do compute somewhat different Boolean functions) !!!
Indeed this is what the following reduction does.

\(R(C)\): First, we choose a set of new variables for our new 3CNF formula: a variable per each input to \(C\) and for each gate of \(C\), So, if circuit \(C\) has in total \(m\) gates and inputs where the output of the circuit is the output of gate \(m\), then we introduce new variables \(g_1 \ldots g_m\) with the intended meaning that variable \(g_1 \ldots g_n\) corresponds to input variables \(x_1 \ldots x_n\) to \(C\), and \(g_j\) for \(n < j \leq m\) corresponds to the output of gate \(j\). Note that the \(g_m\) will have the output of the entire circuit \(C\).

Second, build the following clauses corresponding to each gate of \(C\).

1) For each gate \(j\) for \(n < j \leq m\), we define a CNF \(F_j\) saying that the value of the variable \(g_j\) for that gate is set in accordance to the value of the inputs to that gate and the type of gate. This is done differently for NOT, AND and OR gates.

For example for a NOT gate \(j\) whose input is the output of gate \(i\), we want to write a formula \(F_j\) that says that \(g_j = 1\) if and only if \(g_i = 0\). This can be accomplished as follows. Let \(F_j = (g_i \text{ OR } g_j)(\text{NOT}(g_i) \text{ OR } \text{NOT}(g_j))\). This sub-formula \(F_j\) will be true only when \(g_i = \text{NOT}g_j\). Moreover note that the sub-formula \(F_j\) is in 3CNF form.

Similarly, For an AND gate \(j\) applied to the output of gates \(i\) and \(l\), we can write other 3CNF formulas \(F_j\) which are true only when \(g_j = g_i \text{ AND } g_j\) and Similarly for OR. EXERCISE: do it!

2) We also have a \((m + 1)\)-th sub-formula \(F_{m+1} = g_m\) that amounts to saying that \(F_{m+1}\) evaluates to 1 only when the output of the circuit is 1.

3) Finally outut \(F = \wedge F_j\) for \(j=1\ldots m+1\).

We claim that it is obvious if \(C\) can be satisfiable by some settings of the input variables \(x_1 \ldots x_n\), then \(F\) is satisfiable in all of \(g_1 \ldots g_m\), when \(g_1 \ldots g_n\) are set to \(x_1 \ldots x_n\) and the value of \(g_j\) is set to the output of gate \(j\) in \(C\). Conversely, if \(F\) is satisfiable by some truth assignment to \(g_1 \ldots g_m\), then by construction \(g_m\) must be 1. Take the truth assignment given to variables \(g_1 \ldots g_n\), and assign it to each of the variables of \(C\), i.e. \(x_i = g_i\). We can prove by induction that
the output of gate j in C, for this assignment of the variables, will equal to $g_j$, and therefore the output gate of C, will equal to $g_m$ which is 1.

3.2 From 3SAT to CLIQUE

Recall that the clique in a graph is a fully connected set of nodes. The CLIQUE problem on input a graph G and an integer K, asks whether there is a clique of size $K$ or larger in G.

**Fact:** CLIQUE is NP-complete

Given a 3CNF formula $\phi = C_1 \text{ AND } C_2 \ldots \text{ AND } C_m$ with clauses $C_1 \ldots C_m$ and variables $x_1 \ldots x_n$, we want to reduce it in polynomial time to a graph $G = (V,E)$ and an integer $k$ such that $\phi$ is satisfiable if and only if G has a clique of size $k$.

R: On input $\phi$, an instance $(G,k)$ is constructed as follows.

1. V: Add a vertex for each literal (either $x_i$ or $\text{NOT}(x_i)$ in each clause.
2. E: Add an edge between all pairs of vertices except between:
   (a) two vertices that came from the same clause (in the same triple)
   (b) two vertices corresponding to a literal and its negation.
3. Set $K = m$
4. output $G = (V,E)$ and $K$

We show that $\phi$ with $m$ clauses is satisfiable iff there exists a clique of size $m$ in the graph constructed above.

If $\phi$ has a satisfying assignment, then choose one of the literals that made it true in every clause. The corresponding nodes in the graph form a m-clique since each pair of these nodes has an edge between them (Why? (1) we choose one literal per clause in a satisfying assignment, so all literal chosen
were in different clauses (2) if literal made the clause true and was chosen, its negation is false could not made its clause true and was not chosen.)

Conversely, if G has a m-clique, then for each node in the clique, assign to the literal in the assignment that it came from the value True, and assign the negation of all these literals the value False. The rest of the variables can be all be set to either true or false (it makes no difference). Each vertex in the clique must have come from a literal in a different clause in the formula (since within the same clause there are no edges), and since the clique is of size m, it means that now each of the m clauses in the formula has a literal in it set to True. Thus, all clauses are now satisfied by some literal. Moreover, it is a legal truth assignment since you never will give both a literal and its negation the value True (this follows since they cannot both be in the clique).

QED

3.3 From CLIQUE to IndependentSet

Another type of reduction is from CLIQUE to the IndependentSet (IS) problem. The IS problem is defined as follows. We are given an input graph $G = (V ; E)$ and an integer $K$. We are asked whether there is a subset of $V$ of size at least $K$ such that none of the vertices in it have an edge between them (i.e none are connected to each other). Such a subset is called an independent set.

Fact: IS is NP-complete

Obviously IS is in NP, since on input $G$ and $K$, given a subset of $V$ (a 'certificate'), you can always check that it forms an independent set of size $K$ in polynomial time.

We will reduce CLIQUE to IS. Let $G = (V, E)$ be a graph and $K$ an integer input to the CLIQUE. The reduction from clique to independent set is very simple: We go from the instance $(G, K)$ of clique to the instance $(G', K)$,
where \( G' = (V, E') \) where \( E' \) has exactly those edges that \( E \) did NOT have (\( G' \) is the so called is the complement of \( G \)).

Why does this work? Since a subset of the vertices in \( G \) forms a clique if and only if the same subset it is an independent set in the complement graph.

QED

### 3.4 From IS to VertexCover

A vertex cover of \( G \) is a set \( C \) subset of the vertices \( V \) such that all edges in \( E \) have at least one endpoint in \( C \). The VertexCover(VC) is: Given a graph \( G \) and a number \( K \), does \( G \) have a vertex cover of size at most \( K \)?

**Fact:** VertexCover in NP-complete

It is simple to see that VertexCover is an NP.

The reduction from independent set to vertex cover is quite simple as well, and based on this observation: \( C \) is a vertex cover of \( G = (V, E) \) if and only if \( V - C \) is an independent set! This is because any two nodes not in a vertex cover cannot have an edge between them, this edge would not have an endpoint in the vertex cover.

The reduction follows. Given an instance \((G = (V, E), K)\) of independent set, produce the instance \((G = (V, E), |V| - K)\) of vertex cover. There is an independent set with \( K \) nodes or more if and only if there is a vertex cover of size \( |V| - K \) or less.

QED

### 3.5 From VertexCover to SetCover

Moving away from graphs lets see something a bit more general.

SetCover: Given a universe \( U \) of \( n \) elements, a collection of sets \( S_1...S_m \) which are each a subset of the universe, and an integer \( k \), we ask: do there exist at
most \( k \) of the sets \( \{S_i\} \), \( i \in I \) such that \( \bigcup_{i \in I} S_i = U \) (i.e \( S_i \) for \( i \in I \) ‘cover’ \( U \)). If so, then we say that input \( (U, S_i, i \in I, k) \) is a yes-instance of the set cover problem.

A possible application is. \( U \) is a universe of topics to learn for the exam. Each \( S_i \) is a set of notes that cover some of these topics. Goal is to learn all topics reading the least number of notes.

**FACT:** SetCover is NP-complete

SetCover is an NP. Furthermore it is essentially a generalization of Vertex-Cover and as the special case is NP-complete, certainly the general case is. Formally, this is seen as follows.

R: On input \( G=(V,E) \) (with \( n \) vertices \( m \) edges) and integer \( K \),

1. set \( U = E \)
2. for every vertex in \( V \), define \( S_v = \{ e \in E \ v \text{ is an end-point of } E \} \)
3. output \( (U, S_i, ..., S_m, K) \)

If there is a vertex cover \( S \subset V \) of \( G \) of size at most \( K \), then let \( I = \{ v \in S \} \). The resulting \( I \) is: of size at most \( k \) (since \( S \) is) , and \( \{ S_v, v \in I \} \) contain all \( e \in U \) (since \( S \) covers all edges in the graph and \( U \) corresponds to all of the edges).

Conversely, if there is a set cover \( \{ S_v \} \) for \( v \in I \) of size at most \( k \), then for every \( v \in I \) let \( v \in S \). Obviously \( S \) is of size \( k \) and it is a vertex cover in graph \( G \). QED

To summarize: we saw a few different strategies here for reductions.

- **Being able to express general computation as an instance of a problem:** \( A \leq_p cSAT \) (for any \( A \) in NP)
- **Encoding with gadgets:** \( cSAT \leq_p 3SAT \) and \( 3-SAT \leq_p CLIQUE \)
- **Simple equivalence:** \( CLIQUE \leq_p IS \leq_p VC \).
- **Special Case to General Case:** \( VC \leq_p SET-COVER \).
4 Decision to Search

An interesting property of NP-complete problems is that if you can solve in polynomial time an NP-complete decision problem (i.e. can tell if an instance is a Yes or No quickly), then you can also find a certificate in polynomial time.

Said differently if you can solve the decision problem in polynomial time you can also solve the search problem in polynomial time. This is not always the case, but is true for NP-complete problems.

Ex: if you had a way to decide on input C in polynomial time if C was satisfiable or not by some assignment, then you can actually find y such that C(y)=true in polynomial time.

Proof: Suppose that there exists an algorithm $A(C) = yes$ if $C$ is satisfiable and no otherwise. Let $y = y_1...y_n$ be the Boolean inputs to $C$.

- Set $C_0 = C$. if $A(C) = no$ then output ‘C is unsatisfiable’.
- For i=1...n,
  1. set $C_i = C_{i-1}(True, y_{i+1},...,y_n)$.
  2. If $A(C_i) = yes$, set $y_i = True$, else set $C_i = C_{i-1}(False, y_{i+1},...,y_n)$ and $y_i = False$.

Output $y_1....y_n$

This is true more generally for any NP complete problem.

5 Example of Intractable Problems which are unlikely to be NP-completeness

We have mentioned before there are some problems in NP which we do not polynomial time algorithms for but for which we can do better than exponential time. We already discussed he factoring problem. Another one whose status has intrigued theorists for awhile is the graph isomorphism problem.
**Factoring Problem**: Given an n-bit integer $N$, find a divisor $d$ of $N$ such that $2 \leq d \leq N$.

Decision version is much less natural but can be defined. **Factoring’ Problem**: Given an n-bit integer $N$, and a bound $B$, is there a divisor $d$ of $N$ such that $2 \leq d \leq B$.

It is known how to answer the factoring problem using an algorithm which runs in time $O(e^{O(n^{1/3})})$. This is sub-exponential time. Furthermore we know that quantum computers can solve factoring in polynomial time whereas we do not know any quantum algorithms for the problems above TSP, SAT, HAM, etc.

**Graph Isomorphism Problem**: Given two graphs $G_1$ and $G_2$ decide if there is a mapping $f$ from the vertices of $G_1$ to the vertices of $G_2$ such that there is an edge $(u, v)$ in $G_1$ if and only if there is an edge $(f(u), f(v))$ in $G_2$.

Again, the best known algorithm for graph isomorphism runs in $O(2^{\sqrt{n\log n}})$ which is strictly better than exponential time. we do not know a quantum algorithm for graph isomorphism.

6 Example of Intractable Problems Beyond NP

Finally, some problems are not known to have short and easy verifiable certificates. Namely, they may not even be in NP.

Consider, for example, the problem **non-SAT** problem: Given a formula $\phi$, how would you prove quickly (even if you were a magician) that it is true that there is **no truth assignment** that makes the formula true. Or the **graph-Non-Isomorphism**: How would you prove that there is NO mapping between all the vertices of the first graph to those of the second graph which respects the edge relationship. You could obviously enumerate all of them but that would not take polynomial time but exponential time.

Note that the above problems have exponential size certificates and solutions. For example for non-satisfiability, if you cycle through all the possible solutions
to a formula and if none satisfy it, then you know its not satisfiable. Even worse, some problems can not be solved at all in principle (they are called undecidable). So, its not a matter of inefficiency but of impossibility. An example of such a problem is the halting-problem. Given a program $P$ and an input $x$, decide if $P(x)$ will terminate. For proof, take 6.045.

In the last 2 lectures we let polynomial=time correspond to tractability. What if we let probabilistic polynomial time correspond to tractability, do things change significantly. We shall discuss this in the next lecture.