In the last part of this course, we will discuss problems that are difficult in the theoretical sense. So far one of the bars that we’ve been using to describe an algorithm as efficient is polynomial time: $O(n^c)$ for some constant $c$.

There are many problems that we know how to solve in poly time, such as

1. Shortest Path
2. Minimum Cut
3. Vertex cover / maximum independent set in a bipartite graph

There are also problems that we can solve via exhaustive search type algorithms. Examples include

1. Longest (simple) path, where we can enumerate over all $n!$ simple paths by trying all permutations of vertices, and taking the longest one that’s a path.
2. Maximum cut: we can try all $2^n$ cuts.
3. Integer program: where we can try every combination of variables.

For the last problem of integer program, it becomes important to discuss the bit-complexity of the numbers again: if we have a constraint

$$a \leq x \leq b,$$

we can check all numbers in the range $[a, b]$ in $b - a$ time. If $a$ and $b$ have at most $d$ (binary) digits, this takes $O(2^d)$ time.

So we will now use $n$ to parameterize the bit-complexity of representing the input of one of these problems. Formally, we use the $x$ to represent an input string, which has size $n$ ($n$ bits).
1 Decision Problems and $P$

We can also make one additional simplification: removing optimization objectives. Instead we view the objective as part of the problem specification, and query for whether a solution with better value exists.

For example, instead of asking for the maximum independent set, we ask for whether an independent set of value $k$ exists. This is a decision problem, which can be viewed as assigning every input, $I$, a value that’s either true or false. Such a problem has a polynomial time solution if there is a polynomial time algorithm that provides the right answer each time.

For the independent set problem, it introduces a factor of at most $n$ overhead: we just ask the question for every $k = 1 \ldots n$. It significantly simplifies our definition of problems, as well as polynomial time.

To get the overhead down to $O(n)$ for any input, we can binary search on the answer. Generally, the fact that things have at most $n$ bits means that the objective is the $2^n$ range. Binary searching on such a value takes $O(\log(2^n)) = O(n)$ steps, which gives the bound. Note that for more contrived problems one can do something different, but for most problems whose answers aren’t bigger than $2^n$, this is ok.

2 $NP$: Nondeterministic Polynomial Time

Problem that look harder on the other hand, such as max cut, and minimum vertex cover, have the property that it’s often much easier to verify a solution than computing an answer.

A good example of this is independent set: given a set of vertices $S$ (represented as a binary string of length $n$), we can check in $O(n^2)$ time whether it’s indeed an independent set by looking through all edges. Furthermore, we can check in $O(n)$ time whether the size of the set is at most $k$.

This leads the notion of non-deterministic polynomial time, or NP, where a solution can be verified in polynomial time, but searching through the solutions is much more complicated.

**Definition 2.1.** A decision problem is in NP if there exists a polynomial time verification algorithm $C(I, S)$ that takes both an input and a proposed solution, and verified whether the solution is ok, and satisfies the following properties:

1. the decision problem is true on input $I$ if and only if there exists some $S$ such that $C(I, S)$ returns true.

2. there is some constant $c$ for which $|S| \leq |I|^c$, and

3. the running time of $C(I, S)$ is $|I|^c$ for some constant $c$, aka. polynomial in the length of the input.
A good example of a problem in NP is minimum vertex cover. Here the decision version is whether there exists a vertex cover of size at most \( k \). For the vertex cover problem above, this \( S \) is formalized to the indicator string that’s 1 if a vertex \( u \) is chosen, and 0 otherwise. This string has length \( n \), and the running time of \( C(I, S) \) is \( O(n^2) \).

It can also be checked that any problem in P is also in NP: the verifier can simply take the empty string. On the other hand, anything in NP can be solved in exponential time by trying all \( 2^n \) certificates \( y \) and check if the verifier returns true.

## 3 Reductions

Most of these are problems that we don’t know algorithms that provably do better than brute force. However, we can define reductions between problems: The idea is to directly convert the inputs of the problems. Since we’re dealing with decision problems, these problems have the same output: true and false (equivalent to ‘yes’ and ‘no’ from the previous lecture). This means one strategy that we can take is to create a routine that converts an input of \( A \) to an input of \( B \).

**Definition 3.1.** A decision problem \( A \) is **polynomial time reducible** to a decision problem \( B \) if there exists a polynomial time computable function \( f \) such that for any input \( I \),

\[
A(I) = B(f(I))
\]

We denote this with \( A \rightarrow B \). Note that this means \( B \) is at least as hard as \( A \): an instance of \( A \) can be solved by transforming the input, and invoking \( B \) on the input.

In other words

1. Difficulties flow in the direction of arrows.
2. Algorithms flow in the direction against arrows.

More specifically, a reduction \( A \rightarrow B \) implies that

- if \( B \) can be solved in polynomial time, then \( A \) can also be solved in polynomial time.
- if \( A \) is hard, then so is \( B \).

For a simple example of a reduction, consider the problems:

- **\text{MAXINDEPENDNETSET}(G, k)**: whether \( G \) has a set of at least \( k \) vertices that are pairwise disjoint.
- **\text{MINIMUMVERTEXCOVER}(G, k)**: whether \( G \) has a set of at most \( k \) vertices that cover all edges.

They are interreducible due to the following observation:
Lemma 3.2. If $S$ is an independent set in $G$, then $V \setminus S$ is a vertex cover.

The proof is more or less by checking definitions. With this in mind, the reduction $\text{MAXINDEPENDNETSET} \rightarrow \text{MINIMUMVERTEXCOVER}$ is simply

$$f_{\text{MAXINDEPENDNETSET} \rightarrow \text{MINIMUMVERTEXCOVER}}(G, k) = (G, |V| - k),$$

That is, we check whether $G$ has an independent set of size at least $k$ by checking whether $G$ has a vertex cover of size at most $|V| - k$. It can also be checked that in the reverse direction, the same function also works.

4 3-SAT and NP-Hard Problems

Reductions play a key role in complexity theory. The Cook-Levin theorem states that every problem in NP can be reduced to 3-SAT. This problem takes a set of variables $x_1, \ldots, x_n$, and defines:

- a literal is either an atom $x_i$ or its negation $\neg x_i$.
- A clause is the disjunction ("or") of three literals.

The 3-SAT problem asks, given a propositional formula $\varphi(x_1, \ldots, x_n)$ which is the "and" of finitely many clauses of length 3, does there exist an assignment of either TRUE or FALSE to each $x_i$ which makes $\varphi(x_1, \ldots, x_n)$ evaluate to TRUE?

Theorem 4.1 (Cook-Levin Theorem). For any problem $A$ in NP, we have $A \rightarrow 3-SAT$.

This motivated the definition of NP-hard problems:

Definition 4.2. A problem is NP-hard if every problem in NP can be reduced to it.

This then leads to the extence of NP-complete problems:

Definition 4.3. A problem $A$ is NP complete if:

1. It is in NP.
2. Every problem in NP can be reduced to it. This is usually shown by exhibiting a NP-hard problem $B$ such that $B \rightarrow A$.

Our first NP-hard probelm is then 3-SAT: it is in NP because we can just exhibit a satisfying set of variable assignments. It’s NP hard by the Cook-Levin theorem.