Computational intractability. Many important computational problems are *intractable* — not solvable by any reasonably fast algorithm. Since 1950 or so, computer scientists have developed a fairly deep understanding of intractable problems.

In designing algorithms, it is useful to be able tell when a problem is intractable, or likely to be intractable. If your problem is intractable, there is no algorithm that is guaranteed to compute a correct solution to every input in reasonable time. In this case, you need some other approach. For example, perhaps you only care about solving a few particular instances of the problem. Then you might focus on solving just those instances. Or, you might look for fast algorithms that are guaranteed to return provably good *approximate* solutions. Or, you might revisit your application and try to find a different way to formally model what you need to do.

We focus here on a particular kind of intractability: whether or not the problem has any algorithm whose worst-case running time is bounded by *any polynomial* — that is, for some constant $c > 0$, on any input x, the algorithm takes time $O(|x|^c)$. The class of problems that have such algorithms is called *Polynomial Time*, and is often denoted P (or $\mathbb{P}$). Note that this is a class of *problems*, not algorithms. P is one of many *complexity classes* studied by computer scientists.

We will focus on a particular class of algorithmically important problems, so-called "NP-complete" problems. We have *strong evidence* that these problems are not in $\mathbb{P}$ (have no polynomial-time algorithms) so, are very likely to be intractable in that sense.

Reductions. Reducing a problem that you want to solve to a different problem, one that already has a known algorithm, is an important technique in algorithms. We saw a reduction from Max Bipartite Matching to Max Flow. We also saw reduction from Max Flow to Linear Programming, and from Shortest s-t Path to Linear Programming. These enable us to use any algorithm for Linear Programming to solve Shortest s-t Path, Max Flow, and Max Bipartite Matching.

Here we use reductions for a different purpose. But first we formally define what a reduction is.

A *reduction* from a problem A (such as Max Flow) to another problem B (such as Linear Programming) is a *function* f such that, given any input x for problem A, the function produces an equivalent instance $f(x)$ of problem B. *Equivalent* means that any correct answer to $f(x)$ gives us a correct answer to x. For intuition, recall the reduction from Max Bipartite Matching to Max Flow. It converted any input $G = (U, W, E)$ for Max Bipartite Matching to a flow network (G', s, t), such that, given any max s-t flow in G', we could easily find a maximum bipartite matching in G. We give a precise formal definition of *equivalent* later.

Decision problems and languages. When studying complexity classes, for simplicity, we focus on *decision problems* — problems with yes/no answers. That is, for any input, the output is either "yes" or "no". Most computational problems can be recast as a decision problem in a meaningful way. For example, here is the decision version of Shortest s-t Path:

input: digraph $G = (V, E)$ with edge weights; $s, t \in V$; $W \geq 0$

output: "yes" if there is an s-t path of weight at most W; "no" otherwise

For any decision problem A, the *language* associated with A is the set of inputs for which the answer is "yes". For example, the language for the decision problem for Shortest s-t Path is the set

$$\{(G, s, t, W) \mid \text{there is an } s\text{-}t \text{ path in } G \text{ of weight at most } W\}.$$

To be formally precise, we have in mind here that each input is encoded in a standard way as a binary string. So, each language is, technically, a set of binary strings. But we can safely forget about this most of the time. (It is important mainly in that it gives us a robust and general notion of input size — the size of an input is the number of bits in the encoding of the input.)

If an algorithm for a decision problem says "yes" for a given input x, we say the algorithm *accepts* x. If the algorithm says "no", it *rejects* x. So, if the algorithm is correct, then, for all inputs x the algorithm accepts x if is in the language and otherwise rejects x.

Reductions between decision problems. Above we defined a reduction from problem A to problem B as a function f that, given any instance x of A, produced an *equivalent* instance $f(x)$ of B. Intuitively, equivalent means that any solution to $f(x)$ "easily" gives us a solution to x, but we didn't give a formal definition. Here is the formal definition in the case of decision problems. A correct reduction from a decision problem A to a decision problem B is a function f mapping each input x for A to a corresponding input $f(x)$ for B, where f is such that

> For every possible input x to A:
> the answer to x is "yes" if and only if the answer to $f(x)$ (as an instance of B) is "yes".

That is, the answer to $f(x)$ tells us the answer to x. Using the convention that we identify A and B with their languages, we can express this as follows. By definition, the function f is a reduction from A to B if

$$(\forall x) \;\; x \in A \iff f(x) \in B,$$

where x ranges over all possible inputs for A.

Polynomial-time reductions. Here we focus on *polynomial-time* reductions — reductions that can be computed in polynomial time. That is, a reduction f is a polynomial-time reduction if there is a polynomial-time algorithm that, given any input x, outputs $f(x)$ in time polynomial in the length of x. We write

$$A \leq_P B$$

to mean that there is a polynomial-time reduction from A to B. Informally, it means that A is "no harder" than B. (To be more precise, A is at most "polynomially harder" than B.) Formally, we can say the following:

Lemma 1. *Let A and B be decision problems such that A reduces in polynomial time to B. If B has a polynomial-time algorithm, then A has a polynomial-time algorithm. That is, if $A \leq_P B$ and $B \in P$, then $A \in P$.*

Proof. Assume f is a poly-time reduction from A to B, and that B has a poly-time algorithm. Fix c and c' such that f is computable in time $O(n^c)$ and B has an algorithm running in time $O(n^{c'})$. Consider the following algorithm for A:

> algorithm for A (on input x):
> 1. compute $y = f(x)$, in time $O(|x|^c)$
> 2. run the algorithm for B on input y, taking time $O(|y|^{c'})$
> 3. if the algorithm for B accepts y, then accept x, otherwise reject x

This algorithm for A is correct by the following reasoning:

The algorithm for A accepts x

$\iff$ the algorithm for B accepts $f(x)$ *(by inspection of the algorithm for A)*

$\iff$ $f(x)$ is in B *(as the algorithm for B is correct)*

$\iff$ x is in A. *(because f is a poly-time reduction from A to B)*

So the algorithm is correct.

The algorithm takes polynomial time, $O(|x|^{c\,c'})$, by the following reasoning. Step 1, computing $y = f(x)$, takes time $O(|x|^c)$ because $f(x)$ is computable in time $O(|x|^c)$. Also, this means that y has size $O(|x|^c)$. So Step 2, running the algorithm for B on y, takes time $O(|y|^{c'}) = O((|x|^c)^{c'}) = O(|x|^{c\,c'})$. □

One more formal observation: poly-time reductions compose, so the relation $\leq_P$ is transitive:

Lemma 2. *Let A, B, and C be decision problems. If A reduces in polynomial time to B, and B reduces in polynomial time to C, then A reduces in polynomial time to C.*

That is, if $A \leq_P B$ and $B \leq_P C$, then $A \leq_P C$.

Proof. Let f_1 and f_2 be the poly-time reductions from A to B, and from B to C. Consider the function f_3 defined by composing f_1 and f_2, that is, $f_3(x) = f_2(f_1(x))$. The function f_3 is a correct reduction from A to C by the following reasoning:

$x \in A$

$\iff$ $f_1(x) \in B$ *(because f_1 reduces A to B)*

$\iff$ $f_2(f_1(x)) \in C$ *(because f_2 reduces B to C)*

$\iff$ $f_3(x) \in C$ *(because $f_3(x) = f_2(f_1(x))$)*

So the reduction f_3 is correct.

For the running time, fix c_1 and c_2 such that f_1 is computable in time $O(n^{c_1})$ and f_2 is computable in time $O(n^{c_2})$. The following algorithm computes f_3 in polynomial time, $O(n^{c_1 c_2})$:

algorithm to compute $f_3(x)$:
1. compute $y = f_1(x)$, in time $O(|x|^{c_1})$
2. compute and return $f_2(y)$, taking time $O(|y|^{c_2})$

Clearly the algorithm computes $f_3(x)$. To bound the run time, reason as in the previous proof: Step 1 takes time $O(|x|^{c_1})$, so y has size $|y| = O(|x|^{c_1})$, so Step 2 takes time $O(|y|^{c_2}) = O(|x|^{c_1 c_2})$. □

Clique, Independent Set, and Vertex Cover. Given an undirected graph $G = (V, E)$, a *clique* is a subset $K \subseteq V$ of the vertices such that every pair of vertices in C has an edge in E. An *independent set* is a subset $I \subseteq V$ of the vertices such that no pair of vertices in I has an edge in E. A *vertex cover* is a subset $C \subseteq V$ of the vertices such that every edge in E has an endpoint in C. Here are the three corresponding decision problems.

For each, the input is $G = (V, E)$ and an integer k.

Clique — does G have a clique of size k or more?

Independent Set — does G have an independent set of size k or more?

Vertex Cover — does G have a vertex cover of size k or less?

Each of these three problems reduces in polynomial time to each of the others.

Reducing Clique to Independent Set. Here is a reduction f_1 of CLIQUE to INDEPENDENT SET. Given an instance $(G = (V, E), k)$ of CLIQUE, define $f_1(G, k)$ to be the instance $(\overline{G}, k)$ of INDEPENDENT SET, where $\overline{G} = (V, \overline{E})$ has the same vertex set of G, and has exactly those edges that G does not have.

To verify correctness, observe that f_1 can be computed in time polynomial in the size of its input, and that, for all instances (G, k) of CLIQUE, the answer to (G, k) is "yes" (G has a clique of size at least k) if and only if the answer to the instance $f_1(G, k)$ of INDEPENDENT SET is "yes" ($\overline{G}$ has an independent set of size at least k). This is because a set S of vertices is a clique in G iff S is an independent set in $\overline{G}$.

Reducing Independent Set to Clique. The function f_1 also happens to be a poly-time reduction in the other direction: from INDEPENDENT SET to CLIQUE.

Reducing Independent Set to Vertex Cover. Here is a reduction f_2 of INDEPENDENT SET to VERTEX COVER. Given an instance $(G = (V, E), k)$ of INDEPENDENT SET, define $f_2(G, k)$ to be the instance (G, k') of VERTEX COVER, where $k' = |V| - k$. To verify correctness, observe that f_2 can be computed in time polynomial in the size of its input, and that, for all instances (G, k) of INDEPENDENT SET, the answer to (G, k) is "yes" (G has an independent set of size k) if and only if the answer to the instance $f_2(G, k)$ of VERTEX COVER is "yes" (G has a vertex cover of size at most $k' = |V| - k$). This is because a set S of vertices is an independent set in G iff the complement $V \setminus S$ of S is a vertex cover in G.

Reducing Vertex Cover to Independent Set. The function f_2 also happens to be a poly-time reduction in the other direction: from VERTEX COVER to INDEPENDENT SET.

Reducing Clique to Vertex Cover. And, by Lemma 2 (reductions compose), CLIQUE reduces to VERTEX COVER via the reduction $(G, k) \mapsto f_2(f_1(G, k))$.

Thus, we see that all three of these problems are equivalent, in the sense that if any one of them has a poly-time algorithm, then (via the reductions described above) each of them will have a poly-time algorithm.

Hamiltonian Path and 3-CNF-SAT It would be reasonable to conclude at this point that CLIQUE, INDEPENDENT SET, and VERTEX COVER are, in some sense, just different names for the same problem, and that is why we can reduce each of these problem to the other. But the reason is deeper than that. Next we'll see reductions between problems that, on the surface at least, are not at all similar.

HAMILTONIAN PATH — Given a directed graph $G = (V, E)$, and source and sink vertices s, t, does G have a simple path from s to t that visits each vertex exactly once?

3-CNF-SAT — Given a Boolean formula ϕ in 3-CNF form (an "and" of clauses, each of which is an "or" of at most three literals, each of which is a variable or its negation), is the formula *satisfiable* (that is, can one assign values to the variables to make the formula true).

For example, $\phi = (x \vee \overline{y} \vee z) \wedge (y \vee z) \wedge (\overline{y} \vee \overline{z})$ is satisfiable: take $x = T$, $y = T$, $z = F$.

We remark that 3-CNF-SAT is a very "expressive" problem — computers are built from logic gates, so it is not too surprising that Boolean formulae can model computation. See Lemma 34.6 of the text (CLRS) and surrounding discussion for more details.

Reducing 3-CNF-SAT to Independent Set

Lemma 3. 3-CNF-SAT $\leq_p$ Independent Set.

Proof. Here is the reduction. Let ϕ be any instance of 3-CNF-SAT. Construct the following instance $(G = (V, E), k)$ of Independent Set. For each variable x in ϕ, introduce a "variable gadget" consisting of two new "literal" vertices, labeled x and $\overline{x}$, with an edge between them.

(So far, if there are h variables in ϕ, then there will be 2^h independent sets of size h. Each independent set takes exactly one vertex from each variable gadget, and we have in mind that these 2^h independent sets correspond to the 2^h assignments true/false assignments to the variables of ϕ in the natural way.)

Next, for each clause in ϕ, introduce a "clause gadget" consisting of one new vertex for each literal in the clause, with edges between each pair of vertices within each clause gadget. For example, for the clause $x \vee \overline{y} \vee z$, the clause gadget would consist of three new variables, labeled, respectively, x, $\overline{y}$, and z, connected by three edges.

(So far, if there are h variables and ℓ clauses in ϕ, each independent set of size $h+\ell$ must contain exactly one vertex from each gadget.)

Finally, for every vertex v that occurs in each clause gadget, introduce an edge between that vertex and the "opposite" vertex in the variable gadget. For example, if v is labeled with the variable x, then introduce an edge between v and the vertex labeled $\overline{x}$ in the variable gadget for x. Or, if v is labeled with $\overline{x}$, then introduce an edge between v and the vertex labeled x in the variable gadget for x.

This defines the graph G. Define the threshold k (the desired independent-set size) to be $h + \ell$ (the number of variables plus clauses, that is, the number of gadgets). This defines the reduction: $f(\phi) = (G, k)$, where G and k are defined from ϕ as described above.

Clearly the reduction can be computed in polynomial time. That is, given ϕ, one can construct G and k in time polynomial in $|\phi|$.

It remains to verify that the reduction f is correct. That is, we need to show

$$(\forall \phi) \quad \phi \text{ is satisfiable} \iff (G, k) \text{ has an independent set of size } k, \text{ where } (G, k) = f(\phi).$$

Above ϕ ranges over all instances of 3-CNF-SAT.

Consider an arbitrary instance ϕ of 3-CNF-SAT.

Let h be the number of variables, let ℓ be the number of clauses. Let $(G, k) = f(\phi)$, so that $k = h + \ell$.

We need to show that ϕ is satisfiable if and only if G has an independent set of size at least k.

Only if ($\Longrightarrow$). Suppose that ϕ is satisfiable. We need to show that in this case G has an independent set I of size at least k. Fix any assignment A of the variables of ϕ that satisfies ϕ.

We will describe how to construct I from A. Make I contain one vertex from each gadget of G, as follows. For each variable x in ϕ, if A assigns x the value "true", then make I contain the vertex labeled x from the variable gadget for x in G. Otherwise (A assigns x the value "false"), make I contain the vertex labeled $\overline{x}$ from the variable gadget. For each clause gadget in ϕ, make I contain

the vertex from the clause gadget that corresponds to some literal in the clause that A makes true (there must be at least one such literal, since A satisfies the clause).

Clearly I has size $k = h + \ell$. Also, I is an independent set, because it takes only one vertex from each gadget, and (because of the way G is constructed) never takes a clause vertex labeled x and a variable vertex labeled $\overline{x}$, or vice versa.

If ($\Longleftarrow$). Suppose that G has an independent set of size at least $k = h + \ell$. We need to show that in this case ϕ has some satisfying assignment A. Let I be any independent set of size k. Clearly I has exactly one vertex per gadget in G (it can't have more than one in any gadget, as each gadget is a clique, so it has to have one in each gadget in order to have size k).

Define assignment A from I as follows. For each variable x, consider the variable gadget for x. If I takes the vertex labeled x, then make A assign x the value "true". Otherwise (I takes the vertex labeled $\overline{x}$) make A assign x the value "false". This defines the assignment. To see that it is a satisfying assignment, consider any clause of ϕ. I contains some vertex v from the clause gadget, where v is labeled with a literal in the clause — either a variable x or its negation $\overline{x}$. Consider the case when v is labeled with a variable x (the other case is similar). There is an edge in G between v and the vertex (call it v') labeled $\overline{x}$ in x's variable gadget. So v' cannot be in the independent set I. So the vertex labeled x in x's variable gadget has to be in the independent set I. So the assignment assigns x the value "true". So the clause is satisfied. □

This completes the material covered in the first lecture. More lecture notes on NP will be available for later lecture(s).

External resources on P, NP, and NP-completeness

- CLRS Chapter 34. Dasgupta et al. Chapter 8. Kleinberg & Tardos Chapter 8.
- K & T Chapter 8 slides

  https://www.cs.princeton.edu/~wayne/kleinberg-tardos/pdf/08IntractabilityI.pdf
- Erickson's Lecture 30:

  http://jeffe.cs.illinois.edu/teaching/algorithms/notes/30-nphard.pdf
- MIT lecture videos
  - https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-006-introduction-to-algorithms-fall-2011/lecture-videos/lecture-23-computational-complexity/