Lecture 9

- Non-determinism.
- Computational complexity.
- The class **NP**.
- Some complexity classes other than **NP**.

We now introduce and discuss the notion of *non-determinism*, as a prelude to our discussions on computational complexity, and in particular, the complexity class **NP**.

To characterize non-determinism, we first introduce the notion of the state of an algorithm. The state of an algorithm is a snapshot of the values in all the storage it uses at a given moment. For example, consider the following algorithm to carry out a membership check in an array.

```
IsIn(A[1,...,n], i)
    ret ← false
    foreach j from 1 to n do
        if A[j] = i then ret ← true
    return ret
```

A state in the above algorithm is characterized by the values of j and *ret*. Thus, the algorithm, while running, can be thought of as being in one of $2n$ states — j takes on a value from $\{1, \ldots, n\}$ only, and *ret* takes on a value from $\{\mathsf{true}, \mathsf{false}\}$ only.

Non-determinism is the property that an algorithm is allowed to be in several states simultaneously. The above algorithm, IsIn is deterministic, because

at any given moment, it is in exactly one state, i.e., $\langle j, ret \rangle$ has exactly one value. We can choose to expand our model of computation to allow for non-determinism, thereby yielding so-called non-deterministic algorithms. We will soon see examples of these. Whether a non-deterministic algorithm can be realized in practice, and the manner in which it could be, is an open question in computing. In our discussions, we simply adopt it as a model of computation, even if that is perceived as a thought experiment only.

But first, to further illustrate non-determinism, we consider somewhat simple algorithms called finite-state automata. A Deterministic Finite-state Automaton (DFA) is a 5-tuple, $\langle Q, \Sigma, \delta, q_0, F \rangle$, where:

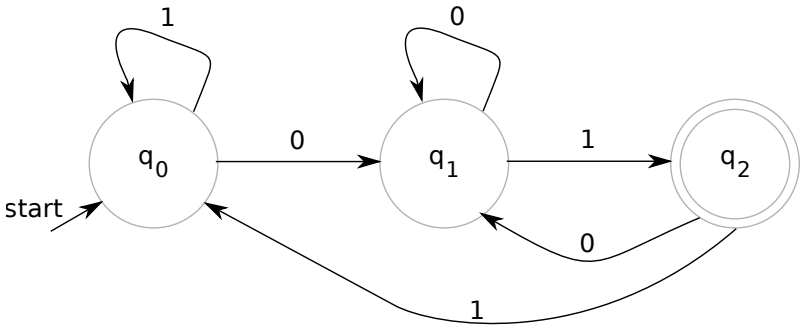
- Q is a set of states.
- Σ is a set of alphabet symbols, or simply, an alphabet.
- δ is the transition function; $\delta \colon Q \times \Sigma \to Q$.
- $q_0 \in Q$ is the start-, or initial-state.
- $F \subseteq Q$ is a set of final, or accepting states.

A sequence of symbols from an alphabet, Σ, is called a string. Given a DFA, we say that it accepts a string s if and only if starting at q_0, and following the transitions for every alphabet symbol in s, in order, leaves us in a state in F, i.e., an accepting state.

For example, consider the binary alphabet, $\{0, 1\}$, and suppose we want a DFA that accepts all binary strings that end in 01. Following is a DFA that does that. We adopt $Q = \{q_0, q_1, q_2\}$, with q_0 as the start-state, and q_2 as the only accepting-state. Our transition function, δ, is:

$$\begin{array}{lcllcl} \delta(q_0, 0) & = & q_1 & \delta(q_0, 1) & = & q_0 \\ \delta(q_1, 0) & = & q_1 & \delta(q_1, 1) & = & q_2 \\ \delta(q_2, 0) & = & q_1 & \delta(q_2, 1) & = & q_0 \end{array}$$

It is customary to perceive a DFA as a directed graph, in which the states, Q, are the vertices, the transitions are the edges, the start-state, q_0 is designated with a "start $\longrightarrow$," and each state in F is designated using a double circle, "◎." The above DFA can be visualized in this manner as shown below.



Note that such a DFA can be perceived as a kind of simple algorithm. For the example, above, we can write an algorithm in our syntax for pseudo-code as follows.

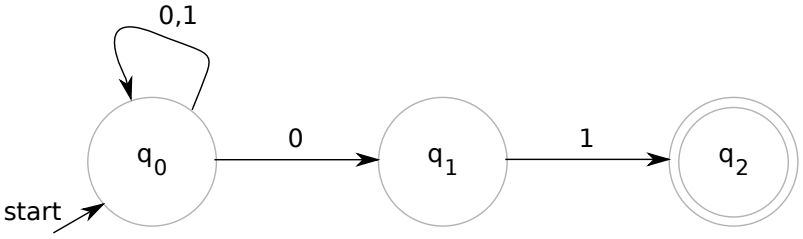
```
EndsWithZeroOne(b)
    Perceive the input b as a bit-string b_0 b_1 ... b_{n-1}
    state ← 0
    foreach i from 0 to n − 1 do
        if state = 0 and b_i = 0 then  state ← 1
        if state = 1 and b_i = 1 then  state ← 2
        if state = 2 and b_i = 0 then  state ← 1
        if state = 2 and b_i = 1 then  state ← 0
    if state = 2 then return true
    else return false
```

A Non-deterministic Finite-state Automaton (NFA) allows for the automaton to be in several states simultaneously. It is specified as a 5-tuple $\langle Q, \Sigma, \delta, q_0, F \rangle$, where four of the components, Q, Σ, q_0 and F, are exactly like in a DFA. The transition function, δ, maps a state and an alphabet symbol to a set of states, rather than a single state. That is, if 2^Q represents the powerset (set of all subsets) of Q, then $\delta \colon Q \times \Sigma \to 2^Q$. We can perceive a transition as effecting several transitions simultaneously: one to each of the states in the member of 2^Q to which we map.

Another important difference between an NFA and a DFA is the condition under which we deem that a string is accepted by an NFA.

> *We say that a string is accepted by an NFA if and only if there exists a sequence of transitions such that when we exhaust the alphabet symbols in the string, we are left in an accepting-state.*

An NFA for the above example, i.e., all binary strings that end in 01, is the following.



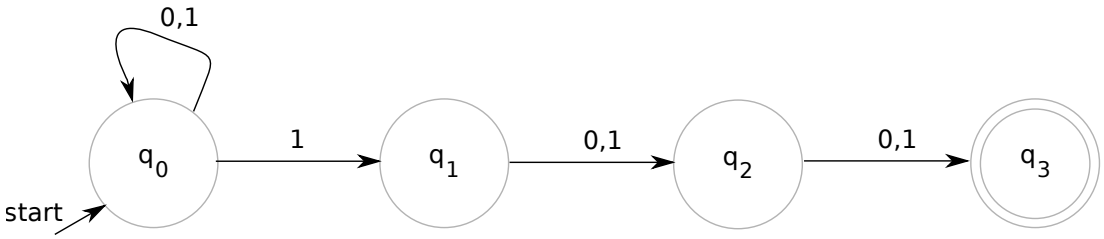
When the above NFA is in state q_0, if it sees a 0 in the input string, it *non-deterministically transitions* to state q_0 or q_1. We use the phrase "non-deterministically transitions to ..." to pick one of the possible next states to refer to. We observe that for every string that ends in 01, there is a sequence of such transitions that leaves us in the accepting state, q_2. For example, on input 001001, we stay in q_0 until we see the last 0, on which we transition to q_1, and then to q_2 on the final 1.

It is important to observe also that on any input string that does not end in 01, there exists no sequence of transitions that leads us to the accepting state, q_2. In this regard, it is important to point out that our above specification for the transition function, δ, allows a transition to the null set of states, which is a subset of Q. We can think of this as a deterministic transition to a "junk" state. For example, in our above NFA, we show no transition out of state q_2. Presumably this means that if we see any input symbol when we are in that state, we immediately transition to "junk," and stay in "junk" forever.

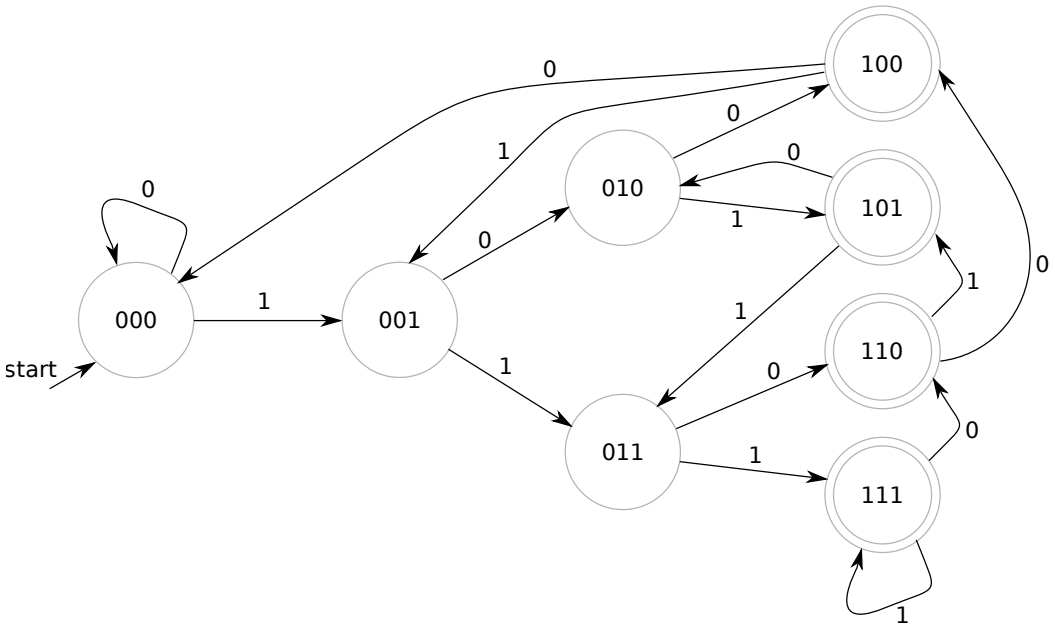
Thus, on input, for example, 0001010, we may be left in state q_0, q_1 or "junk," but not in q_2.

As another example, consider the following. We want an NFA and a DFA for the alphabet $\Sigma = \{0, 1\}$ such that they accept only strings of length at least three, in which the third-from-last symbol is a 1. For example, 001101 should be accepted, but not 0011011.

An NFA for this language is the following. The only non-deterministic transition is from q_0 to q_1 on a 1.



We show a DFA for the language below.



We label each state in the DFA more meaningfully than before. Each state remembers the past three bits we have seen. Given a state $b_0\, b_1\, b_2$, where each $b_i \in \{0, 1\}$, on input symbol $b_3 \in \{0, 1\}$, we transition to the state $b_1\, b_2\, b_3$.

We can prove that a DFA for this language requires eight states. More generally, we can prove that the following language requires only $\Theta(n)$ states for an NFA, but requires $\Theta(2^n)$ states for a DFA. The language is: every binary string whose n^{th}-from-last bit is 1. This shows the power of non-determinism in such simple algorithms. For more general purpose algorithms, however, whether non-determinism adds any power is an open question, as

we discuss further below.

Decision problems Before we discuss more general purpose problems in the context of non-deterministic algorithms, we recall a particular class of problems in which we have interest, decision problems. Recall that we characterized a "problem" as a function, and "solving a problem" to mean "computing a function." A decision problem corresponds to a function whose codomain is $\{0, 1\}$, or $\{\mathsf{true}, \mathsf{false}\}$.

For example, following is a decision problem, which we can call SORTED. Input: an array of n integers. Output: true if the array is sorted non-decreasing, and false otherwise. We can devise a polynomial- (linear-) time algorithm for this problem.

Another example is the following, which we call SHORTSIMPLEPATH. Input: (i) a connected, undirected graph, $G = \langle V, E \rangle$, (ii) distinct $a, b \in V$, and, (iii) an integer $k \in \{1, \ldots, |V| - 1\}$. Output: true if there exists a simple path $a \rightsquigarrow b$ in G of $\leq k$ edges, and false otherwise.

A deterministic polynomial-time algorithm for SHORTSIMPLEPATH is Breadth First Search (BFS). We simply run BFS with a as the source vertex, and up on its termination, compare the input k with $b.d$ that is computed by BFS. Consider, as an alternative, the following non-deterministic algorithm for SHORTSIMPLEPATH.

```
NDSHORTSIMPLEPATH(G = ⟨V, E⟩, a, b, k)
1 c ← a
2 foreach i from 1 to k do
3     Non-deterministically pick a neighbour, n, of c
4     if n = b then return true
5     else c ← n
6 return false
```

Similar to an NFA, the above algorithm meets the following correctness property:

> *If the input instance is true, then there exists a sequence of non-deterministic choices that causes the algorithm to return* true. *If*

the input instance is false, then the algorithm is guaranteed to return false.

For a non-deterministic algorithm, the above is exactly the correctness property we seek. From the standpoint of efficiency, we assume that every non-deterministic choice has the same time-efficiency of its counterpart deterministic choice. For example, given a graph encoded as an adjacency list, and a vertex c in the graph, we know that a neighbour of c can be chosen deterministically in time $\Theta(1)$. In NDShortSimplePath, we assume that a non-deterministic choice of a neighbour can be made in time $\Theta(1)$ as well.

Thus, the above algorithm is linear-time. Its space-efficiency is $\Theta(\log n)$, where n is the size of the input. The reason is that the only auxiliary space it needs is for the variable c, which is constant in $|V|$, and i, which is a counter, and therefore requires $\Theta(\log |V|)$ space only. Thus, the non-deterministic algorithm is more space-efficient than BFS; BFS requires space linear in the size of the input, in the worst-case, which is the space we need for the queue that is used by BFS.

Yet another example is the following problem, which we call LongSimplePath. Input: (i) a connected, undirected graph, $G = \langle V, E \rangle$, (ii) distinct $a, b \in V$, and, (iii) an integer $k \in \{1, \ldots, |V| - 1\}$. Output: true if there exists a simple path $a \rightsquigarrow b$ in G of $\geq k$ edges, and false otherwise. Thus, LongSimplePath is very similar to ShortSimplePath, but seeks a path of maximum length. Following is a non-deterministic algorithm for LongSimplePath.

```
NDLongSimplePath(G = ⟨V, E⟩, a, b, k)
1 c ← a, S ← {c}
2 foreach i from 1 to |V| − 1 do
3     Non-deterministically pick a neighbour, n, of c from V \ S
4     if n = b and i ≥ k then return true
5     else
6         c ← n
7         S ← S ∪ {n}
8 return false
```

The above algorithm runs in time $O(|V|^2 + |E|)$ — the **foreach** loop of Line (2) iterates $\Theta(|V|)$ times in the worst-case, and computing $V \setminus S$ from which we make a non-deterministic choice takes $O(|V|)$ time. The "$+|E|$" is from

an aggregate analysis, that in Line (3), over the entire run of the algorithm, we visit every edge a constant number of times, in the worst-case.

Thus, the algorithm is certainly polynomial-time. How about its space-efficiency? The auxiliary space is for: (i) c, (ii) i, and, (iii) S. (i) is constant in the size of G, (ii) is $\Theta(\log |V|)$, and (iii) takes space $\Theta(|V|)$. Thus, the space-efficiency can be characterized as $\Theta(|V|)$, i.e., linear in the size of the input.

Why consider non-determinism? A natural question one may ask is: why do we consider the non-deterministic model for computation when it appears that no such computer is likely to exist in practice?

An answer lies in features decision problems exhibit. Not all decision problems are, or appear to be, alike. Consider, for example, the problem of solving a Sudoku puzzle. See below for an example.

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

To the left is a puzzle, and to the right is a solution to the puzzle. (Acknowledgement: the above picture is from `https://blogs.unimelb.edu.au/`.) A puzzle comprises a 9×9 board, with some of the slots filled in and the others left blank. A correct solution: (i) is a completely filled board with each slot containing a digit between 1 and 9, (ii) each row and column containing exactly one of each of those digits, (iii) each successive 3×3 square left-to-right, top-to-bottom containing each digit exactly once, and (iv) the slots which were already filled in the input each has exactly that same value it had on

input.

Not every puzzle has a solution; for example, a puzzle which contains the same digit, say, 1, in the same row in the input has no solution. This suggests the following decision problem: given a Sudoku puzzle, does it have a solution?

This decision problem has the following interesting feature. Suppose, given a puzzle, the answer for that puzzle is, 'yes.' Then, we can prove that the answer is indeed 'yes' by providing as evidence a filled-in puzzle which meets all the constraints that are required in a solution. And such an evidence exists only if the answer for that puzzle is indeed 'yes.' The size of the encoding of such an evidence is efficient, in this case linear, in the size of the input puzzle. Furthermore, this proof or evidence can be checked efficiently, in this case in linear-time.

This feature, that every true instance of a decision problem possesses an efficiently- (polynomial-) sized evidence which can be checked efficiently (in polynomial-time) corresponds exactly with the non-deterministic model of computation. In the case of Sudoku, a non-deterministic algorithm simply picks a value for each slot non-deterministically from the set $\{1, 2, \ldots, 9\}$. At the end of making all the choices, the algorithm checks whether all the constraints have been met. If they have, it outputs true, otherwise it outputs false. This algorithm is correct: if there is a solution, there is a sequence of non-deterministic choices which will cause us to correcty return true; if there is no solution, we are guaranteed to return false.

Another, related feature, of decision problems for which there exists a non-deterministic polynomial time algorithm is the following. Suppose we are given an *oracle* for the decision problem. An oracle is a tamper-proof black-box for the problem that given an input, in $\Theta(1)$-time outputs true or false correctly for that input. Given such an oracle, we are able to find a solution, in this case a correctly filled-in solution for the Sudoku puzzle, in polynomial-time. Thus, within a polynomial time factor, the version in which we seek a solution given a puzzle is no more computationally difficult than the decision version which is associated with true or false only.

Following is our algorithm given the oracle. We first query the oracle with the original puzzle. If it outputs false, we know that the puzzle has no solution,

and can stop. Otherwise, we know that the puzzle has a solution and proceed to find one. We guess some value, say, 1, for one of the unfilled slots. We query the oracle as to whether this puzzle has a solution. If it says true, then we proceed with 1 in that slot, and guess a value for one of the other slots. If it says false, then we know that no solution exists in which that value 1 is in that slots. We try with 2 in that slot. We know that one of the nine values must work. Thus, we expect to invoke the oracle at worst 9 times for each unfilled slot.

Note: an important nuance that we elide in the above discussions, which we explicate now, is that we are presumably talking about a generalization of Sudoku, in which each puzzle comprises an $n^2 \times n^2$ board, where $n \in \mathbb{N}$ is unbounded, and we have an alphabet of size n^2 with which to fill the board. If we consider 9×9 puzzles only, then, given that 9 is a constant, we have a constant-time brute-force algorithm for the problem.

There are a number of decision problems that exhibit the above feature, i.e., admit a polynomial-time non-deterministic algorithm, or equivalently, every true instance possesses an efficiently-sized evidence that can be verified efficiently. Examples:

SortedArray: given an array of n items, is it sorted?

VertexCover: given an undirected graph G and an integer k, does G has a vertex cover of size k?

HamPath: given an undirected graph G, does it have a simple path of all its vertices? (Such a path is called a "Hamiltonian path.")

HamCycle: given an undirected graph G, does it have a simple cycle of all its vertices? (Such a cycle is called a "Hamiltonian cycle.")

ShortSimplePath: given an undirected graph G, two vertices in it a, b, and an integer k, does there exist a simple path $a \rightsquigarrow b$ of at most k edges?

LongSimplePath: given an undirected graph G, two vertices in it a, b, and an integer k, does there exist a simple path $a \rightsquigarrow b$ of at least k edges?

SAT: SAT stands for "Boolean satisfiability." Given n propositional variables $p_1, p_2, \ldots, p_n$ and a formula in them with only the operators $\wedge, \vee, \neg$ and parenthesis ( ), does there exist an assignment of true or false to each of

$p_1, \ldots, p_n$ that causes the formula to evaluate to true?

For example, the formula

$$((p_1 \wedge \neg p_2) \vee \neg p_1) \wedge p_3 \wedge \neg(p_4 \wedge p_2)$$

is satisfiable. A satisfying assignment is

$$p_1 = p_4 = 0, p_2 = p_3 = 1$$

The formula $p_1 \wedge p_2 \wedge \neg p_1$ is not satisfiable.

CIRCUIT-SAT: given a *combinational acyclic boolean circuit* of only AND, OR and NOT gates, n inputs wires and one output wire, does there exist an assignment of 0 or 1 to each of the input wires that causes the output wire to be 1? Following are two examples from CLRS.

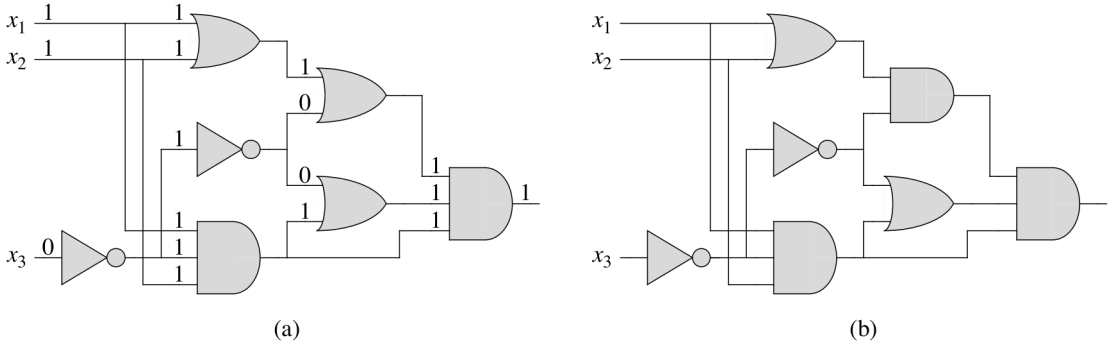


Figure 34.8 Two instances of the circuit-satisfiability problem. **(a)** The assignment $\langle x_1 = 1, x_2 = 1, x_3 = 0 \rangle$ to the inputs of this circuit causes the output of the circuit to be 1. The circuit is therefore satisfiable. **(b)** No assignment to the inputs of this circuit can cause the output of the circuit to be 1. The circuit is therefore unsatisfiable.

SUBSETSUM: given a set of integers, does some subset comprise members that add up to 0?

SETCOVER: given a set $\mathcal{G}$, n subsets of $\mathcal{G}$, $S_1, S_2, \ldots, S_n$, and an integer k, do there exist k of those subsets whose union is $\mathcal{G}$?

Other kinds of decision problems There certainly seem to be decision problems that do not have the same attributes as the ones we dis-

cuss above. Consider the following example. A digital Integrated Circuit (IC), can be seen as an efficiently-sized encoding of a boolean function, $f : \{0,1\}^n \to \{0,1\}^m$. Consider the following decision problem, which we can call IC-EQUIVALENT: given two digital ICs, do they realize the same boolean function?

Suppose the answer is true for a given pair of ICs. Then, there does not appear to be a polynomially-sized evidence. However, if the answer is false, there is an efficiently sized evidence: it is a bit-string of size n on which the two ICs disagree. Thus, the complementary decision problem, given two ICs, do they realize different functions, does seem to have an efficiently sized evidence which can be verified efficiently.

As another example, consider the following decision problem, which we can call NUMSIMPLEPATHS. Given an undirected graph G, two vertices in it, a, b, and an integer k, do there exist k distinct simple paths $a \rightsquigarrow b$? Whether the answer is true or false for a particular input, there does not appear to be an efficiently-sized evidence. The reason is that k may be $\Omega(2^n)$, where n is the number of vertices in the graph.

However, there exists an algorithm for the above problem. An algorithm is one that systematically enumerates every simple path $a \rightsquigarrow b$, and keeps a count. If the count reaches k, then it outputs true; otherwise, it outputs false. Each path is at worst linear in the size of the graph, and once we have enumerated one path, we can reuse the space we used to enumerate the next path. Thus, there is an algorithm whose space-efficiency is at worst linear in the size of the input.

Complexity class

A *computational complexity class*, or simply, complexity class, is a set of decision problems. Following are some complexity classes. We say that a decision problem is *decided* by an algorithm if there exists an algorithm for it that terminates and is correct. Of course our notion of "correct" is different, depending on whether we refer to a deterministic or non-deterministic algorithm. When we say "algorithm" without qualification, we mean a deterministic algorithm. Otherwise, we clearly say "non-deterministic algorithm." The mnemonic n is the size of the input.

L	The set of decision problems for each of which there exists an algorithm whose space-efficiency is $O(\log n)$.
NL	The set of decision problems for each of which there exists a non-deterministic algorithm whose space-efficiency is $O(\log n)$.
P	The set of decision problems for each of which there exists a constant k and an algorithm whose time-efficiency is $O(n^k)$.
NP	The set of decision problems for each of which there exists a constant k and a non-deterministic algorithm whose time-efficiency is $O(n^k)$.
PSPACE	The set of decision problems for each of which there exists a constant k and an algorithm whose space-efficiency is $O(n^k)$.
NPSPACE	The set of decision problems for each of which there exists a constant k and a non-deterministic algorithm whose space-efficiency is $O(n^k)$.
EXP	The set of decision problems for each of which there exists a constant k and an algorithm whose time-efficiency is $O(2^{n^k})$.
NEXP	The set of decision problems for each of which there exists a constant k and a non-deterministic algorithm whose time-efficiency is $O(2^{n^k})$.
DECIDABLE	The set of decision problems for each of which there exists an algorithm.
UNDECIDABLE	The set of decision problems for each of which there exists no algorithm.

There are many more complexity classes, indeed, infinitely many. Our focus is a few of these. We first make some observations about the above classes.

Claim 47. ***UNDECIDABLE*** $\neq \emptyset$.

The above claim is exactly what we establish under "(Non-)existence of algorithms" in Lecture 2 of these notes. We now relate the classes **DECIDABLE** and **UNDECIDABLE**.

Claim 48. ***UNDECIDABLE*** $\cap$ ***DECIDABLE*** $= \emptyset$.

The above claim follows directly from the definitions of **UNDECIDABLE** and **DECIDABLE**. The relationship between the classes from above that all fall under **DECIDABLE** is expressed by the following claim.

Claim 49. ***L*** $\subseteq$ ***NL*** $\subseteq$ ***P*** $\subseteq$ ***NP*** $\subseteq$ ***PSPACE*** $\subseteq$ ***NPSPACE*** $\subseteq$ ***EXP*** $\subseteq$ ***NEXP*** $\subseteq$ ***DECIDABLE***.

The containment of a deterministic class in its counterpart non-deterministic class is easy to intuit. For example, **P** $\subseteq$ **NP**. The reason is that simply because we are allowed the power of non-determinism does not mean that we have to use it. Said differently, a deterministic algorithm is also a non-deterministic algorithm. For example, ShortSimplePath from above $\in$ **P**, and therefore ShortSimplePath $\in$ **NP**.

Containment when we go between a class that is characterized by space-efficiency, to one that is characterized by time-efficiency, and vice versa, is not as straightforward.

Consider, for example, the claim **NL** $\subseteq$ **P**. Let π be a problem $\in$ **NL**. Then, there exists a non-deterministic log-space algorithm, $A_{\pi,\mathbf{L}}$ that decides π. We first observe that $A_{\pi,\mathbf{L}}$ must be polynomial-time. This is because the algorithm can be in at most one of 2^s states, where s is the auxiliary space, and given that it terminates, it must terminate when it arrives in one of these states. And because $s = O(\lg n)$, where n is the size of the input, $2^s = O(n)$. (We assume binary encoding.)

We now devise a polynomial-time deterministic algorithm $A_{\pi,\mathbf{P}}$ that decides π. If $A_{\pi,\mathbf{L}}$ uses auxiliary space s, we allocate space 2^s in $A_{\pi,\mathbf{P}}$.

For each non-deterministic choice $A_{\pi,\mathbf{L}}$ makes, we record in our auxiliary space in $A_{\pi,\mathbf{P}}$ all the possible choices. We then explore each choice just as $A_{\pi,\mathbf{L}}$ would explore its non-deterministic choice. Thus, if $A_{\pi,\mathbf{L}}$ runs in time $T(n)$, where n is the size of the input, $A_{\pi,\mathbf{P}}$ runs in time $T(n) \times 2^s$. As $T(n)$ is polynomial in n, and s is logarithmic in n, $A_{\pi,\mathbf{P}}$ is polynomial-time.

P vs. NP As the above claim expresses, $\mathbf{P} \subseteq \mathbf{NP}$. Whether that containment is strict, i.e., whether $\mathbf{P} \subset \mathbf{NP}$, is arguably the most famous open problem in the world. There are other open problems as well, related to these and other complexity classes.

For example, we know (can prove) that $\mathbf{P} \subset \mathbf{EXP}$. We know also, as the claim says, that $\mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{PSPACE} \subseteq \mathbf{EXP}$. Thus, at least one of those three containments must be strict. It is customarily assumed that all three of those containments are strict, i.e., $\mathbf{P} \subset \mathbf{NP} \subset \mathbf{PSPACE} \subset \mathbf{EXP}$. This conjecture suggests that equating non-deterministic polynomial-time with exponential-time is a mistake.

co-$\mathcal{C}$ Given a decision problem p, we define its *complement*, call it q, as the decision problem whose input is the same as that of p, and for every input instance of p which is true, that input instance of q is false, and for every input instance of p that is false, that input instance of q is true.

For example, the complement of the decision problem, "given a Sudoku puzzle, does it have a solution?" is "given a Sudoku puzzle, does it have no solution?" Similarly, "given two digital ICs, do they realize the same boolean function?" is the complement of "given two digital ICs, do they realize different boolean functions?"

Given a complexity class $\mathcal{C}$, the complexity class denoted **co-$\mathcal{C}$** is defined as: p is in $\mathcal{C}$ if and only if the complement of p is in **co-$\mathcal{C}$**. For example, the complexity class **co-NP** is defined as the set of decision problems, each of whose complement $\in \mathbf{NP}$. So, for example, the following decision problem is in **co-NP**: given an undirected graph G and an integer k, does G not have a vertex cover of size k? Which is the same as: given an undirected graph G and an integer k, is every vertex cover of G of size $> k$?

A problem in **co-NP**, unlike one in **NP**, does not appear to have an efficiently-

sized evidence that can be verified efficiently for every true instance. Consider as an example the problem above which we can call NoVertexCover. The natural evidence that a graph G does not have a vertex cover of size k seems to be enumerate every subset of size k, and demonstrate that none is a vertex cover. But the number of subsets of size k, given G of n vertices, is $\binom{n}{k}$. And $\binom{n}{k}$ is not polynomial in n if k is, for example, a polynomial in n, e.g., $k = \sqrt{n}$.

Note: both $\mathcal{C}$ and **co-$\mathcal{C}$** are sets of decision problems. But one must be careful to observe that **co-$\mathcal{C}$** is not necessarily the set-complement of $\mathcal{C}$, i.e., it is not necessarily $\overline{\mathcal{C}}$. For example, we claim that **UNDECIDABLE** = **co-UNDECIDABLE**. This is because given $p \in$ **UNDECIDABLE**, if its complement is in **DECIDABLE**, then we can simply flip the output of the algorithm for the complement of p from 0 to 1 and from 1 to 0 to get an algorithm for p, which contradicts the assumption that no algorithm exists for p. Similarly, we observe that **P** = **co-P**.

It can also be the case that **co-$\mathcal{C}$** $\cap\, \mathcal{C} \neq \emptyset$, and yet **co-$\mathcal{C}$** $\neq \mathcal{C}$. For example, **P** $\subseteq$ **NP** $\cap$ **co-NP**. However, it seems unlikely that **NP** = **co-NP**. In particular, we observe that we cannot simply adopt the trick of flipping the output of a non-deterministic algorithm, and get a correct algorithm for the complement of the problem. To explain this point, consider NDLongSimplePath that is a correct non-deterministic polynomial-time algorithm for LongSimplePath.

```
NDLongSimplePath(G = ⟨V, E⟩, a, b, k)
 1 c ← a, S ← {c}
 2 foreach i from 1 to |V| − 1 do
 3     Non-deterministically pick a neighbour, n, of c from V \ S
 4     if n = b and i ≥ k then return true
 5     else
 6         c ← n
 7         S ← S ∪ {n}
 8 return false
```

Now consider the complement of LongSimplePath, which we can call NoLongSimplePath: given as input G, a, b, k, does there exist no simple path $a \rightsquigarrow b$ of at least k edges? Now suppose someone, call her Alice, proposes the following as a candidate non-deterministic algorithm for NoLongSim-

plePath.

```
CandidateNDNoLongSimplePath(G = ⟨V, E⟩, a, b, k)
1 c ← a, S ← {c}
2 foreach i from 1 to |V| − 1 do
3     Non-deterministically pick a neighbour, n, of c from V \ S
4     if n = b and i ≥ k then return false
5     else
6         c ← n
7         S ← S ∪ {n}
8 return true
```

Note that CandidateNDNoLongSimplePath is exactly NDLongSimplePath except that we have swapped true for false and false for true in the respective return statements. We now ask: is CandidateNDNoLongSimplePath a correct non-deterministic algorithm for NoLongSimplePath?

The answer is 'no.' The reason is: suppose we are given as input $\langle G, a, b, k\rangle$ that is a false instance of NoLongSimplePath. That is, there is indeed a simple path $a \rightsquigarrow b$ in G of length k or longer. Now, when we run CandidateNDNoLongSimplePath, suppose we make a choice in Line (3) that does not leads us along that particular path $a \rightsquigarrow b$ of length k or longer. We would erroneously return true in Line (8).

Recall that our notion of correctness for a non-deterministic algorithm is that if the input instance is false, we should guarantee that we return false. CandidateNDNoLongSimplePath does not have this property and therefore is not correct.

This should not be surprising. The question as to whether $\textbf{co-NP} \stackrel{?}{\neq} \textbf{NP}$ remains a major open problem, though perhaps not as famous as $\textbf{P} \stackrel{?}{\neq} \textbf{NP}$. The prevailing conjecture is that $\textbf{co-NP} \neq \textbf{NP}$. Another open problem is whether $\textbf{P} \stackrel{?}{\subset} \textbf{co-NP} \cap \textbf{NP}$. We know that $\textbf{P} \subseteq \textbf{co-NP} \cap \textbf{NP}$; the question is whether that containment of **P** in $\textbf{co-NP} \cap \textbf{NP}$ is strict. It is conjectured that it is indeed strict.

For example, consider the following problem, ProdTwoPrimes. Given as input a natural number, is it a product of two primes? It turns out that ProdTwoPrimes $\in \textbf{co-NP} \cap \textbf{NP}$. However, it is conjectured that

ProdTwoPrimes $\notin \mathbf{P}$.