# Notes, 9(c)

ECE 606

## Complexity classes

In the past in this course, we have considered time- and space-efficiency of algorithms. We now consider those notions in the context of problems. That is: we want to ask, given a problem, whether there exists an algorithm for it that meet a certain bar for time- and/or space-efficiency. We focus on decision problems, at least for our basic notions.

A *(computational) complexity class* is a set of decision problems. When we say "there exists an algorithm," we mean one that terminates and is correct. Recall that our notion of "correct" is different for deterministic and non-deterministic algorithms.

When we say "algorithm" without qualification, we mean a deterministic algorithm. When we mean a non-deterministic algorithm, we will explicitly say "non-deterministic algorithm." We use $n$ to represent the size of the input.

Examples of some complexity classes with mnemonics that are customarily used to represent them:

| | |
|---|---|
| **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.

We can identify properties of these classes, and sometimes relate complexity classes to one another. Examples:

**Claim 1.** $\boldsymbol{UNDECIDABLE} \neq \emptyset$.

**Claim 2.** $\boldsymbol{UNDECIDABLE} \cap \boldsymbol{DECIDABLE} = \emptyset$.

**Claim 3.** $\boldsymbol{L} \subseteq \boldsymbol{NL} \subseteq \boldsymbol{P} \subseteq \boldsymbol{NP} \subseteq \boldsymbol{PSPACE} \subseteq \boldsymbol{NPSPACE} \subseteq \boldsymbol{EXP} \subseteq \boldsymbol{NEXP} \subseteq \boldsymbol{DECIDABLE}$.

The issue of **P** vs. **NP**

This may be the most famous open mathematical problem on the planet.

As the above claim states, we know trivially that $\mathbf{P} \subseteq \mathbf{NP}$. The big question is whether that containment is strict. That is, $\mathbf{P} \stackrel{?}{\subset} \mathbf{NP}$.

It is conjectured widely that indeed $\mathbf{P} \subset \mathbf{NP}$. But we do not have a proof (yet). If indeed it is true, this would mean that there exist problems in **NP** for which no efficient (i.e., polynomial-time) algorithm exists. And we already know what such a problem is: next lecture.

There are many other open problems as well. For example, we do not know $\mathbf{NL} \stackrel{?}{\subset} \mathbf{P}$.

We do know that $\mathbf{P} \subset \mathbf{EXP}$. So, in the chain $\mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{PSPACE} \subseteq \mathbf{EXP}$, at least one of the containments must be strict. The widely held conjecture is that all those containments are strict.

**co-$\mathcal{C}$**

We now specify, for every complexity class $\mathcal{C}$, an interesting "mirror" class, which we call **co-$\mathcal{C}$**.

To do so, we first characterize what we mean by the complement of a decision problem, $p$. First recall that an instance $i_p$ of a decision problem $p$ is either true or false. For example, consider the decision problem $p$ that has as input an array of integers, and the output is to be true of that array is sorted and false if it is not. Then an instance of $p$ is a specific array, e.g., the array $[1, 2, 3, 4]$. Such an instance is either true, which in this example it is, or false.

The complement of a decision problem $p$ is a decision problem $q$ with the properties: (i) $i$ is an instance of $p$ if and only if $i$ is an instance of $q$, and, (ii) if $i$ is a true instance of $p$, then $i$ is a false instance of $q$, and if $i$ is a false instance of $p$, then $i$ is a false instance of $q$.

Of course, $q$ is the complement of $p$ if and only if $p$ is the complement of $q$.

Examples of problems that are complements of one another.

1. Input: an integer array, $A$.
   - SortedArray: true if $A$ is sorted; false if not.
   - NotSortedArray: false if $A$ is sorted; true if not.
2. Input: an undirected graph $G = \langle V, E \rangle$, and an integer $k$.
   - kVertexCover: true if $G$ has a vertex cover of size $k$; false otherwise.
   - NokVertexCover: true if no $C \subseteq V$ with $|C| = k$ is a vertex cover for $G$; false otherwise.
3. Input: an $n^2 \times n^2$ Sudoku puzzle.
   - SudokuSolvable: true if the puzzle is solvable; false otherwise.
   - SudokuNotSolvable: true if the puzzle has no solution; false otherwise.

We can now define **co-$\mathcal{C}$** for a complexity class $\mathcal{C}$: a decision problem $p \in \mathcal{C}$ if and only if its complement $q \in$ **co-$\mathcal{C}$**.

Warning: **co-$\mathcal{C}$** is not necessarily the same as the set complement, $\overline{\mathcal{C}}$. Recall that $\mathcal{C}$ is a set. So, $\overline{\mathcal{C}} = U \setminus \mathcal{C}$, where $U$ is the set of all decision problems.

**Claim 4.** *(i)* ***P*** $=$ ***co-P****. (ii)* ***UNDECIDABLE*** $=$ ***co-UNDECIDABLE****.*

*Proof.* For (i), we prove (a) $\mathbf{P} \subseteq$ **co-P**, and (b) $\mathbf{P} \supseteq$ **co-P**.

For (a) let $p \in \mathbf{P}$. We need to prove that $p \in$ **co-P**. Let $q$ be the complement of $p$. Then $q \in \mathbf{P}$, because we can take the polynomial-time algorithm for $p$, and flip its output bit to get a polynomial-time algorithm for $q$. Therefore, from the definition of **co-$\mathcal{C}$**, $p \in$ **co-P**.

For (b) let $p \in$ **co-P**. We need to prove that $p \in \mathbf{P}$. Because $p \in$ **co-P**, $p$'s complement $q \in \mathbf{P}$. But this implies that $q$'s complement $p$ is also $\in \mathbf{P}$.

The proof for (ii) is similar, except that we leverage contradiction: if some decision problem $p \notin$ **UNDECIDABLE**, then $p \in$ **DECIDABLE**, i.e., there exists an algorithm for $p$. □

What about **NP** vs. **co-NP**? We first observe that $\mathbf{NP} \cap$ **co-NP** $\neq \emptyset$. In particular, $\mathbf{P} \subseteq \mathbf{NP} \cap$ **co-NP**.

Notwithstanding, it seems unlikely that **NP** = **co-NP**. We cannot prove this — this is another major open problem. However, we point to two things below.

Consider SudokuSolvable and its complement problem SudokuNotSolvable. The former is in **NP** and the latter is in **co-NP**. The former has a feature we discussed before: every true instance is associated with a nice 'proof,' which is the solved puzzle.

Now we ask: how would be prove that a true instance of SudokuNotSolvable is indeed true? There does not appear to be a terse proof like in the case of SudokuSolvable.

As the second observation, the simple trick of invoking the non-deterministic algorithm for SudokuSolvable and then flipping the output bit does not appear to work to get a correct algorithm for SudokuNotSolvable.

SudokuSolvable($p = n^2 \times n^2$ puzzle)

1 **foreach** *unfilled slot in* $p$ **do**
2 Non-deterministically pick $i \in \{1, \ldots, n^2\}$
3 Check if constraints are satisfied correctly
4 **if** *yes* **then return** true
5 **else return** false

Now imagine as a candidate algorithm for SudokuNotSolvable, we propose exactly the above algorithm, but with a return of false in Line (4) and true in Line (5). That would not be a correct non-deterministic algorithm for SudokuNotSolvable.

The reason is: suppose the input puzzle $p$ indeed does have a solution. That is, it is a false instance of SudokuNotSolvable. There may well exist a set of non-deterministic choices that can be made in Lines (1)–(2) for which the Sudoku constraints are not satisfied. This would cause us to erroneously output true instead of false.

Recall what correctness means for a non-deterministic algorithm for a decision problem: for a true instance, there must exist a sequence/set of non-deterministic choices that causes an output of true. For an input instance that is false, we must guarantee that the output is false.

*Important note on this page: I've edited it in these two places since making the video to emphasize that the property is an "if and only if."*

## An alternative, popular definition for **NP**

We defined **NP** as the class of decision problems for each of which there exists a polynomial-time non-deterministic algorithm.

Another, equivalent definition is the following. We use $|\cdot|$ to indicate the size of a string.

> The set **NP** comprises those decision problems such that for every $p \in \mathbf{NP}$, there exists a two-input polynomial-time algorithm $V_p(\cdot, \cdot)$ such that: an instance $i$ of $p$ is true if and only if there exists a string $c$ such that (i) the size of $c$, $|c|$, is at worst polynomial in $|i|$, and, (ii) $V_p(i, c) =$ true, i.e., $V_p$ when invoked with input $\langle i, c \rangle$ outputs true.

The string $c$ is called a *certificate* or *witness*. Because it is evidence to the fact that $i$ is true. The algorithm $V_p$ is called a *verification algorithm* — it verifies that $c$ is indeed a valid certificate for $i$.

For example, given a true instance of an $n^2 \times n^2$ Sudoku puzzle, a natural certificate is the puzzle completely filled with a correct solution. We observe that such a certificate exists if and only if the puzzle is solvable. Such a certificate is linear in the size of the instance, under a natural encoding of the puzzle, which is of size $\Theta(n^4 \lg n)$. The verification algorithm, for such a certificate, checks all the constraints: (i) that the slots that were already pre-filled indeed contain the same values in the certificate, and, (ii) that every row, column and $n \times n$ square left-to-right and top-to-bottom, indeed contains every symbol exactly once. The algorithm runs in linear-time.

Why is this characterization of **NP**, as "an efficiently-sized certificate exists for every true instance which can be verified efficiently," equivalent to "efficient non-deterministic algorithm exists?"

Because our certificate is exactly a correct sequence of non-deterministic choices, and our verification algorithm is exactly the check, after we make all our non-deterministic choices, that we should output true from the non-deterministic algorithm. As we associate each non-deterministic choice with a cost of 1, the size of the certificate can be at worst polynomial in the size of the input problem instance. And the checks after the non-deterministic choices must run in polynomial-time — those checks are exactly our verification algorithm.