# Review

1. A *problem* for which we seek an algorithm is a function.

* The mathematical notion of function.” Not a function in a programming language such as C.

1. We adopted the imperative, or procedural, style to specifying algorithms.

* Imperative style: first do this, then do that, then repeat this until some condition becomes true, …
* There exist other styles, e.g., the declarative style.
* Imperative style is natural in some ways, but it can be difficult to match the procedure with a property.
* E.g., correctness proofs are not necessarily easy.

1. Customarily, we assess the efficiency of an algorithm along two axes: time and space. Both are typically characterized as functions of the size of the input.

* Characterizing the size typically requires assumptions about the manner in which inputs, and more generally strings, are encoded.

1. Rather than considering exact functions for time- and space-efficiency, we adopt asymptotics, i.e., “in the limit.”

* .
* We adopt other abstractions as well. E.g., time for 1-bit operation is constant, i.e., .

1. A class of algorithms of particular interest: polynomial-time algorithms.

* An algorithm is said to be polynomial time if there exists a constant such that the time-efficiency of the algorithm is , where is the size of the input.
* E.g., computing the product given by repeated addition, i.e., computing , is not polynomial time if encoded with base .
* But computing by “repeated doubling” is polynomial time.

1. We discussed algorithms with numbers, e.g., those for division, exponentiation, and modular arithmetic.

* Algorithms with modular arithmetic heavily leverages facts/theorems from number theory.
* E.g., Fermat’s little theorem.

1. We discussed hash tables, specifically, the problem of resolving collisions using chaining.
2. We discussed four design strategies explicitly, and one implicitly.

* Explicit: randomization, divide-n-conquer, greedy, dynamic programming.
* Implicit: incremental.
* Each strategy can be seen as intimately tied to an underlying property of the problem.
* Huge warning: not every algorithmic problem lends itself to one of those strategies. Need to study the problem carefully and perhaps write down and prove a theorem about it.

1. We discussed also reduction to Linear Programming (LP) as a strategy, even though we suspect that no polynomial time algorithm exists for Integer Linear Programming (ILP), or even ZOE.
2. We segued from ILP to CIRCUIT-SAT. And from that problem to computational complexity.
3. Computational complexity class or complexity class or class: a set of decision problems, i.e., functions whose codomain is .
4. Classes of interest that have been studied typically relate to the kind of algorithm that exists for each of the problems in the class.

* **L**: decision problems for each of which an algorithm exists whose space-efficiency is , where is the size of the input.
* **P**: decision problems for each of which a polynomial time algorithm exists.
* **PSPACE**: decision problems for each of which an algorithm whose space-efficiency is upper-bounded by a polynomial in the size of the input exists.

1. Notion of non-determinism: a possibly significant change to the underlying model of computation.

* **NL**: decision problems for each of which a non-deterministic algorithm exists whose space-efficiency is , where is the size of the input.
* **NP**: decision problems for each of which a non-deterministic polynomial-time algorithm exists.
* **NPSPACE**: decision problems for each of which a non-deterministic algorithm whose space-efficiency is upper-bounded by a polynomial in the size of the input exists.

1. When we say “algorithm” without qualification, we mean a deterministic algorithm. If we intend to say non-deterministic algorithm, then we explicitly qualify as such.
2. We know: **L** **NL** **P** **NP** **PSPACE** **NPSPACE**.
3. But we do now know about the strictness of inclusion in many cases. E.g., **L** **NL**, N**L** **P**, **P** **NP**. But we conjecture that those inclusions are indeed strict.
4. We do know, provably, that **PSPACE** **NPSPACE** – Savitch’s theorem.
5. Membership in a complexity class can be seen legitimately as an upper-bound computational hardness of a decision problem.
6. Notion of a reduction, , to compare the computational hardness of one problem to another.

* E.g., VERTEX-COVER CLIQUE
* E.g., CNF-SAT SET-COVER

1. Notion we adopted for this course: existence of a function between instances of the two problems that: (i) satisfies “if and only if” property, and (ii) is polynomial time computable.
2. Notion of -hard for a complexity class is always under some (presumably meaningful) notion of a reduction .
3. When we say **NP**-hard without specifying what reduction, we mean under the particular notion of that we discussed in this course.
4. Notion of -complete: both a lower- and upper-bound on computational hardness of a problem.

* A decision problem is -complete if it is: (i) , and (ii) -hard under a particular notion of .

1. We went over a number of examples of reductions.