Lecture 5: Introduction to Complexity Theory

1 Complexity Theory

1.1 Resource Consumption

Complexity theory, or more precisely, Computational Complexity theory, deals with the resources required during some computation to solve a given problem.

The process of computing involves the consumption of different resources like time taken to perform the computation, amount of memory used, power consumed by the system performing the computation, etc. A theory of resource consumption looks at how much resources are needed to solve any particular computational problem. In order to consider such a theory, we first have to decide what the resources are.

In this class the resources that we consider will be time and space (in terms of memory). What does it mean to measure 'time' and 'space'? One possibility is that we run algorithm A on machine I and measure the time t in seconds that the program executes for and space k in KB as the amount of memory used. However, this method has the drawback of being machine-dependent. Thus, the same algorithm run on two different machines may give us different results for time and space consumption.

To overcome this limitation, we should use an abstract model of a computer. The model we use is the *Turing machine*. Time is measured in terms of the number of the steps taken by the Turing machine, and space is measured in terms of the number of cells used. In reality, it is tedious to construct the equivalent Turing machine for a given algorithm. Instead, we perform a rough analysis for the algorithm itself, under the assumption that calculations involving fixed input sizes take constant time and space. Our goal is to have an analysis that is *technology independent*.

The performance of an algorithm needs to be measured over all inputs. We aggregate all instances of a given size together and analyze performance for such instances. For an input of size n, let t_n be the maximum time consumed and s_n be the maximu space consumed. Typically, we want to find functions t and s, such that $t(n) = t_n$ and $s(n) = s_n$. This is *worst-case analysis*. We can also do *average-case analysis*, but that is typically harder and less meaningful.

One might wonder why don't simply use benchmarks. The two major problems with using benchmarking as the basis of our performance analysis are:

- Benchmarks do not convey any information about the scalability of the problem.
- Algorithms may be optimized specifically for the set of benchmarks. We will not know whether our performance analysis holds outside that set. This is called *design for benchmarking*.

By focusing on all instances of a given length, and measuring *scalability*, we get an analysis that is *instance independent*.

Asymptotic Notations

As mentioned in the earlier section, we are interested in finding the behavior of a program without considering architectural details of a machine. We normally use asymptotic notations to compare different algorithms as stated below.

Θ notation (Tight Bound)
f = $\Theta(g(x))$ given that there exist constants c_1, c_2 and k such that $c_1.g(x) <= f(x) <= c_2.g(x)$ for all $k <= x$ where $c_1, c_2 >= 1$ and $k >= 0$

O notation (Upper Bound)
$f = O(g(x))$ given that there exist constants c and k such that $f(x) <= c.g(x)$ for all $k <= x$ where $c >= 1$ and $k >= 0$

Ω notation (Lower Bound)
$f = \Omega(g(x))$ given that there exist constants c and k such that $c.g(x) <= f(x)$ for all $k <= x$ where $c >= 1$ and $k >= 0$

We apply these asymptotic notations to measure space and time complexities of a program. Note that this approach ignores multiplicative constants.

1.2 Scalability and Complexity classes

Scalability deals with the following issue: How much more resources (space and time) do I need when my problem size increases?

So the scalability of the problem is the issue of the growth rate of the time and space requirements with increase in the input size. In this class we are not interested in a very sophisticated performance analysis. For any problem, we will be interested in knowing whether it falls into one of three broad categories: slow, moderate and fast growth, which we define as follows:

- Slow — Logarithmic
- Moderate — Polynomial
- Fast — Exponential

Table 1: **Complexity Classes and Scalability**

	Slow	Moderate	Fast
Time	LOGTIME	PTIME	EXPTIME
Space	LOGSPACE	PSPACE	EXPSPACE

Thus, we can define categories of problems according to their scalability. This is summarized in table 1.

Informally, the classes can be explained as follows:

- PTIME — Problems that can be solved in polynomial amount of time.
- EXPTIME — Problems that can be solved in exponential amount of time.
- LOGSPACE — Problems that can be solved in logarithmic amount of space.
- PSPACE — Problems that can be solved in polynomial amount of space.
- EXPSPACE — Problems that can be solved in exponential amount of space.

Defining LOGTIME is problematic since reading an input of length n takes time n. Thus this class is machine dependent. In this course we will ignore this class and focus on the other classes when we talk about complexity. The categories above are called *complexity classes*.

1.3 Relationships between the classes

By definition,

$$\text{PTIME} \subseteq \text{EXPTIME}$$

$$\text{LOGSPACE} \subseteq \text{PSPACE} \subseteq \text{EXPSPACE}$$

It is easy to see that

$$\text{PTIME} \subseteq \text{PSPACE}$$

and

$$\text{EXPTIME} \subseteq \text{EXPSPACE}$$

This is because consuming space takes time, so any computation that takes polynomial space must take at least polynomial time. Therefore any computation that takes at most polynomial time cannot take more than polynomial space.

Next we consider LOGSPACE. If a problem is in LOGSPACE, it means that in solving an instance of the problem at most $c \log n$ cells will be used, where c is a positive constant and n is the size of input. Let the alphabet of the machine have size d. (Real-world computers have an alphabet of size 2 consisting of

the binary digits 0 and 1.) Then the number of possible configurations of the memory is $d^{c \log n} = n^{c \log d}$, which is polynomial in n. So we can run through all possible configurations of the memory in polynomial time. Since a solution to the problem must be one of those possible configurations, we can compute it in polynomial time. Thus we have

$$\text{LOGSPACE} \subseteq \text{PTIME}$$

and similarly,

$$\text{PSPACE} \subseteq \text{EXPTIME}$$

Putting together all the relationships between the classes, we get,

$$\text{LOGSPACE} \subseteq \text{PTIME} \subseteq \text{PSPACE} \subseteq \text{EXPTIME} \subseteq \text{EXPSPACE}$$

An immediate question that arises is whether the containments above are strict. It was discovered in the 1960's that an exponential gap matters. So,

$$\text{LOGSPACE} \subsetneq \text{PSPACE} \subsetneq \text{EXPSPACE}$$

$$\text{PTIME} \subsetneq \text{EXPTIME}$$

So we know that the long chain of containments has gaps. The question is where exactly the gaps lie. This has been the central question of complexity theory for the last 40 years and it is still unresolved.

2 Truth Evaluation

Truth evaluation is concerned with the following question:
Given $\varphi \in Form$ and $\tau \in 2^{AP(\varphi)}$, does $\tau \models \varphi$?

To answer the question above, we can equivalently evaluate $\varphi(\tau)$. We can write a simple recursive algorithm to evaluate $\varphi(\tau)$.

$\varphi(\tau) =$
case
φ is $p \in Prop$: $\varphi(\tau) = \tau(p)$
φ is $(\neg\theta)$: $\varphi(\tau) = \neg(\theta(\tau))$
φ is $(\theta \circ \psi)$: $\varphi(\tau) = \circ(\theta(\tau), \psi(\tau))$
esac

From this algorithm we immediately get: **Theorem**: Truth evaluation is in PTIME.

A more clever algorithm yields: **Theorem**: Truth evaluation is in LOGSPACE.

3 Logical Extremes

In the set of all formulas there are two extreme types of formulas: those that are always true, and those that are always false. More formally, if $\varphi \in$ FORM then φ can be put in one of three categories:

- φ is *valid* or a *tautology* if for every $\tau \in 2^{AP(\varphi)}$, $\tau \models \varphi$.
- φ is *unsatisfiable* or a *contradiction* or for every $\tau \in 2^{AP(\varphi)}$, $\tau \not\models \varphi$.
- φ is *satisfiable* if there is a $\tau \in 2^{AP(\varphi)}$ such that $\tau \models \varphi$.

For example, $((a \wedge (a \rightarrow b)) \rightarrow b)$ is a tautology representing the "Barbara" syllogism, which states that if Socrates is a man and all men are mortal, then Socrates is mortal.[1] $(a \wedge \neg a)$ is a contradiction, since there can be no truth assignment in which a is true and false at the same time. Finally, $(a \wedge b)$ is satisfiable, but not a tautology, since it is true if a and b are true, but false otherwise. See also the Venn diagram in Figure 1.

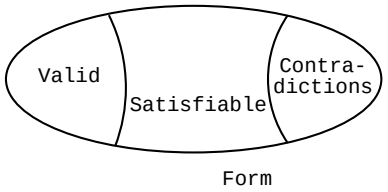

Figure 1: Venn diagram representing the classification of elements in FORM.

Which elements in this classification are the most 'interesting' depends on the point of view:

- Philosophers have been mostly interested in tautologies and contradictions. These are the things that are always true and never true. They represent universal truths in some sense.
- For electrical engineers, tautologies and contradictions are of little interest because the corresponding circuits have a constant output value, i.e. they are 'stuck' either at 0 or at 1. Engineers are generally interested in formulas that are satisfiable, but not valid.
- For software engineers, valid means $models(\varphi) = 2^{Prop}$, and contradiction means $models(\varphi) = \emptyset$. They, too, are mostly interested in formulas that are satisfiable, but not valid.

There is an intuitive relationship between the different classes:

[1] The name bArbArA is a mnemonic. It represents the fact that there are three affirmative statements in the syllogism.

Lemma 1

1. *ψ is valid $\iff$ $(\neg\psi)$ is not satisfiable.*
2. *ψ is not valid $\iff$ $(\neg\psi)$ is satisfiable.*
3. *ψ is satisfiable $\iff$ $(\neg\psi)$ is not valid.*

These can be easily proved simply from the definitions. We can now begin classifying formulas according to the above definitions. Given φ, we would like to know

- is φ satisfiable?
- is φ valid?

These questions are related as seen in the above lemmas and are fundamental to logic.