Lecture 1: Overview

built on 2021/01/06 at 11:15:11

Welcome to Computation Theory, Term II/2020–2021 edition.

Computation theory is mainly about the question: *What are the fundamental capabilities and limitations of computers?* Fundamental in the sense that these are capabilities and limitations of *any* computers or computing devices whatsoever.

To get a sense of what we're going to study: Sorting is this simple problem of rearranging a sequence of items so that they are ordered from small to large. Is sorting hard? Apparently not. We have seen $O(n \log n)$ algorithms and they're pretty darn fast. Compare this with another seemingly-simple question: find a schedule of classes for the whole university such that no two classes meet at the same time in the same room. This is is apparently harder—why?

Computation theory is often taught with three focus areas: automata theory, computability theory, and complexity theory. It is unapologetically a Math class. We begin our journey with automata theory, rigorously modeling computers with different levels of capability. Then, we'll look at the study of computability, which determines which problems, or classes of problems, can be solved in each model of computation, ultimately establishing classes of problems that cannot be solved in any reasonable model of computation. Finally, we'll look at computational complexity theory, which aims to find out the intrinsic amount of resources (e.g., time and space) necessary to solve a problem.

We'll cover important ideas (some more theoretical, some more practical) that have shaped computer science as a field. Though many of these ideas have found real-world applications, one central focus of this course is to understand fundamental limits of computing—for example, what can and what can't be done efficiently? what can and what can't be computed? In many ways, this course complements what you've seen in the Data Structures course, as well as the Algorithms course—although we'll only assume you have mastered the lessons of Data Structures and Discrete Math.

As many students in the past have pointed out, this course has no practical utility in the short term, but it has significant long-term value, one that is central to your CS education: it makes you a better abstract thinker; it is the gym for problem solving; and Math is beautiful.

1 Administrivia

1.1 Basic Info

- Instructor: That's Me. Kanat Tangwongsan (`kanat dot tan AT mahidol dot edu`)
- Office Hours: By appointment.
- TA: New. This is an upper-level course. You will also help run this course.
- Website: the main site is at `https://cs.muic.mahidol.ac.th/courses/toc`, but we use Canvas for gradebook, homework submissions, discussions, etc.
- *Textbook:* No textbook for the class. We are giving you homegrown notes. There are also a few references that we will put up on the course website.

1.2 General Expectations

- You have taken (and mastered the concepts of) Discrete Math and Data Structures & Algorithms. We prefer that you have had Algorithms.
- This course will present to you perhaps the most abstract ideas you have seen in your whole CS career. Understanding them requires some dedication on your part. The only means I know to have a firm grasp of these topics is to practice. Hence, I expect you to take the problem sets seriously. It is strongly advised that you first work on the problems by yourself—only after a day or two of thinking should you begin discussing them with friends.

- Ask a lot of questions!
- There will be reading assignments (and in-class reading "checkpoints")

1.3 Grades

Your letter grade will be given at the end of the term. It is determined holistically to reflect your performance for the whole term on the following components:

(bi-weekly) Assignments (6 sets)	18%
Tests (4 tests, one during final)	76%
Participation and In-class activities	6%

Per OAA, if your term score is x,

Scores	Letter Grade
$x \geqslant 90$	A
$85 \leqslant x < 90$	B+
$80 \leqslant x < 85$	B
...	...
$x < 60$	F

1.4 Assignments and Late Policy

- Assignments due electronically at 11:59PM Bangkok time
- Encouraged to hand them in well ahead of the deadline.
- You are allotted **THREE (3)** late days for the term at no grade penalty.
- At most **ONE (1)** late day may be used per assignment.
- If you have used up these late days or used more than one for a given assignment, your submission will not be graded.

Each problem on your assignments will be graded using the following scheme (out of 2 points).

Symbol	Our Thoughts...	Numerical Score
✓	Good answer (or better)	2
$\checkmark^-$	Seriously flawed	1
✗	Ouch and/or not attempted	0

1.5 Collaboration Policy

- Working together is important; the goal is to learn. Hence, we interpret collaboration very liberally.
- You may work with other students. However, each student must write it up separately. Be sure to indicate who you have worked with (refer to the hand-in instructions).
- Sharing code/writeups: not okay.
- No collaboration whatsoever on exams

2 Three Unrelated Things

We will look at three seemingly unrelated examples that hopefully give you a feel for what we're studying this term. The discussion today is intended to be high-level; we'll come back and flesh out the details later in the term.

To ask questions such as what a computer can and cannot do, we'll need to define more formally what a computer is. For today, though, we'll use our intuitive understanding and defer that discussion to a little later.

2.1 Parenthesis Matcher

Consider the following simple task: write a program that given a string consisting of only open- and close- parens (i.e., `(` and `)`), decides whether the expression is well-formed. We've seen this problem from Data Structures. A moment's thought reveals that it is pretty easy to do with a single counter. In Python:

```python
def is_well_formed(st_expr):
    depth = 0
    for tok in st_expr:
        if tok=='(':
            depth += 1
        else:
            if depth>0: depth -= 1
            else: return False
    return depth==0
```

It is also easy to see that if the input string has length n, this program runs in $O(n)$ time and stores only a number—the `depth` variable. Let's be a bit more pedantic. We wish to be a lot more precise about our space usage. We'll count the exact number of bits used by this program.

As it turns out, the seemingly-simple `depth` variable doesn't take just a constant amount of space. When storing a large number, we intuitively expect to need more bits to represent it. But just how many bits? By a simple counting argument, we know that $\Theta(\log T)$ bits is necessary and sufficient to store a number T. Since T never exceeds n, the length of the input string, we know that the variable `depth` doesn't occupy more than $O(\log n)$ bits of space.

The more involved question—but a fun and important one nonetheless—is, could we have solved the same problem for *any* given input using fewer than $O(\log n)$ bits of space? For example, is it possible to use only a constant number of bits? If not, can we demonstrate convincingly (i.e., rigorously) that this is not at all possible?

2.2 Does This Halt?

One nice tool for programmers is a program that can indicate whether a given function F will terminate or not on input I. That is, given a function F and an input I, return True/False depending on whether $F(I)$ will eventually terminate. In some simple cases, this problem is pretty easy. For example, by inspection, we know that the program

```python
while True: pass
```

will not terminate unless interrupted, whereas we know with certainty that the one-liner `print "Hello, World"` will terminate. Can we design this tool and sell it for $$$?

Let's think about it for a bit. Can you come up with a solution?

It is indeed impossible. But how do you prove to yourself (or anyone for that matter) that it is impossible to come up with such a scheme? This is called the *halting problem*. Here's a (very) rough sketch of the idea (lots of lies and imprecision).

Assume for a contradiction that we manage to write a function WILL_IT_HALT(F, I) that answers the question, does F halt on input I? In other words, if you remember something from OPL, the type of WILL_IT_HALT is the following:

$$\text{WILL_IT_HALT}\colon (\texttt{Program}, \texttt{Input}) \to \{\texttt{True}, \texttt{False}\}$$

Consider the following program: The function CONFUSE takes no argument and if it is to return, it returns nothing. The function is straightforward:

CONFUSE():

```
    **if** (WILL_IT_HALT(CONFUSE, ())) **then** enter a forever loop
    **else return**
```

What happens when we ask WILL_IT_HALT if CONFUSE terminates? If WILL_IT_HALT says it doesn't terminate, WILL_IT_HALT would be wrong because CONFUSE would indeed terminate. If WILL_IT_HALT says it *does* terminate, WILL_IT_HALT would, again, be wrong because CONFUSE would go into an infinite loop. What can we conclude? A contradiction! There isn't a function with that capability after all.

2.3 Is Factoring Hard?

Can you factor 4921 into primes? After a few seconds of sitting down, you can work out that $4921 = 7 \times 19 \times 37$. So here's an easy-to-state problem: given a number N, factor it into primes. What is a good algorithm for this? Can you come up with one?

Let's talk also about a related question: if I give you a number 12345679, is it a prime? Apparently not. You can check that $12345679 = 37 \times 333667$. In general, how do we check if a number is prime? How efficient can we do it?

Perhaps, you still remember this code from Intro to Programming:

```python
def is_prime(n):
    if n < 2: return False
    for trial in range(2, n):
        if n % trial == 0: return False
    return True
```

What's the running time of this? Well, you know that there is an optimization where instead of going all the way up to n, we could simply stop at $\sqrt{n}$. The running time is roughly speaking $O(\sqrt{n})$

But it should be noted that we normally represent a number in binary, so the number N takes $m = \log N$ bits to represent. So what's the running time in terms of the number of bits? The above algorithm runs in $O(2^{m/2})$ time, i.e., exponential in the number of bits—the actual problem size.

As it turns out, the most "efficient" algorithm for this problem (the so-called primality testing problem) runs in $O(m^c) = O(\log^c N)$ for some constant c. Hence, in some sense, it's pretty efficient.

Relative Hardness: It is not difficult to see that if one has a routine for factoring, then this can be used to solve primality testing. Hence, in terms of hardness, we know that

$$\text{primality testing} \leqslant \text{factoring}$$

It is unclear whether having a good primality testing routine at hand helps us solve factoring. Indeed, factoring is apparently difficult and no one knows really just how difficult it is. On the belief that it is difficult, we can turn the problem on its head: if something is difficult, what could we use it for? As it turns out, using the belief/assumption that factoring is hard, we build cryptography systems on it. We'll see more later.

3 Next Time

We'll review the idea of proofs and proof techniques; and we'll also look at some connection (or the lack thereof) between proofs and computations.

First assignment: Out tomorrow (Wed).