Lecture 1: Course Overview

built on 2019/04/23 at 15:53:34

Welcome to Data Structures & Algorithms, Term III/2018–2019 edition.

Your Intro to Programming course taught you the mechanics of writing a program. It also introduced you to the world of problem solving using computers. Your Intermediate Programming gave you more comfort writing larger programs in an object-oriented style. It also taught you Java. This course picks up where these courses left off: we will learn systematic ways to think about problems and generalizable techniques for solving them—and we will practice solving a lot of (algorithmic) problems.

Course Goals: Data structures and algorithms are the fundamental building blocks of any software system. In this course, we will learn to design, analyze, and program algorithms and data structures. In the end, you can select and apply appropriate data structures and algorithms for different applications.

We'll focus on fundamental concepts that can be applied across problem domains, programming languages, and computer architectures. There will be a significant programming component, which will be in Java.

We're not teaching you proper Java—that's the job of your Intermediate Programming class. We'll look at enough Java concepts to get you through. You're encouraged to learn more by yourself and practice good programming style nonetheless. To create a uniform environment, we're fixing the version number: we're using Java 8.

In a word: this course aims to teach you basic techniques and flex your problem-solving muscle so that you can come up with solutions that are

(1) correct (and show that they're correct)—this is the correctness focus.
(2) fast (and quantify how fast)—this is the performance focus. and...
(3) beautiful—this is the interface-design focus.

1 Administrivia

1.1 Basic Info

- Instructor: Kanat Tangwongsan (`kanat dot tan AT mahidol dot edu`)
- Office Hours: Posted office hours or by appointment in 1409
- TA: Gift and Kaotoo have graciously volunteered to help with this course. They can be reached in person—or electronically via Canvas discussions.
- Website: the main site is at `https://cs.muic.mahidol.ac.th/courses/ds`, but we use Canvas for gradebook, homework submissions, discussions, etc.
- *Textbook:* No textbook for the class. There are a few references that we will put up on the course website.

1.2 General Expectations

- You know Python from Programming I/Intro to Programming... and you know Java (or are willing to study it on your own).
- You have taken Discrete Math (ICMA 242/ICCS 342).
- You can expect *three* things: (1) a lot of work (2) challenging work (3) a lot of help/guidance. You're expected to take responsibility for your journey in this course. You're expected to work hard on it—and please ask questions to help you learn. And by the end, you will have learned a lot.

1.3 Grades

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

Assignments (≈8 sets)	16%
Mastery (3 exams)	42%
Quizzes (5 sets)	35%
Participation	7%

We have no exact formula for letter grades, but 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

Exercises on your assignments will be graded using the following scheme (out of 2 points). Note that problems are not graded.

Symbol	Our Thoughts...	Numerical Score
✓	Good answer (or better)	2
✓⁻	Fair answer but has serious flaws	1
✗	Ouch	0

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 **FIVE (5)** 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.

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 What's an algorithm?

According to Wikipedia, an *algorithm* is a self-contained sequence of actions to be performed that can be expressed within a finite amount of space and time and in a well-defined formal language. The precise definition doesn't matter much here. We will get a sense of it by way of examples.

```
1. Form groups of 4 students
2. One person from each group raises hand; you will be handed an A4
```

Here's an algorithm:

Step 1: Fold the paper along the long side—tear it along the fold.

Step 2: With two pieces of paper, fold each of them again along the long side—tear along the fold.

Each group should now have 4 pieces of paper.

As an algorithm, we specify **input** (a piece of A4), a process, and an **output** (4 pieces of papers).

3 Three Unrelated Examples

We will look at three unrelated examples that demonstrate the power of algorithms and data structures. These examples are intended to be high-level. In the back of your mind, you should think how you will implement these ideas as computer programs (say in Python or Java).

3.1 Example: Who Has The Highest Number?

In our first example, we'll answer a simple question about a collection of numbers. We'll see firsthand what differences good algorithms can make.

Everyone has a piece of paper (we just made a stack). Write your name and *one* number between 1 and $10,000$ on it.

The question is, *in this class, who has the highest number?*

That is, the *input* is everyone's number, and the expected *output* is the highest number. Such input/output behaviors define a *problem*. We're just asking the question: find the largest number from a collection of numbers given as input.

Let's try this on real-human subjects!

3.1.1 Approach I: Everyone Calls Out His/Her Number

There are variants of this approach, but to keep it simple, let's say there's someone willing to keep track of the largest number announced so far (initially $-\infty$). Then, each person calls out the number, in turn. At the end of this process, we'll have the largest number.

3.1.2 Approach II: Pairing Up

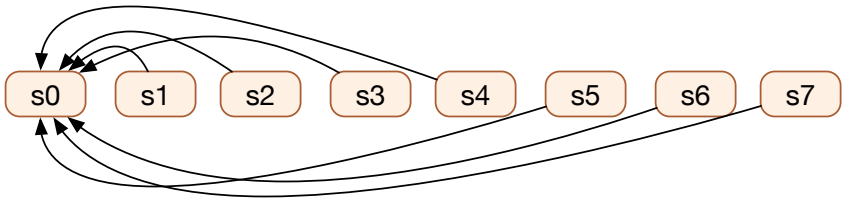
We can take a more social approach. Instead of subjecting one person to that menial bookkeeping task, we will ask everyone to actively participate in a computation. To start this process (algorithm), everyone will have to stand up. We'll carry out this process until only *one* person is left standing:

1. Everyone standing pairs up (if you don't have a partner, just stand there)
2. Talk with your pair: if you have a smaller number than your partner's, you sit down. Otherwise, continue to stand.

The claim is, *the last person standing has the largest number.* Why? This is like a tournament. We pair up the participants and the weaker one (one with a smaller number) is eliminated.

3.1.3 Back of An Envelope Analysis

In Approach I, we have no idea where the largest number is, so we have to wait for all the numbers to be announced. The communication pattern can be illustrated using the figure below:



Hence, if there are n people in the room, this will be about n rounds of exchanges. This means, if we have a room of 1024 people, we will finish after over a thousand(!!) rounds of communication.

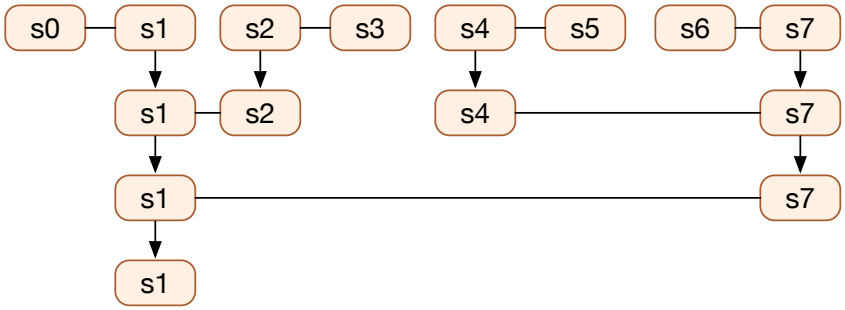
Approach II is more clever. The crux of it is the following simple observation:

> **Observation:** If your friend has a number larger than you, you can't be the maximum, so you can safely leave the process.

Iter #	# Standing
0	32
1	16
2	8
3	4
4	2
5	1

Then, notice that because one person per pair will sit down, in every round, half of the participants will sit down. Therefore, if we start with $n = 32$ participants, we'll have 16 left after the first round, then 8 after the second, 4 after the third, 2 after the forth, and 1 after the fifth.

For example, when we start with $n = 8$, the process can be schematically depicted as follows:



In general, to figure how many rounds this takes, we just have to ask the question: *How many times can we divide n by* 2 *until it is at most 1?* We can answer this question using simple arithmetic: what's the smallest t such that $n/2^t \leqslant 1$? Because

$$\frac{n}{2^z} \leqslant 1 \iff z \geqslant \log_2 n$$

we have that the smallest such z is $\log_2 n$, so the answer is $t = \log_2(n)$ rounds.

3.2 Example: Caller ID Lookup

Let's look at a data structure problem that we use practically everyday. When someone gives us a call, our cell phone looks up in the address book, trying to match the caller id (the phone number of the caller) with a name associated with that number which we have in our address book. We will examine this in more detail now.

First, we'll specify the problem more precisely:

- **Input:** a caller id (e.g., 0871443145) + an address book
- **Output:** a name matching the caller id (e.g., John Doe) or report unknown.

The main data structuring question here is: *how to store the address book so that we can answer this kind of question quickly?*

Users may want other features from this address book data structure, such as we want it to be small (use very little memory) or that it can be created quickly. For now, let's focus on the retrieval aspect.

That is, for now, our prime concern is, how to make look-up's fast (ignoring other considerations)?

3.2.1 Approach I: A Pile Of Numbers

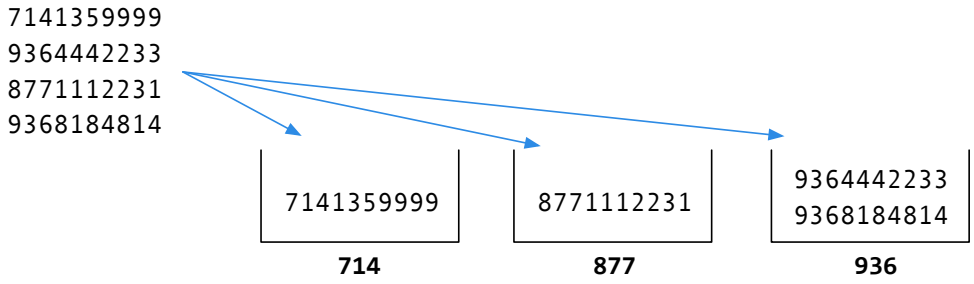
The first approach is really simple: Keep the address book's entries in an unordered pile—imagine a stack of cards in an arbitrary order.

This is obviously easy from an upkeeping point of view. If we want to add a new contact, it's just a matter of putting that "card" into the pile. But how does it fare in terms of the look up speed (which is the measure we particularly care about)?

Let's try this manually for now. We'll soon also attempt to analyze this analytically.

3.2.2 Approach II: Prefix-Based Buckets

As an alternative, we'll try the following: Bucket numbers by their 3-digit prefixes. For example, the number 7141359999 goes into a bucket labeled 714. As a more elaborate example, we have the following:



This is a bit sophisticated to upkeep but it's still simple. Adding a new contact only requires adding that contact to the right bucket.

Let's once again try this manually. And we'll attempt to analyze both approaches analytically.

3.2.3 Back of An Envelope Analysis

In Approach I, we don't know where the contact we're looking for is in the pile, so in the worst case, we have to look at every single card. If there are n cards, we'll need to look at all n of them.

In Approach II, let's make a simplifying assumption. We'll see later on that we can make sure that this assumption actually holds: we'll assume that the prefixes are uniformly distributed, meaning that if there are n entries and k buckets, each bucket gets around n/k entries. Therefore, to look in a bucket, we have to look at no more than n/k entries. This can be a big win for a large address book.

We'll come back these ideas later this term. In particular, we'll look at their performance theoretically. We'll also implement variants of these ideas and benchmark their empirical performance.

3.3 Example: Data Compression

Has anyone used a data compression program (e.g., zip, bzip2, etc.) before? In this example, we'll take a look at the basic ideas behind such programs. So, how might one compress data? Note that by compress, we mean to reduce the space requirement without losing any data (i.e., it can be uncompressed later). This is typically called 'lossless' compression.

Let's look at a few easy cases first:

- Say your data looks like this `aaaaaaaaaaaaaaaaaaaa`. Can we come up with a scheme that uses fewer bytes than storing this many `a`'s? One way to do this is to devise a representation that says "`a` has come up 20 times consecutively", for example, `a` : 20. We'll call this scheme *run-length encoding* (RLE) because we encode the length of a run of a symbol.
- But this is no good if we have a mix of different things interleaving with each other, say `ababaacbabacabaaa`. There are still a lot of `a`s but RLE doesn't really help. A new idea is needed. It turns out we can work at the bit level (as opposed to a byte level). Because the symbol `a` more frequently than others, we want to use fewer bits to represent each of them. That is,

  If something shows up a lot, make the representation for it short.

  As a concrete example, suppose the frequencies of the symbols are as follows:

Symbol	# of Occurences	Proposed Bit Representation
a	100	0
b	2	10
c	10	11

The original data representation would require $8 \times 100 + 8 \times 2 + 8 \times 10 = 896$ bits, or 112 bytes. Whereas, using the proposed bit representation, we'll need only $1 \times 100 + 2 \times 2 + 10 \times 2 = 124$ bits, or roughly 16 bytes. That's a lot of saving.

4 Things You Know, and Things You Don't (Even) Know (They Exist)

One of the main goals of this class is to teach you to solve real-life problems using computers. And much of what we'll do in this class and beyond will involve piecing together things that we know to accomplish what we want. To give you a sense of what this means, let's look at the following puzzle:

Many of you may have seen a device for measuring time known as an *hourglass* and various other names such as sand glass, sand timer, and egg timer. An hourglass has two connected vertical glass bulbs allowing a regulated trickle of material (usually fine-grained sand) from the top to the bottom. It is designed to measure the passage of a certain duration, say 7 minutes, 30 minutes, or an hour. At first, one of the bulb is full. To start timing, we set it so that the full bulb is at the top. The duration as designed has elapsed when the top bulb runs out—and it can be inverted to begin timing again.

For this puzzle, you're given *two* hourglasses, one measuring 7 minutes and one measuring 11 minutes. Now you have a picky grandpa who wants his egg cooked for exactly T minutes. If he wants $T = 18$ minutes, you can describe a simple process as follows:

Step 1: Light the stove
Step 2: Begin timing on the 11-minute hourglass
Step 3: When that is done, start timing on the 7-minute hourglass.
Step 4: Turn off the stove

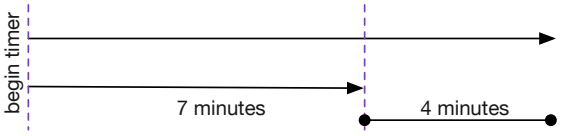
And lo and behold, that's 18 minutes.

Now onto the real puzzle: *cook an egg for $T = 15$ minutes* using these two devices (no cheating).

Let's think about it for a few minutes.

And here's a solution. First, we know how to measure 7 minutes and 11 minutes already. To measure 15 minutes, we can try to measure $7 + 8$ or $11 + 4$. But one clue we have is that 4 is $11 - 7$. So why don't we try measuring 4 minutes first?

We'll start both timers at the same time. When the 7-minute timer finishes, we know that the 11-minute timer has 4 more minutes to go. Therefore, we have a 4-minute timer from this point on.



Attempt #1: To make $T = 15$ minutes, we'll do 11 minutes first then 4 minutes. First, start the 11-minute timer. Once done, what now? We can't quite do anything.

Even though we know how to make a 4-min timer, we need a lead time of 7 minutes. This creates a constraint on where we can use this construction.

Attempt#2: Let's do 4 minutes first then 11 minutes.

1. Start both 7-min and 11-min timers.
2. When the 7-minute timer finishes, we turn on the stove, knowing that the 11-minute timer has 4 more minutes to go.
3. When that expires, we invert it and wait another 11 minutes.
4. When that expires, we turn off the stove and serve the egg `:)`