Lecture 1

Before the Lecture

Have the following URLs open in a browser:

- https://www.cityofchicago.org/city/en/depts/mayor/iframe/plow_tracker.html
- http://clearstreets.org/

Introduction

- Welcome/introduction
- Minor housekeeping items:
    - All the information about the class is posted on our course website. Bookmark it, you'll be coming back to it many times. http://bit.ly/cmsc12100-aut-18
    - If you are registered for the class, you should also be on our Piazza site. Important announcements are made through that site.
    - **No discussion sessions this week**. Instead, we have reserved time on the CSIL labs for you to work through the first lab (details on Piazza about this). Starting 2nd week, you have to attend the discussion session you are registered for on my.UChicago.
    - If you are not registered for the class, but want to get into the class, there is a waiting list to get in. You can get on the waiting list on https://waitlist.cs.uchicago.edu/
- We will go through other course logistics at the end of the lecture

Computational Thinking

This may seem like a surprising statement, but the ultimate goal of this class isn't to teach you programming. You *will* learn how to program, and you do *a lot* of programming in this class, but the ultimate goal of this class is to teach you **computational thinking**.

What is the difference between "just knowing how to program" and "computational thinking"?

- Computer programming is just a tool.

- Last year, I took a 3-hour workshop on how to use the woodshop tools at the university's Fab Lab. I know how to saw, drill, and sand. I have no idea how I'd build a cabinet, or a table, etc.
- Similarly, learning the basics of programming is not that hard. However, applying those skills to solve a problem computationally is a very different ball game. That's where it is important to know how to think computationally ("think like a computer scientist")

In fact, another goal of the class is to empower you to take problems that matter to you in your fields (economics, finance, public policy, etc.) and solve them computationally. If you take CS 122, you'll actually be able to work on a project of your own choosing (in this class, you will work through programming assignments based on real problems from a variety of fields)

So, what exactly is computational thinking?

⇒ SLIDE: Jeanette Wing definition

The field of Computer Science is huge, but we will focus on four specific concepts/skills that are integral to computational thinking.

⇒ SLIDE: Four aspects of Computational Thinking

Decomposition/Abstraction

Computational Thinking can help us solve large and complex problems in a more manageable way.

Problem: How can we find out, after a snow storm in Chicago, whether a given street in the city has been plowed?

The City of Chicago has a Plow Tracker website that shows real-time information of plows.

⇒ WEBSITE: Plow Tracker

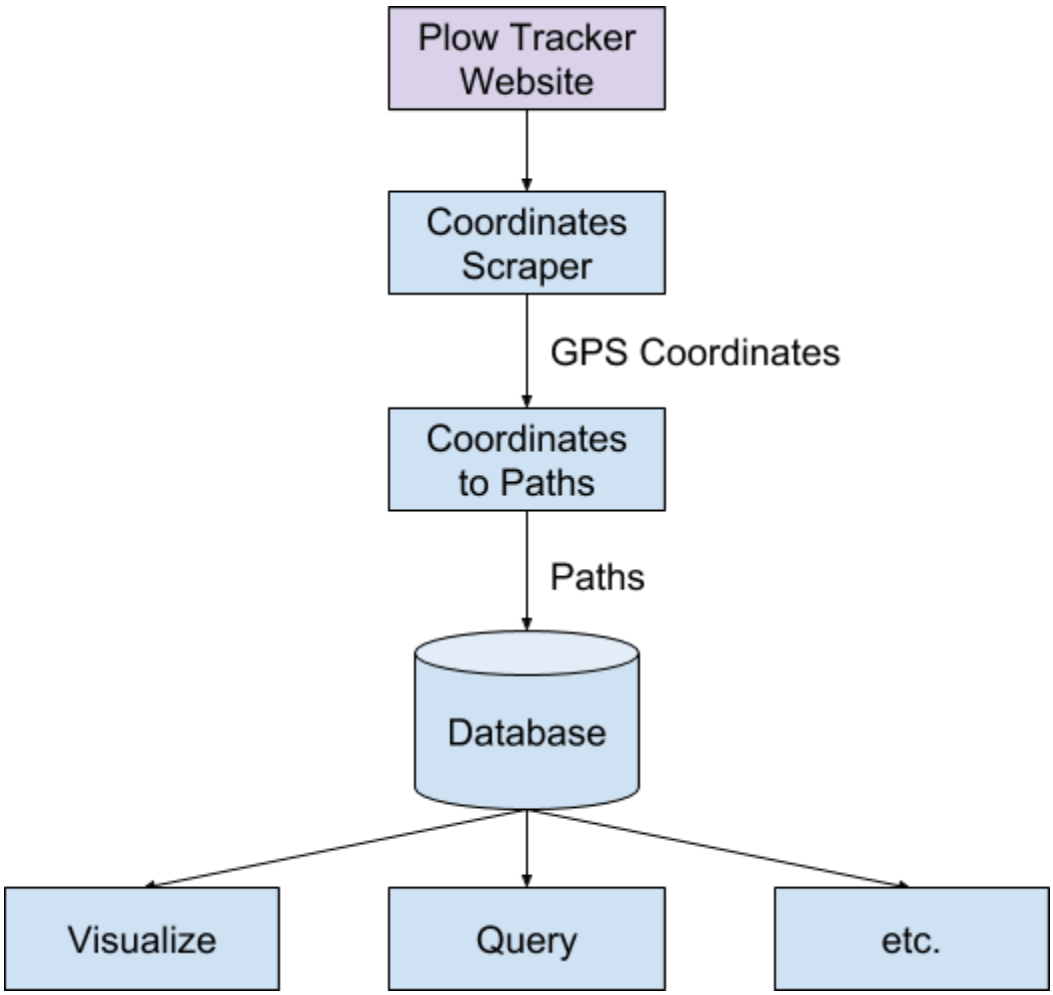
Let's say we have the ability to extract a stream of positions

⇒ SLIDE: GPS Coordinates

ASK: How do we go from that to the following?

⇒ SLIDE: Map
⇒ WEBSITE: ClearStreets. Note that there's also the ability to query the data

Arrive at something similar to this (additional tasks may come up in discussion):



Notes:

- Don't go into discussion of how map snapping is done exactly. It can be a big time sink. Mention it is a non-trivial problem, and explained in a little bit more detail in the book.

ASK: We've *decomposed* this problem into smaller problems. What is the advantage of doing this?

Arrive at the following:

- Each component only has to worry about a specific task, without being concerned with how the other parts of the program work.

- Easier to test
- Easier to reuse (point out that converting coordinates to paths is a very well-studied problem, and ClearStreets just used an existing piece of software called OSRM)
- Easier to build new applications on top of those components (harder to do if we had built a single monolithic system)
    - Only if there is time: Mention anecdote of how, since they had the data, they were able to query it in many different ways, including asking what streets were plowed more than others. When you exclude major streets, there was one street that was plowed more frequently than others. Turns out an alderman lives on that street.
- When we "put together" all the various components, we can *abstract* away the details of how each component works internally (e.g., we don't need to know what OSRM works internally). We still care about the *interface* of each component (the data it expects, and the data it produces).

Modeling

Another way in which we can make a problem more manageable is by distilling it into its essential components, so that we don't waste time and effort on aspects of the problem that are irrelevant to solving it.

Given a problem, we need to come up with a model that captures the essence of the problem, while allowing a computer program to solve it.

Modeling is not unique to Computational Thinking (there are also economic models, epidemiological models, statistical models, etc.) but it is integral to it.

Problem: Matching medical students to residency programs (National Residency Matching Program, or "The Match").

ASK: What aspects of this problem would we need to model to be able to solve it? Caveat: not yet thinking about *how* to solve it. Just what information we need to keep track of. Another way to think about this question is "What will the input to my program be?" (i.e., what data will it have to work with)

- Answers may provide a lot of factors that are not part of NRMP: region where the hospital is located, the ranking of the student's medical school, the student's specialty, etc.

- We could come up with a model that captures all of this but, when we distil this to its essence, we just have the following:
    - Students provide a ranking of hospitals
    - Hospitals provide a ranking of students
- We can represent this with a "graph" ⇒ SLIDE: Graph. Circles are students/hospitals, lines are potential matches (student and hospital mutually listed each other)
    - Note: Don't attempt to define or describe what graphs are. "Circles and lines" is enough for now.
- As we'll see later in the class, there are a number of data structures we can use to model the information in a problem. A graph is one such data structure.

Only if time:

- This is probably apocryphal, but still fun: For the Lord of the Rings movies, they came up with a computational model of how combatants would behave in the battlefield (so they wouldn't have to animate each combatant individually; they would just place them in the battlefield, and they would behave according to the model). The initial model weighed two factors: desire to harm the opponent and self-preservation (e.g., you want to fight your enemy, but not if you've sustained so much damage you can't continue). When they did the first simulation, they placed two armies in front of each other, and they each just turned around and walked away. Clearly the model needed to be refined.

Algorithms

In a sense, the model is the input to our program. In this case, we ultimately want to find a matching between students and hospitals.

⇒ SLIDE: Input -> Computation -> Output

(in the right-hand graph, a blue edge represents a match, as opposed to a potential match)

At its most basic level, a program takes some input, performs some computation on that input, and produces some output. The exact steps we follow in that computation is known as an *algorithm*.

This kind of matching problem is well-known, and there are already many solutions to it. We are going to apply an algorithm that is guaranteed to produce something known as a *stable match*. The exact definition is in the book, but you can think of it simply as a "a match that is not necessarily the best one, but is likely to be pretty good".

- Note: Don't try to define a stable match. It can be hard for them to wrap their head around it, and it's not really essential to follow the steps in the algorithm.

In this algorithm, we go through each unmatched student, and match them with the first hospital that also lists that student *and* where that hospital hasn't been already been matched with a student they prefer more (we'll see in a few steps what happens if the hospital was already matched, but with a less preferable student).

⇒ SLIDES: Steps 1-4 involve applying this rule

Now we get to a point where all of E's tentative hospitals are already matched with a student. We go through the hospitals (in the order in which E ranked them), and if any hospital ranked E higher than its current match, we bump the current match and match E instead. This happens right away with H1.

⇒ SLIDES: Step 5 as we've just explained. Step 6 is the same: A bumps D from H4 (but not E from H1)

We reach a stable match when we make a pass through the students and nothing changes (no one is assigned to a hospital, no one is bumped from their current hospital). This happens after Step 6.

Exact algorithm, and definition of stable match is in the book. This kind of algorithm is known as a greedy algorithm: at every step, we took a locally optimal step, in the hopes that it will lead to a globally optimal outcome.

Complexity

Computational Thinking also involves understanding the complexity of a problem, and whether it can be solved efficiently. As it turns out, this problem can be solved pretty efficiently with the greedy algorithm (it takes about N·M steps, where N is the number of students and M is the number of hospitals).

However, the line between "easy" and "virtually impossible" can be crossed pretty easily in CS.

⇒ SLIDE: xkcd

In this case, if we introduce couples into the problem (making sure that couples are matched to the same hospital or to adjacent hospitals), it turns out the problem becomes "NP-Complete". The only way to find a good solution is to check all possible matches, which requires an amount of time that growns exponentially with the number of students/hospitals.

Only if time:

- Similarly, given a map where cities are connected by roads, finding the shortest path between two cities requires R + C log C (where R is the number of roads, and C is the number of cities)
- If instead we want to the shortest path that visits every city once, that takes $C^2 \cdot 2^C$ time. For 250 cities, that would involve roughly 10^{80} steps (which also happens to be roughly the number of atoms in the universe)

When you write a solution to a problem, it is important to understand how efficient it is (we will revisit this concept a few times throughout the quarter), and whether it can even be solved efficiently (when finding the optimal solution is impossible, we sometimes have to rely on algorithms that use heuristics or probabilistic methods to produce a good, but maybe not optimal, solution)

Wrap-up

Throughout the quarter, we will often call back to these four aspects of computational thinking. We'll see that they are often relevant when trying to decide how to solve a given problem. And, while you will use programming (and, more specifically, Python) as a powerful tool, ultimately your ability to solve the problems we give you will depend more on your ability to think computationally.

Final housekeeping

- Reminder: No discussion sessions this week.
- Reminder: Make sure you are on our Piazza site. All important announcements are made there.
- This class will involve doing seven programming assignments. First one will be posted on Friday. You get two 24-hour extensions to use at your discretion.
- Academic Honesty policy on our website. Read it carefully.
- We also have a series of ungraded lab assignments. The first one covers basic Linux skills you will need to do your work in this class, so we have reserved time in the CSIL lab for you to work on them (there will also be TAs there at those times). Times have been posted on Piazza. After that, subsequent labs are "take-home labs"; you can do them on your own either in CSIL or in your VM. We strongly encourage you to do so, even though they are not graded: they will allow you to practice and better understand concepts that are essential to do well in the programming assignments.
- Two exams: Monday, October 29th from 7:00pm-9:00pm and Monday, December 10th from 7:00pm-9:00pm
- Your final grade is 60% PAs, 20% Exam 1, 20% Exam 2.