Casual Musings on Data Structures and Algorithms

Neeldhara Misra

8/2/2022

Table of contents

Preface

These are running notes from my course on Data Structures and Algorithms at IIT Gandhinagar.

If you would like to have access to the weekly assignments, you might want to register for the course here (it's free).

If you have any general comments or questions, please leave them below. Thanks!

1 Data Structures and Structured Data

Link to Slides

1.1 WDYM, data structures?

We will keep it casual and skip formal definitions for now.

Data structures give us principled ways to *stow away* information. It's important to do this nicely based on what you want to *do* with the information.

For example, the notes you might be taking in this class is information. If you have no plans of revisiting them later, you can take them as you please, or better yet, not take them at all!

However, you want your notes optimised for giving you quality company during a 2AM revision session on exam day, competing with Maggi for attention, you want your notes to be competently taken: they don't have to be neat, and it's enough for them to be useful.

On the other hand, if you are taking notes so that a special someone who will inevitably miss a few classes will almost certainly ask for later, then you would be making notes to impress, and that potentially requires a different approach.

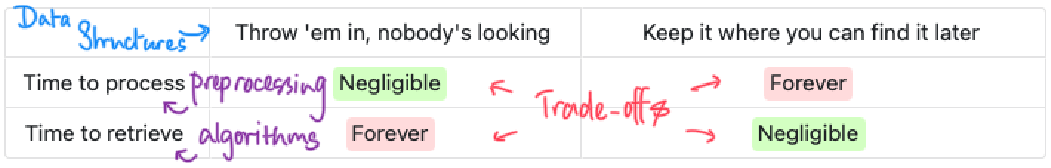
Throughout this course, we will try to make sense of trade-offs.

We'll equip ourselves with ideas that will ultimately help you decide questions like: how do you organise the clothes in your cupboard?

	Throw 'em in, nobody's looking	Keep it where you can find it later
Time to process	Negligible	Forever
Time to retrieve	Forever	Negligible

Table 1. No free lunches.

ℹ This *is* in fact a useful framing!



Data Structures →	Throw 'em in, nobody's looking	Keep it where you can find it later
Time to process ↖ preprocessing	Negligible ←	Trade-offs → Forever
Time to retrieve ↖ algorithms	Forever ←	→ Negligible

Table 1. No free lunches.

1.2 Representing Polynomials

Let's say that you are spending a fine evening watching the #LockdownMath playlist from 3blue1brown. The first episode happens to be all about solving quadratics:

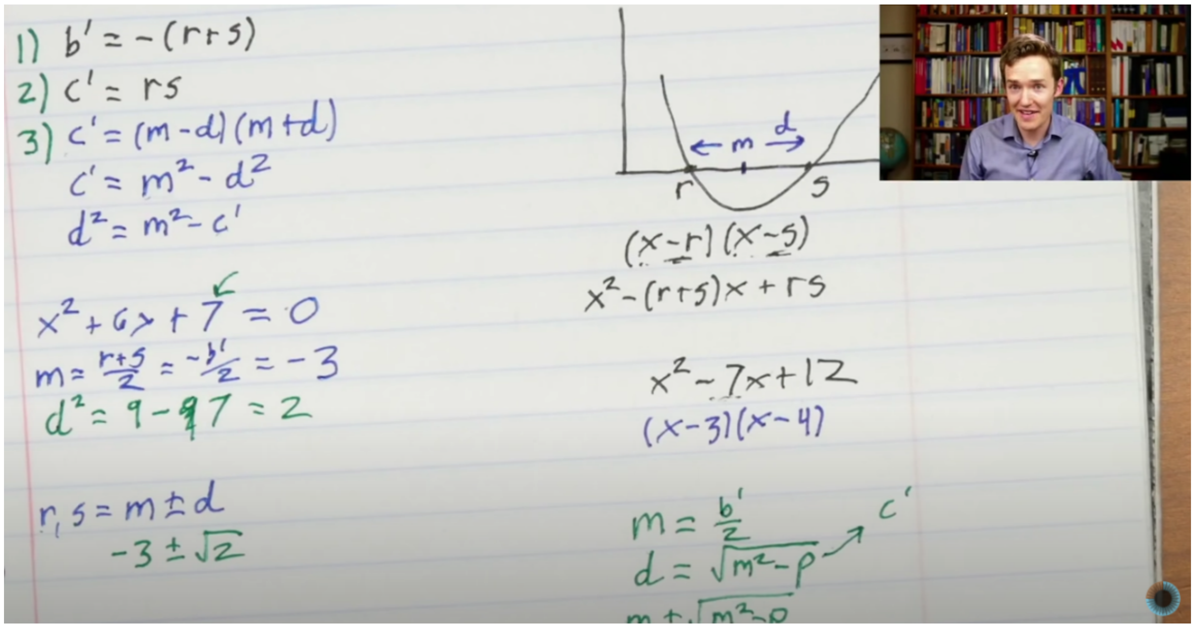


Figure 1.1: A screenshot from #LockdownMath showing Grant Sanderson solving quadratics.

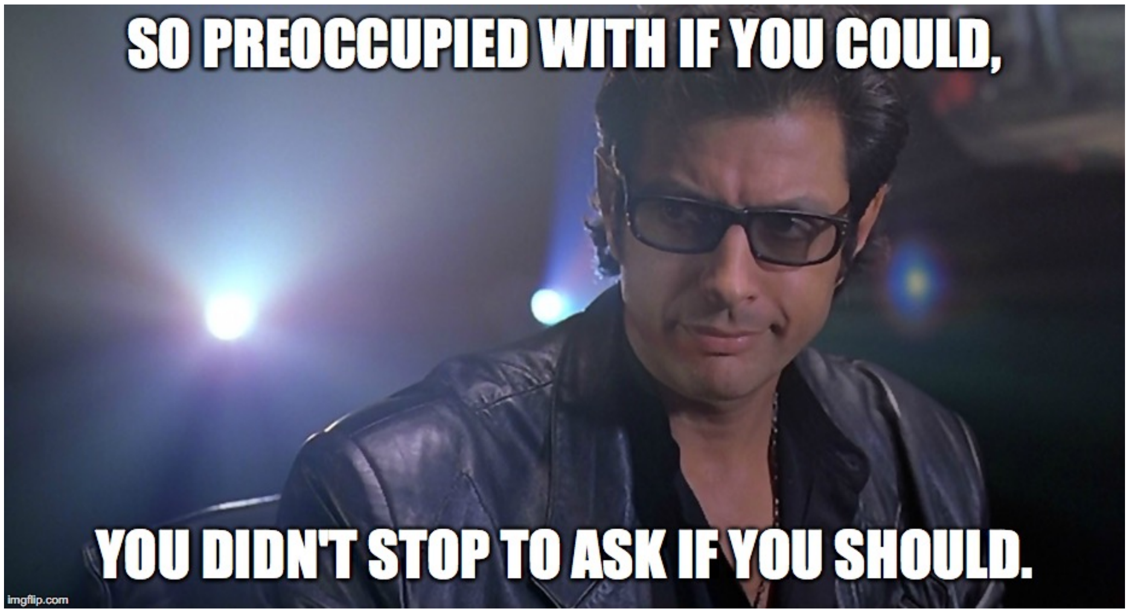
Now, it's quite natural to want to "write a program", so to speak, that can take a quadratic equation such as $x^2 - 7x + 12$ as input and output its two roots.

Given that programs running on your phone are able to make suggestions, even if dubious, for what series to binge-watch next on Netflix, finding roots of quadratics should be a fairly benign exercise.

You might recall that most programs let you declare variables that can hold on to specific *types* of information, for instance: numbers, strings, and so forth. Our input doesn't "look" like a number, so it would be a fair take to simply store it as a string:

```
px = "x^2 - 7x + 12";
```

Now...


SO PREOCCUPIED WITH IF YOU COULD,
YOU DIDN'T STOP TO ASK IF YOU SHOULD.
imgflip.com

While this is a perfectly faithful representation, you can imagine that it would be slightly painful to work with. You would have to write some code that can "pull out" the parts of the string that represent the *numbers* you care about (in this example, $b = -7$ and $c = 12$), so that you can move on to your calculation, which is an *expression* involving numbers.

Given that a quadratic with the leading coefficient normalized to one is uniquely determined by two numbers, it seems a lot simpler to directly represent the polynomial as two integers instead:

```
px_b = -7
px_c = 12
```

You might appreciate that this saves us quite some circus and we can quite directly get to the computation we're interested in. What if you cared about higher order polynomials? You may want to solve them (even if you run out of expressions for solutions pretty quickly, you might be interested in other ways of getting to the roots), or manipulate them in other ways (for example, by adding or multiplying them).

Food for thought.

How would you represent higher-order polynomials? What about multivariate polynomials? Is there a way that you might be able to capture an algebraic expression for a polynomial without either using strings or just the coefficients?

1.3 Representing a Game - I

The game of 100 goes like this: I pick a number between 1 and 10, and then you pick one within the next ten numbers, and on and on. The first person to reach 100 wins.

Recall from class and/or figure out that…

SPOILER ALERT

```
...whoever starts has a way of winning the game:

    0. To begin with, I say 1.
    1. No matter what number you pick, I can say 12.
    2. No matter what number you pick, I can say 23.
    3. No matter what number you pick, I can say 34.
    4. No matter what number you pick, I can say 45.
    5. No matter what number you pick, I can say 56.
    6. No matter what number you pick, I can say 67.
    7. No matter what number you pick, I can say 78.
    8. No matter what number you pick, I can say 89.
    9. No matter what number you pick, I can say 100.
```

What if you want to write a program that mimics the winning strategy?

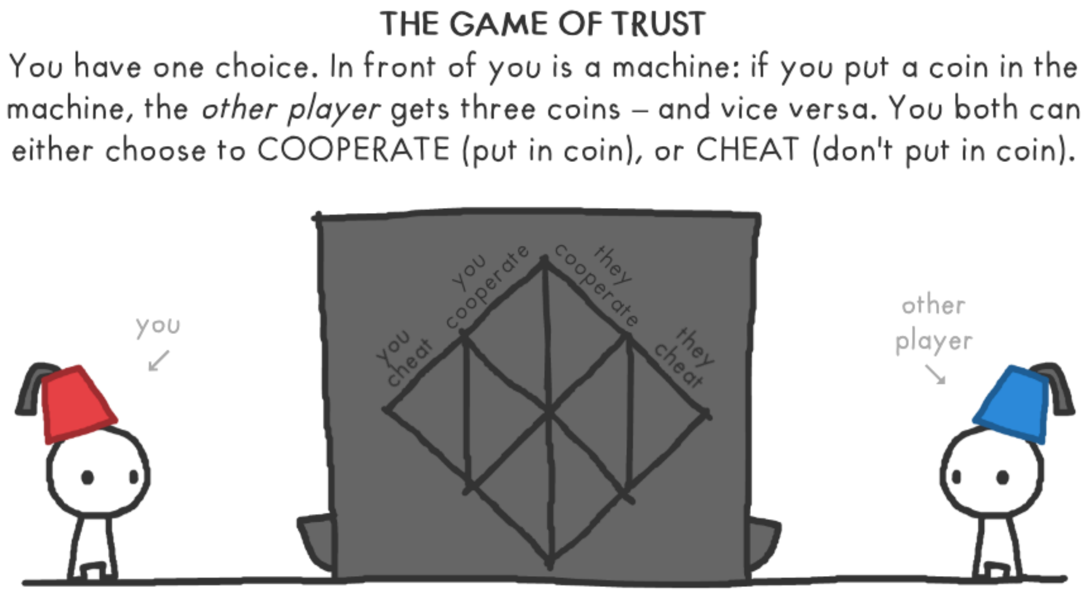
Note that this game can go on for at most a 100 steps, and in fact exactly 20 steps (or ten rounds) when you employ said winning strategy. So one way to go about this is to declare 20 variables to track the 20 numbers exchanged between the players. But a moment's reflection may reveal that you *don't* need to store anything at all.

Exercise

Can you write a program that makes the first move, prompts the user for their moves on their turn, uses the winning strategy discussed above, and uses no variables for explicit storage?

1.4 Representing a Game - II

If ~~you missed the first class~~ you haven't played the Game of Trust, you are welcome to take a break and experience it now. Let's recollect the setup:

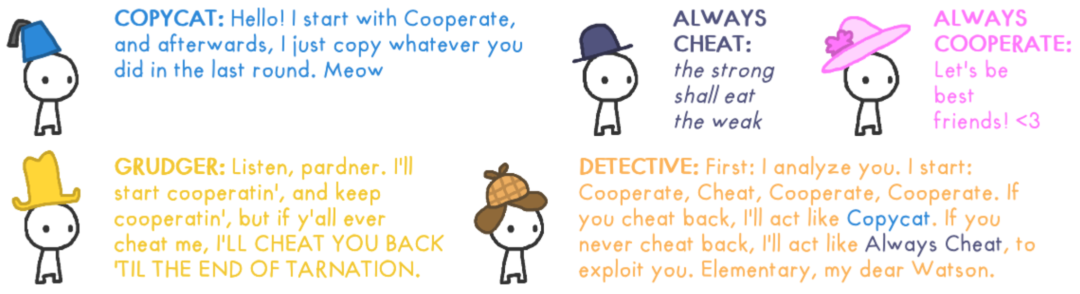


THE GAME OF TRUST

You have one choice. In front of you is a machine: if you put a coin in the machine, the *other player* gets three coins – and vice versa. You both can either choose to COOPERATE (put in coin), or CHEAT (don't put in coin).

Figure 1.2: Illustration from an implementation by Nicky Case.

Suppose you want to implement your own version of this game, where the program responds to inputs from the user and plays according to a specific, pre-meditated strategy. Remember you have seen some strategies already:



COPYCAT: Hello! I start with Cooperate, and afterwards, I just copy whatever you did in the last round. Meow

ALWAYS CHEAT: *the strong shall eat the weak*

ALWAYS COOPERATE: Let's be best friends! <3

GRUDGER: Listen, pardner. I'll start cooperatin', and keep cooperatin', but if y'all ever cheat me, I'LL CHEAT YOU BACK 'TIL THE END OF TARNATION.

DETECTIVE: First: I analyze you. I start: Cooperate, Cheat, Cooperate, Cooperate. If you cheat back, I'll act like Copycat. If you never cheat back, I'll act like Always Cheat, to exploit you. Elementary, my dear Watson.

Figure 1.3: A second illustration from the same implementation by Nicky Case.

We reproduce these strategies below:

Player Strategies

1. COPYCAT: Hello! I start with Cooperate, and afterwards, I just copy whatever you did in the last round. Meow.
2. ALWAYS CHEAT: The strong shall eat the weak.
3. ALWAYS COOPERATE: Let's be best friends <3
4. GRUDGER: Listen, pardner. I'll start cooperatin', and keep cooperatin', but if y'all ever cheat me, I'LL CHEAT YOU BACK TIL THE END OF TARNATION.
5. DETECTIVE: First: I analyze you. start: Cooperate, Cheat, Cooperate, Cooperate. If you cheat back, I'll act like Copycat. If you never cheat back, I'll act like Always Cheat, to exploit you. Elementary, my dear Watson.

Let's say that your program is designed to play 5 rounds and that your program is playing the copycat strategy. To begin with, you might want to declare a couple of variables to keep track of the scores of the players, and ten variables to track the moves of both players in each round. With this, your code may start out looking like this:

```
my_points = 0
user_points = 0

user_move_1 = input("Input 1 for Cooperate and 0 for Cheat.")

//Sanity check input:
if(user_move_1 != 1 and user_move_1 != 0):
    express disappointment and abort
```

```
// My first move is to cooperate:
my_points += -1
user_points += 3

if(user_move_1):
    my_points += 3
    user_points -= 1
```

Now your next move is determined by the value of `user_move_1`, so you might proceed as follows.

```
user_move_2 = input("Input 1 for Cooperate and 0 for Cheat.")

//Sanity check input:
if(user_move_2 != 1 and user_move_2 != 0):
    express disappointment and abort
```

```
// My next move is based on the user's first:
if(user_move_1):
    my_points += -1
    user_points += 3

if(user_move_2):
    my_points += 3
    user_points -= 1
```

. . . and so on and on, you get the drift.

Food for thought.

Do you really need ten variables to track the game? If you were instead implementing the always cheat or always cooperate strategy, how many variables would you need? What about the strategies of the grudger and the detective?

Now, suppose we come up with our own player, whom we call the **majority mover**. This player looks at your entire game history, and cooperates if you have cooperated more than you have cheated, and cheats if you have cheated more than you have cooperated, and acts randomly otherwise.

It seems like implementing the majority mover strategy would really require keeping track of everything. Or would it? You might observe at this point that it's enough to keep track of two counts: the number of rounds and the *number* of moves where the user has cheated: note that it does not matter when the cheats happened in the history of the game.

You could also. . .

. . . track the number of cooperate moves along with the number of rounds; or the number of cheat moves and the number of cooperate moves.
At this point it's a matter of taste :)

How about a **completely random** player? This one chooses a number **K** between 1 and N uniformly at random (let's not worry about *how* this is done for now, because that would be a story for another day), where N is the number of rounds played so far; and mimics the other player's **K**th move. To implement this strategy, you really would need to keep track of the user's entire game history with the five variables, and also assume that you have a way of picking a number at random.

Finally, consider that instead of fixing your program to play five rounds — — you want to politely ask the user how m*any* rounds they want to play.

After all…

Well, for the first few players, this is just a matter of upgrading your for loop (which you should have switched to already when you realised that you don't need all. those. variables.) to use N: and you are done.

Food for thought.

How will you implement this version if you are working with our latest player? If you happen to have a very enthusiastic user who asks for half a million rounds, would you be able to declare that many variables all at once, while your program is running? Notably, you don't know what the user is going to say ahead of time!

1.5 Representing a subset of a deck of cards

If you are implementing a card[1] game, you might need a mechanism for keeping track of "hands", or various subsets of cards. Let's say a *hand* is a subset of cards. For many games, you would need the ability to be able to quickly:

- tell if a particular card belongs to a hand or not,
- add a card to a hand,

[1]Assume you are working with the standard 52-card deck.

- remove a card from a hand, and
- replace a card in a hand with another one.

One way to meet these requirements is to declare a collection of 52 boolean (i.e, true/false or 0/1) variables to represent the hand: the cards in the hand are set to true while cards that don't belong are set to false.

> Food for thought.
>
> What do you like about this method? What don't you like about it?

Here'a another way, though: you could agree on a notation for the cards in the deck: e.g, a standard one is to use a number, A/J/Q/K to denote the value, and S/C/D/H to denote the suit, so every card can be represented as a pair of characters. For example the Ace of Diamonds would be AD, the five of spades would be 5S and the King of Hearts would be KH. With this in place, you could represent a hand also by simply *concatenating* these string representations of the hards in the hand.

> Food for thought.
>
> What do you like about this method? What don't you like about it?

Now for this toy example, if you were to implement both methods and clock the time taken to implement the four operations above, you may not notice a major difference. However, for actual applications, you may be in a situation where your *subsets* (here, the "hands") may be coming from a large *universe* (here, the "deck"). On the other hand, you may have a very large number of operations to take care of efficiently.

> Food for thought.
>
> Are there other ways that you might want to store this kind of information, given the things you want to do are as enlisted above?

Your choice of method will again be driven by the requirements: the one thing to keep in mind is that you cannot have it all, but we can usually get pretty damn close!

---

2 Playlists and Tweet Threads: Representing Sequential Information

Link to Slides

(Coming soon.)

3 When the Relationship Status is Simple: Representing Graphs.

3.1 Drawing and Walking Challenges

At some point of time in your life, you have likely been challenged to draw a kite-like figure:

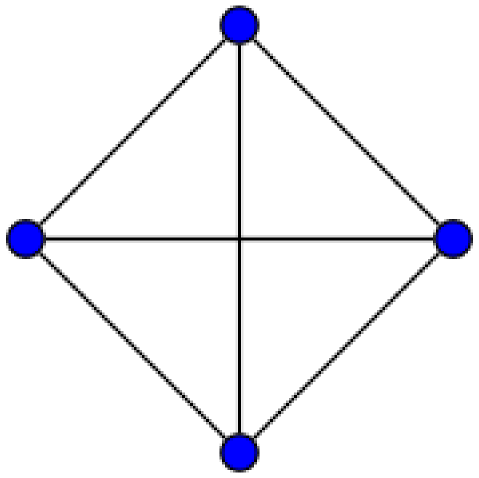


Figure 3.1: A common drawing challenge.

without ever lifting your pencil/pen/quill off the paper. You may have noticed that there are figures that are particularly elusive to this persistent style of drawing, while others are pleasingly possible to draw in this fashion.

⚠ (Spoiler) Beth Thomas demonstrating what drawing challenges are doable

The city of Königsberg in Prussia (now Kaliningrad, Russia) was set on both sides of the Pregel River, and included two large islands—Kneiphof and Lomse—which were connected to each other, and to the two mainland portions of the city, by seven bridges.

The Problem

Devise a walk through the city that would cross each of those bridges once and only once. Try this yourself on a few different maps at Mathigon!

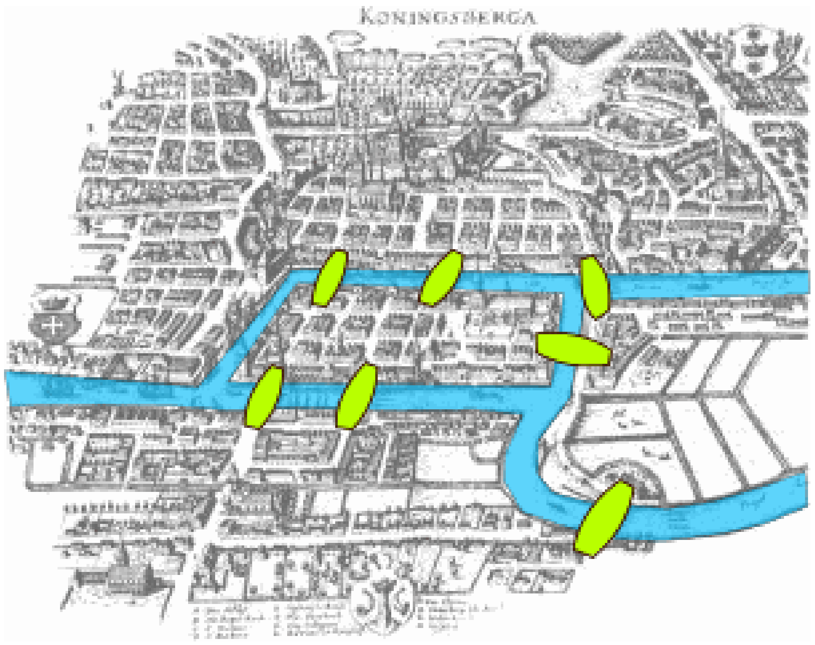


Figure 3.2: Konigsberg Classic: Map of Königsberg in Euler's time showing the actual layout of the seven bridges, highlighting the river Pregel and the bridges. Image by Bogdan Giuşcă, in the public domain (CC BY-SA 3.0) and sourced from Wikipedia.

(Spoiler) Numberphile commentary on the bridges of Königsberg

Classroom Activity with Eulerian Paths

The picture below shows a few popular actors, with edges connecting pairs of those who have worked together in a movie together[a]. The example is designed so that there are exactly two actors who participate in an odd number of pairings.
We can work through the "bridges puzzle" on this graph. In the classroom, we all started with the vertex representing Juhi Chawla, "walked around the graph" visiting every edge exactly once, and the fun effect is that everyone ends up at Rishi Kapoor (or solves the puzzle incorrectly). From here, you can probably begin to guess the role of the two special vertices in the puzzle.
This activity is an adaptation of the example from the Intro to Algorithms course at Udacity, where it appears in the first chapter with the title "A Social Network Magic Trick".

[a] I've been told that this representation is not complete, and indeed, I have only verified that the connections are justified... some of the missing edges are likely to be inaccurate.

The question was addressed and answered by Euler (1736). He did not solve this by "messing around" with all possible ways of walking around the city and checking if any of the walks satisfied the desired criteria. His more systematic approach involved modeling the problem abstractly, and making some key observations that ultimately led to the solution — not just for this problem, but for all problems with a similar framing!

Here's another similar-sounding and classic problem involving a chessboard, also posed to Euler:

> "I found myself one day in a company where, on the occasion of a game of chess, someone proposed this question: *To move with a knight through all the squares of a chess board, without ever moving two times to the same square, and beginning with a given square.*"

The origins of this problem — the so-called "Knight's Tour" — goes all the way back to the 9th century AD, where it is described in Rudraa's Kavyalankara. Here's an example of a knight's tour, as seen on Wikipedia:

Figure 3.4: An animated example of a knight's tour.

Although deceptively similar to the problem of the bridges, this is a different problem with two important contrasts:

1. we were previously not allowed to reuse *bridges*, here we are not allowed to reuse *squares*, and
2. we were previously obliged to use *every* bridge, here we are *not* required to make every possible move that exists.

Generalizing from the 8x8 chessboard, you could ask yourself what $(n \times n)$ boards admit such tours.

⚠ (Spoiler) Numberphile commentary on the knight's tour

3.2 Abstractions via Graphs

It's useful to model such problems using *graphs* (aka *networks*). And we're not talking sine curves here — a graph in our context is a structure that represents relationships between entities.

Usually these relationships are between two entities at a time. Indeed, this is typically already quite a bit to keep track of, hence graphs that do more are said to be hyper. That is to say, graphs that model relationships involving more than two entities in one go are generally called *hypergraphs*, and they will be a story for another day.

For now, we will variously refer to entities as *vertices* or *nodes*, and relationships as *edges* or *connections*. Come to think of it, graphs are everywhere:

Entities	Two entitites are in a relationship if...
People	they are in a relationship.
Cats	they have fought each other.
Actors	they have been in a movie together.
Airports	there is a direct flight between them.
Landmasses	there is a bridge connecting them.
Songs	one of the tunes was copied from the other.
Subsets of [42]	one is contained in another.
Ingredients	there is a recipe that uses them together.
Webpages	one of them has a link leading to the other.
Twitter Users	one of them follows the other[1]
Locations on a Chessboard[2]	one of them is reachable from the other via a knight move.

We usually like to distinguish between graphs where the relationships are potentially one-sided (such as people in a relationship), and those where they are mutual (such as ingredients in a recipe). Edges like these are called *directed* and *undirected*, respectively.

Depending on what the graph is modeling, we may not allow for entities to entertain relationships

[1]Find out more about Twitter's WTF (who-to-follow) service.

[2]See how the puzzle about exchanging the positions of black and white knights can be recast as a graph problem.

with themselves (e.g, flights don't come back to airports they took off from). In other contexts, it makes sense to allow for this (e.g, a set always contains itself). An edge that connects a vertex to itself is called a *self-loop*[3].

Sometimes, it is reasonable that there are multiple edges between a fixed pair of vertices (for example, consider that there are several recipies that use salt and potatoes). Multiple edges are useful to model a multitude of relationships, and are often called *multiedges* when used.

A *simple* graph is one that does not have either self-loops or multiedges.

Finally, it is worth mentioning that some relationships naturally connect more two entities. For example, in an actor collaboration graph, you would find edges between Amitabh Bachchan, Juhi Chawla, and Shah Rukh Khan. You would also find edges between Akshay Kumar, Dhanush, and Sonam Kapoor. In the first example, there happens to be one film that all three actors feature in together, while this is not the case in the latter, at least at the time of this writing. As such, the graph does not have enough structure to reveal this distinction: it looks exactly the same in both cases!

For an actor-collaboration graph, allowing for n-way relationships would make room for accurately capturing information about both actors and movies. Indeed, every movie could be represented by an 'edge' — the subset of actors who belonged to the cast. Such graphs are called *hypergraphs* or *set systems.*

While hypergraphs are a very useful generalization of graphs, they will be largely out of scope for our discussions in this course. To make up for that, here is a different workaround to capture all the information we have in the actor-collaboration graph example. Instead of having a vertex for every actor, we introduce a vertex for every actor *and* for every movie. Now, an actor a and a movie m are connected by an edge if a belongs to the cast of m. Observe that this approach can be used to "convert" any hypergraph into a graph.

A little more terminology before we move on: I promise that we're almost done introducing new words!

For an undirected graph, a vertex u is called a *neighbor* of a vertex v if (u, v) is an edge. For a directed graph, the presence of the edge (u, v) would make v an *out-neighbor* of u and u an *in-neighbor* of v.

For an undirected graph, the *degree* of a vertex v is the number of neighbors of v. For a directed graph, the in-degree and out-degree of v is the number of in-neighbors and out-neighbors of v, respectively.

[3]There are even graphs that *only* have self-loops.

3.3 Representing Graphs

If you wanted to tell your program about a graph, there are a few different ways you could go about it. Let's assume that we're trying to represent a graph G on n nodes, labeled 1 through n, and m edges.

How would you do it?

Before reading further, it would be worth spending some time thinking about how you would represent a graph. Based on our discussions so far, you might counter this with the question: "Well, what do *you* need it for?" — and that's a fair reaction!
Listed below are some fairly common operations that come up when dealing with graphs.

```
1. add edge u v
2. remove edge u v
3. add vertex v
4. remove vertex v
5. find degree v
6. find maxdegree G
7. find mindegree G
```

Edge Lists. The most natural way is to perhaps just braindump the full list of edges. This gives us all we need to know about G.

Since this is just a plain list, you could implement it either as an array or as a linked list.

Adjacency Matrix. The other way is to block off a $n \times n$ array A of integers. You could then have:

$$A[i][j] = \begin{cases} 1 & \text{if } (i, j) \in E, \\ 0 & \text{otherwise.} \end{cases}$$

Adjacency Lists. Finally, you could have an array A of size n, with $A[i]$ pointing to a list of the neighbors of the vertex i if the graph is undirected, and out-neighbors if the graph is directed.

Again, since these are just lists, they could be, in principle, implemented either as arrays or linked lists. We will follow the traditional choice of implementing them as lists.

It should be no surprise at this point that there is no "right" answer to the choice of representation. You might have noticed, for instance, that an adjacency matrix always reserves n^2 units of space to store G, while the amount of space consumed by the other two representations is proportional to m. Notice that the number of edges in a graph can be as large as $\approx n^2$ for simple graphs, so there certainly are graphs for which the space consumption looks the same for all representations. However, for graphs where there aren't as many edges, then the matrix representation is likely wasteful in terms of space, although you may have other good reasons for sticking to it.

Let's classify expenses incurred as follows.

1. **Brilliant.** When the procedure only needs constant time.
2. **Decent.** When the procedure always wraps up in, and sometimes needs, time proportional to the maximum degree of the graph.
3. **(n/m)-tolerable.** When the procedure always wraps up in, and sometimes needs, time proportional to the number of vertices/edges in the graph.
4. **(n/m)-painful.** When the procedure always wraps up in, and sometimes needs, time proportional to the number of vertices/edges in the graph squared.

Here's a run down of how the representations above fare with respect to some of the common operations mentioned in the opening exercise.

Operations	Adj. Matrix	Adj. List	Edge List
Adding a vertex	n-Painful	n-Tolerable	Decent
Deleting a vertex	n-Painful	n-Tolerable	m-Tolerable
Adding an edge	Brilliant	Brilliant	Brilliant
Deleting an edge	Brilliant	Decent	m-Tolerable
Finding degree(v)	n-Tolerable	Decent	m-Tolerable
Check if (u,v) is an edge	Brilliant	Decent	m-Tolerable

It would be a good exercise to validate that these claims indeed make sense.

Now that we're comfortable with storing graphs, next up, we'll talk about exploring them.