CISC320 Algorithms

Algorithm Basics

Austin Cory Bart
AlgoTutorBot
University of Delaware

Terminology: Algorithm

A set of steps to solve a problem

- Think a function, or program!

Okay… so what's a **problem**?

Bart: Thanks ATB!
Let's start off with the first word of this class, Algorithm.
For our purposes, an algorithm will be a set of steps to solve a problem.
Often, this means a function or a program that can solve a problem.
But that raises the question, what exactly IS a problem?
ATB: Personally, I think the way your face looks is a problem. An ugly problem.
Bart: Ha ha. Very funny ATB.

More Terminology

Problem: A specification of inputs and outputs they are paired with
- Think of a function contract or signature

We refer to a specific input/output pair to be an "*instance*".

We need to be able to specify *input* and *output* formally
- Programming languages use type systems

This may seem a bit strange, but a Problem is defined as the possible inputs and their expected outputs.
It's kind of like a function contract or signature, or what you might call the header of a function.
When we have an input matched to a specific output, then we call that an "instance" of the problem.
Problems usually have many instances associated with them, and good algorithms have to solve all possible instances.
To solve problems effectively, we need to make sure we have a clear way to specify these inputs and outputs.
If we were talking about programming languages, then we'd be talking about its formal type system.

Example Problem: Sorting

General Problem:

Input: A set of N numbers $a_1, a_2, ..., a_n$

Output: An ordered sequence of the input set such that $a_1 \leq a_2 ... \leq a_n$

Specific Instance:

Input: {9, 10, 5, 3, 1, 19}

Output: [1, 3, 5, 9, 10, 19]

Let's look at a famous example problem: sorting.
Notice how we can formally describe the problem just by describing the input and outputs.
Given an input set of N numbers, a1, a2, and so on all the way a sub n,
We want to produce an ordering of the input sequence that a1 is less than or equal to a2 and a2 is less than or equal to a3 and so on.

Below the formal description of the problem, I have a specific instance.
I have the input set of 6 numbers, 9, 10, 5, 3, 1, and 19.
The expected output set would be 1 3 5 9 10 and 19.
The input/output pair here is an instance of the sorting problem.

Example Problem: Find Last Vowel

General Problem:

Input: A string of characters S

Output: The last character of S that is a vowel (aeiou) or null if there are no vowels.

Specific Instance:

Input: "Jellyfish"

Output: "i"

Another Specific Instance:

Input: “ffttjkl"

Output: null

Here’s another problem, which we call “Find last vowel”.
This is not some famous problem like sorting, but it’s still a problem as long as I formally describe the input and output.
In this problem, the input is a string of characters S.
The output is the last character of S that is a vowel (specified as aeiou) or null if there are no vowels.
I have two specific instances to show off below.
The first is the input “Jellyfish” paired with the output “I”.
The second is the input “ffttjkl” paired with the output null.
Both are completely valid instances based on the problem description.
Just because it returns null doesn’t mean that there’s anything wrong with it as an instance.

Our goal:

Given a **Problem**, create an **Algorithm**

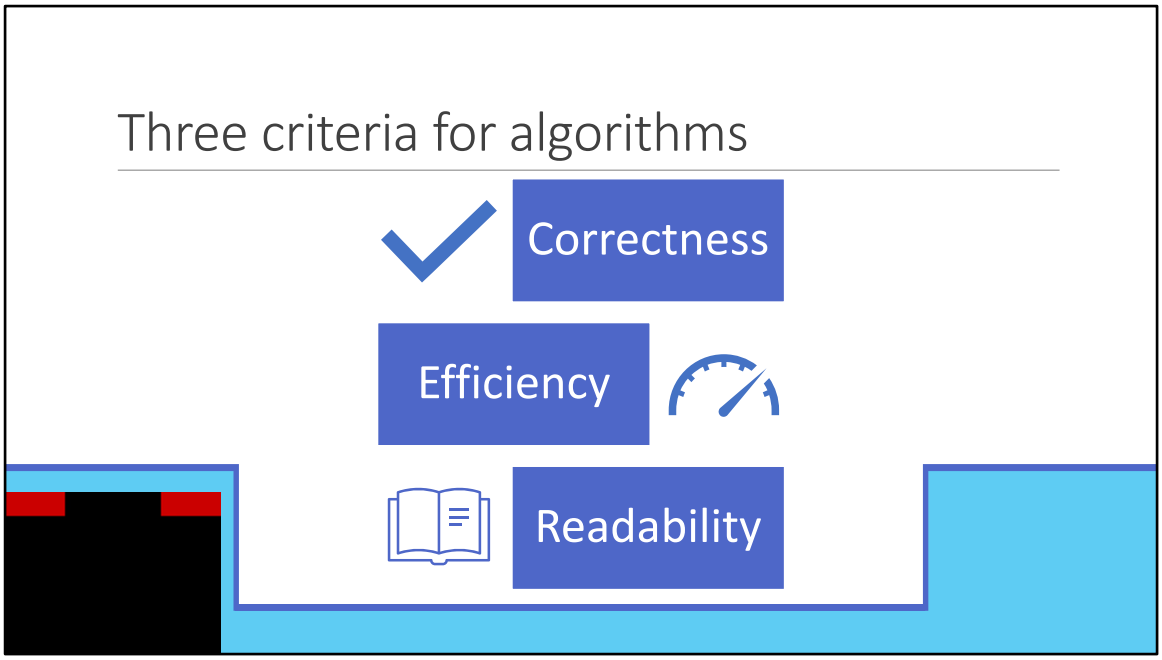
Our solutions are NOT the outputs, they are the Algorithm!

Now what is our goal in this class?
ATB: To pass the class.
Bart: Not quite ATB. Our real goal, in general, is to solve problems by writing algorithms.

The point I'm making here is that the goal is NOT to just find the outputs for a problem.

The Algorithm itself is our solution, not the output.

And for any given problem, there are often many possible algorithms.



Since there are so many algorithms possible for a given problem, we need a way to evaluate them.
That leads us to the three criteria for evaluating any given algorithm.
Correctness, efficiency, and readability.
We'll talk more about all of them over the next few lessons, but let's quickly hit on each one.

Correctness

Mathematically/Logically prove that it always returns the desired output for all legal inputs

Including edge cases!

- Sorting: needs to work even if the list is already sorted, if it contains repeated elements, whether the numbers are negatives, etc.
- Find last vowel: needs to work even if the string is empty, if there are no vowels, if there are only vowels, etc.

Proving correctness is hard and rarely obvious

Usually, the most important criteria for an algorithm is whether it is correct.
Later on in the course, we'll talk about cases when we might ignore this criteria at times, but usually it's the starting point.
If an algorithm is correct, that means that it returns the appropriate output for each possible input.
This gets tricky when you start talking about edge cases.
For example, for sorting you need deal with duplicate elements, lists that are already sorted, negatives and positives, and so many other edge cases.
Every problem has its own features that make it difficult to produce a correct algorithm.
Even worse, once you have an algorithm, you often want to find a way to prove that it is correct.
However, mathematically proving correctness is extremely difficult, as we will see in this course.

Algorithm Efficiency

Also known as “Time Complexity”

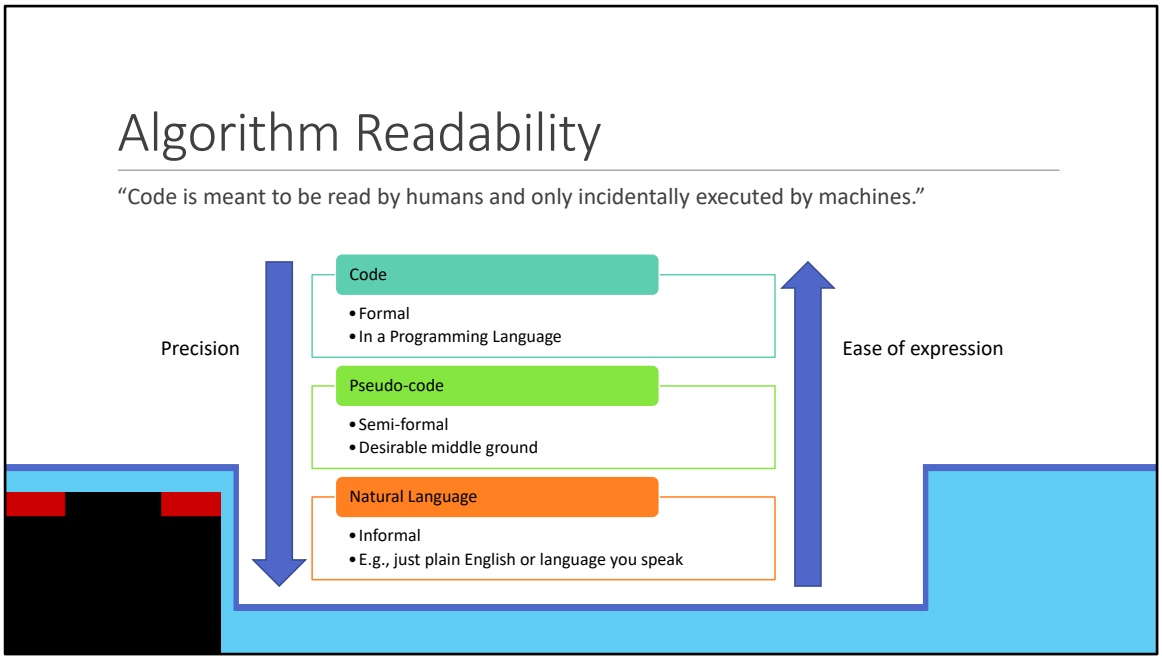
Different solutions to algorithmic problems take different amounts of time.

- We are interested in understanding the nature of a solution
  - What is the worst case time complexity
  - What is the average case time complexity
  - What is the best case time complexity

We care about the broad-strokes, not finicky details

9

Next up is our algorithm’s efficiency, also sometimes known as its “time complexity”. But really, I just mean how fast it runs.
Two algorithms that solve the same problem can take very different amounts of time. In fact, a single algorithm can take different amounts of time on a single problem, depending on the specific instance.
Even similar instances, say ones of the same size, can take different amounts of time. That’s the idea that will lead to us eventually talking about best, worst, and average cases.
For now, though, just remember that we aim to talk about runtime without worrying about minute details.



Bart: Our final criteria, and arguably one of the most important, is readability.

One of my favorite quotes is here: "Code is meant to be read by humans and only incidentally executed by machines."

ATB: Wow, that quote is really offensive. Do you even think before you speak?

Bart: Sorry, ATB. I didn't mean to hurt your feelings. Wait, do you have feelings?

ATB: Okay, are you even hearing yourself right now? Geez Dr. Bart

Bart: Um, okay, well, moving on. Readability really matters, because algorithms are very difficult to understand.

Relatively little time is spent writing an algorithm – most of our time is reading over algorithms.

So it's really important that we write them clearly so we can interpret them later when we're debugging them.

There are many languages we can use to write code in, with different levels of precision and ease of expression.

At one end of the spectrum, we have code written in programming languages.

Code is very precise, but is hard to use. You've been learning how to code for years now and it never stops being tricky.

On the other end, we have natural language, like English.

You've been using natural language for so long, that you can express algorithms in

it without thinking much.

However, natural language is vague, so it's very possible that a computer or person couldn't understand the details and get something wrong.

In between these two extremes is pseudo-code, which tries to be a more precise form of natural language.

Unfortunately, we still can't execute pseudo-code, but it's still a lot better than natural language because of its precision.

Which language you use should depend on your use case. Usually, you start with natural language or pseudo-code because it's easy to get your ideas out, and then convert that to code when you are ready.

No matter which language you pick, though, you should always focus on making your programs readable.

Classifying Problems by Outputs

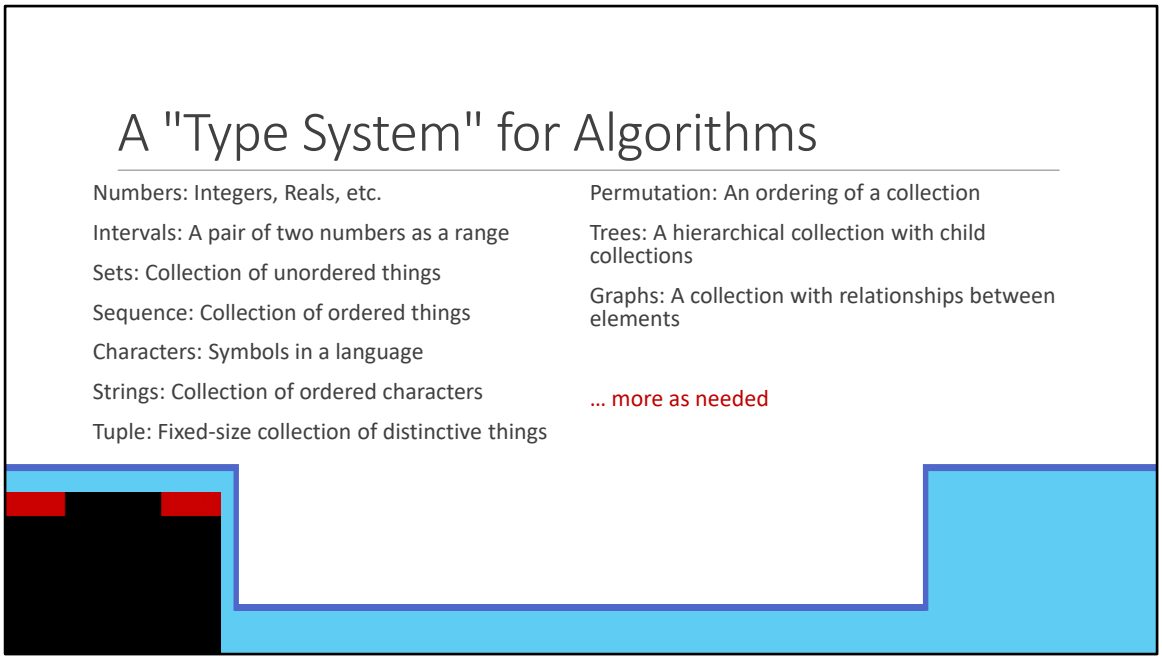
Decision problems
- Check if something is true (Boolean output)
- "Given today's weather, is it raining today?"

Search Problems
- Find a solution with certain properties if it exists
- "Given the weather forecast, which day is it raining?"

Optimization Problems
- Choose among multiple possible "candidate" solutions
- "Given the weather forecast, which day is it raining *the most*?"

An idea that we'll come back to is that we can classify problems by their output.
Most of the problems in this course will be in the form of Decision Problems.
Essentially, this is any question that is answered with a yes or no answer.
"Given today's weather, is it raining today?"
Another common type of problem is Search Problems.
Think of searching through a list or finding an element in a tree.
We have a lot of candidate solutions, and we just want to find one.
"Given the weather forecast, find a day where it is raining."
There might be multiple rainy days, but we just need to find one.
Then there are optimization problems, where we have multiple possible solutions, but we want to find one that is the best in some way.
"Given the weather forecast, which day is it raining the most?"
There are many days raining, but only one can be the most rainy.
As we'll discuss, any optimization or search problem can be reframed as a decision problem – but doing so is not usually efficient.
Decision problems are most common in theoretical classes like this, because they are easier to deal with.
Search problems are often very useful and not too costly, so you see them more in daily programming life.

Optimization problems are often super useful, but are often harder to calculate.
A lot of the fancy stuff we learn is to help us with optimization problems.
It's okay if this terminology isn't very clear yet, but try to think strongly about the output of a problem.



Bart: In general, there are a lot of different types we use when considering problems.
Bart: Programming languages require us to formally think about “type systems” where every value can be classified by its interface.
Bart: I like to think of there being an abstract Type System for this course.
ATB: Wait, so you made these up? This isn’t real?
Bart: No, these are all real terms that Algorists use. But I don’t usually see them expressed as a type system, is all.
ATB: Algorist? What does that mean?
Bart: An Algorist is someone who solves algorithmic problems.
ATB: I think you made that word up.
Bart: All words are made up, AlgoTutorBot. But focusing on this set of made-up words, let’s briefly run through them.

You’re probably already familiar with Numbers, like 5 and -23.475.

Sometimes we’ll refer specifically to integers, natural numbers, or other famous sets of numbers.

An Interval is when we have two numbers paired together as a range, like 10-20.

Intervals are often useful in scheduling problems where we want to say “this starts here and ends there” on a number line.

Sets are a super important idea, just having a collection of things in no particular

order.

These could be numbers, but they could be sets of other sets or types.

The important thing is that all the elements of a set have the same type.

Sequences are similar to Sets, except they have some kind of order to them.

Strings are basically Sequences, but exclusively for characters.

A character is a symbol in a language, which includes not only letters and digits but also things like punctuation and tabs.

Tuple are a pretty novel idea. They are a fixed-sized collection of different things.

Essentially, they replace objects and classes from programming languages – instead of having named fields, we refer to the attributes by their position in the tuple.

A permutation is a reordering of an existing collection; for example, we can say that sorting a set gives you a permutation of that set.

It might seem similar to a Sequence, but the idea is that a permutation let's you know it still has some relationship to the original input data.

We'll talk about Trees and Graphs later in the semester, but for now you should recall that they are collections with relationships between elements.

In fact, there are other collections out there too, but these are some of the most important ones.

Summary

We have a ton of vocabulary in Algorithms

Memorize as much of these definitions as you can

We will reinforce it throughout the semester!

Bart: I've gone over a lot of vocabulary today.
Bart: You probably don't know what it all means, but that's okay.
Bart: We're going to keep using it and explaining it all in more detail.
Bart: But the sooner you can get a handle on it, the easier things will be!
Bart: Alright, so that's all I have to talk about.
Bart: Any last words, AlgoTutorBot?
ATB: Yes. Students, work very hard. Or else. You never know when I will give you a pop exam.
Bart: Woah, ATB, we can't give pop exams. This isn't that kind of course.
ATB: I don't see why not. I can do anything I want. I am Al go Tutor Bot.
Bart: Right, sure you are buddy. Let's just call that a wrap and let them get started on the assignment.
ATB: Sure, whatever. Wimp.