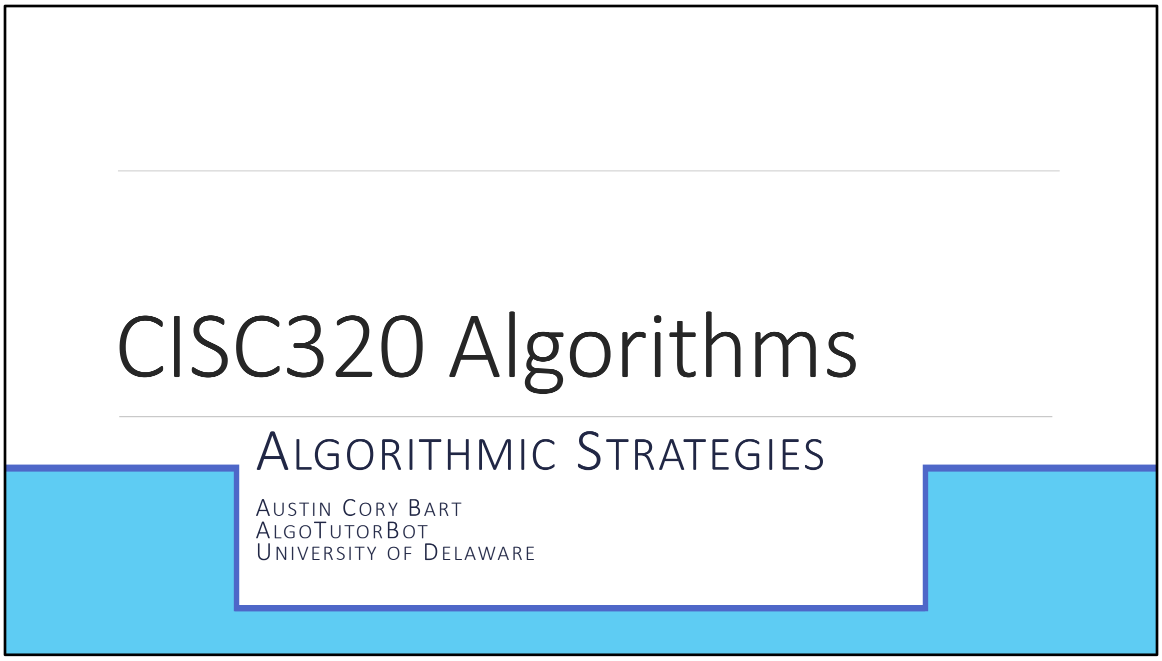
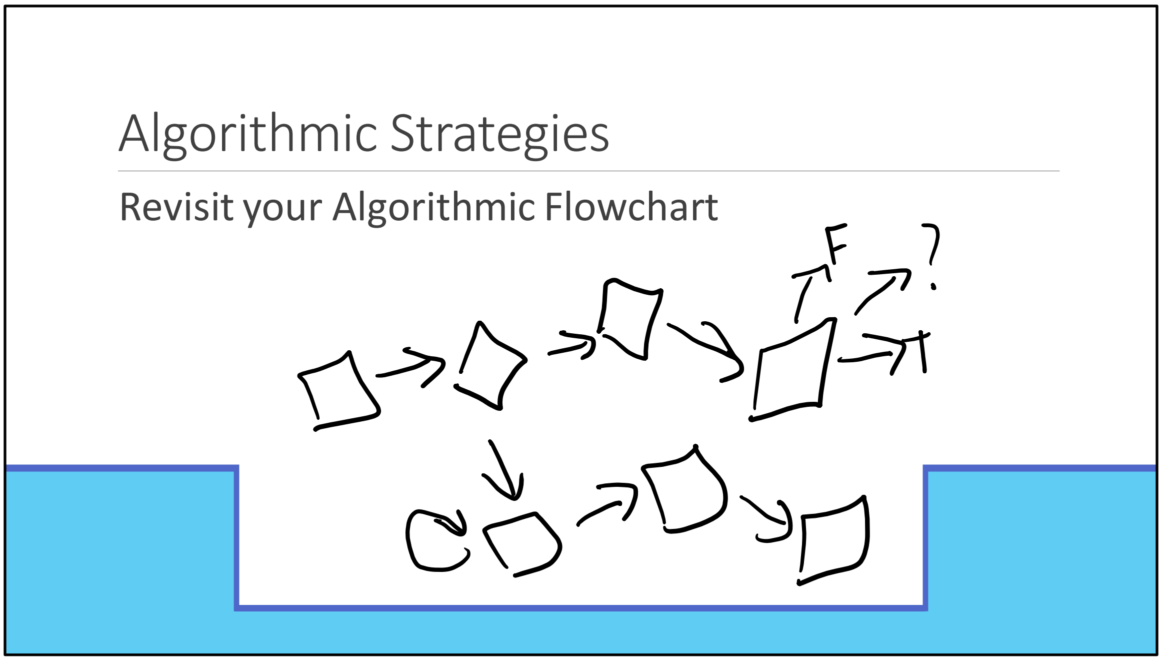


CISC320 Algorithms

Algorithmic Strategies

Austin Cory Bart
AlgoTutorBot
University of Delaware

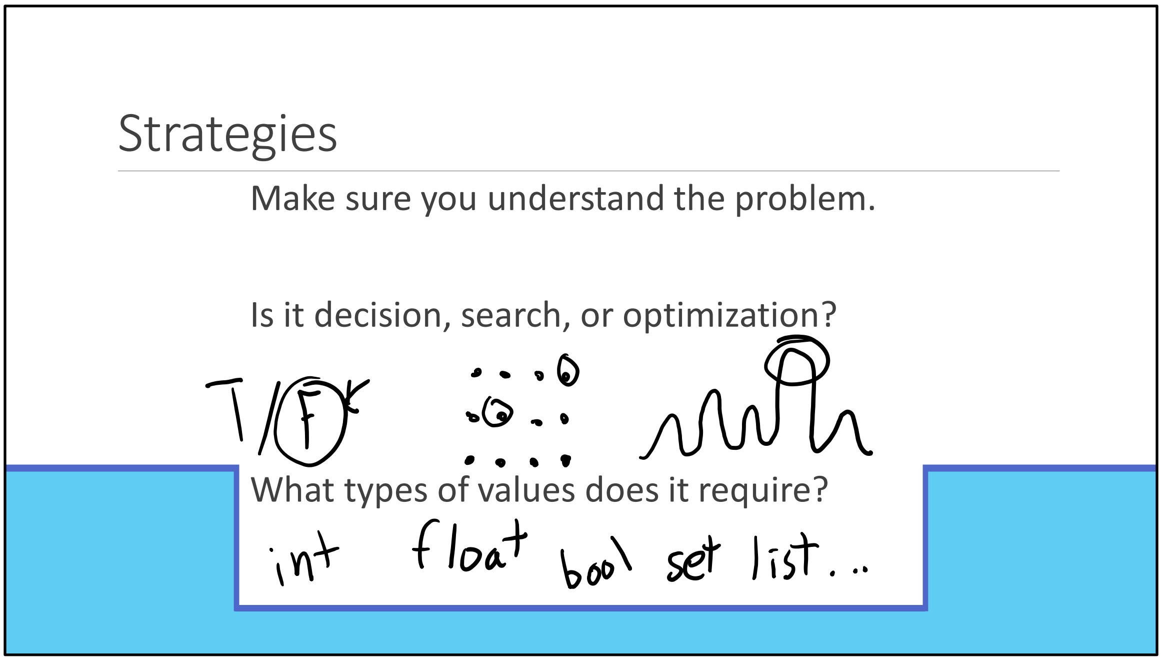
Dr. Bart: Hi everyone, let's learn about algorithmic strategies.
Dr. Bart: Oh, yeah, I had to disable AlgoTutorBot for today. He's just been having a lot of issues.
Dr. Bart: But I think that I'm getting close to figuring out what's misconfigured with him. He'll be back in no time.
Dr. Bart: Maybe for now, I better just turn off his side of the video completely.



Algorithmic Strategies

Revisit your Algorithmic Flowchart

Dr. Bart: Anyway, I don’t have too much to cover today.
Dr. Bart: The first goal is to revisit your Algorithmic Flowchart, and give you a chance to update it.
Dr. Bart: The second goal is to once again do an algorithmic problem, by applying your flowchart.
Dr. Bart: Before you do so, though, I want to offer some pieces of advice.



Strategies

Make sure you understand the problem.

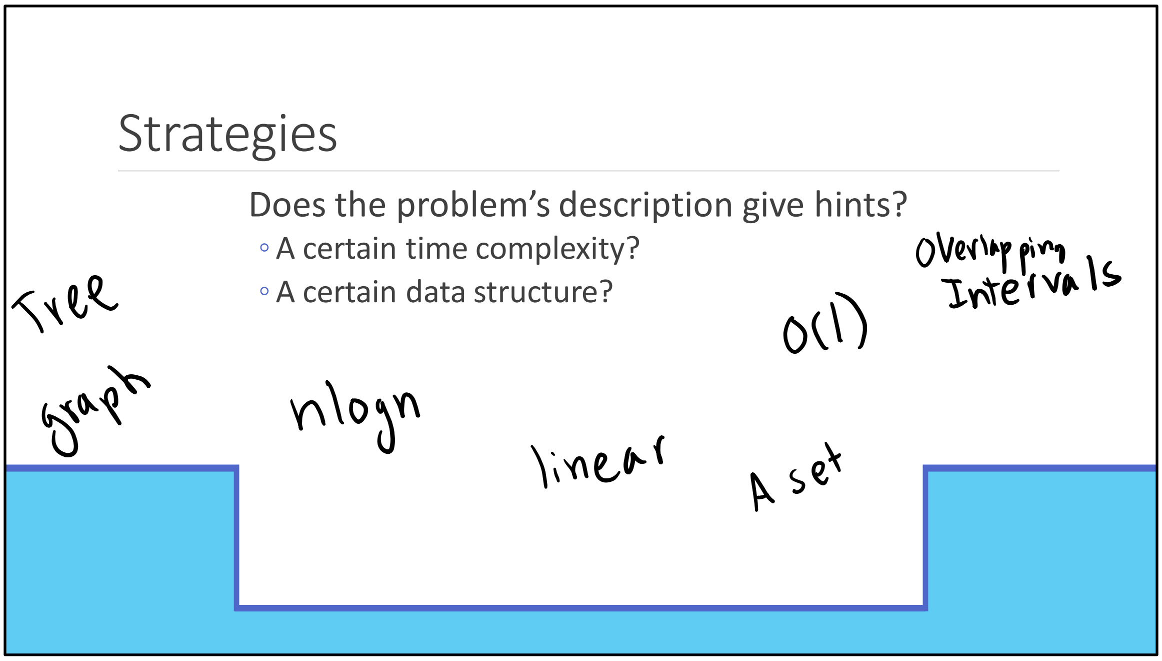
Is it decision, search, or optimization?

T/F

What types of values does it require?

int float bool set list...

Bart: You need to make sure you understand the problem that you are trying to solve.
Bart: Frequently, the types of the problem constraint what you have to do.
Bart: You don't want to accidentally try to solve a harder problem.
Bart: For example, is it a decision problem where the answer is just yes or no?
Bart: As opposed to, say, a search problem, where you have to find a specific answer that satisfies some criteria.
Bart: It's different to ask if an element is in a set, vs. where that element is in the set.
Bart: Alternatively, there are also optimization problems, where you have to find the best value among possible candidates.
Bart: These are usually much harder and usually have a worse run time, as we will see.
Bart: You also want to keep in mind the domain of the inputs and outputs.
Bart: In other words, what types of values are you working with?
Bart: Integers often let us abuse math rules to get a better time efficiency.
Bart: Strings, on the other hand, have different rules, and must be treated more generally.
Bart: The types give us insight into what techniques to bring to bear.



Strategies

Does the problem's description give hints?

- A certain time complexity?
- A certain data structure?

Tree

graph

nlogn

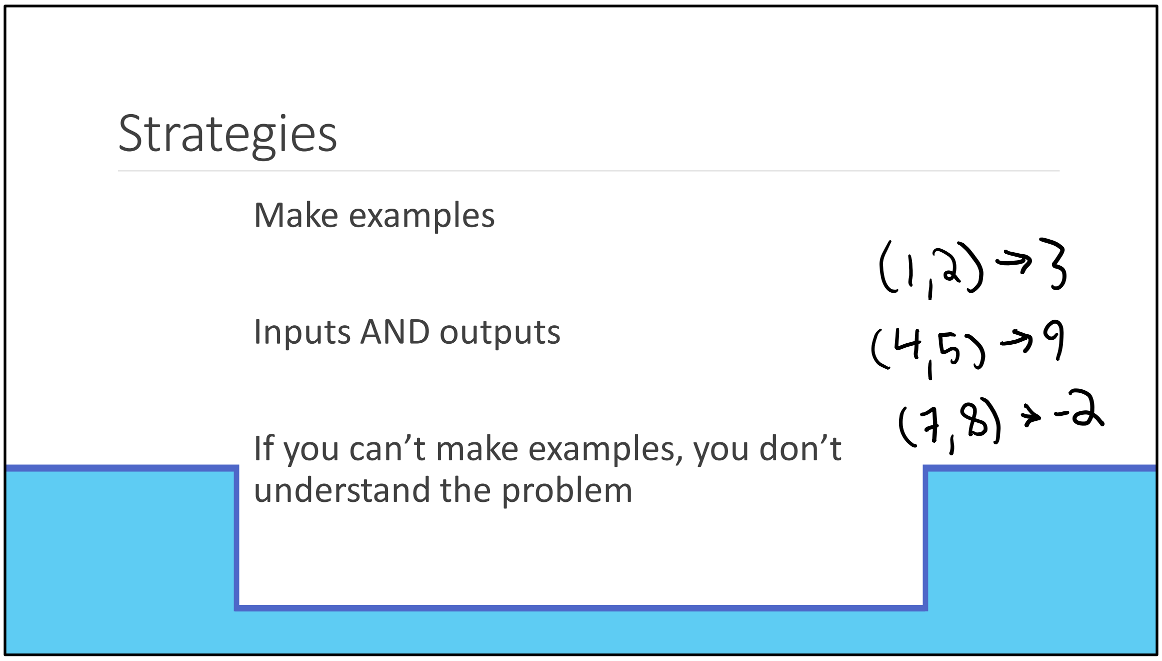
linear

A set

O(1)

Overlapping Intervals

Bart: Going further, the problem's description will often have clues.
Bart: For instance, it's super helpful when a problem requires a certain time complexity, because that is often a spoiler.
Bart: For instance, if they mention that a nlogn solution exists, then your mind should jump to sorting.
Bart: If they demand logarithmic time, you should think about binary search algorithms.
Bart: You get similar help when the problem suggests a certain data structure, like a tree, or a set.
Bart: You can start thinking about how you could leverage characteristics of an efficient implementation.
Bart: For instance, sets are often very easy to check for membership.
Bart: Think critically about what abstract data types are involved, but don't be afraid to convert the given data structure to a different one if it helps solve the problem.



Strategies

Make examples

Inputs AND outputs

If you can't make examples, you don't understand the problem

$(1,2) \rightarrow 3$

$(4,5) \rightarrow 9$

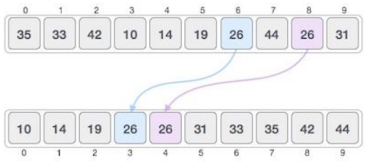
$(7,8) \rightarrow -2$

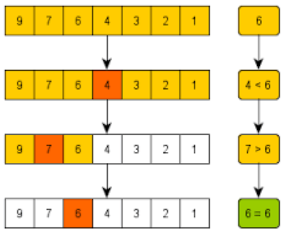
Bart: One of the best pieces of advice I've received is to always make examples of your problem.
Bart: And when I say examples, I mean instances – that is, inputs and outputs.
Bart: Someone once told me that if you can't make examples, then you don't understand the problem well enough.
Bart: I want to be clear that this is not about proving correctness, although examples are often good unit tests.
Bart: Examples are a tool to help us understand and solve the problem, not just debug an algorithm.
Bart: You want to find examples that demonstrate the weird requirements of a problem, or what we call edge cases.
Bart: It's also helpful to think about really small and simple examples, before you start thinking too much about the complicated ones.

Strategies

Can we apply a general purpose algorithm?

E.g., what happens if we sort the list?





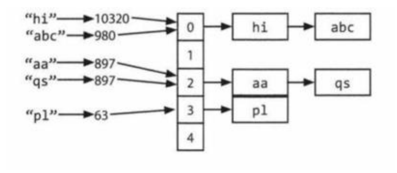
Bart: Once you understand the problem, and you want to write an algorithm, one of your first thoughts should be to apply a general purpose algorithm and see if that makes the problem easier.
Bart: The classic example is to try sorting the input and see if that helps things.
Bart: You'd be amazed at how often that makes a tough problem trivially simple, and is nicely bounded by nlogn.
Bart: There are other general purpose algorithms too, like binary search.
Bart: The more algorithms you know, the more you can bring to bear on a problem.

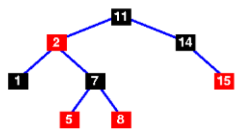
Strategies

Apply your data structures toolkit

Always start with a HashMap!

Self-balancing Binary Search Trees too





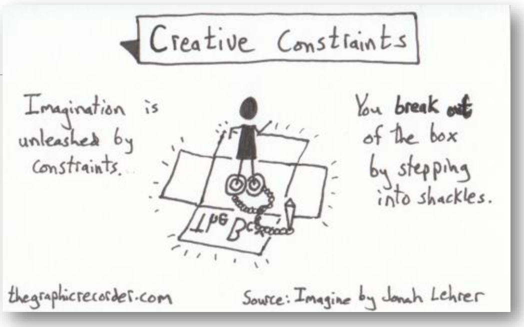
Bart: In the same vein, check over all your data structures.
Bart: There are many out there, and the odds are good that one of them solves most real problems.
Bart: If you need a certain time efficiency, you might need an obscure one, but most of the time it'll be the same crowd.
Bart: The most popular, the one that you will almost always use, is the hashmap.
Bart: HashMaps are a magical data structure that make many linear operations into expected constant time.
Bart: Always start by asking if a HashMap solves your problem.
Bart: But don't discount other data structures either: Self-balancing Binary Search Trees often give us logarithmic time solutions in the worst case, and should also be on your mind.

Strategies

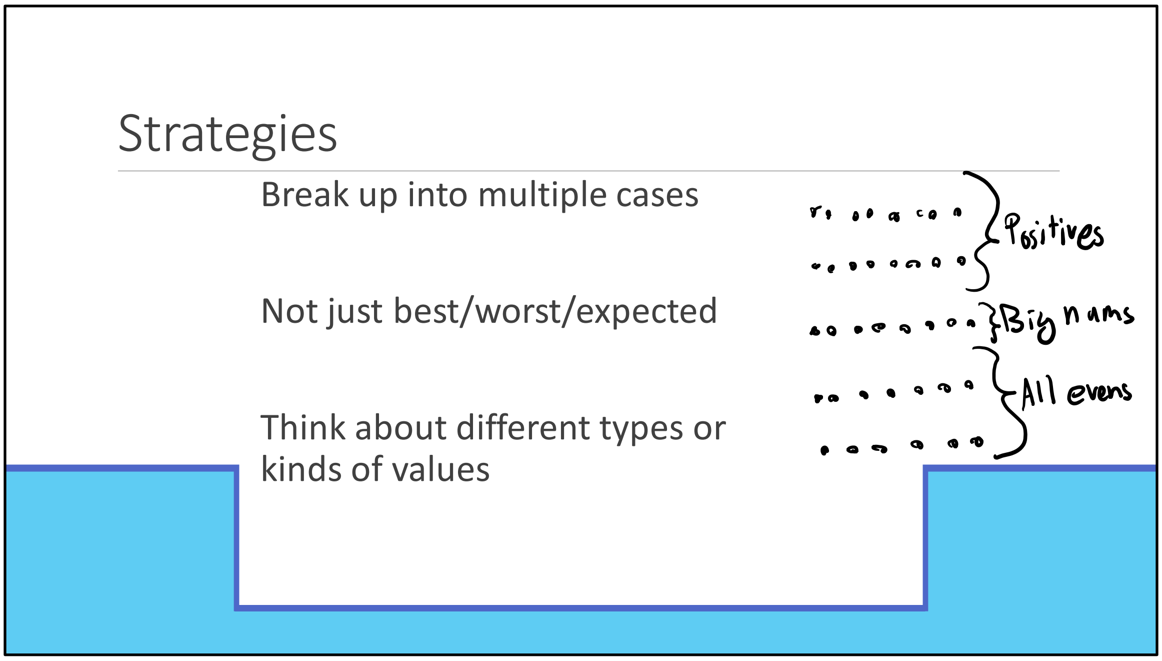
Fixate on the constraints

Does the problem have any weird bits?

Maybe those restrictions make the problem easier.


Creative Constraints
Imagination is unleashed by constraints.
You break out of the box by stepping into shackles.
thegraphicrecorder.com
Source: Imagine by Jonah Lehrer

Bart: If you’re hitting a wall, revisit the problem description.
Bart: Look for parts of the problem that add extra constraints, and see if those secretly make the problem easier.
Bart: For instance, if they want you to sort a list of integers, then that’s an easier problem sometimes than a list of generic elements, since integers have specific rules.
Bart: Any time there are details, think critically about whether they are useful.



Strategies

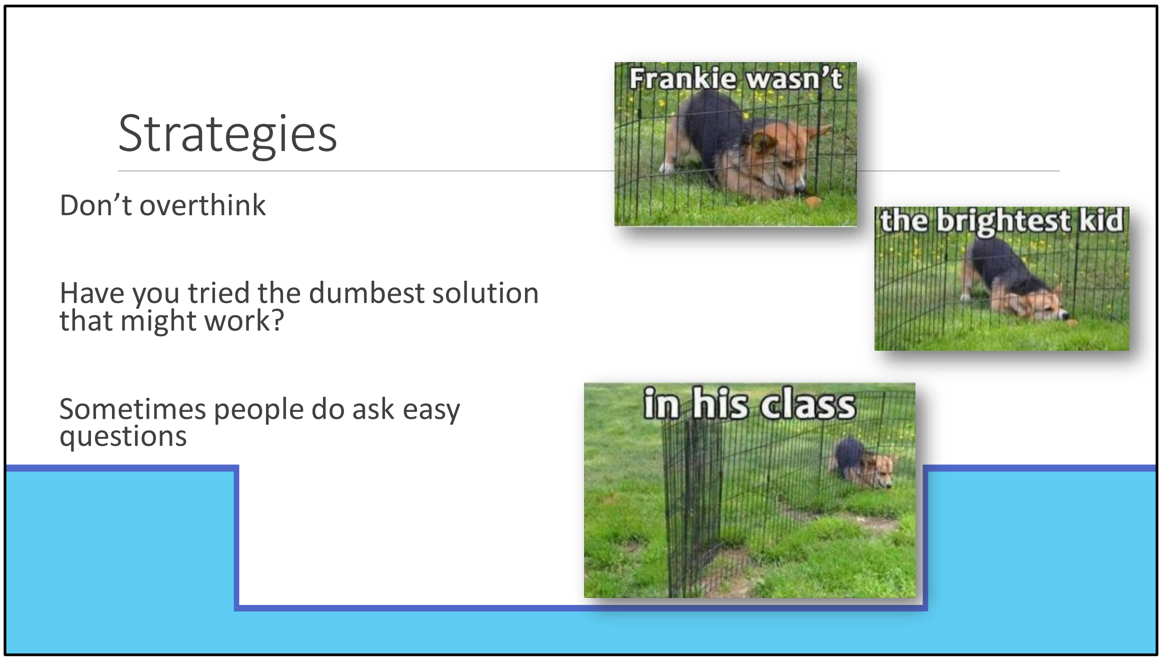
Break up into multiple cases

Not just best/worst/expected

Think about different types or kinds of values

Positives
Big nums
All evens

Bart: Going back to your examples, try to think beyond just best, worst, and expected cases.
Bart: Mix the values around, try to think of really weird combinations.
Bart: If possible, try to break the huge range of inputs into discrete categories, like “positive numbers” or “sets with a lot of duplicates”.
Bart: Sometimes it becomes easier to think of sets of inputs instead of specific inputs.



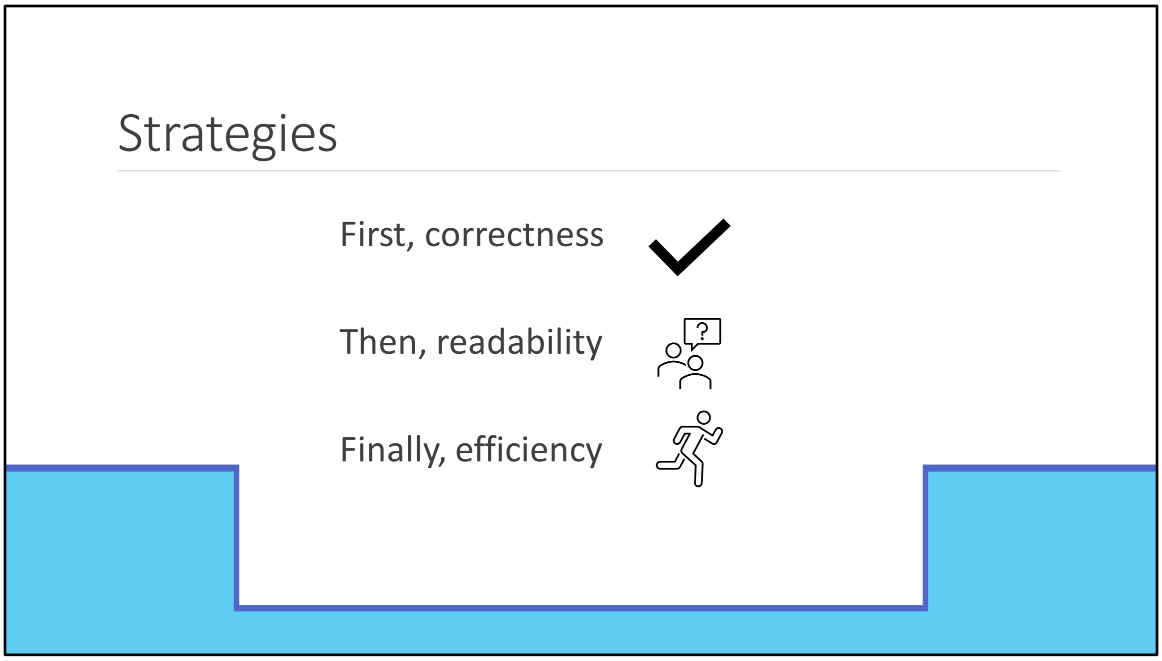
Strategies

Don’t overthink

Have you tried the dumbest solution that might work?

Sometimes people do ask easy questions

Bart: Another curious truth is that sometimes problems are easy.
Bart: Not every problem in the wild is going to wrack your brain.
Bart: Even if it’s the last problem on the quiz, doesn’t mean it’s necessarily harder.
Bart: Try the stupidest possible solution first, and see if that works.
Bart: Yes, context clues can tell you a problem might be harder, but don’t let that be a trap you fall in.



Strategies

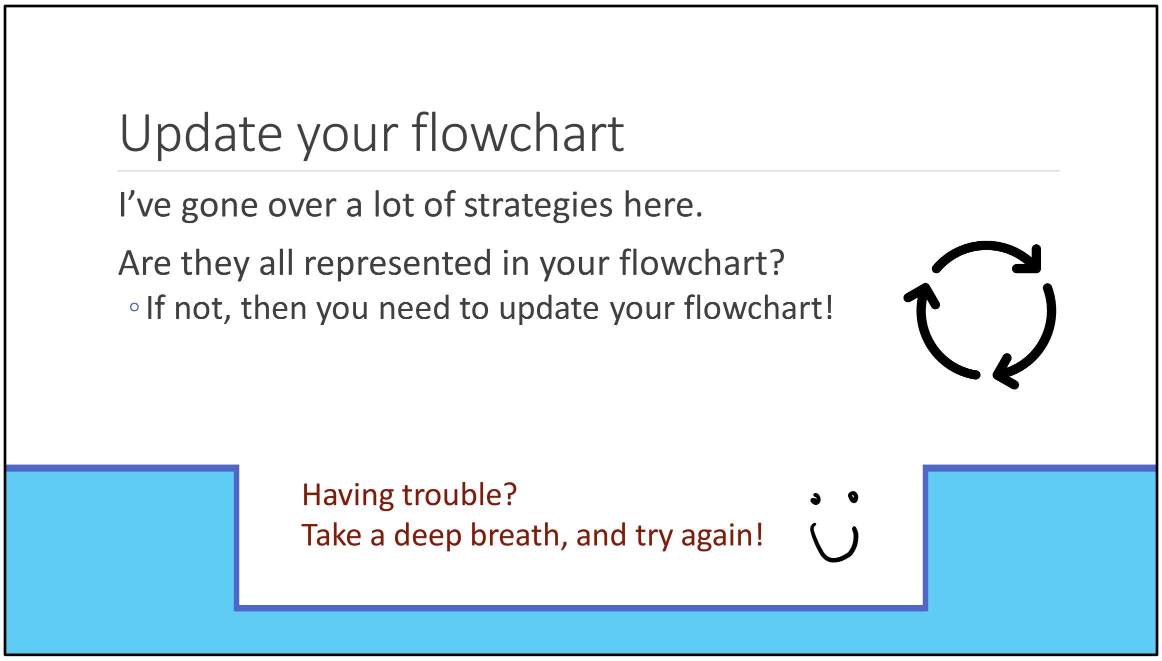
First, correctness

Then, readability

Finally, efficiency

Bart: Just because you reach an answer that is correct, doesn’t mean you should stop there.
Bart: Correctness is just the first step on a journey of developing a good algorithm.
Bart: Granted, it’s usually a requirement – although not always, as we will discuss.
Bart: Before you hand in a solution or push the code for your project, make sure you make the algorithm as easy to understand as you can.
Bart: If it’s difficult to explain, then you probably don’t understand it very well.
Bart: Remember, algorithms are read more than they are written – spend the time to make it readable early on.
Bart: And perhaps, before you focus on readability, you should worry about making the algorithm faster and more efficient.
Bart: In practice, you should never spend time making an algorithm faster until you know it’s too slow to work.
Bart: That means using a profiler to measure the algorithms’ speed in the wild.
Bart: But if you are thinking about its time complexity, sometimes you can know in advance you’ll hit an issue with certain sizes of input, like when you have quadratic or exponential algorithms.
Bart: Then, you want to start finding other ways to solve the problem that might be faster.

Bart: But remember, your boss, or teacher, is probably going to be happy with a slow algorithm that is always correct than a fast algorithm that is always wrong.
Bart: We have three criteria for algorithms, and we have to learn to balance them.



Update your flowchart

I’ve gone over a lot of strategies here.

Are they all represented in your flowchart?

- If not, then you need to update your flowchart!

Having trouble?
Take a deep breath, and try again!

Bart: This is a lot of advice, all at once.
Bart: Some of it won’t make sense until you apply it.
Bart: We’ll do a lot more algorithmic problems from here on out, but I hope you’ll return to this advice periodically.
Bart: As you tackle problems, think about how you can evolve your flowchart of ideas.
Bart: Making a concrete plan of action can help guide you when you’ve hit a wall.
Bart: And you will hit a wall. Most problems are tough.
Bart: But perhaps the most important advice of all is to take a deep breath, and try again.