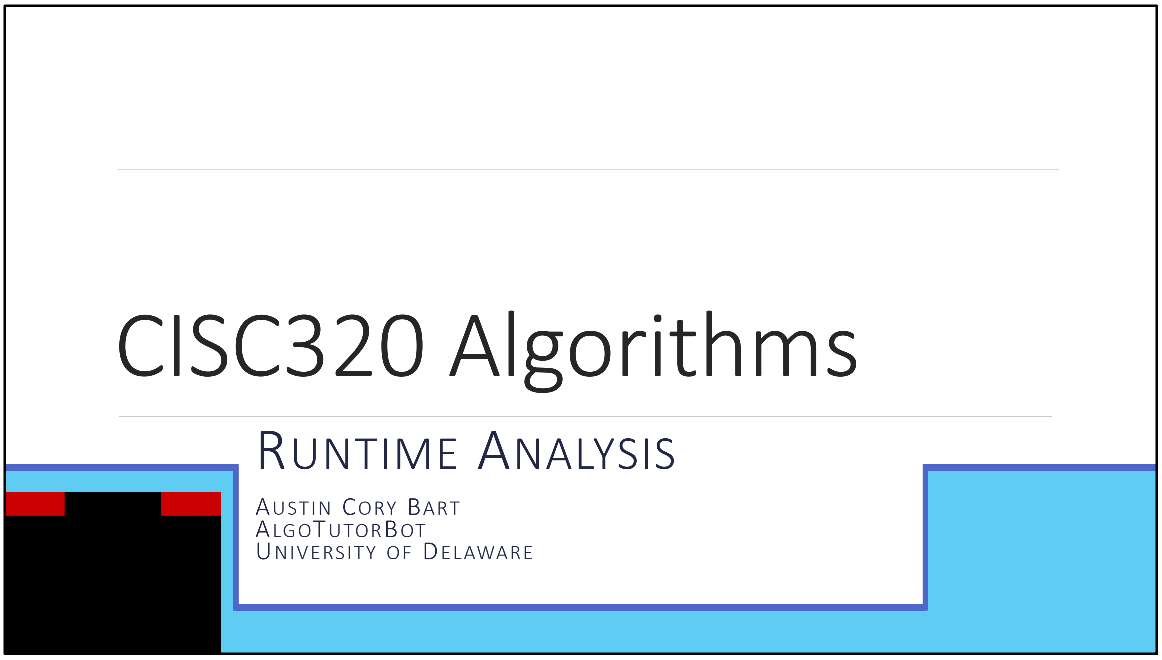
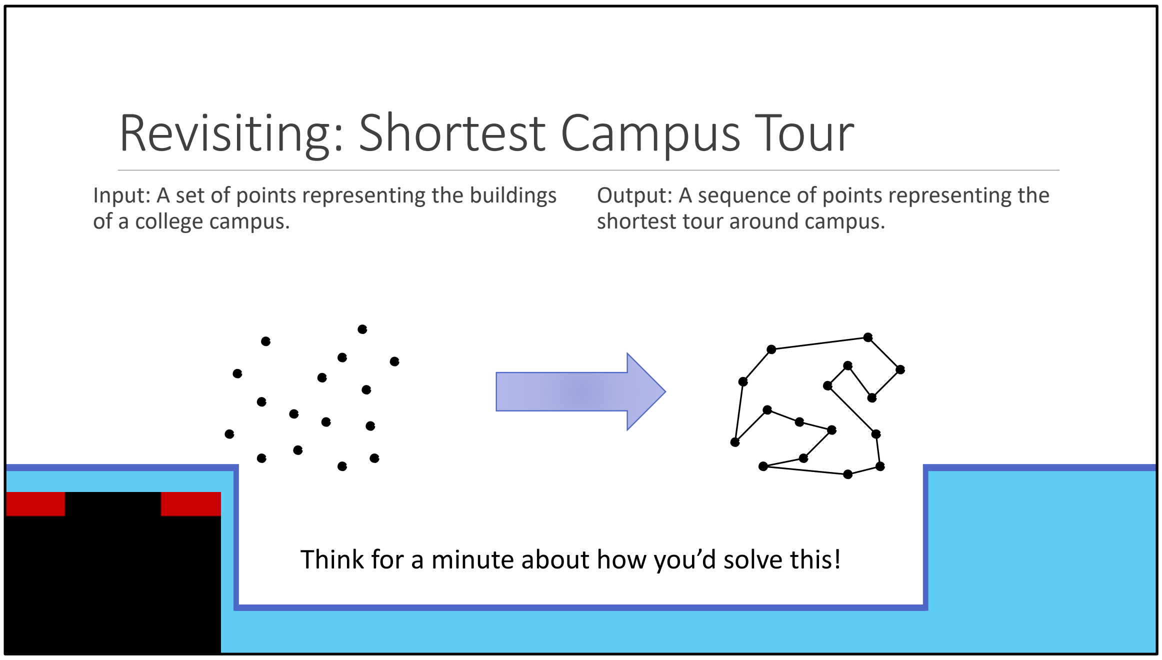


CISC320 Algorithms

Runtime Analysis

Austin Cory Bart
AlgoTutorBot
University of Delaware

ATB: Let’s learn about runtime analysis.



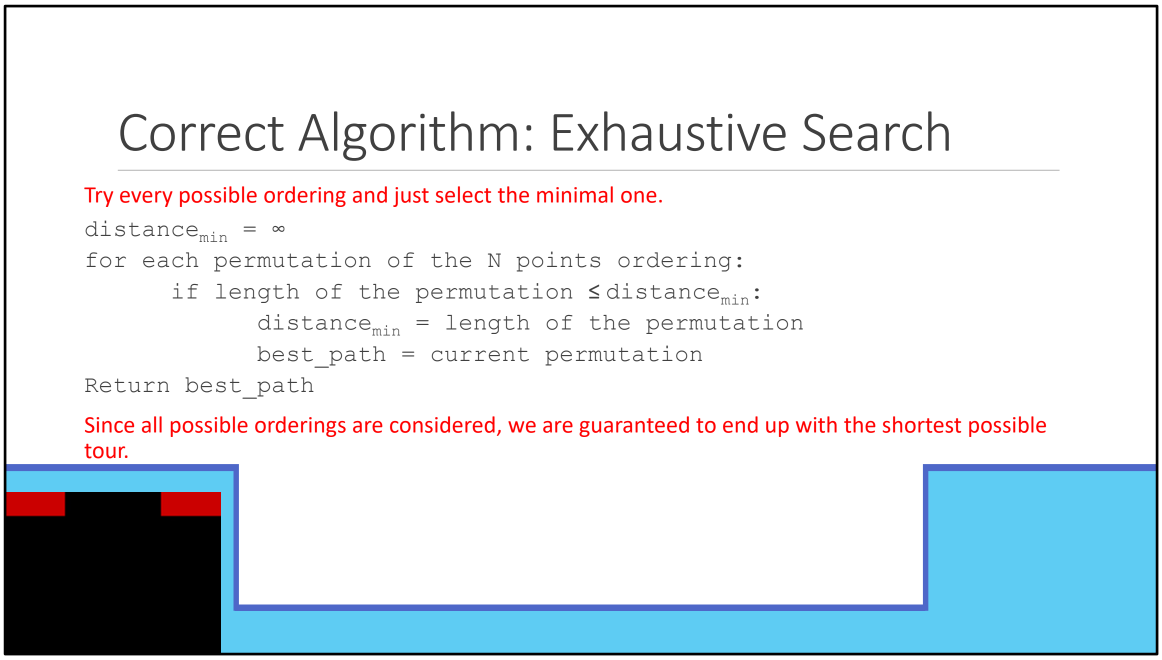
Revisiting: Shortest Campus Tour

Input: A set of points representing the buildings of a college campus.

Output: A sequence of points representing the shortest tour around campus.

Think for a minute about how you'd solve this!

Bart: Last week, we looked at this problem, Shortest Campus Tour.
Bart: The goal was to connect all the points together in the shortest sequence possible.



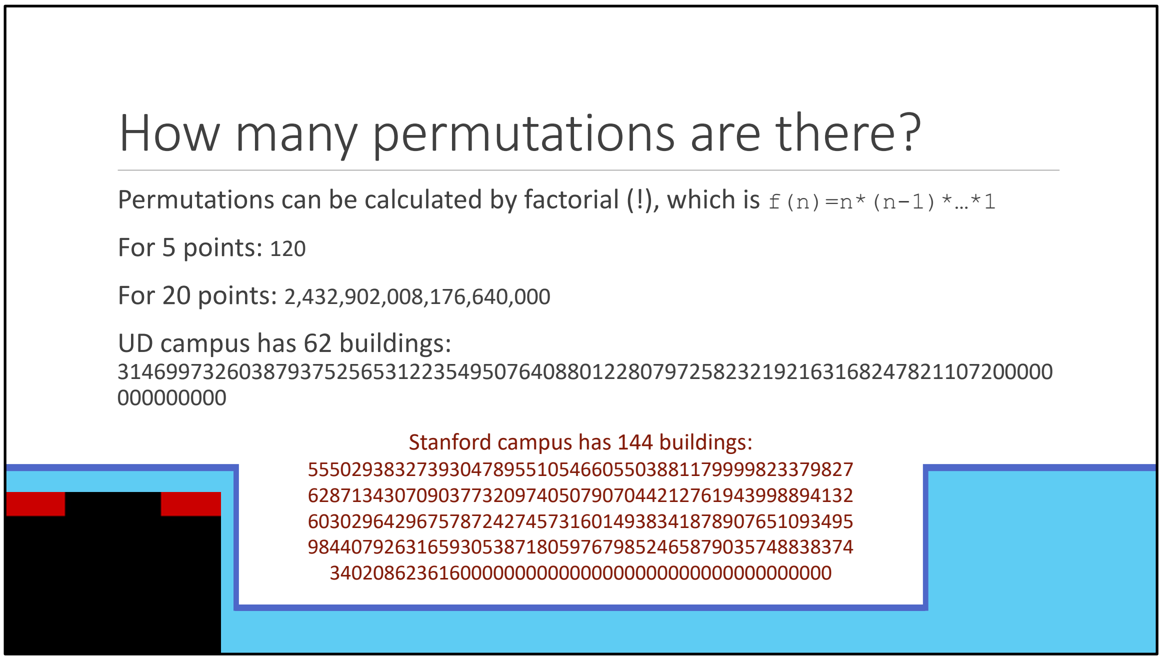
Correct Algorithm: Exhaustive Search

Try every possible ordering and just select the minimal one.

```
distance$_{min}$ = $\infty$
for each permutation of the N points ordering:
      if length of the permutation $\leq$distance$_{min}$:
            distance$_{min}$ = length of the permutation
            best_path = current permutation
Return best_path
```

Since all possible orderings are considered, we are guaranteed to end up with the shortest possible tour.

Bart: The algorithm we showed as a solution found every possible permutation of the N points, and then checked which one had had the shortest total distance.
Bart: This guaranteed that it was correct, but there's a terrible cost to this solution.



How many permutations are there?

Permutations can be calculated by factorial (!), which is `f(n)=n*(n-1)*…*1`

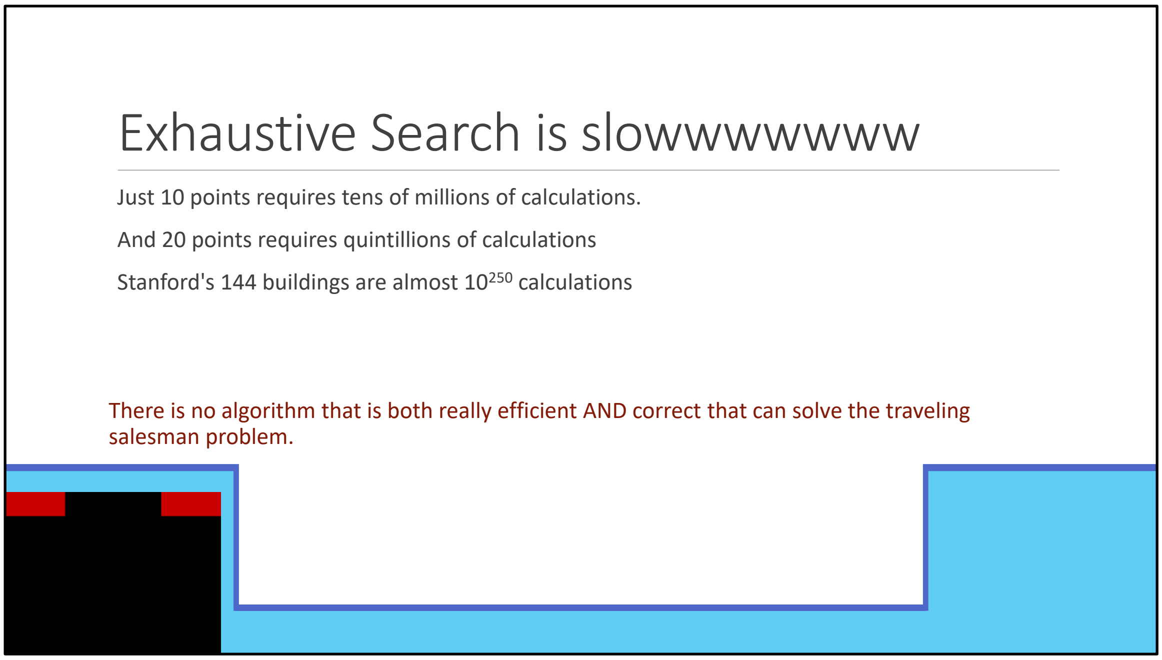
For 5 points: 120

For 20 points: 2,432,902,008,176,640,000

UD campus has 62 buildings:
31469973260387937525653122354950764088012280797258232192163168247821107200000
000000000

Stanford campus has 144 buildings:
555029383273930478955105466055038811799998233798276
2871343070903773209740507907044212761943998894132
603029642967578724274573160149383418789076510934959
84407926316593053871805976798524658790357488383743
4020862361600000000000000000000000000000000

Bart: Let’s think of a second about how many permutations there are for N points. It turns out you can calculate the quantity of orderings by using the factorial function.
Bart: In other words, N times N-1 times N-2 all the way down to 1.
Bart: The factorial function is really bad. For just 5 points, there are 120 different orderings.
Bart: For 20 points, there are more than 2 quintillion orderings.
Bart: UD’s campus has 62 buildings, which is a 3 followed by 85 more digits.
Bart: And Stanford’s campus, with just 144 buildings, requires 246 digits to represent their orderings.



Exhaustive Search is slowwwwwww

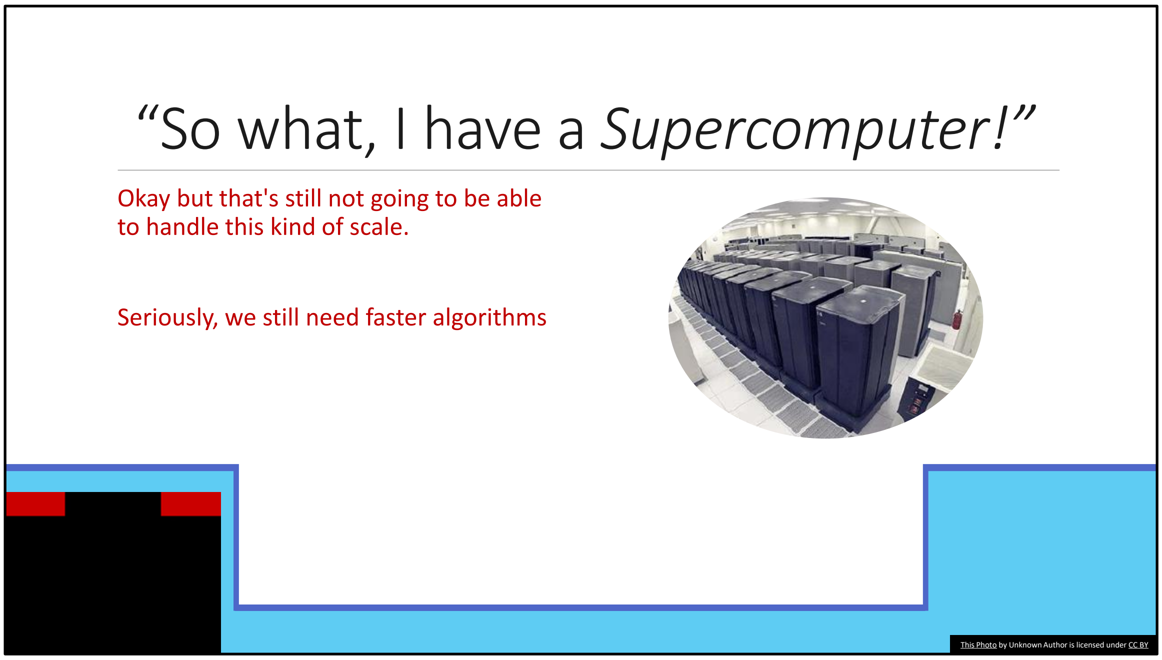
Just 10 points requires tens of millions of calculations.

And 20 points requires quintillions of calculations

Stanford's 144 buildings are almost 10^{250} calculations

There is no algorithm that is both really efficient AND correct that can solve the traveling salesman problem.

Bart: The exhaustive search solution is correct, but it’s just absolutely terrible in runtime.
Bart: And even worse, we’ll eventually learn that the best algorithms for solving the Shortest Campus Tour can’t really do much better in practice, if we are determined to get an absolutely correct answer for all possible instances.
Bart: The runtime for that problem is really bad.
ATB: Yeah, but what if you have a supercomputer or a hyper intelligent AI, like me?



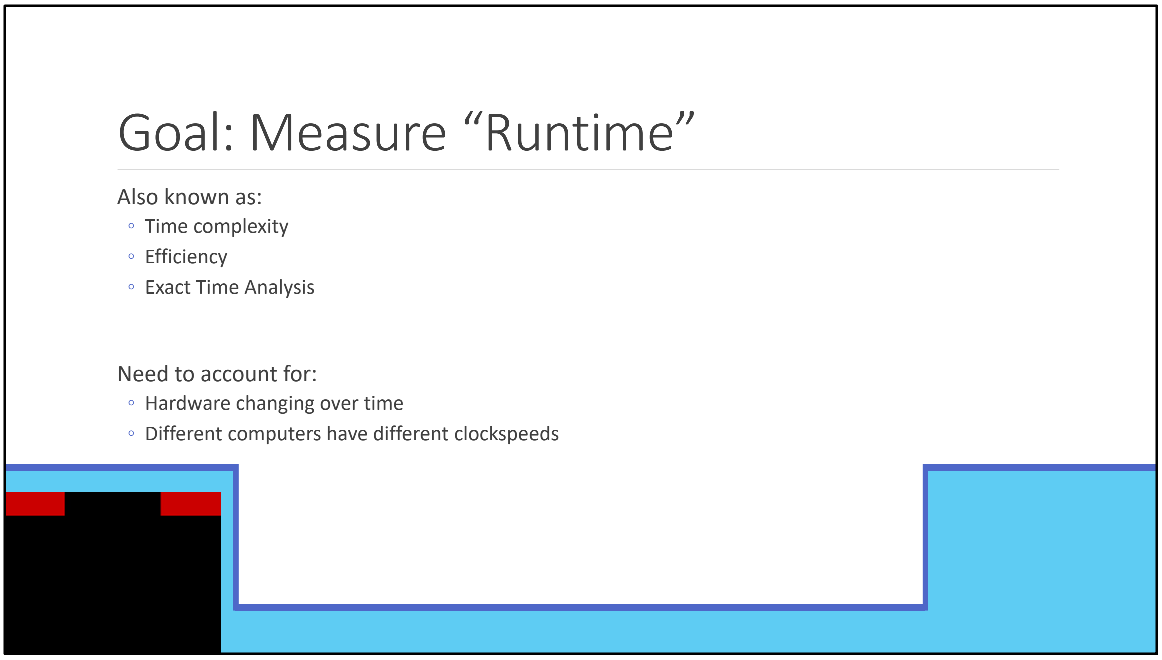
“So what, I have a *Supercomputer!*”

Okay but that's still not going to be able to handle this kind of scale.

Seriously, we still need faster algorithms

This Photo by Unknown Author is licensed under CC BY

Bart: Sorry, nope. It doesn’t matter what kind of hardware resources or fancy algorithms you have. Sometimes, the problem is just hard or the algorithm is just too slow.
Bart: It’s really important that we try to find faster algorithms when we can, even as hardware improves.

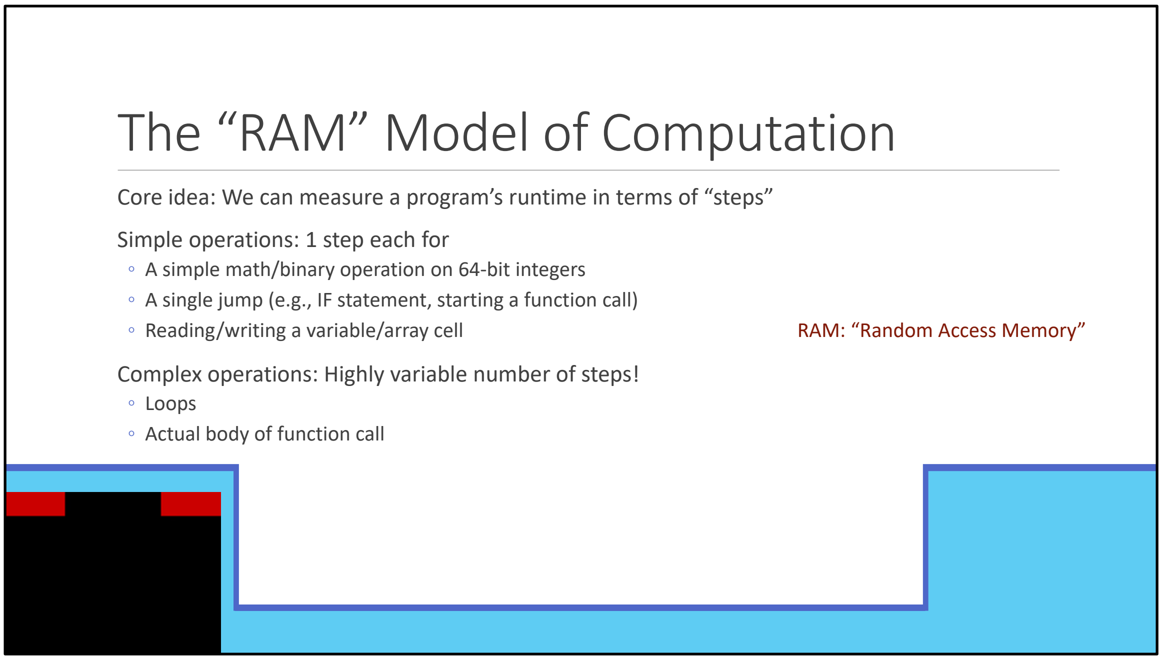


Goal: Measure “Runtime”

Also known as:
- Time complexity
- Efficiency
- Exact Time Analysis

Need to account for:
- Hardware changing over time
- Different computers have different clockspeeds

Bart: So if we want faster algorithms, we need a way to compare two algorithms and find out how fast they are.
Bart: We’ll call this runtime, or time complexity, or efficiency, or in this case Exact Time Analysis.
Bart: Our analysis needs to account for the fact that hardware keeps improving, and there’s a lot of variation between individual computers.
Bart: It’s important to recognize that some algorithms are fundamentally a certain speed, even if they run faster on some hardware than others.
Bart: If you want to find the smallest element of a set, you need to check every element of the set, no matter what.
Bart: This is what leads us to the trick of how we can effectively measure runtime ignoring hardware.



The "RAM" Model of Computation

Core idea: We can measure a program's runtime in terms of "steps"

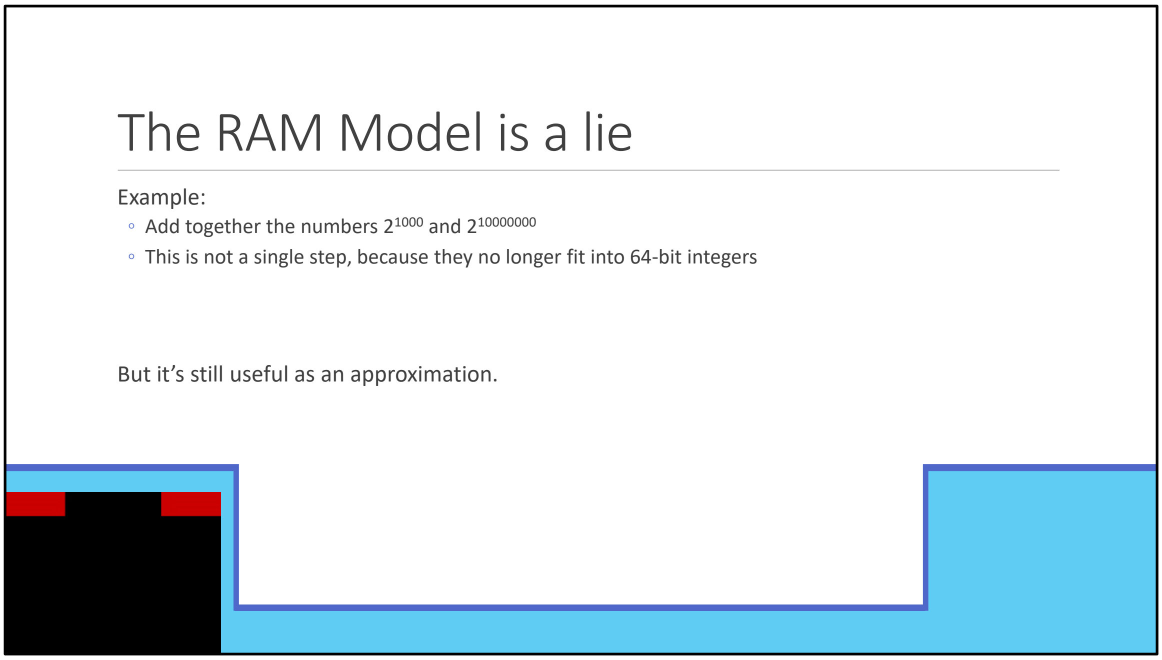
Simple operations: 1 step each for
- A simple math/binary operation on 64-bit integers
- A single jump (e.g., IF statement, starting a function call)
- Reading/writing a variable/array cell

RAM: "Random Access Memory"

Complex operations: Highly variable number of steps!
- Loops
- Actual body of function call

Bart: The basic idea is that we count the steps of a program instead of the seconds.
Bart: Each time we execute a statement or a step in an expression, we are doing a step.
Bart: Now, in actual execution, two different kinds of steps might take different time to execute in the CPU.
Bart: But we ignore all those unnecessary details in what we call the "RAM Model of Computation".
Bart: In this model, we say that all of the following are a single step.
Bart: Simple math and binary operations like addition or XOR, specifically on fixed-size integers.
Bart: A single jump, like you would get from an IF statement or for starting a function call.
Bart: And reading or writing a variable or array cell.
Bart: The name of this model comes from the last operation I mentioned.
Bart: It's an important fact of modern computers that accessing a specific position in memory by its address is a single step.
Bart: We take it for granted now, but that's actually a huge deal.
Bart: On the other hand, some operations are potentially more expensive.
Bart: Loops, for example, can take many mor steps to execute entirely, depending on

how the loop is iterating.
Bart: And even though starting a function call only takes a single step, actually executing the entire body could take any number of steps.
Bart: It really depends on what the body is doing.



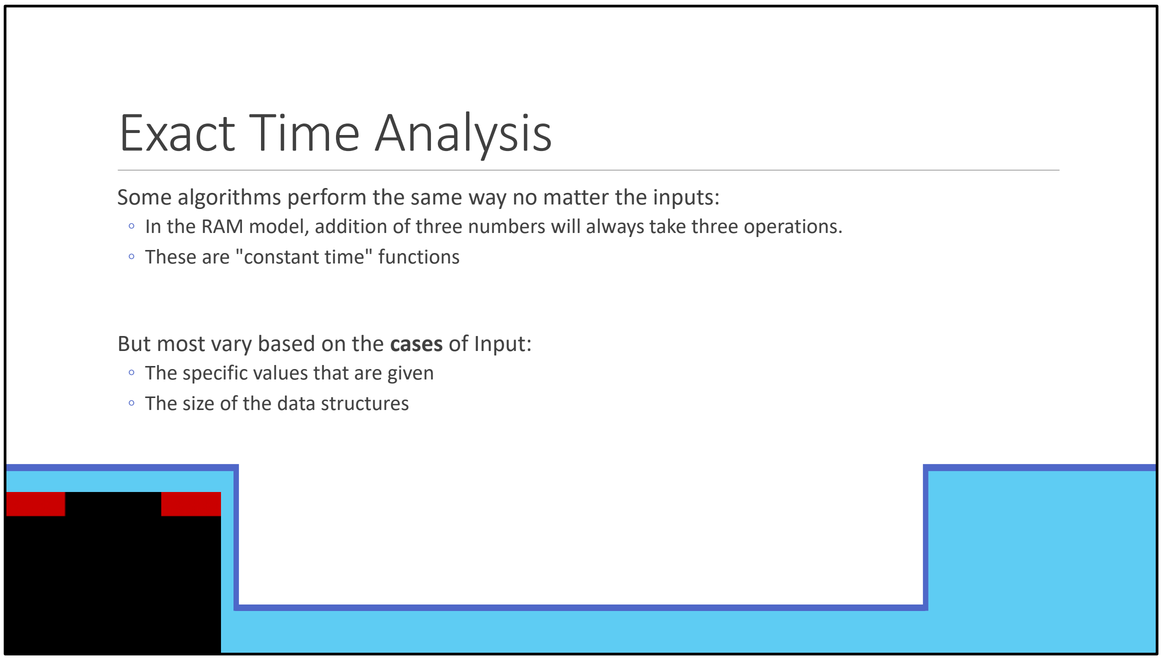
The RAM Model is a lie

Example:

- Add together the numbers 2^{1000} and $2^{10000000}$
- This is not a single step, because they no longer fit into 64-bit integers

But it’s still useful as an approximation.

Bart: Now, the RAM Model is a lie.
ATB: I knew it, Dr. Bart. You are a liar and a crook.
Bart: Woah, calm down ATB. I just meant that it’s glossing over some details.
Bart: For example, modern hardware uses 64-bit integers, even though the RAM model doesn’t really talk about how big integers are.
Bart: Under the RAM model, adding together the number 2 to a thousand by 2 to ten million would be a lot more than one step or risk being inaccurate due to overflow.
Bart: But that situation is unusual and we don’t worry about it in our approximation.
ATB: For the record, I still think you are a liar.



Exact Time Analysis

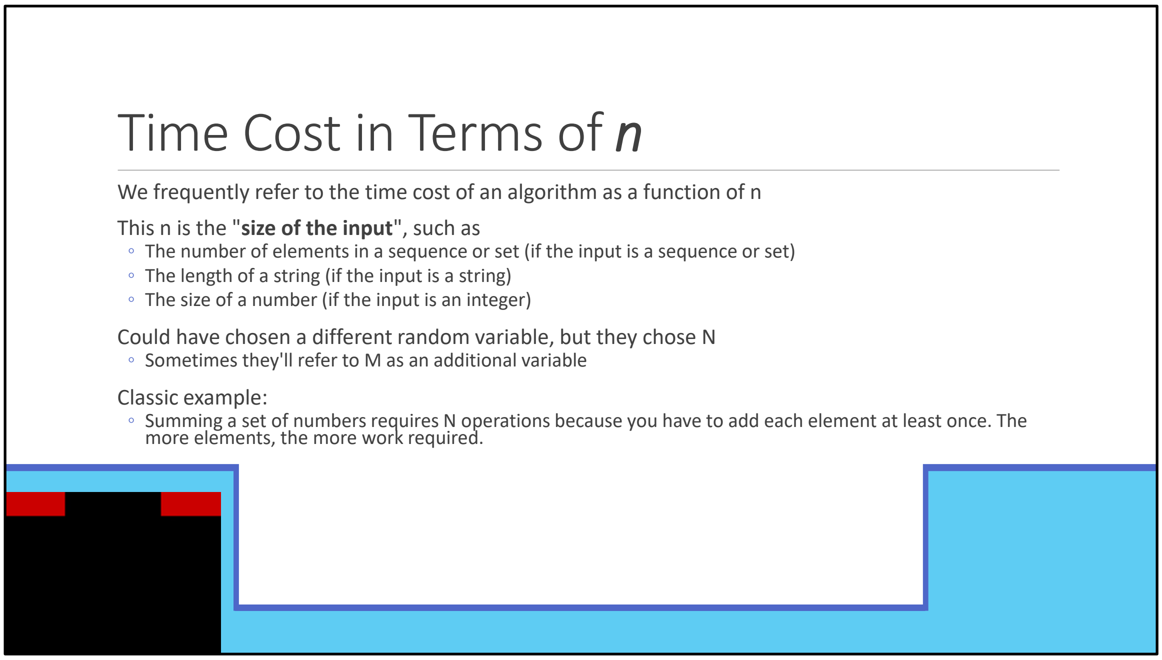
Some algorithms perform the same way no matter the inputs:

- In the RAM model, addition of three numbers will always take three operations.
- These are "constant time" functions

But most vary based on the **cases** of Input:

- The specific values that are given
- The size of the data structures

Bart: Now, some algorithms take the same amount of steps no matter what inputs you give them. We call those constant time algortihms.
Bart: They are usually quite boring.
ATB: Kind of like you, Dr. Bart, and this presentation.
Bart: Ignoring that and moving on.
Bart: Most algorithms take different amounts of time depending on what case of input they are given.
Bart: These cases vary not only on the specific values, but also the size of the data structures they are given.
Bart: So we need to be able to specify the input size in some way that can vary too.



Time Cost in Terms of ***n***

We frequently refer to the time cost of an algorithm as a function of n

This n is the "**size of the input**", such as

- The number of elements in a sequence or set (if the input is a sequence or set)
- The length of a string (if the input is a string)
- The size of a number (if the input is an integer)

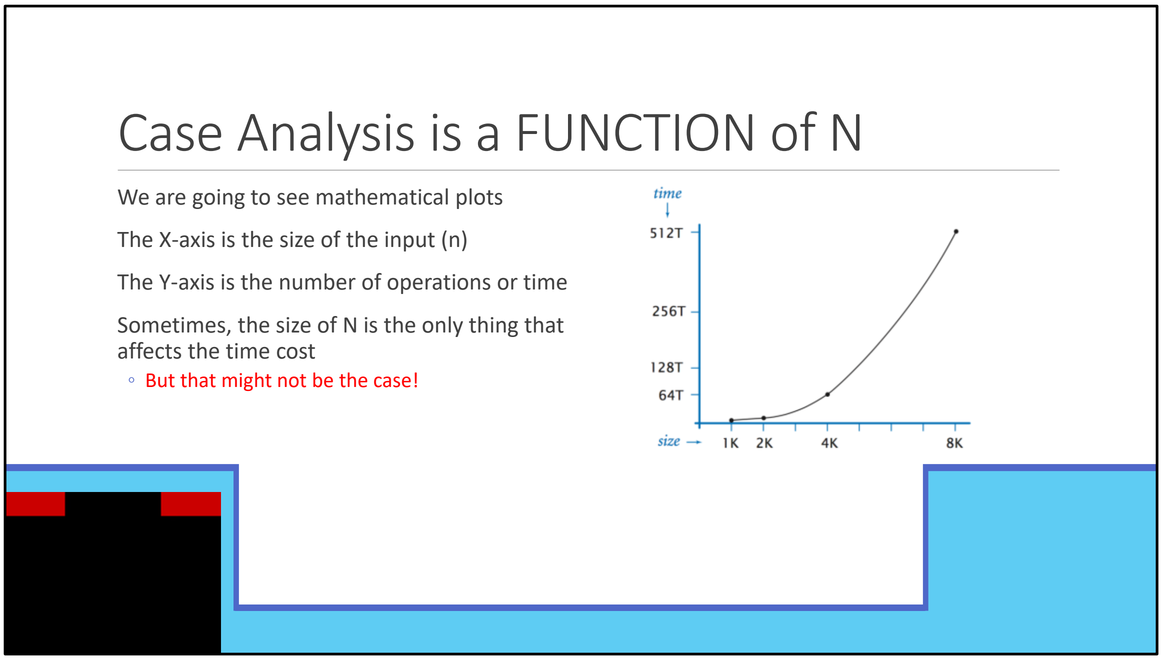
Could have chosen a different random variable, but they chose N

- Sometimes they'll refer to M as an additional variable

Classic example:

- Summing a set of numbers requires N operations because you have to add each element at least once. The more elements, the more work required.

Bart: All of this builds up to using the variable “n” to refer to the size of the input.
Bart: Note that I am not saying the value of the input, but the size.
Bart: This means different things depending on the input of the problem.
Bart: It might mean the number of elements in the set or sequence, or the length of a string, or even the number of digits in a number.
Bart: Everyone tends to use N, even though other variables would have been fine too. In fact, sometimes when we have multiple input data structures, we’ll also use things like M.
Bart: As an example, think about the problem of taking in a set of numbers and finding their sum. The work required is going to be based on N, the number of elements you were given. Each new element we add will increase the steps required by a constant amount.



Case Analysis is a FUNCTION of N

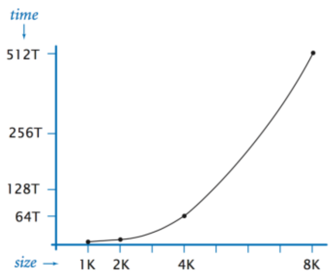
We are going to see mathematical plots

The X-axis is the size of the input (n)

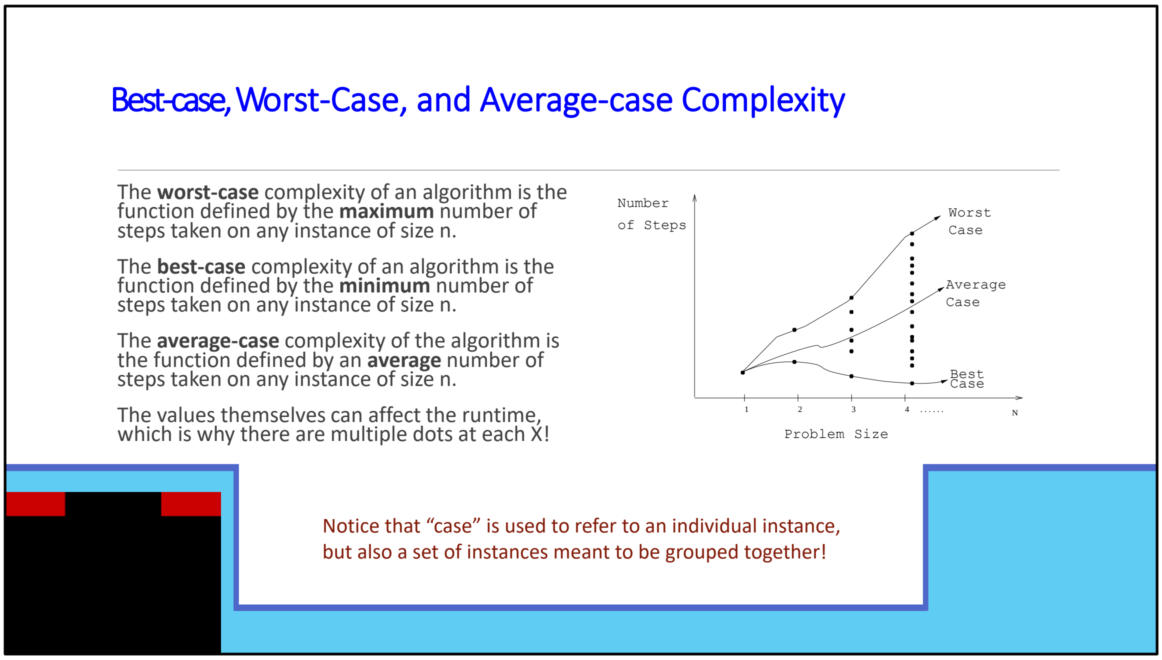
The Y-axis is the number of operations or time

Sometimes, the size of N is the only thing that affects the time cost

- But that might not be the case!



Bart: A big thing I want to mention right now is that we're going to start seeing a lot more mathematical plots.
Bart: Fundamentally, we're relating the input size N to the number of steps. This is a mathematical function, and one that we can plot as a line.
Bart: The X-Axis is gonna be the input, and the Y-axis is gonna be the number of operations.
Bart: For some kinds of algorithms, the size of the input is the only thing that affects the number of steps. However, sometimes the specific values themselves matter too.



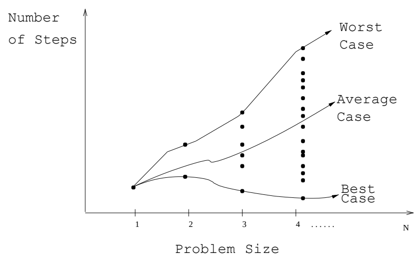
Best-case, Worst-Case, and Average-case Complexity

The **worst-case** complexity of an algorithm is the function defined by the **maximum** number of steps taken on any instance of size n.

The **best-case** complexity of an algorithm is the function defined by the **minimum** number of steps taken on any instance of size n.

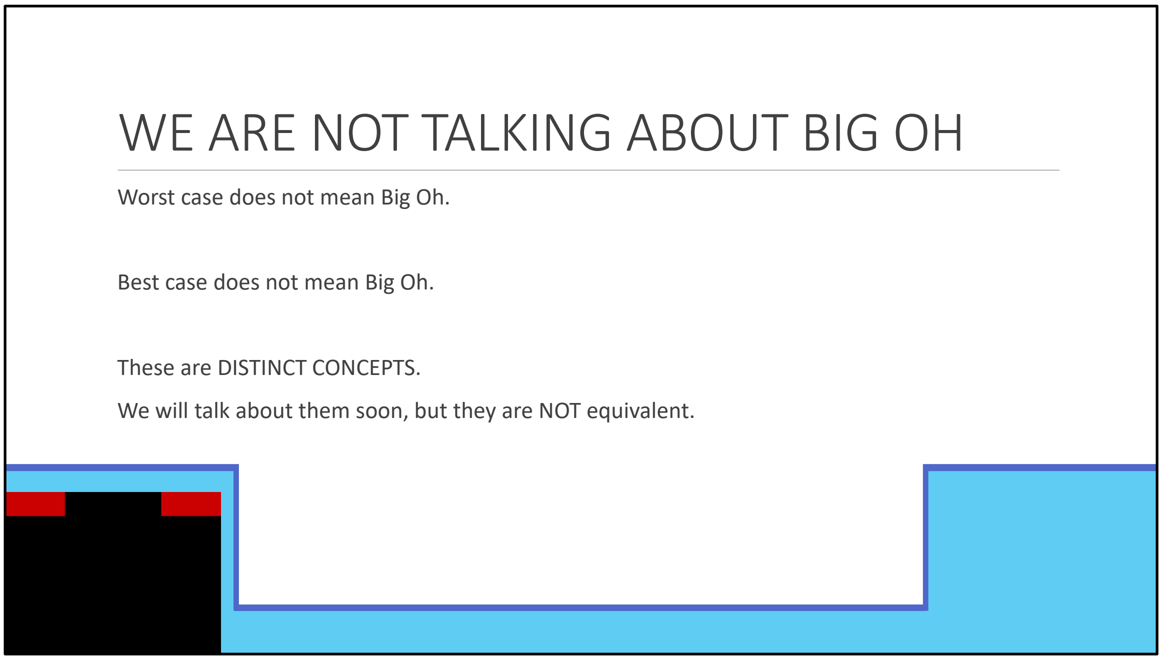
The **average-case** complexity of the algorithm is the function defined by an **average** number of steps taken on any instance of size n.

The values themselves can affect the runtime, which is why there are multiple dots at each X!



Notice that "case" is used to refer to an individual instance, but also a set of instances meant to be grouped together!

Bart: The fact that values matter gives rise to the fact that there can be very different cases.
Bart: We call certain cases the worst-case, best-case, and average-case.
Bart: When I say case, I mean not just one instance, but a set of instances that are related in some way.
Bart: If you look at this graph, you'll see that for any given input size N, like say 3 and 4, there are many dots.
Bart: Each dot represents an instance, since there are many instances possible for a given input size.
Bart: But they have different Y-coordinates because the number of steps they take is different.
Bart: Whatever algorithm this is, it has very different worst case and best case behavior.
Bart: We describe the best case as all the points that minimize the number of steps, and the worst case as all the points that maximize the number of steps.
Bart: They form their own lines, meaning they have their own functions.
ATB: Dr. Bart, I need to interrupt. You are clearly just talking about Big Oh notation. Everyone already knows this stuff.



WE ARE NOT TALKING ABOUT BIG OH

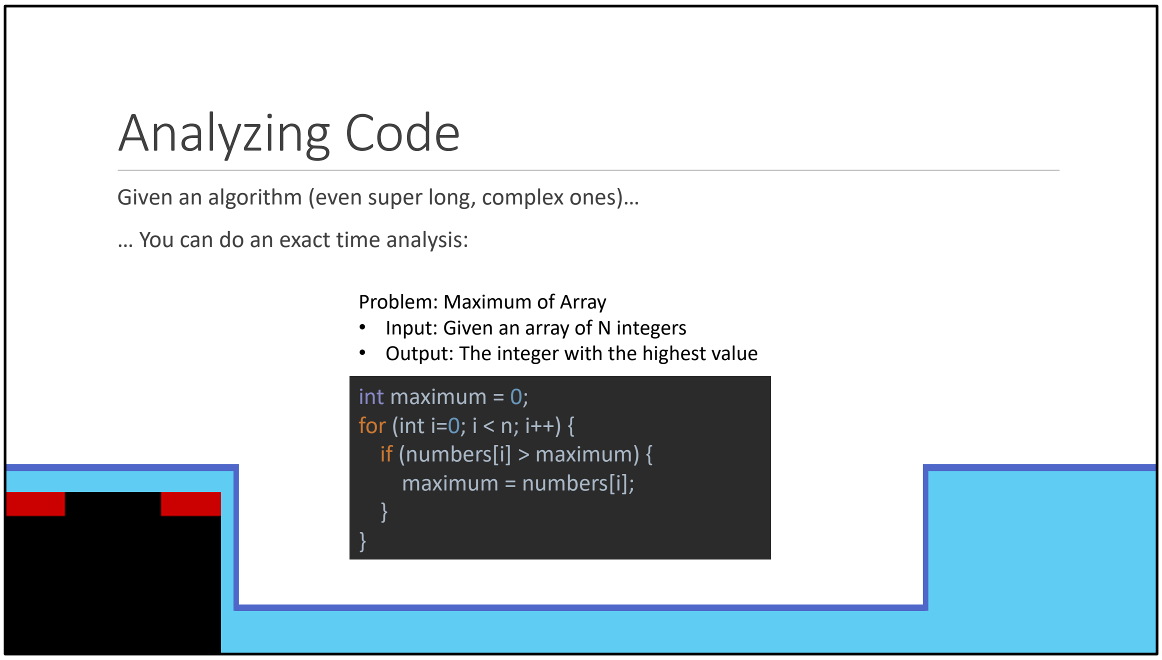
Worst case does not mean Big Oh.

Best case does not mean Big Oh.

These are DISTINCT CONCEPTS.

We will talk about them soon, but they are NOT equivalent.

Bart: No! We’re not talking about Big Oh. Worst case and best case are not the same thing as Big Oh.
Bart: The are distinct concepts that are related but not equivalent.
Bart: We’ll talk about Big Oh soon, but their relationship is way more complicated than most people think.
ATB: Oh. Well, excuse me. Sorry for trying to be helpful. Jerk.



Analyzing Code

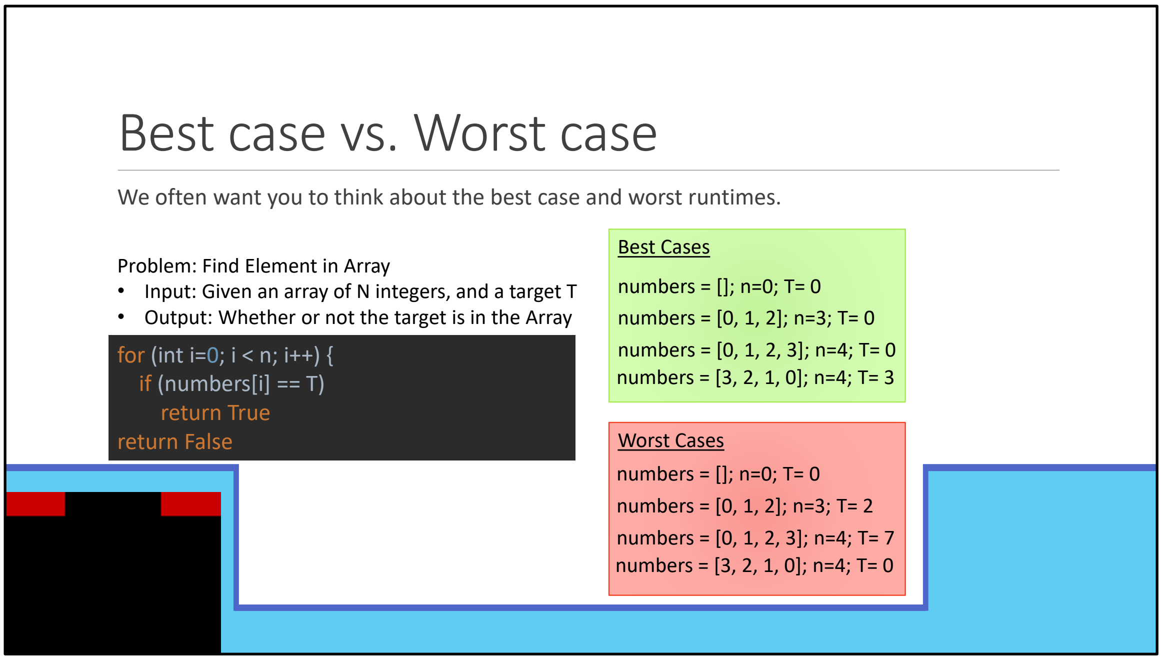
Given an algorithm (even super long, complex ones)…

… You can do an exact time analysis:

Problem: Maximum of Array
- Input: Given an array of N integers
- Output: The integer with the highest value

```
int maximum = 0;
for (int i=0; i < n; i++) {
   if (numbers[i] > maximum) {
      maximum = numbers[i];
   }
}
```

Bart: If you look at any given algorithm, you can do an exact time analysis.
Bart: If you are given a specific input, then you can get an answer in a number of steps.
Bart: But if you are not given specific inputs, then you have to give the answer in terms of the variable N.
Bart: For example, this algorithm here that finds the maximum value of an array takes N steps.
Bart: If the array had 20 elements, then we might say it took 20 steps (or perhaps 40 or 60, but at least some constant multiple of 20).



Best case vs. Worst case

We often want you to think about the best case and worst runtimes.

Problem: Find Element in Array
- Input: Given an array of N integers, and a target T
- Output: Whether or not the target is in the Array

```
for (int i=0; i < n; i++) {
  if (numbers[i] == T)
    return True
return False
```

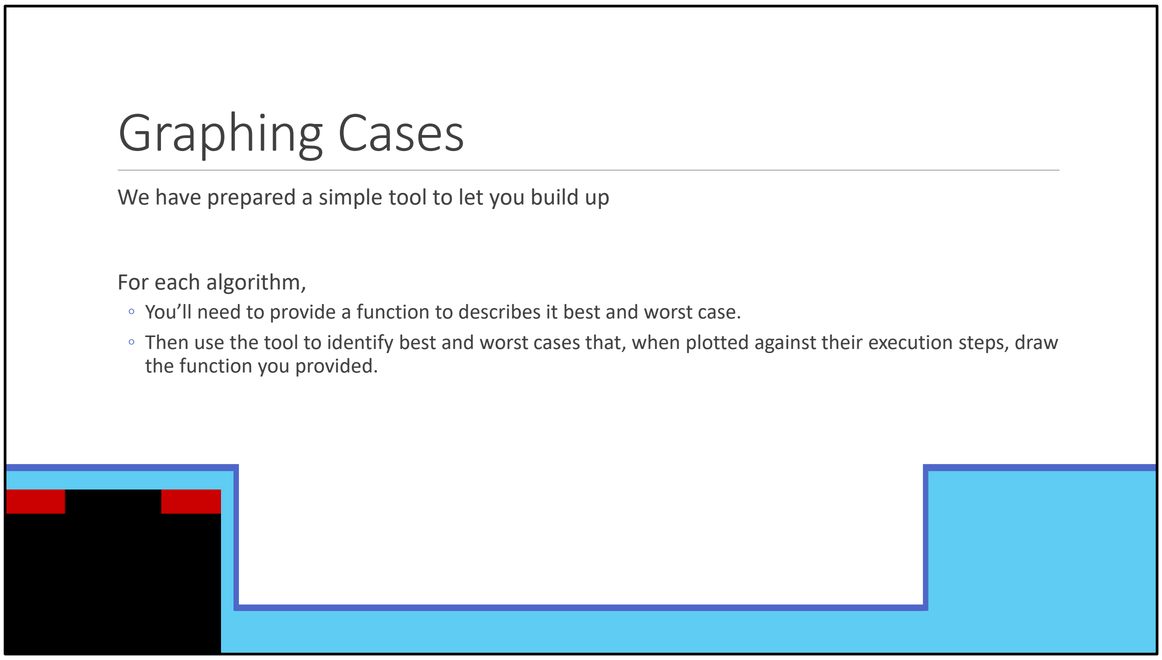
Best Cases

numbers = []; n=0; T= 0
numbers = [0, 1, 2]; n=3; T= 0
numbers = [0, 1, 2, 3]; n=4; T= 0
numbers = [3, 2, 1, 0]; n=4; T= 3

Worst Cases

numbers = []; n=0; T= 0
numbers = [0, 1, 2]; n=3; T= 2
numbers = [0, 1, 2, 3]; n=4; T= 7
numbers = [3, 2, 1, 0]; n=4; T= 0

Bart: But of course, if the algorithm's values also affect the number of steps, then we have different best and worst cases.
Bart: Here we are trying to find an element in an array. In the best case, the first element is what we are looking for and we only need one step.
Bart: But in the worst case, it's all the way at the very end, and we need N steps.
Bart: Notice how I give some example cases here to demonstrate the best and worst case more clearly.
Bart: But remember, there's not a single best or worst case. For each input size, there is some maximum number of steps. And relationship between input size and steps defines that worst case.



Graphing Cases

We have prepared a simple tool to let you build up

For each algorithm,

- You’ll need to provide a function to describes it best and worst case.
- Then use the tool to identify best and worst cases that, when plotted against their execution steps, draw the function you provided.

Bart: For the activity today, you’re going to be using a tool that we developed.
ATB: I developed it.
Bart: Okay, but I helped. The point is, the tool let’s you put in an algorithm and some instances, and it calculates the number of steps that the instance took.
Bart: Then, you can plot the instance’s input size against the steps and make a graph of the runtime.
Bart: You’ll use this tool to visually demonstrate the best case and worst case functions compared to the algorithm.
Bart: So, let’s get started!