Tractability

15-110 – Wednesday 10/23

Learning Goals

Analyze and improve the **efficiency** of different programs

- Recognize problems that are **intractable** and (probably) cannot be solved quickly

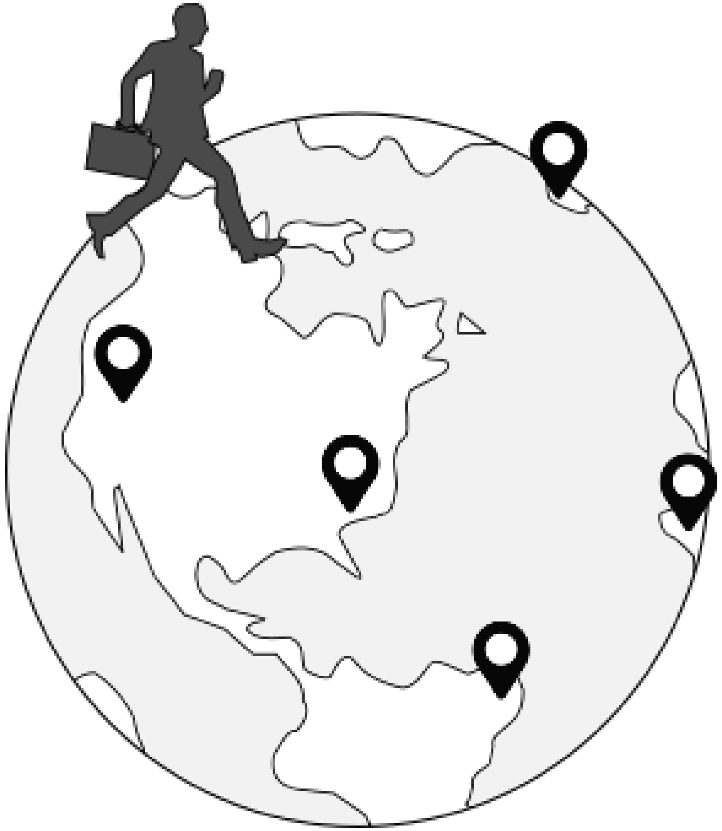
- Understand the question and potential ramifications of **whether P = NP**

Tractability

Graph Example: Travelling Salesperson Problem

There are many classic algorithmic problems that involve graphs, because graphs can be difficult to interact with.

One especially well-known problem is called the **Travelling Salesperson problem**. You're given a list of cities to visit, and a graph of distances between cities. You want to find the shortest possible route that lets you visit every city, then gets you back home.



One Solution: Brute Force

One intuitive way to solve the Travelling Salesperson problem is to plan out **every possible route** from the starting city across all the others, then choose the shortest route of them all.

This type of approach- generating all possibilities and then comparing them- is called a **brute force approach**. It's a simple and intuitive way to solve problems, but it does have drawbacks.

Brute Force Efficiency

Consider the efficiency of a brute-force approach. Let's say that generating a path of n stops counts as one action. How many possible paths are there?

We have n possible first stops on the route. For each of those routes, there are n-1 possible second stops. Then there are n-2 third stops per route, etc... until there is only one city left for the last stop.

This means that the number of possible routes is n * (n-1) * (n-2) * ... * 1. **It's O(n!). That's really inefficient!**

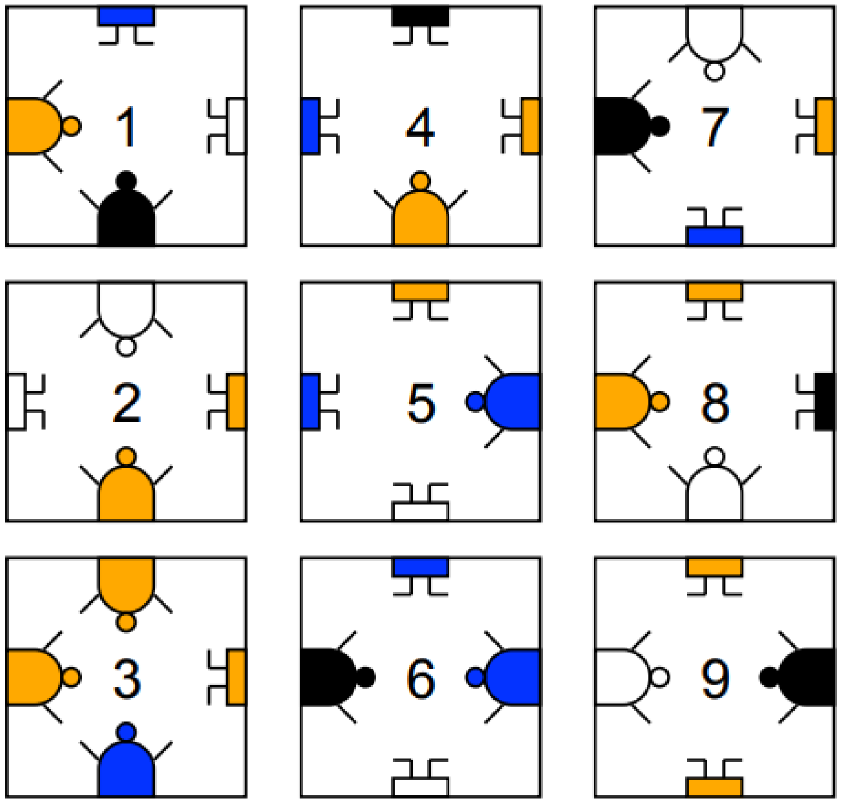
It turns out that there are a lot of problems in computer science that share this property- we can solve them, but it takes a long time. Let's go through some examples...

Example: Puzzle Solving

First, let's consider a simple problem. We want to solve a basic puzzle by putting together square pieces (like the ones shown to the right) so that any two pieces that are touching each other make a figure with a head and feet of the same color.

To make this even simpler, let's make a rule that pieces cannot be rotated, and the final result must be a m x m square.

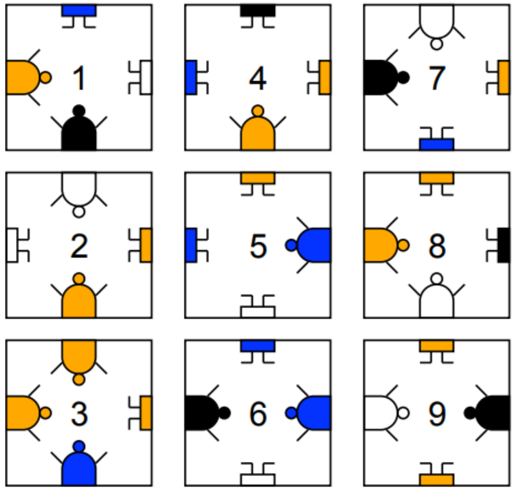
Here's our question: is it possible to make a solution that follows these rules?



Brute Force on Puzzle Solving

We can again use brute force to solve the puzzle problem, just like we did with Travelling Salesperson. We need to try all possible placements for each piece.

In the example to the right, there are 9 options for the first position, 8 for the second, 7 for the third, etc.... it's **O(n!)** time again.



9 choices	8 choices	7 choices
6 choices	5 choices	4 choices
3 choices	2 choices	1 choice

O(n!) is really bad

It turns out that O(n!) is a *really bad* runtime. For example, let's assume that it takes 1 microsecond (1/1000th of a second) to set up a specific ordering of pieces of a puzzle and check if it's correct.

If we have nine pieces (like in our example before), it will take **6.048 minutes** to solve the puzzle.

If we increase the size to a 4x4 puzzle (16 pieces), it will take **663.46 years!**

O(n!) is awful. Let's see if we can find problems that do a bit better.

Example: Subset Sum

In the problem Subset Sum, we are given a list of numbers and a target number, x. We want to determine if there's a subset of the list that sums to x.

Brute force solution: generate all possible subsets, see if any of them sum to x.

How do we do this? Check the example on the right. If we have all four subsets of the list [2, 3], we can use them to create all 8 subsets of [1, 2, 3]. For each subset, make one version that includes 1, and one version that doesn't.

We double the number of subsets with each new number- this is **$O(2^n)$**.

Subsets of [2, 3]:

- []
- [2]
- [3]
- [2, 3]

Subsets of [1, 2, 3]:

- []
- [1]
- [2]
- [1, 2]
- [3]
- [1, 3]
- [2, 3]
- [1, 2, 3]

Example: Boolean Satisfiability

A similar problem commonly encountered in computer science, called **Boolean Satisfiability**, asks: for a given circuit with n inputs (X_1 to X_n), is there a set of assignments of X_i to 1 or 0 that makes the whole circuit output 1?

Instead of generating all possible subsets, we instead generate all possible combinations of input values.

But this also doubles every time we add a new input (as we must try all possible combinations with the input set to 0, then set to 1)- it's **$O(2^n)$**!

Inputs for 2 elements

- 0, 0
- 0, 1
- 1, 0
- 1, 1

Inputs for 3 elements

- 0, 0, 0
- 0, 0, 1
- 0, 1, 0
- 0, 1, 1
- 1, 0, 0
- 1, 0, 1
- 1, 1, 0
- 1, 1, 1

Real-life Example: Exam Scheduling

Here's one final example: scheduling final exams. Given a list of classes, a list of students who are each taking a certain set of classes, and a list of timeslots over the period of a week, you need to generate a schedule that fits within the period and results in no student having two exams in the same period.

We can generate all possible schedules by assigning each class to each possible timeslot. Then we just need to look for one schedule that has no conflicts. But every time we add a new class, we need to try adding it to **every** possible schedule in **every** possible timeslot.

If we say there are k timeslots and n classes, then we have to turn one schedule into k different schedules for every new class added. This is $\mathbf{O(k^n)}$!

Semester & Mini-2 Final Exams: December 9, 10, 12, 13, 15 & 16(Make-Up Day)

Course	Section	Title	Date	Time (USA EST)	Classroom(s)
Architecture					
48116	A	BUILDING PHYSICS	Sunday, December 15, 2019	01:00 pm - 04:00 pm	To Be Announced (TBA)
48315	1	ENVIR I: CLIM & ENG	Thursday, December 12, 2019	08:30 am - 11:30 am	To Be Announced (TBA)
48432	A	ENV II	Thursday, December 12, 2019	08:30 am - 11:30 am	To Be Announced (TBA)
48531	A	FABRICATNG CUSTOMZTN	Monday, December 9, 2019	01:00 pm - 04:00 pm	To Be Announced (TBA)
48558	A	RLT COMP	Thursday, December 12, 2019	08:30 am - 11:30 am	To Be Announced (TBA)
48568	A	ADV CAD BIM 3D VISLZ	Tuesday, December 10, 2019	08:30 am - 11:30 am	To Be Announced (TBA)
48635	1	ENVIRO I M.ARCH	Thursday, December 12, 2019	08:30 am - 11:30 am	To Be Announced (TBA)
48655	A	ENV II GRAD	Thursday, December 12, 2019	08:30 am - 11:30 am	To Be Announced (TBA)
48714	A	DATA ANL URBN DSNG	Friday, December 13, 2019	01:00 pm - 04:00 pm	To Be Announced (TBA)
48729	A	PROD HLTH QUAL BLDGS	Thursday, December 12, 2019	01:00 pm - 04:00 pm	To Be Announced (TBA)
48734	A	RCTV SP MD ARC	Friday, December 13, 2019	05:30 pm - 08:30 pm	To Be Announced (TBA)
48743	A	INTRO ECO DES	Friday, December 13, 2019	01:00 pm - 04:00 pm	To Be Announced (TBA)
48749	A	CD SPECIAL TOPICS	Tuesday, December 10, 2019	01:00 pm - 04:00 pm	To Be Announced (TBA)
48785	A	MAAD RES PROJ	Sunday, December 15, 2019	05:30 pm - 08:30 pm	To Be Announced (TBA)
48798	A	HVAC & PS LOW CARB B	Monday, December 9, 2019	05:30 pm - 08:30 pm	To Be Announced (TBA)
Art					
60157	A	DRAWING NON-MAJORS	Tuesday, December 10, 2019	05:30 pm - 08:30 pm	CFA TBD
60218	A	REAL-TIME ANIMATION	Monday, December 9, 2019	08:30 am - 11:30 am	To Be Announced (TBA)
60220	A	TECH CHARACTER ANIM	Thursday, December 12, 2019	05:30 pm - 08:30 pm	To Be Announced (TBA)
60220	B	TECH CHARACTER ANIM	Thursday, December 12, 2019	05:30 pm - 08:30 pm	To Be Announced (TBA)
60333	A	CHARACTER RIGGING	Sunday, December 15, 2019	08:30 am - 11:30 am	BH 140F

$O(2^n)$ and $O(k^n)$ are still really slow

$O(2^n)$ is a bit better than O(n!)... but not *that* much better. Let's say we want to solve the subset sum problem, and it again takes us 1 microsecond to generate a specific subset and see if it's equal to the target.

If n = 10, we find the solution in **1.024 seconds**. Much better!

But if n = 20, we find the solution in **17.48 minutes**...

And if n = 30, it will take us **12.43 days**. By the time n = 40, it takes **35 years**.

$O(2^n)$ is not as bad as O(n!), but it's still really bad.

Improving Algorithms

For each of the problems we discussed, we can try to be clever and shave some time off by improving the algorithm.

For example, in subset sum, we could try adding the numbers from smallest to largest, and keep track of the intermediate sums. If the sum becomes larger than the target, we can stop generating new sublists from the too-big sublist.

Another example: for the puzzle, we can keep an eye on bordering pieces as we add them. As soon as we add a piece that doesn't match the pieces it touches, we can go back and try something different.

This kind of improvement does help, but it tends to shave off a **constant** amount of time. In the worst case, we'll still run in $O(2^n)$ or $O(n!)$.

Tractability

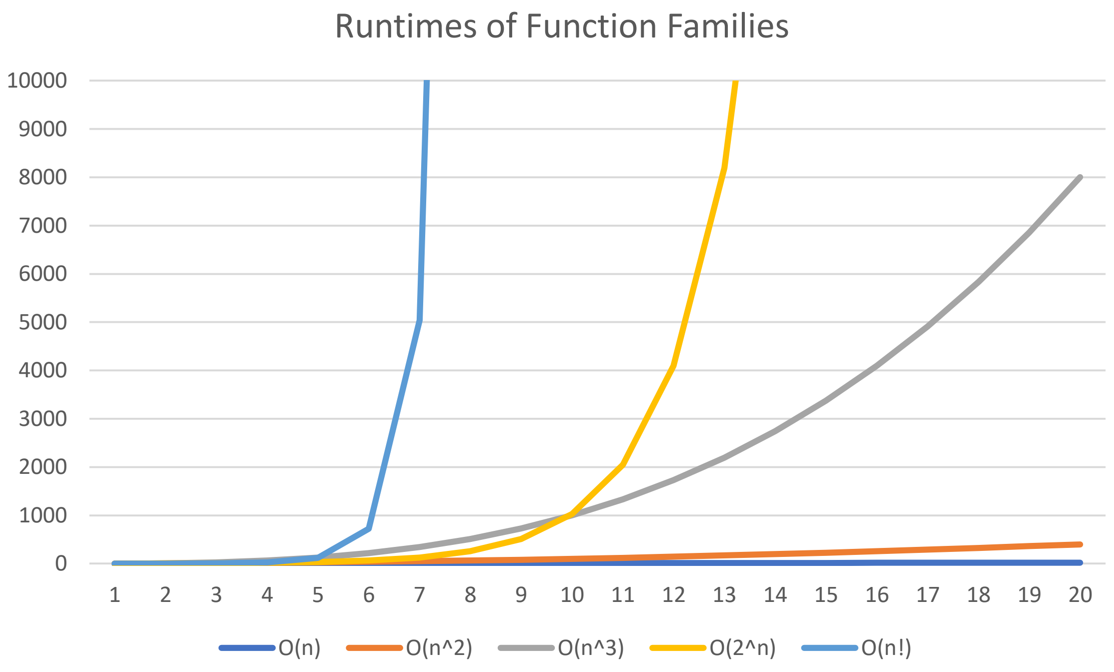
This leads us to a new concept: **tractability**. A problem is said to be **tractable** if it has a reasonably efficient runtime, so that we can use it for practical input sizes.

We say that a runtime is reasonable if it can be expressed as a **polynomial** equation. This means an equation of the form

$ax^k + bx^{k-1} + \ldots + cx + d$

O(1), O(logn), O(n), O(nlogn), $O(n^2)$, and $O(n^k)$ are all tractable. $O(2^n)$, $O(k^n)$, and O(n!) are not- they're **intractable**.

We can see the difference in growth quickly using the graph to the right.



Complexity Classes

Goal: Find Tractable Solutions

Now we know just how bad the brute-force solutions to this set of problems are. But maybe there's a different approach that doesn't require us to generate every possible answer!

That will be our goal for the rest of the lecture: to see if we can find a **tractable** solution to these hard problems.

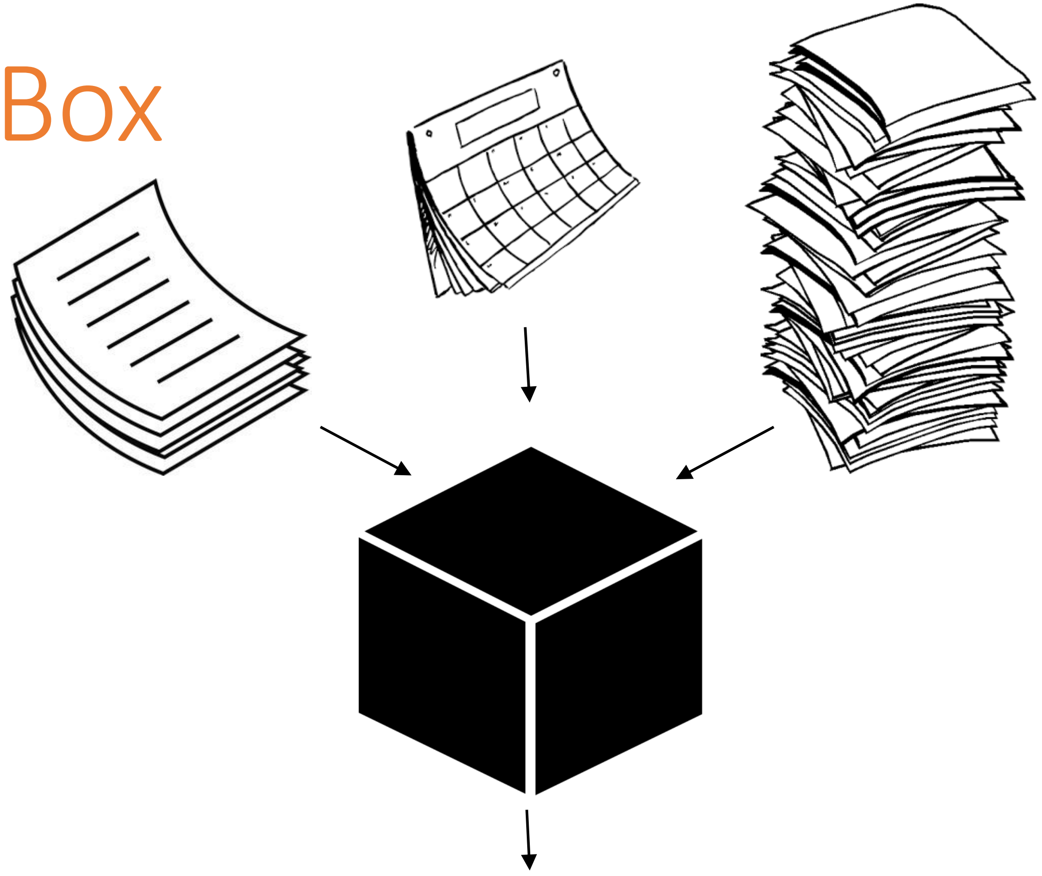
Until now, we've only discussed how long it takes to **find** the solution to a problem. Let's take a different approach.

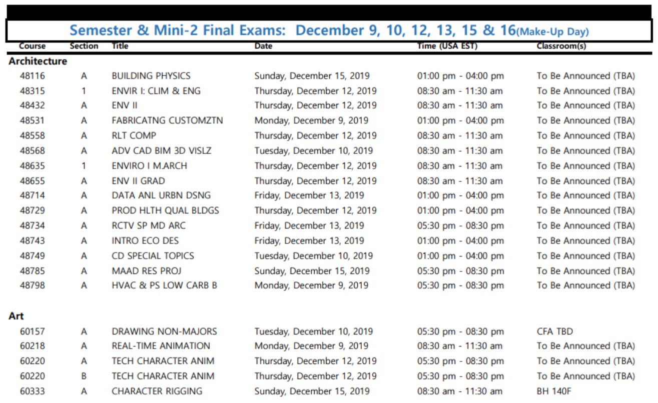
Magical Schedule-Making Box

Suppose a magical black box descends from the sky onto campus one day.

Someone discovers that if you feed the box a list of all the classes in a semester, all the final exam timeslots, and every student's schedule, the box will spit out a final exam schedule for CMU.

If CMU has n classes, how long would it take us to check if this schedule has any conflicts in it?





Semester & Mini-2 Final Exams: December 9, 10, 12, 13, 15 & 16(Make-Up Day)

Course	Section	Title	Date	Time (USA EST)	Classroom(s)
Architecture					
48116	A	BUILDING PHYSICS	Sunday, December 15, 2019	01:00 pm - 04:00 pm	To Be Announced (TBA)
48315	1	ENVIR I: CLIM & ENG	Thursday, December 12, 2019	08:30 am - 11:30 am	To Be Announced (TBA)
48432	A	ENV II	Thursday, December 12, 2019	08:30 am - 11:30 am	To Be Announced (TBA)
48531	A	FABRICATNG CUSTOMZTN	Monday, December 9, 2019	01:00 pm - 04:00 pm	To Be Announced (TBA)
48558	A	RLT COMP	Thursday, December 12, 2019	08:30 am - 11:30 am	To Be Announced (TBA)
48568	A	ADV CAD BIM 3D VISLZ	Tuesday, December 10, 2019	08:30 am - 11:30 am	To Be Announced (TBA)
48635	1	ENVIRO I M.ARCH	Thursday, December 12, 2019	08:30 am - 11:30 am	To Be Announced (TBA)
48655	A	ENV II GRAD	Thursday, December 12, 2019	08:30 am - 11:30 am	To Be Announced (TBA)
48714	A	DATA ANL URBN DSNG	Friday, December 13, 2019	01:00 pm - 04:00 pm	To Be Announced (TBA)
48729	A	PROD HLTH QUAL BLDGS	Thursday, December 12, 2019	01:00 pm - 04:00 pm	To Be Announced (TBA)
48734	A	RCTV SP MD ARC	Friday, December 13, 2019	05:30 pm - 08:30 pm	To Be Announced (TBA)
48743	A	INTRO ECO DES	Friday, December 13, 2019	01:00 pm - 04:00 pm	To Be Announced (TBA)
48749	A	CD SPECIAL TOPICS	Tuesday, December 10, 2019	01:00 pm - 04:00 pm	To Be Announced (TBA)
48785	A	MAAD RES PROJ	Sunday, December 15, 2019	05:30 pm - 08:30 pm	To Be Announced (TBA)
48798	A	HVAC & PS LOW CARB B	Monday, December 9, 2019	05:30 pm - 08:30 pm	To Be Announced (TBA)
Art					
60157	A	DRAWING NON-MAJORS	Tuesday, December 10, 2019	05:30 pm - 08:30 pm	CFA TBD
60218	A	REAL-TIME ANIMATION	Monday, December 9, 2019	08:30 am - 11:30 am	To Be Announced (TBA)
60220	A	TECH CHARACTER ANIM	Thursday, December 12, 2019	05:30 pm - 08:30 pm	To Be Announced (TBA)
60220	B	TECH CHARACTER ANIM	Thursday, December 12, 2019	05:30 pm - 08:30 pm	To Be Announced (TBA)
60333	A	CHARACTER RIGGING	Sunday, December 15, 2019	08:30 am - 11:30 am	BH 140F

Verifying a Final Exam Schedule

For every student, we need to go through all pairs of their classes, to see if any of their classes are in the same timeslot. Each student is likely enrolled in no more than 5 classes, so that's a constant number of checks – 10.

How many students are there? We can probably find a constant relation between the number of classes in a semester and the number of students on campus. Let's say if there are n classes, there are 6*n students.

That means that overall, we have to do 6*n*10 work. That's 60n, which is O(n). **Verifying the solution is tractable!**

Oracles

In computer science, we call the magical schedule-producing box an **oracle**. In ancient Greece, an oracle was a person who would make predictions about the future. In computer science, an oracle is a hypothetical algorithm that can produce a solution to a problem in a reasonable amount of time.

Oracles let us consider what we could do with a solution if one was produced quickly for us.

Complexity Classes

Now that we've talked about both solving and verifying problems, we can start putting problems into different groups.

We call these groups **complexity classes**. These are just collections of problems that have similar efficiency.

Complexity Class P

First, we define the complexity class P to be **the set of problems that we know can be solved in polynomial time**.

Our earlier examples (subset sum, puzzle solving, exam scheduling) don't fall into this category yet. But plenty of other algorithms do- linear search, selection sort, etc.

Complexity Class NP

Next, we'll define the complexity class NP to be **the set of problems that can be verified in polynomial time**.

This includes all problems in P- if you can solve something in polynomial time, you can check it as well.

But it also includes most of the problems we discussed before! We already showed that we can check exam scheduling in linear time. We can also check subset sum, Boolean satisfiability, and puzzle solving this way.

Not all Problems are in P or NP

Some problems are so difficult, we can't even verify them in polynomial time!

Travelling Salesperson is an example of this. If we're given a solution, we can't verify that it's the **best** path- it's just one possible path that exists.

We can turn Travelling Salesperson into an NP problem by changing the prompt: instead of finding the best path, just try to find a path that is less than X total distance, for some number X.

P vs NP

Big Question: Does P = NP?

Here's our big idea for the day. **Wouldn't it be nice if the set of problems P was the same as the set of problems NP?**

If this was true, we could put together CMU's final exam schedule in a day, instead of needing to wait half a semester to find out when exams will happen. We'd be able to solve a lot of hard problems really quickly!

Whether or not P = NP is a core question in the field of computer science, but it's still unsolved.

The first person who proves whether or not P = NP will win a million dollars, but no one has proved it yet...

Proving P != NP

Let's assume that P != NP. How would we prove this?

You'd need to definitively prove that a problem in NP exists that **cannot** be solved in polynomial time. But how can we show that it's impossible to do this?

This is tricky!

Proving P = NP

Let's assume P = NP. How would we prove this?

You need to show that **every** problem in NP can be solved in polynomial time. That's a lot of problems!

To make this easier, computer scientists try to find problems in NP that are related to each other.

Transforming Problems

Consider subset sum and Boolean satisfiability. We can **transform** subset sum into satisfiability. We just need to make a circuit that uses each value in the list as an input (0 if it isn't included, 1 if it is), and make the circuit output 1 if the included values sum to the target.

In fact, this mapping can be done in **polynomial time**. This means that if we can find a tractable solution to Boolean satisfiability, we can also use it to make a tractable solution to subset sum

Complexity Class NP-Complete

The complexity class NP-Complete is a set of problems where **every problem in NP-Complete** can be mapped to **any problem in NP** in polynomial time.

Why bother to make this complexity class? **If we find a tractable solution to any NP-Complete problem, we can make all problems in NP tractable;** we'll be able to solve all NP problems in polynomial time.

That will mean that P = NP!

NP-Complete Problems

If you use the limited version of the Travelling Salesperson problem, **all the problems we discussed today are in NP-Complete!**

There's lots more problems in this complexity class too: https://en.wikipedia.org/wiki/List_of_NP-complete_problems

If you can find a tractable solution to any of these problems, you'll prove P = NP and will become rich and famous.

Possible Outcomes

What happens if we prove P = NP?

We'll be able to solve a lot of hard problems very quickly. NP problems show up everywhere, so nearly everything in the world will get radically faster!

On the other hand, this might also wreck how modern security and encryption is implemented (as it will get easier to guess passwords).

What happens if we prove P != NP?

Not much- we still have to use slow or good-enough solutions to hard problems. But a lot of computer scientists can turn their focus to other problems.

Most people think P != NP, but we don't know how to prove it.

Learning Goals

Analyze and improve the **efficiency** of different programs

- Recognize problems that are **intractable** and (probably) cannot be solved quickly

- Understand the question and potential ramifications of **whether P = NP**