#14: Limits of Computation

SAMS SENIOR CS TRACK

Last Time

Used a **time loop** to create time-based animations.

Used **randomness** to simulate events.

Today's Learning Goals

Understand how **efficiency** changes how long a program takes to run.

Recognize that some problems **probably can't be solved efficiently**.

Recognize that other problems **probably can't be solved at all!**

Efficiency

Program Runtimes

So far, we've written programs that can run pretty quickly, usually because we're testing them on fairly small inputs.

When we test on larger inputs, programs can take longer to run...

Example: isLegalSudoku

```
def isLegalSudoku(board):
    for row in range(len(board)):
        for col in range(len(board)):
            for block in range(len(board)):
                if not isLegalRow(board, row):
                    return False
                if not isLegalCol(board, col):
                    return False
                if not isLegalBlock(board, block):
                    return False
    return True
```

```
def isLegalSudoku(board):
    for row in range(len(board)):
        if not isLegalRow(board, row):
            return False
    for col in range(len(board)):
        if not isLegalCol(board, col):
            return False
    for block in range(len(board)):
        if not isLegalBlock(board, block):
            return False
    return True
```

Algorithmic Efficiency

These two implementations have one core difference: **the number of operations that need to be performed.**

If we say `N = len(board)`, then the left `isLegalSudoku` calls each `isLegal` function `N**3` times. The right `isLegalSudoku` only calls each `isLegal` function `N` times!

In general, to improve algorithm runtime (and efficiency), we want to **reduce the number of operations we need to perform**. We especially want to pay attention to operations that **grow larger as the input grows larger**- that is, we care about operations that depend on N!

Calculating Runtime

We can actually test how long a program takes to run using the **time** library:

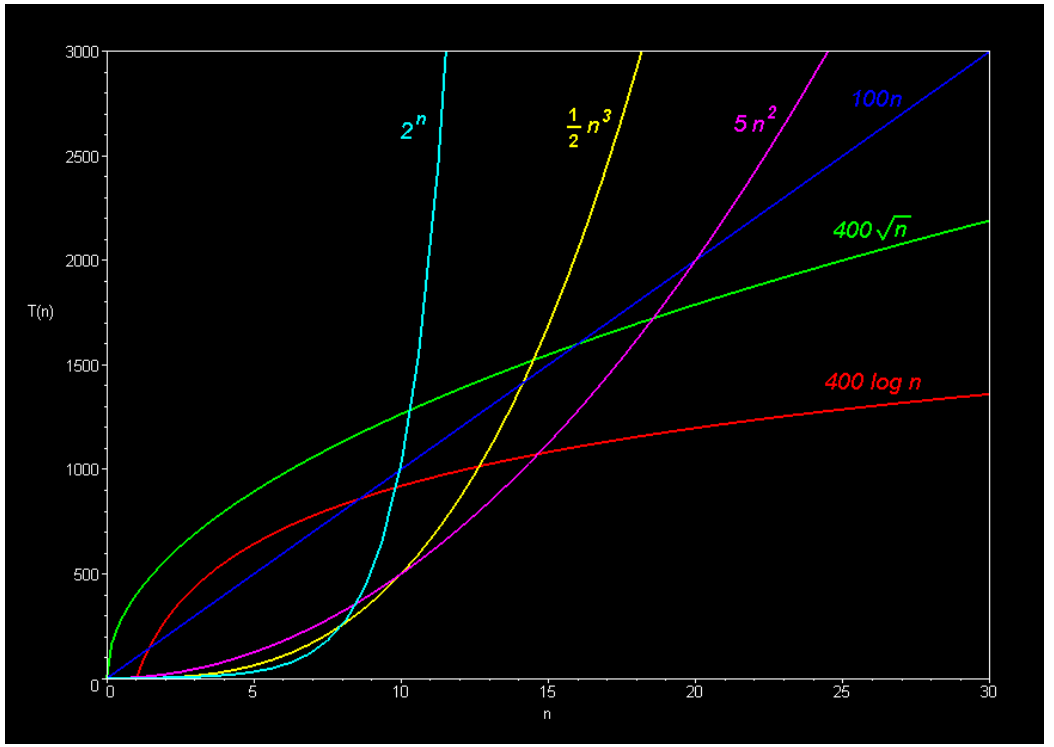
https://docs.python.org/3/library/time.html

`time.time()` returns the number of seconds since the epoch (when computers started recording time). We can use it to measure how long it takes a function to run:

```
t1 = time.time()
runFunction()
t2 = time.time()
print("Time passed: " + str(t2 - t1))
```

Big-O Notation and Function Families

Computer scientists use **Big-O notation** to mathematically approximate the runtime of a program. Big-O notation tells us the **function family** a program runs in assuming the **worst possible case**.



Constant time ($O(1)$) and linear time ($O(N)$) are both fairly fast.

Polynomial time ($O(N^2)$, $O(N^3)$) is slow, but okay.

Exponential time ($O(2^N)$, $O(N!)$) is horrible!

Improving Efficiency

If you have a program that takes a while to run, try to improve its efficiency by considering whether the program does **unnecessary work**, especially work that grows with the size of the input.

If you can reduce the order of the program's function family, you'll find that it starts running much faster!

What is the efficiency of the function on the right? Can we improve it?

```
def areLegalValues(lst): # assume len(lst) is a square num
    for i in range(len(lst)):
        for j in range(len(lst)):
            if i != j and lst[i] == lst[j] and lst[i] != 0:
                return False # check repeating values

            found = False
            for num in range(0, len(lst)+1):
                if num == lst[i]:
                    found = True
            if found == False: # check out of range
                return False
    return True
```

Example: subsetSum

Problem: Given a list lst of N elements and an integer target, can we find a sublist of lst that sums to target?

Solution: produce all possible sublists (using **recursion**), return the first one that sums to the target sum.

This works, but is slow on even medium-sized inputs! It runs in $O(2^n)$ time, because the number of possible sublists doubles every time we add a new element.

This is exponential, which is bad. **Can we do better?**

```
def subsetSum(lst, target):
    if sum(lst) == target:
        return lst
    elif len(lst) == 0:
        return None
    tmp = subsetSum(lst[1:], target - lst[0])
    if tmp != None:
        return [ lst[0] ] + tmp
    return subsetSum(lst[1:], target)
```

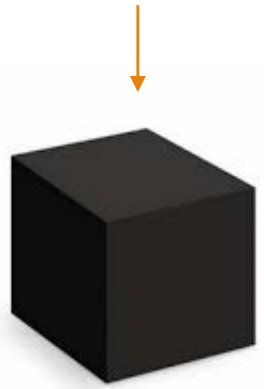
Verifying a Solution

Assume we have a magic box that can take in a list and produce an answer to subsetSum for that list. We want to check if this box is legit.

Discuss: How long does it take to verify that that answer is correct (is a subset of lst and sums to target)?

Answer: $O(N^2)$ for checking the subset, O(N) for checking the sum. **Verifying is polynomial!**

```
subsetSum([16,37,6,40,96,34,16,66], 112)
```



```
[16, 6, 40, 34, 16]
```

P vs. NP

Complexity Classes P and NP

There is a class of problems that can have solutions **verified** in polynomial time. This class is called NP, short for “non-deterministic polynomial time”. A function like solveSudoku is in NP.

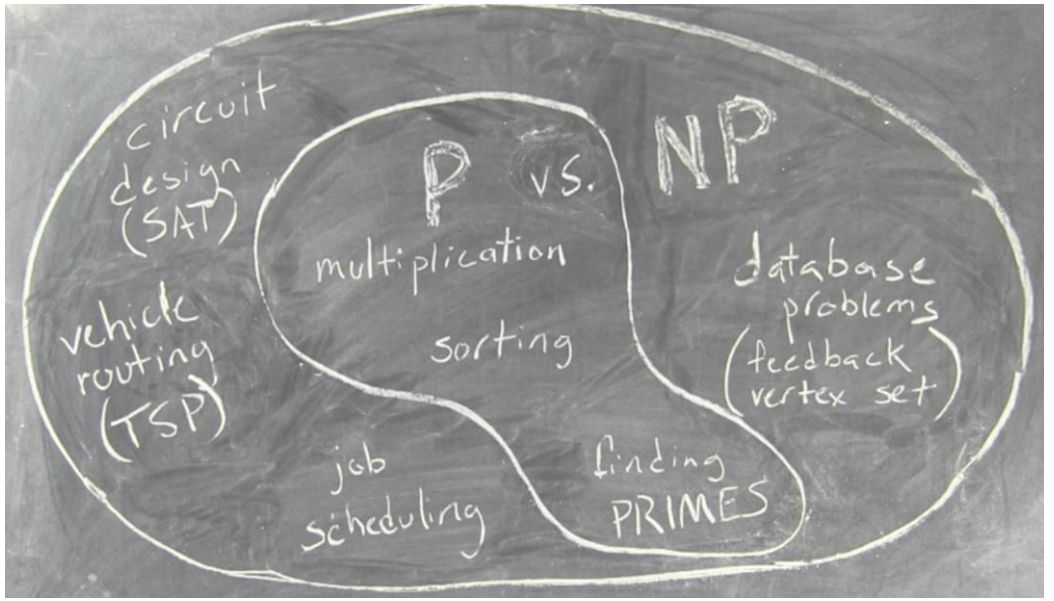
There is another class of problems that can be **solved** in polynomial time. This class is called P, for polynomial. A function like isPrime is in P.

So far we’ve established that subsetSum is in NP. We don’t yet know whether it’s in P- maybe we could make a faster solution, and then it would be. In general, we know that **P is a subset of NP** (if we can solve in polynomial time, we can verify too!).

Problems in NP

There are lots of common and useful problems in the class NP, problems that we don't have a polynomial-time solution for (yet). These include:

- Subset sum
- Optimized packing of items
- Route-planning (Travelling Salesman)
- Coloring a graph (solveSudoku)
- Scheduling with constraints (final exams)
- And many more...



P vs. NP

Big Idea: wouldn't it be nice if all problems in NP were also in P? Then we could have fast solutions to lots of problems! (Though this would also break most encryption methods…) In other words, **could P = NP?**

Alternatively, if we can't have this nice thing, wouldn't it be great to prove that it's impossible for some problem like subsetSum to have a polynomial-time solution, so we can stop wasting time trying to find one? In other words, **can we show P != NP?**

This question of whether or not P = NP is one of the most important problems in computer science. It's also one of the seven Millenium Prize problems.

How Can We Show P = NP?

If you want to demonstrate that P = NP, you need to show that **all problems in NP are also in P**.

There are a lot of problems in NP! To make this easier, computer scientists have identified a set of problems called **NP-Complete** that can make one solution usable across multiple NP problems.

We call a problem NP-Complete if it meets two requirements:

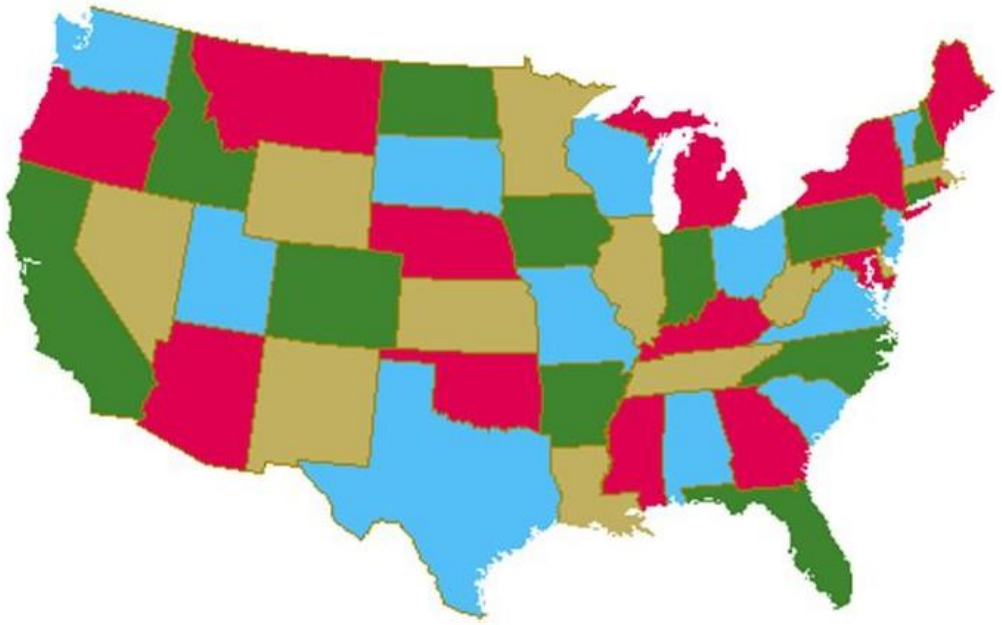
- it is in NP
- we can transform **any problem in NP** into it in polynomial time

Example: Graph Coloring

A classic NP-Complete problem is **graph coloring**. Given a graph, find a way to color the nodes so that no two connected nodes have the same color, using only a specific number of different colors.

We can demonstrate how this works by thinking about coloring a map. The states are nodes, and bordering states are connected by edges.

Let's show how we can use a solution to graph coloring to solve another NP problem, solveSudoku.



Graph Coloring to Sudoku Solving

1. Start with an unsolved Sudoku board.
2. Map the numbers 1-9 to 9 individual colors.
3. Create a graph with a node for each space in the Sudoku board. Give nodes colors if they already had numbers assigned.
4. Connect all nodes that were in the same row, column, or block with edges.
5. Send the graph to the graph coloring algorithm, and get the result.
6. Transform the result graph back into a Sudoku table, and the colors back into numbers.
7. You're done!

Note that all of the steps **except step 5** can definitely be done in polynomial time.

How Can We Show P = NP?

If you find a solution to an NP-Complete problem (like graph coloring or subsetSum), you can make step 5 polynomial time.

That means that every other NP problem can be solved in **polynomial time**. In other words, you can solve all NP problems with just one NP-Complete solution!

Most researchers who are trying to show that P = NP take this approach.

How Can We Show P != NP?

If you want to demonstrate that P != NP, you need to prove that at least one NP problem **cannot be solved in polynomial time**.

How do we prove that it's impossible to find a better solution? You need to consider all possible situations so you don't miss an unusual, clever algorithm. Writing proofs like this is a large part of theoretical computer science.

Most computer scientists think that P != NP, but proving this is very tricky.

Conclusion

We can't yet prove whether or not P = NP, even though computer scientists have been working on this problem for years.

However, there are many other things that we can prove! Let's start with one basic question: **are there any programs we can't write?**

Computability

Computability

Computability theory is the study of defining algorithms/procedures, usually in a mathematical context.

Let's start with a more practical context. **For a given program, can be ensure that that program works correctly?**

Motivation

Two Boeing 737 jets have crashed in the past year, both seemingly due to technical errors in the automation system.

Why did this happen? Shouldn't we be able to write code that we can be **100% sure** will work correctly?

The Perfect Test Function

Goal: we want to write the ideal test function, one that will verify whether a given program returns the correct result on **all possible inputs**. Let's call it testAll(f). Note that in this context, f will be a function- we can do this because **function names are references**, like variable names!

To test all possible inputs, we first need to make sure we don't get infinite loops/infinite recursion on any input to our function. If we don't check this, testAll(f) may take forever to run!

```
def testAll(f):
    if not alwaysHalts(f):
        return False
    ...
```

The Perfect Halting Function

New goal: write the program alwaysHalts(f), which returns True if f 'halts' (stops and returns a value) on all possible inputs to f.

To solve this, we must write the program halts(f, inp), which returns True if f halts on the given function and input, and False otherwise.

```
def alwaysHalts(f):
    for inp in allPossibleInputs(f):
        if not halts(f, inp):
            return False
    return True
```

The Halting Problem: can we write a program to do this?

No.

Let’s use a Proof by Contradiction to show why.

Proof by Contradiction

To show that the program halts() cannot exist, we only need to find one program f and one input inp such that it is **impossible** for halts(f, inp) to return the correct result.

To do this, let's design a program, breakHalts(f), which **uses halts to break itself**.

Here's the big question: **does breakHalts halt when given itself as an input, or not?**

```
def breakHalts(f):
    inp = f
    if halts(f, inp):
        print('Running forever!')
        while True: pass
    else:
        print('Halting!')
        return
```

Case One: breakHalts() halts

Assume that breakHalts(breakHalts) **does** halt.

Therefore, halts(breakHalts, breakHalts) should return True, and we enter the if case.

Then we enter an infinite while loop… and the program never halts.

CONTRADICTION!

```
def breakHalts(f):
    inp = f
    if halts(f, inp): # True
        print('Running forever!')
        while True: pass
    else:
        print('Halting!')
        return
```

Case Two: breakHalts() Loops Forever

Assume that breakHalts(breakHalts) will not halt, and will instead loop forever.

Therefore, halts(breakHalts, breakHalts) should return False, and we enter the else case.

But then we immediately return, which means the program halts!

CONTRADICTION!

```
def breakHalts(f):
    inp = f
    if halts(f, inp): # False
        print('Running forever!')
        while True: pass
    else:
        print('Halting!')
        return
```

Some Functions are Uncomputable

We just showed that it is impossible to write the program breakHalts and call it on itself. But the program we used only had one unusual bit of code- the call to halts(). Therefore, **it is impossible to write the program halts()**.

Since we can't write halts(), we can't write alwaysHalts(), and since we can't write alwaysHalts(), we can't write testAll(). These problems are **uncomputable**- we cannot write a program to compute them, no matter how clever we are.

Takeaway: there are some programs that are simply impossible to write!

We still need to write programs.

Can we make them fast and correct?

How to Solve a Problem in NP?

Option 1: only run your function on small inputs. (Then bad efficiency doesn't matter)

Option 2: use heuristics to find a 'good-enough' solution

Example: scheduling final exams at CMU

Big Idea: if you can identify when your algorithm is non-polynomial, you can find workarounds to deal with it!

How to Verify that your Code Works?

We can’t write a universal test function for code.

But we can **prove** that certain functions will behave as expected on certain classes of inputs.

We can also use **contracts** to ensure that functions only accept certain types of input and only return certain types of output.

Big Idea: test your code well and often and it will be robust, if not perfect.

Today's Learning Goals

Understand how **efficiency** changes how long a program takes to run.

Recognize that some problems **probably can't be solved efficiently**.

Recognize that other problems **probably can't be solved at all!**