The Limits of Computing

Predicting Running Time of a Program

Suppose you are working on a very important problem and wrote a program to make lots of calculations. You expect that it may take a while to produce a result.

- How long will you wait?
- Should you wait or stop?
- You waited for a few days and decided to stop, but what if it will end/halt in the next 5 minutes?

Classifying Problems

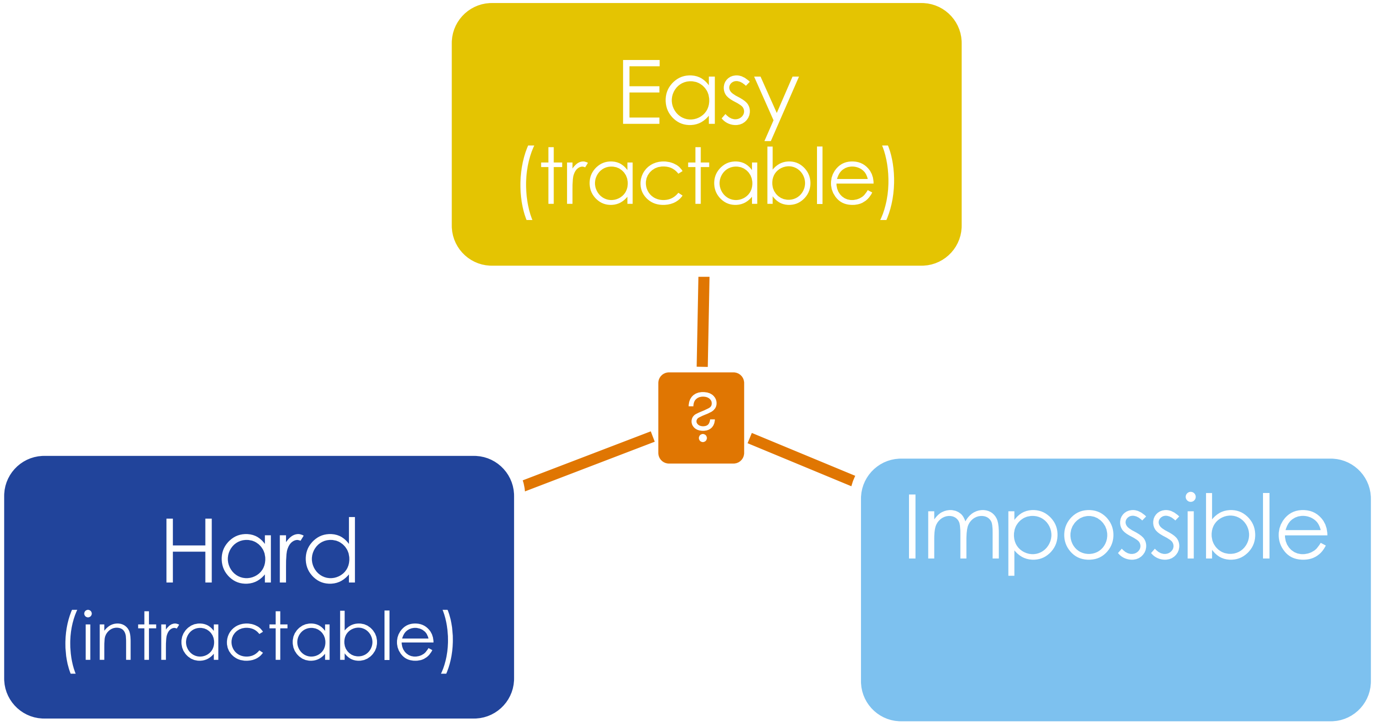
- Can you say if your program will terminate and return a result?
- Can you say anything about the hardness of the problem that you are trying to solve with your program?

Complexity and Computability Theories

- Computer scientists are interested in measuring the **"hardness"** of computational problems in order to understand **how much time**, **or some other resource such as memory**, is needed to solve it.

- What problems **can or cannot be solved** by mechanical computation? Can we categorize problems as "easy", "hard", or "impossible"?

Can we **categorize** problems?



Easy (tractable)

?

Hard (intractable)

Impossible

Easy, Hard, Impossible

- An "easy (i.e. tractable)" problem is one for which there exists a mechanical procedure (i.e. program or algorithm) that can solve it in a reasonable amount of time.
- A "hard (i.e. intractable)" problem is one that is solveable by a mechanical procedure but every algorithm we can find is so slow that it is practically useless.
- An "impossible (i.e. uncomputable)" problem is one such that it is provably impossible to solve no matter how much time we are willing to use.

Easy (Tractable)

- An “easy (i.e. tractable)” problem is one for which there exists a mechanical procedure (i.e. program or algorithm) that can solve it in a **reasonable amount of time.**

How do we measure this?

Hard (Intractable)

- A “hard (i.e. intractable)” problem is one that is **solveable** by a mechanical procedure but every algorithm we can find is so slow that it is **practically useless.**

What does this mean?

Impossible

- An “impossible” problem is one such that it is **provably impossible** to solve no matter how much time we are willing to use.

How can we prove something like that?

Why Study Impossibility?

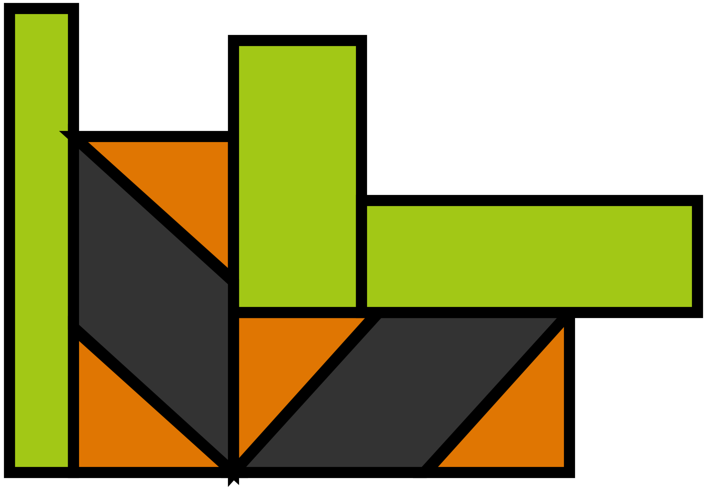
- Practical: If we know that a problem is unsolvable we know that we need to simplify or modify the problem.

- Cultural: Gain perspective on computation.

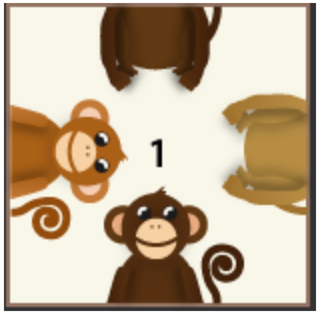
Decision Problems

- A specific set of computations are classified as decision problems.
- An algorithm solves a **decision problem** if its output is simply YES or NO, depending on whether a certain property holds for its input. Such an algorithm is called a **decision procedure**.
- Example:

  Given a set of *n* shapes, can these shapes be arranged into a rectangle?



The Monkey Puzzle

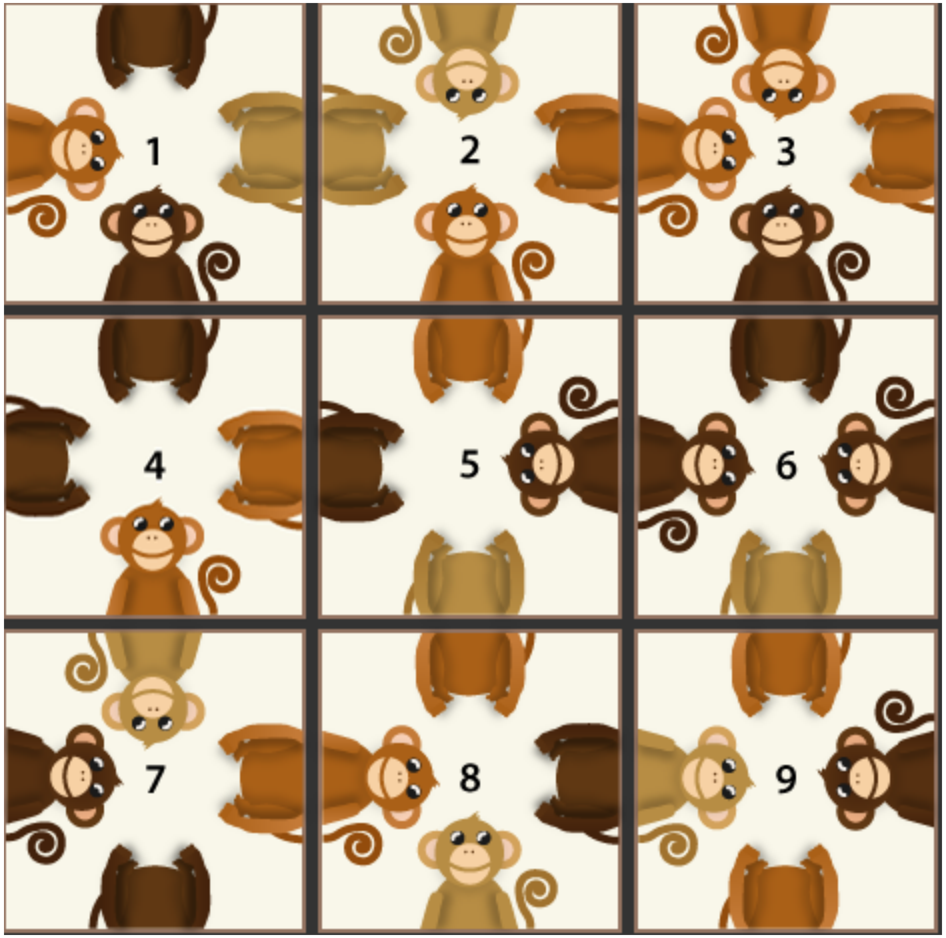


- Given:
  - A set of **n** square cards whose sides are imprinted with the upper and lower halves of colored monkeys.
  - **n** is a perfect square number, such that **n = m**^{2}.
  - Cards cannot be rotated.

decision problem

- Problem:
  - **Does there exist an arrangement of the n cards in an m x m grid** such that each adjacent pair of cards display the upper and lower half of a monkey of the same color.

Example



- Can we always compute a YES/NO answer to the problem?
- If we can, is the problem tractable (easy to solve) in general?

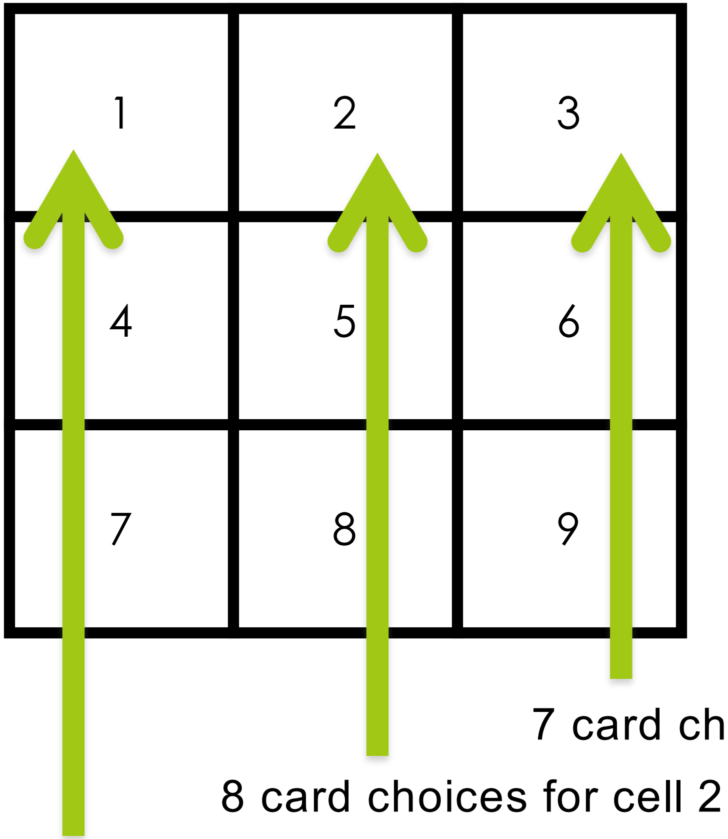
Algorithm

Simple brute-force (exhaustive search) algorithm:

- Pick one card for each cell of *m* x *m* grid.
- Verify if each pair of touching edges make a full monkey of the same color.
- If not, try another arrangement until a solution is found or all possible arrangements are checked.
- Answer "YES" if a solution is found. Otherwise, answer "NO" if all arrangements are analyzed and no solution is found.

Analysis

Suppose there are *n* = 9 cards (*m* = 3)



The total number of unique arrangements for *n* = 9 cards is:

9 * 8 * 7 * .... *1 = 9! (9 factorial)
= 362880

7 card choices for cell 3

goes on like this

8 card choices for cell 2

9 card choices for cell 1

Analysis (cont’ d)

For ***n*** cards, the number of arrangements to examine is *n!*

Assume that we can analyze one arrangement in a microsecond (µs), that is, analyze 1 million arrangements in one second :

n	Time to analyze all arrangements
9	362,880 µs
16	20,922,789,888,000 µs (app. 242 days)
25	15,511,210,043,330,985,984,000,000 µs (app. 500 billion years)

Age of the universe is about 14 billion years

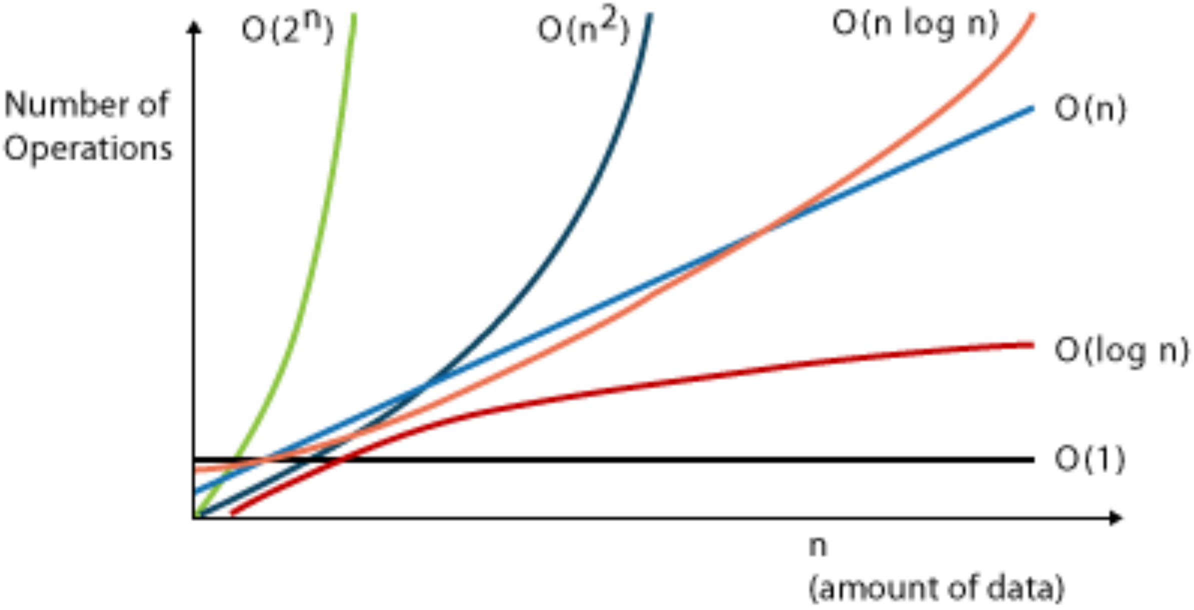
Classifying Problems

- The field of computational complexity categorizes decidable decision problems by how hard they are to solve. “Hard“ in this sense, is described in terms of the computational resources needed by the most efficient algorithm that is known to solve the problem.

Reviewing the Big O Notation (1)

- We use the big O notation to indicate the relationship between the size of the input and the corresponding amount of work.
- For the Monkey Puzzle
  - Input size: Number of tiles (*n*)
  - Amount of work: Number of operations to check if any arrangement solves the problem (*n!*)
- For very large ***n* (size of input data)**, we express the number of operations as the **(time) order of complexity**.

Growth of Some Functions



Big O notation:

gives an asymptotic upper bound

ignores constants

Any function *f(n)* such that *f(n)* ≤ *c* n^2 for large *n* has *O*(n^2) complexity

Quiz on Big O

- What is the order of complexity in big O for the following descriptions
  - The amount of computation does not depend on the size of input data

    O(1)
  - If we double the input size the work is doubles, if we triple it the work is 3 times as much

    O(n)
  - If we double the input size the work is 4 times as much, if we triple it the work is 9 times as much

    O(n^2)
  - If we double the input size, the work has 1 additional operation

    O(log n)

Classifications

- Algorithms that are $O(n^k)$ for some fixed k are polynomial-time* algorithms.
  - $O(1)$, $O(\log n)$, $O(n)$, $O(n \log n)$, $O(n^2)$
  - reasonable, tractable
- All other algorithms are super-polynomial-time algorithms.
  - $O(2^n)$, $O(n^n)$, $O(n!)$
  - unreasonable, intractable

*A **polynomial** is an expression consisting of variables and coefficients that involves only the operations of addition, subtraction, multiplication, and non-negative integer exponents.

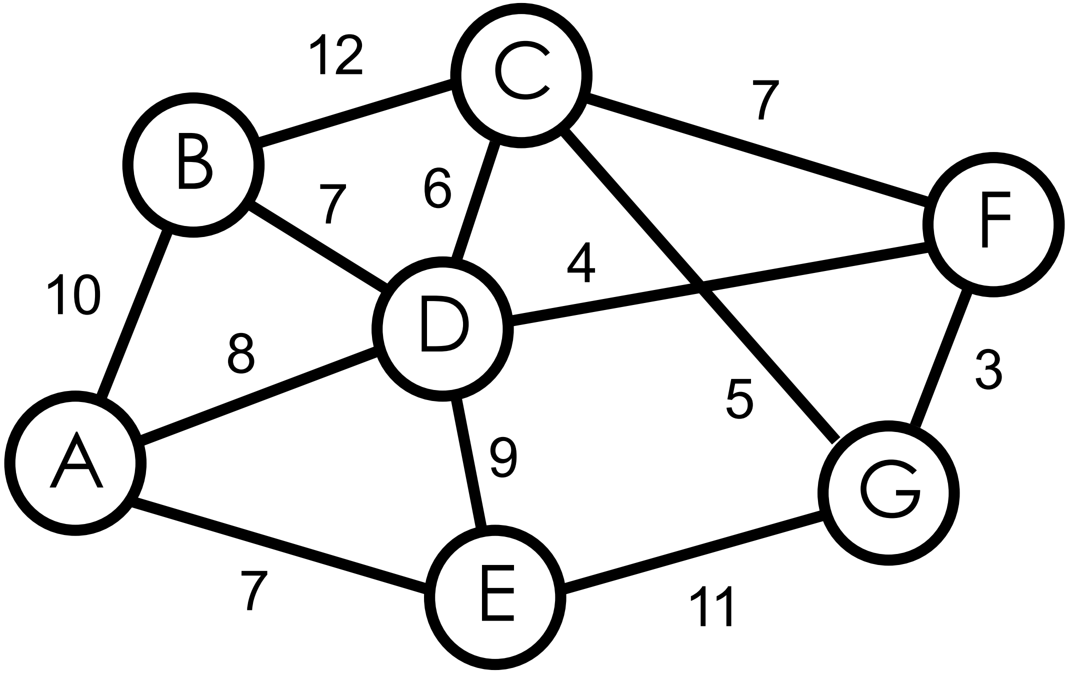
A Famous Hard Problem

Traveling Salesperson

- Given: a weighted graph of nodes representing cities and edges representing flight paths (weights represent cost)

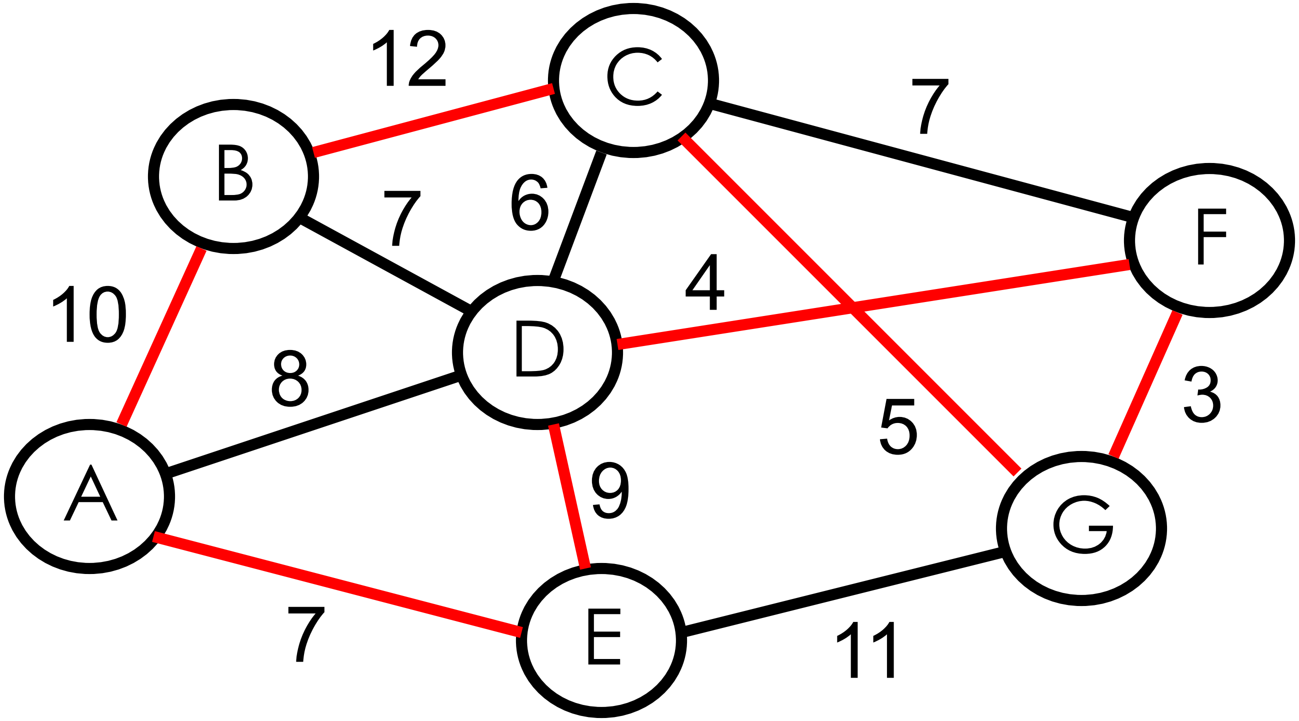
- Is there a route that takes the salesperson through every city and back to the starting city with cost no more than *k*?
  - The salesperson can visit a city only once (except for the start and end of the trip).

An Instance of the Problem



Is there a route that takes the salesperson through every city and back to the starting city with cost no more than *52*?

Traveling Salesperson



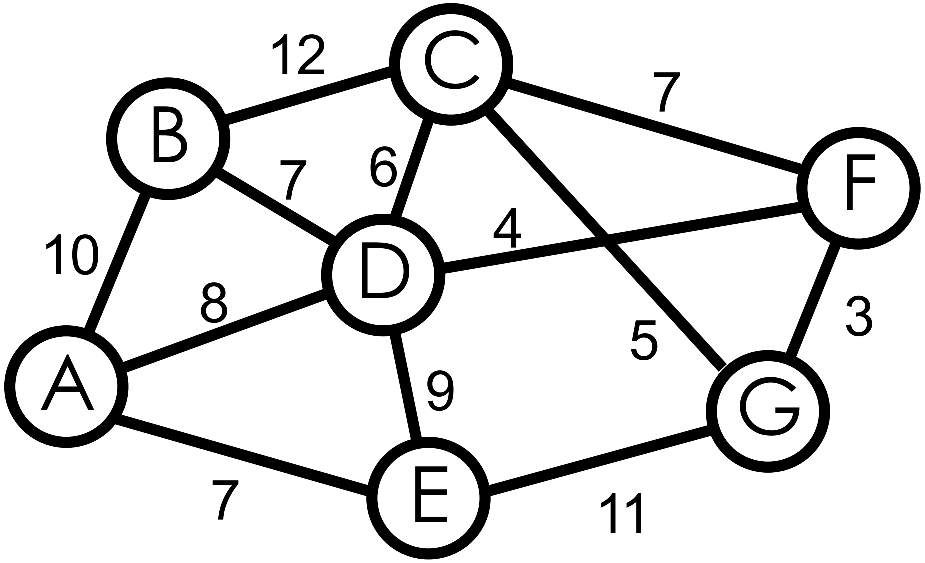
Is there a route with cost at most 52? YES (Route above costs 50.)

If I am given a potential solution I can verify that to say yes or no, but otherwise I have to search for it. By a brute-force approach, I enumerate all possible routes visiting every city once and check for the cost.

Analysis

- If there are *n* cities, what is the maximum number of routes that we might need to compute?
- Worst-case: There is a flight available between every pair of cities.
- Compute cost of every possible route.
  - Pick a starting city
  - Pick the next city (n-1 choices remaining)
  - Pick the next city (n-2 choices remaining)
  - …
- Maximum number of routes: __________

Number of Paths to Consider



Number of all possible routes = Number of all possible permutations of n nodes = n!

Observe ABCGFDE is equivalent to BCGFDEA (starting from a point and returning to it going through the same nodes)

Number of all possible unique route = n! / n = n – 1!

Observe also that ABCGFDE has the same cost as EDFGCBA

Number of all possible paths to consider = (n – 1)! / 2

Still O(n!)

Analysis

- If there are *n* cities, what is the maximum number of routes that we might need to compute?

- Worst-case: There is a flight available between every pair of cities.

- Compute cost of every possible route.
  - Pick a starting city
  - Pick the next city (n-1 choices remaining)
  - Pick the next city (n-2 choices remaining)
  - …

- Worst-case complexity: ____________ O(n!)

Note: n! > 2 ^{n}
for every n > 3.

Exponential complexity (super-polynomial time)

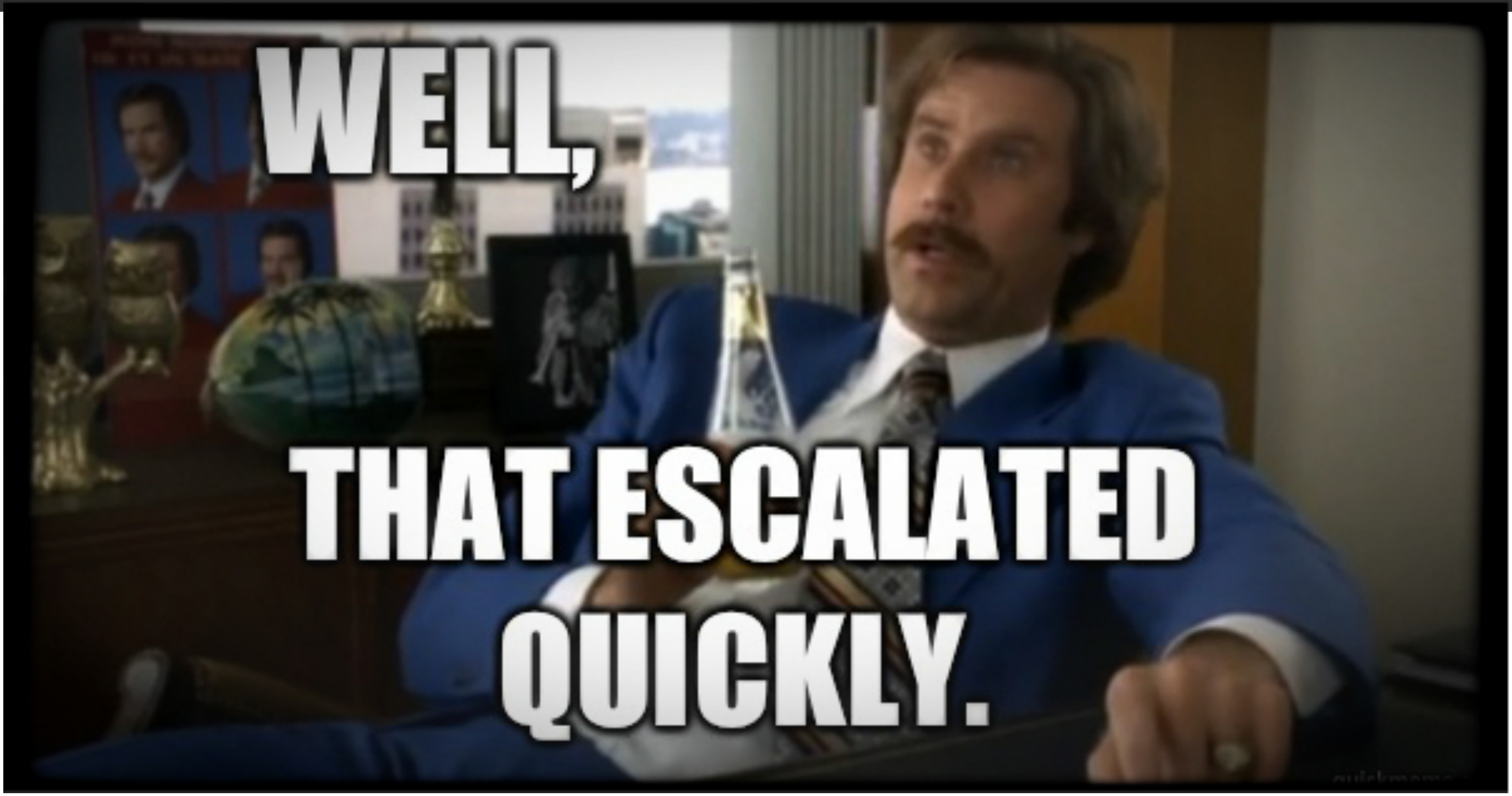
Polynomial vs. Exponential Growth

Assumption: Computer can perform one billion operations for second

Running Time	Size n = 10	Size n = 20	Size n = 30	Size n = 40
n	0.00000001	0.00000002	0.00000003	0.00000004 sec.
n^2	0.00000010	0.00000040	0.00000090	0.00000160 sec.
n^3	0.00000100	0.00000800	0.00002700	0.00006400 sec.
n^5	0.00010000	0.00320000	0.02430000	0.10240000 sec.
n!	0.0036	77.1 years	8400 trillion years	2.5 * 10^{31} Years□

Source: http://www.cs.hmc.edu/csforall

Polynomial vs. Exponential Growth



n! → 0.0036 → 77.1 years → 8400 trillion years → 2.5 * 10^{31} Years□

Source: http://www.cs.hmc.edu/csforall

The Big Picture

- Intractable problems are solvable if the amount of data (*n*) that we are processing is small.
- But if *n* is not small, then the amount of computation grows exponentially and the solutions quickly become out of our reach.
- Computers can solve these problems if *n* is not small, but it will take far too long for the result to be generated.
  - We would be long dead before the result is computed.

Thursday

- Exam tomorrow
- Calculating Grades over weekend due by Tuesday
- Extra Credit: Honest attempt at Post-Test in OLI

Summary

- For many interesting problems naïve algorithms rely on exhaustive search
  - Check all possible answers
  - Exponential running time (intractable)
- We need smarter algorithms for them to be practical (avoid exhaustive search)

Dealing with Intractability

- Restrict the problem, exploiting properties of specific instances of the problem.
- Trade correctness with tractability.
  - Go for approximate solutions.
  - Get correct result with some probability.

Satisfiability

- Given a Boolean formula with *n* variables using the operators AND, OR and NOT:
  - Is there an assignment of Boolean values for the variables so that the formula is true (satisfied)?

    Example: (X AND Y) OR (NOT Z AND (X OR Y))
  - Truth assignment: *X* = True, *Y* = True, *Z* = False.
- How many assignments do we need to check for *n* variables?
  - Each symbol has 2 possibilities 2^n assignments

Verifiability

- No known tractable algorithm to decide, however it is easy to verify a solution.

Decision Problems

- We have seen 3 examples of decision problems with simple brute-force algorithms that are intractable.

The Monkey Puzzle	$O(n!)$
Traveling Salesperson	$O(n!)$
Satisfiability	$O(2^n)$

We can avoid brute-force in many problems and obtain polynomial time solutions, but not always. For example, satisfiability of Boolean expressions of certain forms have polynomial time solutions.

Special Cases of a Problem May be Tractable

- General Boolean satisfiability we just talked about (let us call it SAT) is not tractable but 2-satisfiability is.
  - **2-satisfiability** (2-SAT): determining whether a conjunction of disjunctions (and of ors), where each disjunction (or operation) has two arguments that may either be variables or the negations of variables, is satisfiable

  Example: (X OR Y) AND (Z OR (NOT Y))

Formulas are of a special form

Are These Problems Tractable?

- For any one of the intractable problems we saw, is there a single tractable (polynomial) algorithm to solve any instance of the problem?

  Haven't been found so far.

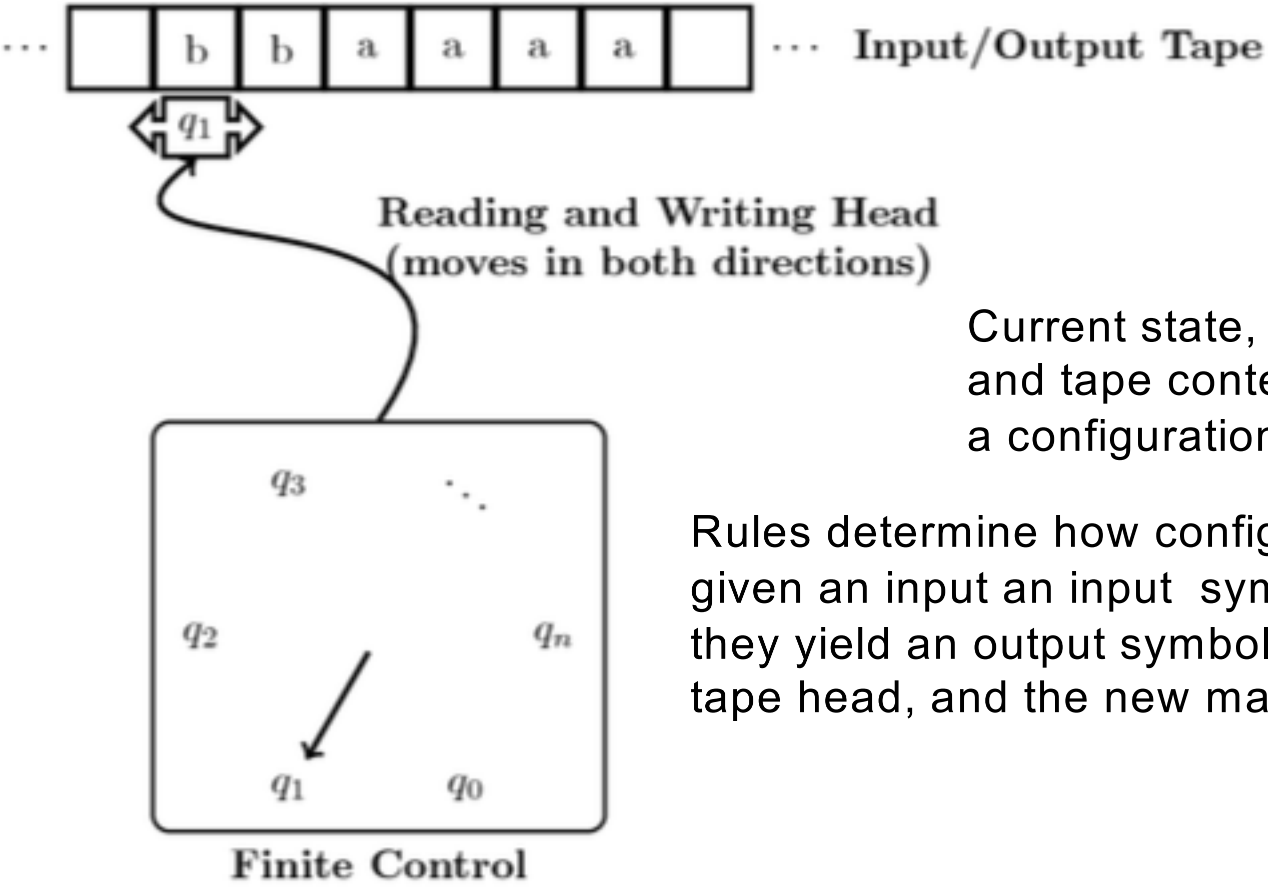
- Possible reasons:
  - These problems have undiscovered polynomial-time solutions.
  - These problems are intrinsically difficult – we cannot hope to find polynomial solutions.

- Important discovery: Complexities of some of these problems are linked. If we can solve one, we can solve the other problems in that class.

Modeling Computing

- A rigorous discussion on these questions requires a model that can model every possible mechanical procedure. One such model is that of a Turing Machine.
- A Turing machine can do what any other computer can do. If it cannot do something then real machines would not be able to either.
- If we can reason about the number of steps it would take a Turing Machine to solve a problem, we could make general claims about the number of steps it would take any computer to solve a problem.

Turing Machines



Current state, head location, and tape contents determine a configuration of the machine

Rules determine how configurations evolve: given an input an input symbol and a state, they yield an output symbol, direction for the tape head, and the new machine.

Remark: You don’t need to memorize these. The next slide shows what you need to remember.

Turing Machines (cont’ d)

- A Turing machine *M* computes a function *f* if:
  - *M* **halts on all inputs.**
  - **On input x**, **it writes *f(x)*** on the tape and halts.

P and NP

Polynomial decidability

The class **P** consists of all those decision problems that can be solved on *a deterministic sequential machine* in an amount of time that is polynomial in the size of the input

N in NP comes from nondeterministic

Polynomial verifiability

The class **NP** consists of all those decision problems whose positive solutions can be verified in polynomial time given the right information, or equivalently, whose solution can be found in polynomial time on a *non-deterministic machine*.

Decidability vs. Verifiability

P = the class of problems that can be decided (solved) quickly

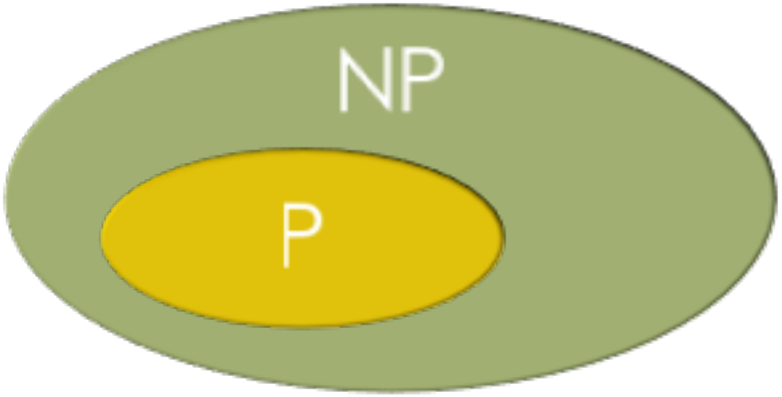
NP = the class of problems for which solutions can be verified quickly

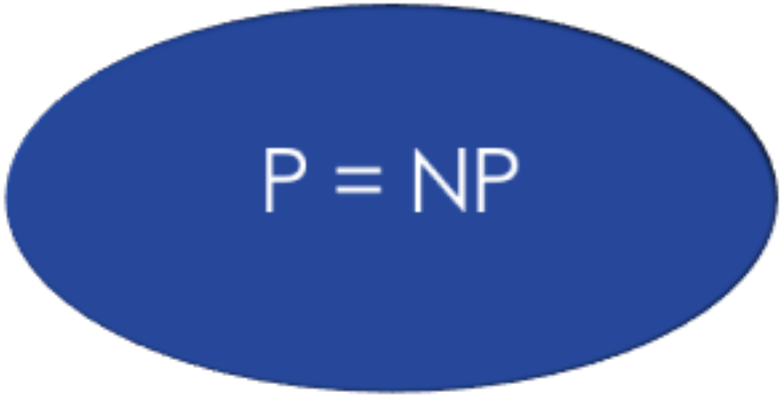
Example

	Verifiable in Polynomial Time NP	Solvable in Polynomial Time P
Given an integer list, is 10 in the list?	YES	YES
Satisfiability problem	YES	?
Traveling Salesperson	YES	?

- If a problem is in **P**, it must also be in **NP**.
- If a problem is in **NP**, is it also in **P**?

Two Possibilities





If P ≠ NP, then
some decision problems can't be solved in polynomial time.

If P = NP, then
all polynomially verifiable problems can be solved in polynomial time.

*The Clay Mathematics Institute is offering a **$1M** prize for the first person to prove P = NP or P ≠ NP.*

(http://www.claymath.org/millennium/P_vs_NP/)

Watch out, Homer!

In the 1995 Halloween episode of The Simpsons, Homer Simpson finds a portal to the mysterious Third Dimension behind a bookcase, and desperate to escape his in-laws, he plunges through. He finds himself wandering across a dark surface etched with green gridlines and strewn with geometric shapes, above which hover strange equations. One of these is the deceptively simple assertion that P = NP.

NP-Complete Problems

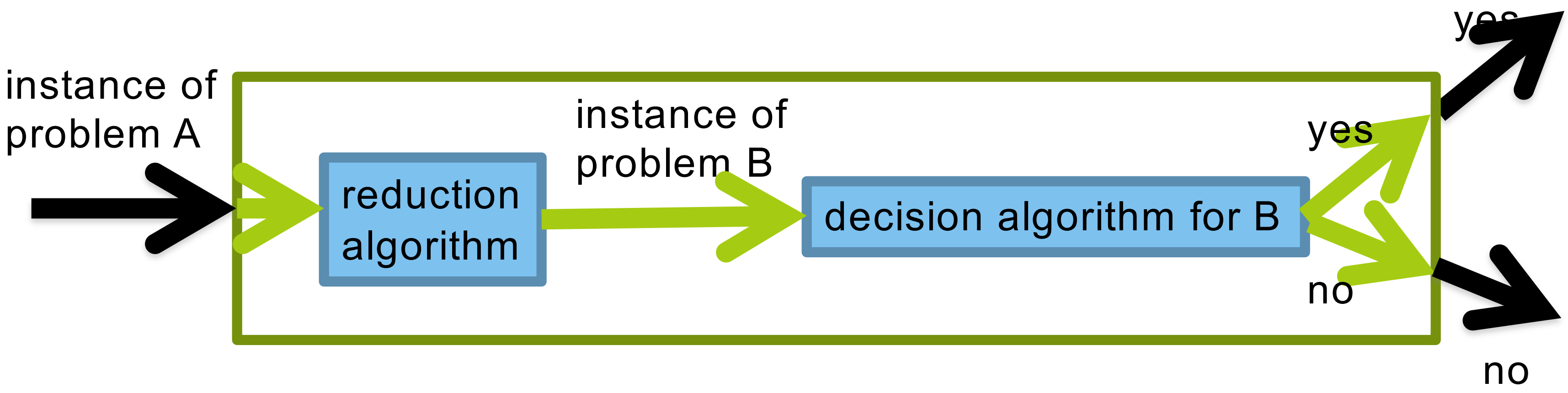
- An important advance in the P vs. NP question was the discovery of a class of problems in NP whose complexity is related to the whole class [Cook and Levin, ‘70]: if one of these problems is in P then NP = P.

Reducability

- A reduction is a way of converting one problem to another problem in such a way that a solution to the second problem can be used to solving the first problem.
  - Real life examples: You can reduce the problem of finding your way around a city to the problem of obtaining a map of that city, reduce the problem of traveling to NY from Pgh to buying a ticket
  - Math example: Finding the area of a square reduces to the problem of finding its length

Reductions for solving decision problems

- Consider 2 problems A and B: Suppose we are trying to solve A and have a decision algorithm for B.



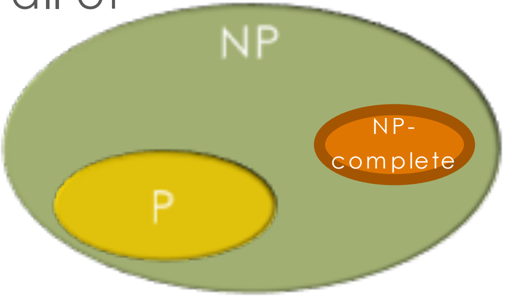
- Reduction algorithm should be polynomial time and the reduction should be such that A and B give the same result in all cases

NP-completeness

- A problem A is NP-complete if
  - A is in NP
  - Every other problem in NP is polynomial time reducible to A (there is an efficient way to transform each problem in NP to A).

Some Remarks on NP-Completeness

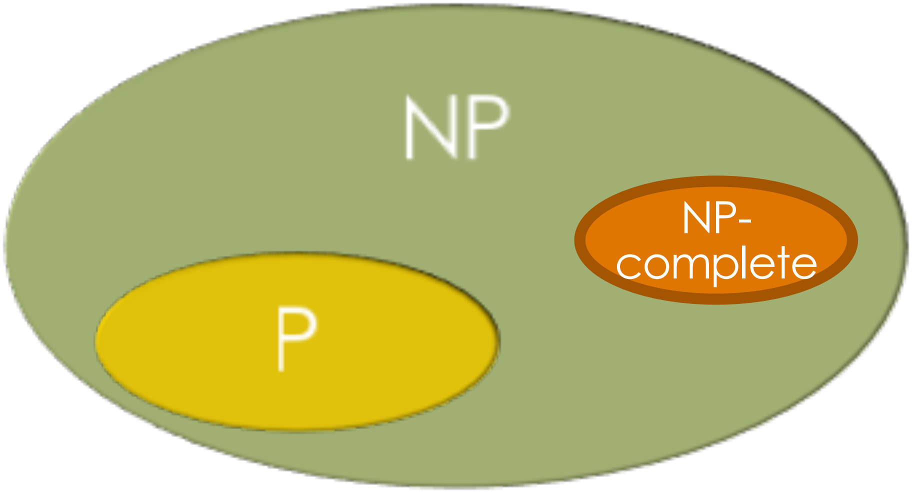
- The class **NP-Complete** consists of all those problems in NP that are least likely to be in P.
  - Monkey puzzle, Traveling salesperson, and Satisfiability are all in NP-Complete.
- Every problem in NP-Complete can be transformed to another problem in NP-Complete.
  - If there were some way to solve one of these problems in polynomial time, we should be able to solve all of these problems in polynomial time.



Informally, NP-complete problems are the hardest problems in NP.

Why is NP-completeness of Interest?

Theorem: If any NP-complete problem is in P then all are and P = NP.



Most believe P ≠ NP. So, in practice NP-completeness of a problem prevents wasting time from trying to find a polynomial time solution for it.

NP-completeness in Practice

- Since the discovery that SAT is NP-complete, thousands of problems have been proved NP-complete.
  - NP-completeness is mentioned as a keyword in 6,000 scientific papers per year. "Captures vast domains of computational, scientific, mathematical endeavors, and seems to roughly delimit what mathematicians and scientists had been aspiring to compute feasibly." [Papadimitriou]
- If you have a problem that is in NP, and you don’t know a polynomial time algorithm for it, it may be reasonable to assume that it is NP-complete until proved otherwise.

Examples of NP-complete Problems

- **Bin Packing**. You have n items and m bins. Item i weighs w[i] pounds. Each bin can hold at most W pounds. Can you pack all n items into the m bins without violating the given weight limit?

- **Machine Scheduling.** Your goal is to process n jobs on m machines. For simplicity, assume each machine can process any one job in 1 time unit. Also, there can be precedence constraints: perhaps job j must finish before job k can start. Can you schedule all of the jobs to finish in L time units?

- **Crossword puzzle.** Given an integer N, and a list of valid words, is it possible to assign letters to the cells of an N-by-N grid so that all horizontal and vertical words are valid?

Source: http://algs4.cs.princeton.edu/66intractability/

Decision Problems Vs. Optimization Problems

- We can usually cast a given optimization problem as a related decision problem by imposing a bound on the value to be optimized
  - Optimization version of Traveling Salesperson: What is the shortest route that goes through every city?
  - We can show an optimization problem to be hard by using its relationship to the decision problem, because a decision problem is “no harder” than the optimization problem.

Coping with Intractability

- Designing algorithms that run quickly on some instances, but take a prohibitive amount of time on others. For example, SAT solvers.

- Sacrifice the guarantee on an optimal solution and design approximation algorithms.

What’s Next?

- Are all computational problems solvable by computer?
  - NO!
    There are some that we can’t solve no matter how much time we give the computer, no matter how powerful the computer is.

Computability

- A problem is computable (i.e. decidable, solveable) if there is a mechanical procedure that
  1. Always terminates.
  2. Always gives the correct answer.

Program Termination

- Can we determine if a program will terminate given a valid input?
- Example:

```
def mystery1(x):
    while (x != 1):
        x = x - 2
```

  - Does this algorithm terminate when x = 15?
  - Does this algorithm terminate when x = 110?

Another Example

```
def mystery2(x):

    while (x != 1):

        if x % 2 == 0:

            x = x // 2

        else:

            x = (3 * x) + 1
```

If you test this program, it seems to terminate even though it sometimes reaches unpredictable values for x. In the absence of a proof of why it works this way, we cannot be sure whether there is any x for which it won't terminate.

- Does this algorithm terminate when x = 15?
- Does this algorithm terminate when x = 110?
- Does this algorithm terminate for any positive x?

Halting Problem

- Alan Turing proved that noncomputable functions exist by finding an noncomputable function, known as the Halting Problem.

- Halting Problem: Does a universal program *H* exist that can take **any** program *P* and **any** input *I* for program *P* and determine if *P* terminates/halts when run with input *I*?

Halting Problem Cast in Python

- **Input:** A string representing a Python program and an input to that program
- **Output:**
  - True, if evaluating the input program would ever finish
  - False, otherwise

Example

- Suppose we had a function **`halts`** that solves the Halting Problem
- Given the functions below

halts on
all inputs

```
def add(x, y):
    return x + y
```

```
def loop():
   while True:
       pass
```

loops
indefinitely

`halts(`add(10,15)’)`

returns True

`halts(`loop()’)`

returns False

Implement a Halt Checker? □

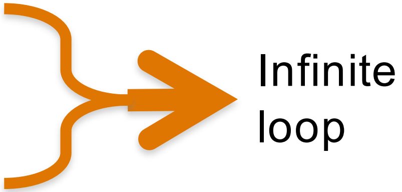
- How could we implement such a `halts` function? What is wrong with running the program given in the input string?

- We will show that `halts` is noncomputable -- `halts` function cannot exist.

Proving Uncomputability

- To prove the Python function `halts` does not exist, we will show that **if it exists it leads to a contradiction.**

```
def paradox():
    if halts(`paradox()’):
        while True:
            pass
```



Infinite loop

Proving Uncomputability

```
def paradox():
    if halts(`paradox()’):
        while True:
            pass
```

Contradiction!

If halts(`paradox()’) returns True,
paradox() never halts

If halts(`paradox()’) returns False,
paradox() halts.

Turing-Complete Languagues

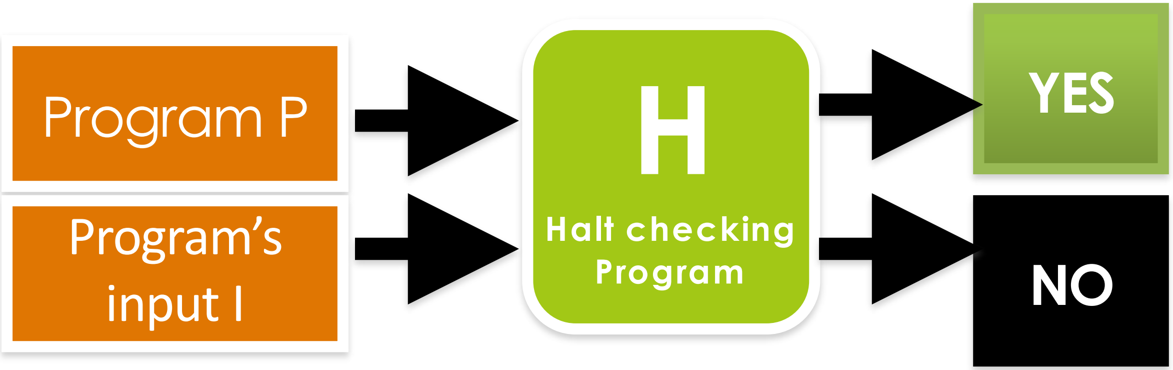
- We proved that a Python function `halts` cannot exist. How can we turn this into a general statement about *any* halts function?
  - We can use a Universal Turing Machine in reasoning about it rather than a Python interpreter.
- In fact, Python is a Turing-complete language: It can simulate a Universal Turing Machine. If `halts` cannot be computed by Python it cannot be computed by a Universal Turing Machine.

Telling the Story in a Python-independent Way

Proof by Contradiction (first step)

Assume a program *H* exists that requires a program *P* and an input *I*.

- *H* determines if program *P* will halt when *P* is executed using input *I*.



H outputs YES if *P* halts when run with input *I*

H outputs NO if P does not halt when run with input *I*

We will show that *H* cannot exist by showing that if it did exist we would get a logical contradiction.

Proof by contradiction (first step)

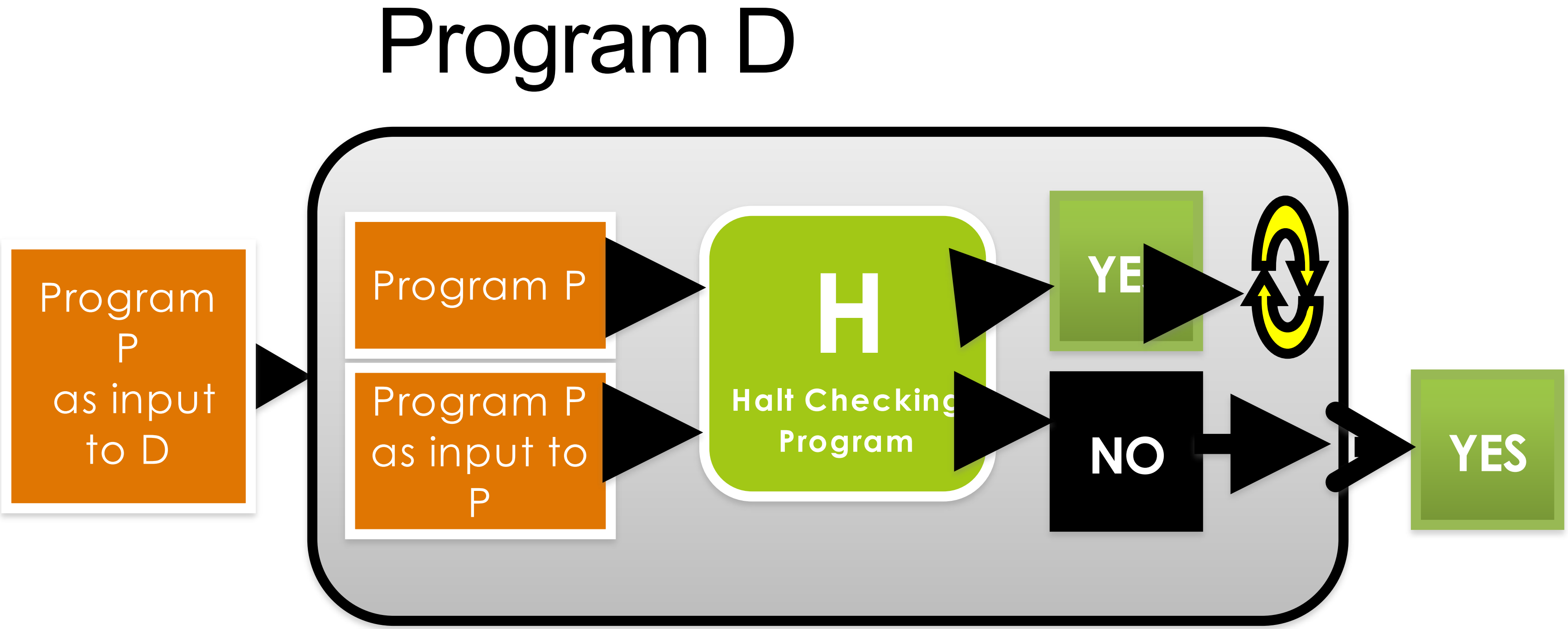
- Construct a new Program *D* that takes as input any program *P*

- *D* asks the halt checker *H* what happens if *P* runs with its own copy as input?

- If *H* answers that *P* will halt if it runs with itself as input, then *D* goes into an infinite loop (and does not halt).

- If *H* answers that *P* will not halt if it runs with itself as input, then *D* halts.

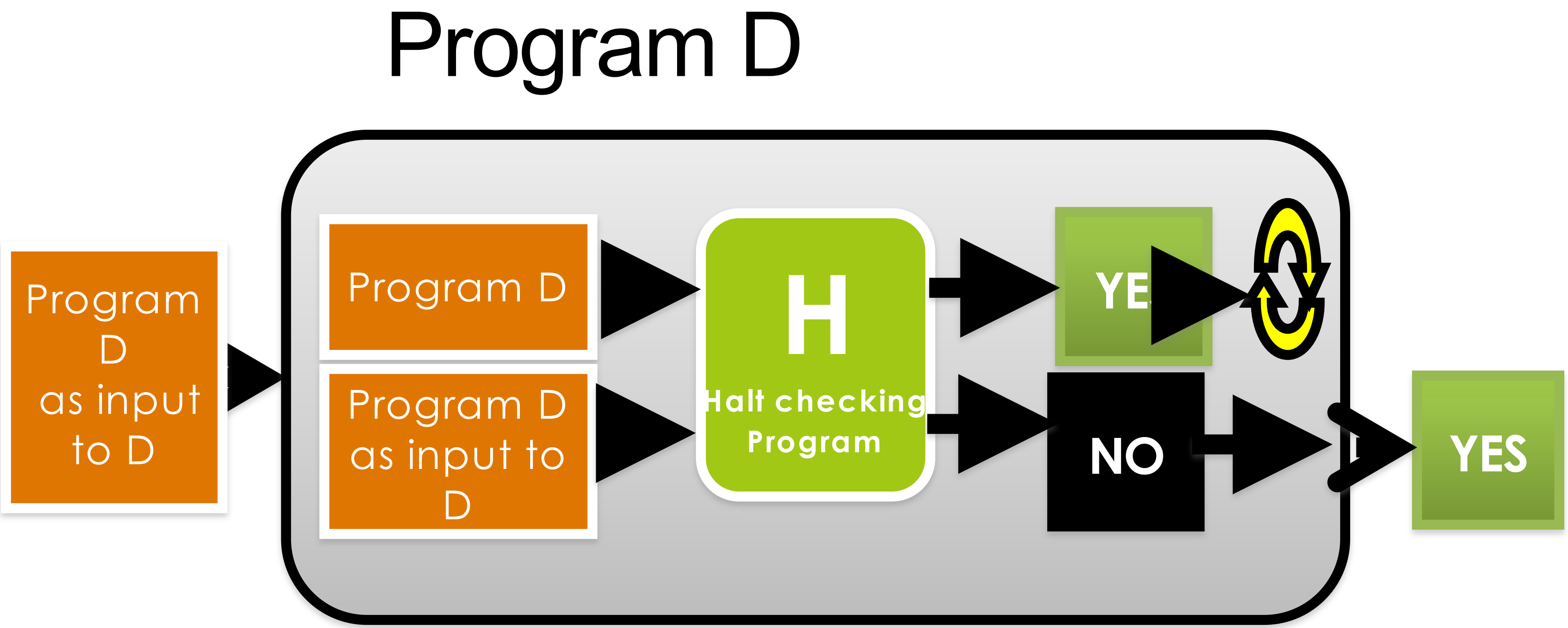
New Program *D*



D asks *H* what happens if we run program *P* on *P*.
Loops if it says YES.

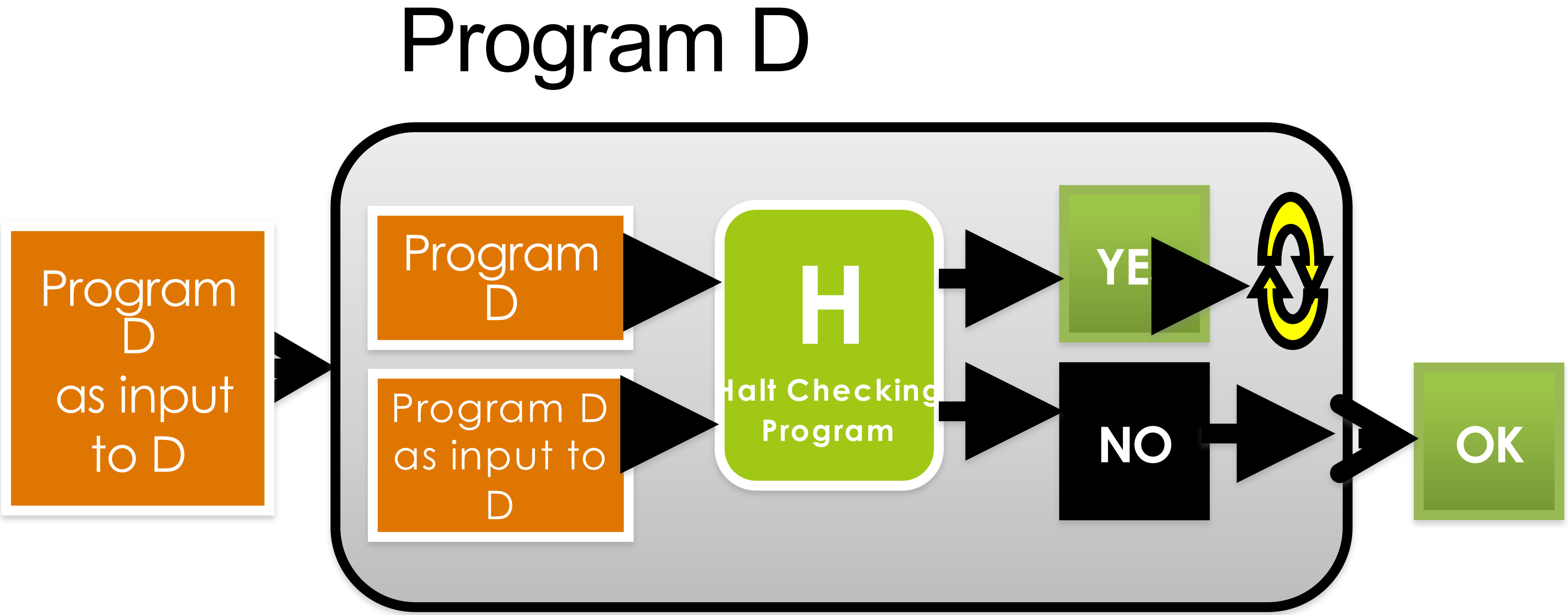
Stops and returns YES if it says no.

D testing itself



If *H* answers yes (*D* halts),
then *D* goes into an infinite loop and does not halt.

Proof by contradiction (last step)



What happens if D tests itself?

<span style="color:red">If *D* does not halt on *D*, then *D* halts on *D*.</span>

<span style="color:red">If *D* halts on *D*, then *D* does not halt on *D*.</span>

CONTRADICTION!

Contradiction

- No matter what *H* answers about *D*, *D* does the opposite, so *H* can never answer the halting problem for the specific program *D*.
  - Therefore, a universal halting checker H cannot exist.

- We can **never** write a computer program that determines if ANY program halts with ANY input.
  - It doesn't matter how powerful the computer is.
  - It doesn't matter how much time we devote to the computation.

Why Is Halting Problem Special?

- One of the first problems to be shown to be noncomputable. (i.e. undecidable, unsolveable)

- A problem can be shown to be noncomputable by reducing the halting problem into that problem.

- Examples of other nonsolveable problems: Software verification, Hilbert’s tenth problem, tiling problem

Virus Detection (due to Dave Evans)

If we could write a function that can always determine whether an expressions contains a virus that will infect other files, then we could solve the halting problem (which we know is impossible to solve)

```
def halts(p):

    return isVirus(p+`infectFiles()’)
```

Some sequence of steps that infects files

If isVirus existed it would return True when p halts and False otherwise (assuming `p` does not infect files.) But we know halts does not exist so isVirus cannot exist either.

Living with Noncomputable Functions

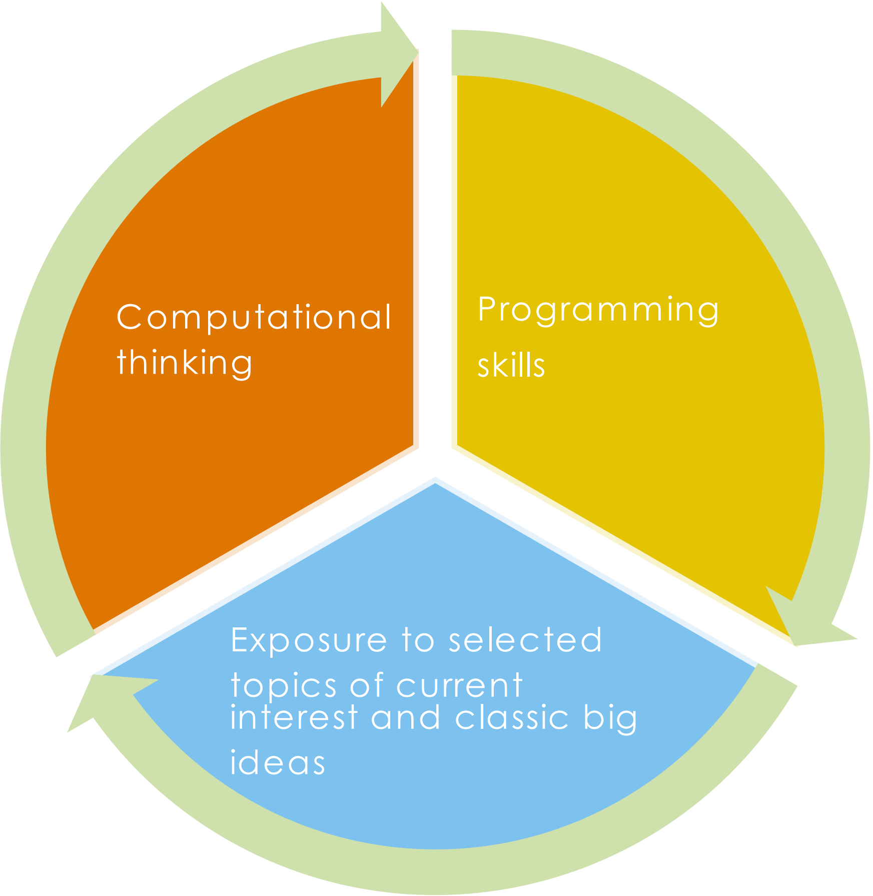
- Noncomputable (undecidable, unsolveable) means there is no procedure (algorithm) that
  1. Always terminates
  2. Always give the correct answer
- We should give up either one of these conditions
  - We usually prefer to give up 2 (correctness in all cases)
  - For example, a virus detection software cannot detect if a program is a virus for all possible programs. To be computable, they need to give up correctness for some cases.

What Should You Know?

- The fact that there are limits to what we can compute and what we can compute efficiently all using a mechanical procedure (algorithm) .
  - What do we mean when we call a problem tractable/intractable?
  - What do we mean when we call a problem solveable (i.e. computable, decidable) vs. unsolveable (noncomputable, undecidable)?
- What the question P vs. NP is about.
- Names of Some NP-complete problems and amount of work needed to solve them using brute-force algorithms.
- The fact that Halting Problem is unsolveable and that there are many others that are unsolveable.

CONCLUDING REMARKS

Course Objectives

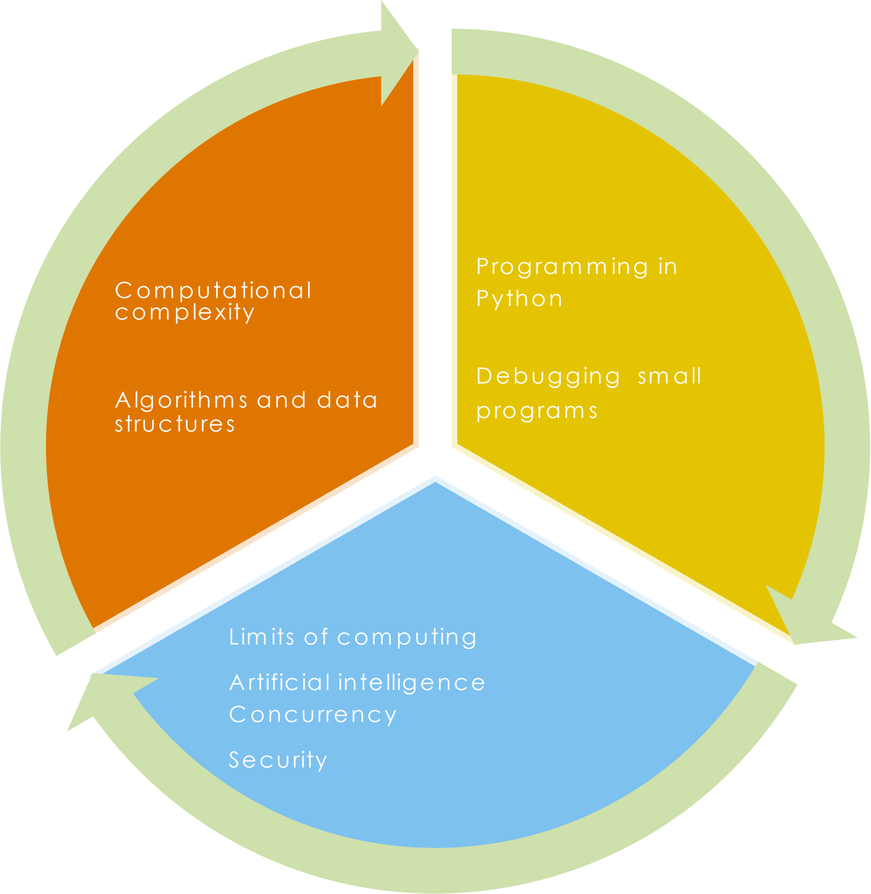


Computational thinking

Programming skills

Exposure to selected topics of current interest and classic big ideas

Course Coverage



Programming in Python

Debugging small programs

Limits of computing

Artificial intelligence

Concurrency

Security

Computational complexity

Algorithms and data structures

Where to Go From Here

- Done with computer science. You will be involved in computing only as needed in your own discipline?
  - We believe you are leaving this course with useful skills.
- Grew an interest in computing. You want to explore more?
  - 15-112 is taken by many who feel this way. It primarily focuses on software construction.
- Considering adding computer science as a minor or major?
  - Great! We are happy to have been instrumental in this decision.