The Limits of Computing

Intractability

Limits of Computing

Announcement

- Final Exam is on Friday
  - 9:00am – 10:20am Part 1
  - 4:30pm – 6:10pm Part 2
- If you did not fill in the course evaluations please do it today. 7 August is the last day
- Thanks

To wait or not to wait?

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?
  - Wouldn't it be good to know if it will end or not

More information would help

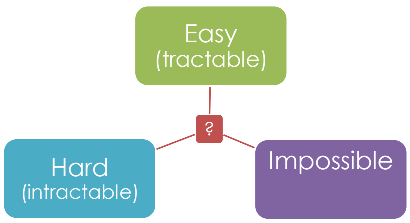
- Can you say if your program will return a result or not?
- Can you say anything about the hardness of the problem that you are trying to solve?

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 and cannot be solved** by mechanical computation?

Principles of Computing, Carnegie Mellon University

Can we **categorize** problems?



Principles of Computing, Carnegie Mellon University

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?

Principles of Computing, Carnegie Mellon University

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?

Principles of Computing, Carnegie Mellon University

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?

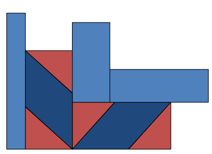
Principles of Computing, Carnegie Mellon University

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**.

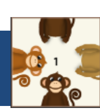
Example:

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



Principles of Computing, Carnegie Mellon University

The Monkey Puzzle



Given:

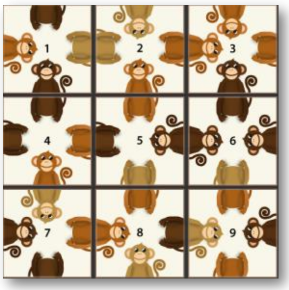
- A set of **n** square cards whose sides are imprinted with the upper and lower halves of colored monkeys.
- **n** is a square number, such that **n = m²**.
- Cards cannot be rotated.

decision problem

Problem:

- **Determine if an arrangement of the n cards in an m x m grid exists** such that each adjacent pair of cards display the upper and lower half of a monkey of the same color.

Example



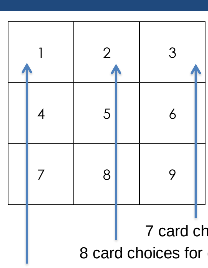
- Can we always compute a YES/NO answer to the problem?
- If we can, **is the problem tractable (easy to solve)** in general?

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

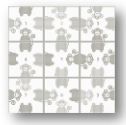
9 card choices for cell 1

Analysis (cont' d)

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

Assume that
we can analyze one arrangement in one microsecond (μs),
that is, we can 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

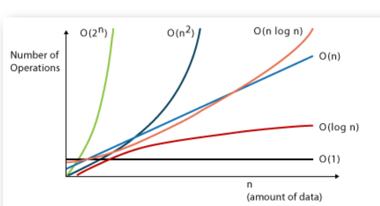
Principles of Computing, Carnegie Mellon University

Reviewing the Big O Notation (1)

- We use 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**.

Principles of Computing, Carnegie Mellon University

Growth of Some Functions



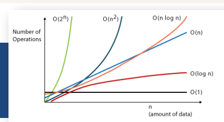
Big O notation:
gives an asymptotic upper bound
ignores constants

Any function $f(n)$ such that $f(n) \le c\, n^2$ for large n has $O(n^2)$ complexity

Quiz on Big O

- What is the order of complexity in big O
  - 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, 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**

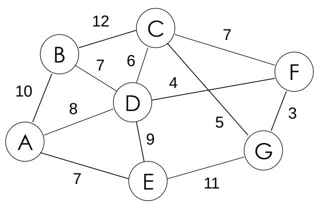
** A **polynomial** is an expression consisting of variables and coefficients that involves only the operations of addition, subtraction, multiplication, and non-negative integer exponents.*

A Famous (Hard) Problem

Traveling Salesperson

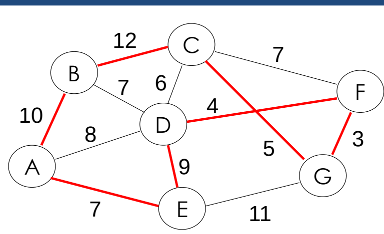
- **Given:** a weighted graph of nodes representing cities and edges representing flight paths (weights represent cost)
- Is there a route that takes the salesperson through every city and back to the starting city with cost no more than *k*?
  - The salesperson can visit a city only once (except for 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 candidate 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.

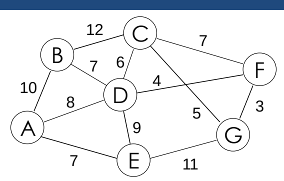
Principles of Computing, Carnegie Mellon University

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: __________

Principles of Computing, Carnegie Mellon University

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!)

Principles of Computing, Carnegie Mellon University

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)

Principles of Computing, Carnegie Mellon University

Map Coloring

- Given a map of N territories, can the map be colored using **3** colors such that no two adjacent territories are colored with the same color?

Principles of Computing, Carnegie Mellon University

Analysis

- Given a map of N territories, can the map be colored using **3** colors such that no two adjacent territories are colored with the same color?
  - Pick a color for territory 1 (3 choices)
  - Pick a color for territory 2 (3 choices)
  - ...
- There are 3^N possible colorings.

Principles of Computing, Carnegie Mellon University

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

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
n^2	0.00000010	0.00000040	0.00000090	0.00000160
n^3	0.00000100	0.00000800	0.00002700	0.00006400
n^5	0.00010000	0.00320000	0.02430000	0.10240000
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**.

Polynomial time *e.g. $3n^4 + n - 5$*

- **If *n* is not small**, then the amount of computation grows exponentially. Computers can solve these **intractable problems**, but it will take far too long for the result to be generated.

super-polynomial time *e.g.* ***4^n***

We would be long dead before the result is computed.

P vs NP

The Limits of Computing

Decision Problems

- We have seen different decision problems with simple ***brute-force algorithms*** that are **intractable**.
  - The Monkey Puzzle O(n!)
  - Traveling Salesperson O(n!)
  - Map Coloring $O(3^N)$
  - Satisfiability $O(2^n)$

Can avoid brute-force in many problems?

Special cases of an intractable problem may be tractable

- **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.

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

2-satisfiability: determining whether a conjunction of disjunctions (AND of ORs), where each disjunction (OR) has 2 arguments that may either be variables or the negations of variables, **is satisfiable**.

Are These Problems Tractable?

- For any one of these problems, is there a single tractable (polynomial) algorithm to solve any instance of the problem?
  - Haven't been found so far.
  - Haven't been proved that they do not exist
- Possible reasons are that these problems
  - have undiscovered polynomial-time solutions.
  - are intrinsically difficult – we can't hope to find polynomial solutions.

Are These Problems Tractable?

Important discovery:

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

Verifiability

- No known tractable algorithm to decide, however it is easy to verify a candidate (i.e. proposed) solution.

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 candidate solutions can be verified quickly

Principles of Computing, Carnegie Mellon University

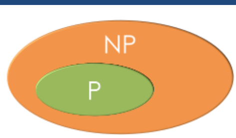
Example

	Verifiable in Polynomial Time **NP**	Solvable in Polynomial Time **P**
Finding the maximum value in an array	YES	YES
Satisfiability problem	YES	?
Map coloring problem	YES	?

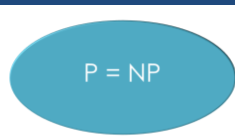
- If a problem is in **P**, it must also be in **NP**.
- If a problem is in **NP**, is it also in **P**?

Principles of Computing, Carnegie Mellon University

Two Possibilities



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

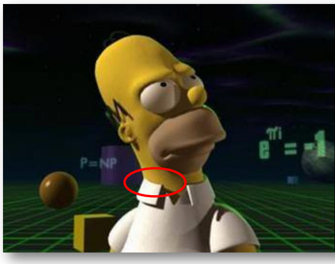


If P = NP, then all polynomially verifiable problems can be solved in polynomial time.

*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.

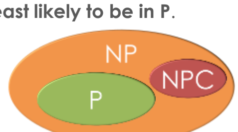
NP-Complete Problems

- 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.

NP-Complete

- The class **NP-Complete** consists of all those problems in NP that are **least likely to be in P**.
  - Monkey puzzle,
  - Traveling salesperson,
  - Mapo coloring, and
  - Satisfiability

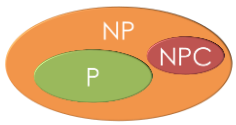


- Each of these problems are called **NP-Complete**
- Informally, NP-complete problems are the hardest problems in NP.

Principles of Computing, Carnegie Mellon University

NP-Complete

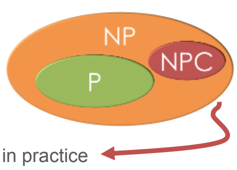
- Every problem in NP-Complete can be transformed to another problem in NP-Complete.
- If there were some way to solve one of these problems in polynomial time, we should be able to solve all of these problems in polynomial time.



Principles of Computing, Carnegie Mellon University

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.

Principles of Computing, Carnegie Mellon University

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/

Noncomputable Functions

The Limits of Computing

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
- **Output:**
  - True, if evaluating the input program would ever finish
  - False, otherwise

A Halt Checker

- 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
```

halts('add(10,15)') returns True

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

loops indefinitely

halts('loop()') returns False

Proving Uncomputability

- To prove the Python function `halts` does not exist, we will show that **if it exists it leads to a contradiction.** (such as "*This sentence is false*")

```
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.

Principles of Computing, Carnegie Mellon University

Turning into a general statement

- We proved that a Python function **`halts`** cannot exist.
- How can we turn this into a general statement about *any* halts function?

Principles of Computing, Carnegie Mellon University

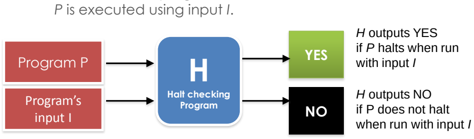
Telling the Story in a Python-independent Way

Principles of Computing, Carnegie Mellon University

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*.



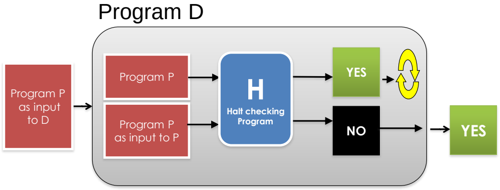
We will show that *H* cannot exist by showing that if it did exist we would get a logical contradiction.

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*

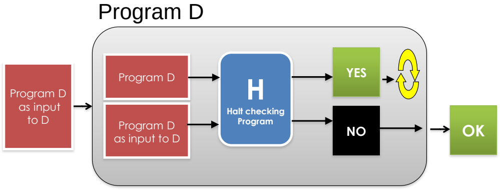
Program D



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

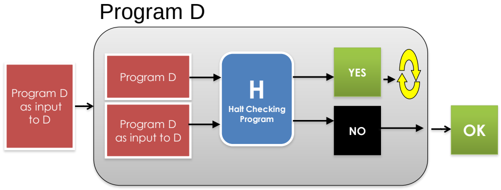
Program D



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

Principles of Computing, Carnegie Mellon University

Proof by contradiction (last step)



What happens if D tests itself?

If *D* does not halt on *D*, then *D* halts on *D*.
If *D* halts on *D*, then *D* does not halt on *D*.

CONTRADICTION!

Principles of Computing, Carnegie Mellon University

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.

Principles of Computing, Carnegie Mellon University

Why Is Halting Problem Special?

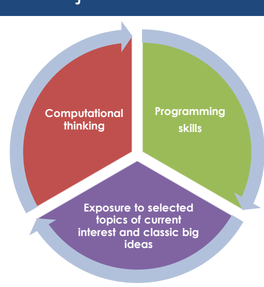
- One of the first problems to be shown to be noncomputable (i.e. undecidable, unsolveable)
- A problem can be shown to be noncomputable by reducing the halting problem into that problem.
- Examples of other nonsolveable problems: Software verification, Hilbert's tenth problem, tiling problem

What Should You Know?

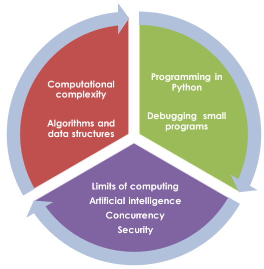
- The fact that there are limits to what we can compute and what we can compute efficiently all using a mechanical procedure (algorithm) .
  - What do we mean when we call a problem tractable/intractable?
  - What do we mean when we call a problem solveable (i.e. computable, decidable) vs. unsolveable (noncomputable, undecidable)?
- What the question P vs. NP is about.
- Name some NP-complete problems and reason about the 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



Course Coverage



Principles of Computing, Carnegie Mellon University

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.