UNIT 14C

The Limits of Computing: Uncomputable Functions

Problem Classifications

- Tractable Problems
  - Problems that have reasonable, polynomial-time solutions
- Intractable Problems
  - Problems that may have no reasonable, polynomial-time solutions
- Uncomputable Problems
  - Problems that have no algorithms at all to solve them

RECALL FROM LAST LECTURE

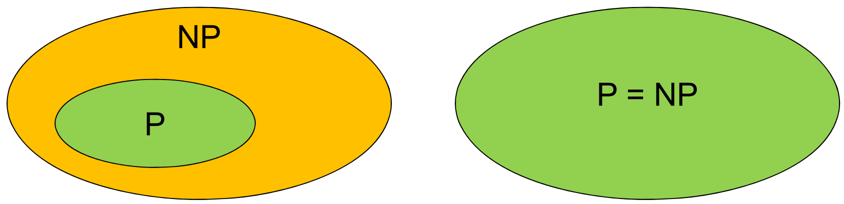
Decidability vs. Verifiability

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

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

Two Possibilities

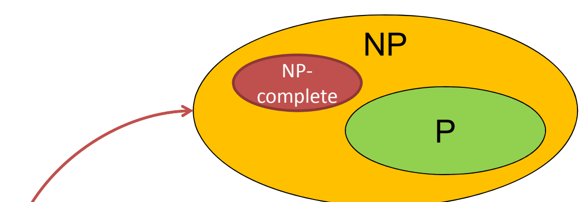
We do not know which of these possibilities is true.



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

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

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.

Today's Lecture

- We will look the Halting Problem that is a canonical problem in the study of limits of computing.
- We will show using proof by contradiction that it cannot be solved.
- Along the way, we will think about termination and programs that have some form of self-reference.

The Barber Paradox

- Suppose there is a town with just one barber, who is male. In this town, every man keeps himself clean-shaven, and he does so by doing exactly one of two things:
  1. Shaving himself, or
  2. Going to the barber.
- Another way to state this is: The barber is a man in town who shaves those and only those men in town who do not shave themselves.
- Who shaves the barber?

Program Termination

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

```
def mystery1(x)
  while (x != 1) do
    x = x – 2
  end
end
```

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

Another Example

```
def mystery2(x)
  while (x != 1) do
    if x % 2 == 0 then
      x = x / 2
    else
      x = 3 * x + 1
  end
end
```

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

The 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*?
- Alan Turing showed that such a universal program *H* cannot exist.
  - This is known as the Halting Problem.

Proof by Contradiction (example)

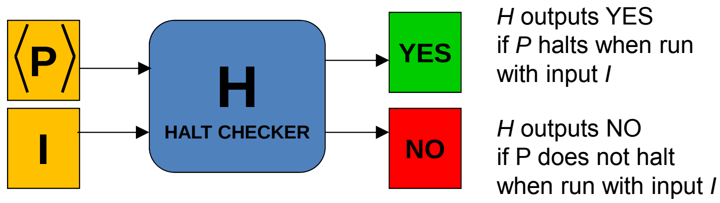
Suppose you want to prove the proposition "One cannot get an A in this course without doing the homeworks".

1. You first assume the opposite: "One can get an A in this course without doing the homeworks".
2. From that assumption and using what you know about the course you arrive at a conclusion, which is not true (e.g. Homeworks are worth less than 10%).
3. Since you know that this conclusion is false (contradicts with what is known), the initial assumption must be wrong.

   "One can get an A in this course without doing the homeworks". ← Must be false

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.

13

Programs Computing with Their Own Representation

- A compiler is a program that takes as its input a program that needs to be translated from a high-level language (e.g. Ruby) to a low-level language (e.g. machine language).
  - In general, a program can process any data, so it can have a program as its input to process.
- Can a compiler compile itself? YES!

15110 Principles of Computing, Carnegie Mellon University

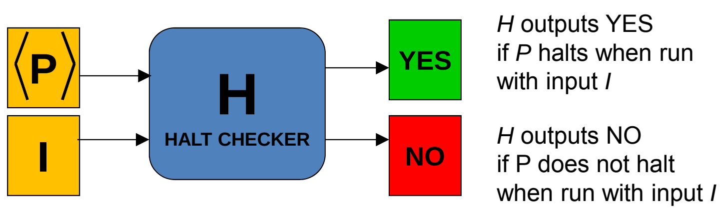
14

Proof (cont'd)

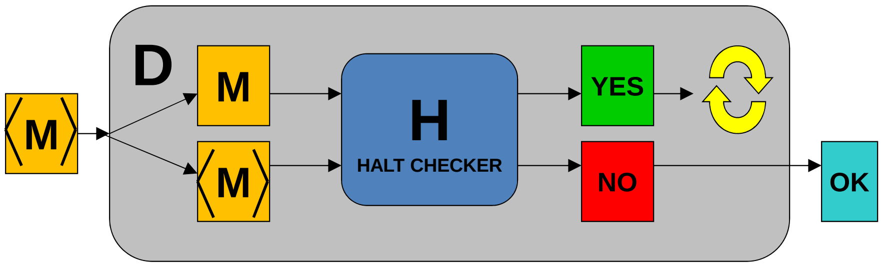
- Let *D* be a program that takes input <*M*> where <*M*> is a program description.
- *D* asks the halt checker *H* what happens if *M* runs with itself <*M*> as input?
- If *H* answers that *M* will halt if it runs with itself as input, then *D* goes into an infinite loop (and does not halt).
- If *H* answers that *M* will not halt if it runs with itself as input, then *D* halts.

Recall the Halt Checker

- 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*.



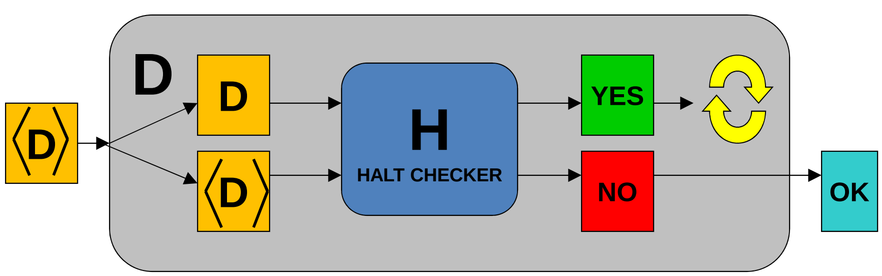
How to Construct D



D asks *H* what happens if we run program *M* on with input <*M*> .
Loops if it says yes.
Stops and returns OK if it says no.

D gets evil

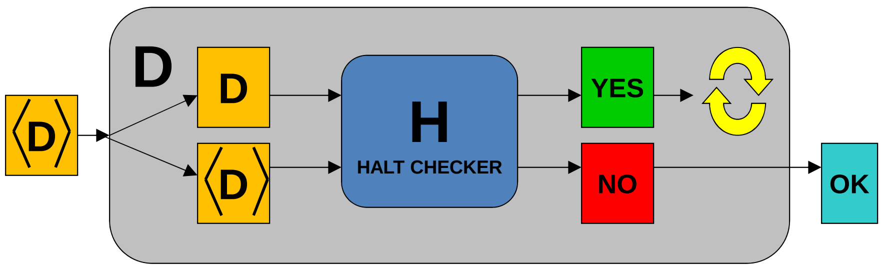
- What happens if *D* tests itself?
  - If *H* answers yes (*D* halts), then *D* goes into an infinite loop and does not halt.



Proof By Contradiction (last step)

contradiction

- 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

- 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 <u>never</u> 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 uncomputable by reducing the halting problem into that problem

Do We Give up on Uncomputable Problems

- Uncomputable (undecidable, unsolveable) means there is no procedure (algorithm) that
  1. Always terminates
  2. Always give the correct answer

Living with Uncomputable Functions

- 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 at all and what we can compute efficiently.
  - 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.