CS 70 Discrete Mathematics and Probability Theory
Summer 2025 Tate

DIS 3D

1 Computability Intro

Note 12

Computability: The main focus is on the Halting problem, and programs that provably cannot exist.

The *Halting problem* is the problem of determining whether a program P run on input x ever halts, or whether it loops forever. It turns out that there does not exist any program that solves this problem.

Using this information, we can prove that other problems also cannot be solved by a computer program, through the use of *reductions*. The main idea is to show that if a given problem can be solved by a computer program `TestX`, then the Halting problem can also be solved by a computer program `TestHalt` that uses `TestX` as a subroutine.

The primary template we'll use for this course is as follows. Suppose we want to show that a program `TestX` does not exist, where `TestX(Q, y)` tries to determine whether a program Q on input y does some task $\mathscr{X}$ (i.e. it outputs "True" if $Q(y)$ does the task $\mathscr{X}$, and it outputs "False" if $Q(y)$ does not do the task $\mathscr{X}$). We can define `TestHalt` as follows (in pseudocode):

```
def TestHalt(P, x):
    def Q(y):
        run P(x)
        do 𝒳
    return TestX(Q, y)  # for some given y
```

Note that this template will be sufficient for our purposes in CS70, but more complex reductions will require more sophisticated programs—you'll learn more about this in classes like CS170 and CS172.

(a) Consider the reduction template given above. Let's break down what it's doing.

We follow an argument by contradiction—we assume that there is a program `TestX(Q, y)` that is able to determine whether another program Q on input y does some task $\mathscr{X}$.

There are two cases: either $P(x)$ halts, or it loops forever. We'd like to show that `TestHalt` as defined above returns the correct answer in both of these cases.

(i) Suppose $P(x)$ halts. What does `TestHalt` return, and why?

(ii) Suppose $P(x)$ loops forever. What does `TestHalt` return, and why?

(iii) What does this tell us about the existence of `TestX`? Briefly justify your answer.

Solution:

(a) (i) If $P(x)$ halts, then $Q(y)$ will finish executing $P(x)$, and eventually do the task $\mathscr{X}$. This means that `TestX` would return "True", since $Q(y)$ does eventually do $\mathscr{X}$.

(ii) If $P(x)$ loops forever, then $Q(y)$ will get stuck while executing $P(x)$, and will never get to doing the task $\mathscr{X}$. This means that `TestX` would return "False", since $Q(y)$ never does $\mathscr{X}$.

(iii) These answers returned by `TestHalt` exactly solve the Halting problem! However, we've already shown that the Halting problem cannot be solved by a computer program—this is a contradiction. As such, `TestX` cannot exist.

2 Hello World!

Note 12

Determine the computability of the following tasks. If it's not computable, write a reduction or self-reference proof. If it is, write the program. Throughout this problem, you are allowed to execute programs while surpressing their print statements.

(a) You want to determine whether a program P on input x prints "Hello World!". Is there a computer program that can perform this task? Justify your answer.

(b) You want to determine whether a program P prints "Hello World!" while or before running the kth line in the program. Is there a computer program that can perform this task? Justify your answer.

(c) You want to determine whether a program P prints "Hello World!" in the first k steps of its execution. Is there a computer program that can perform this task? Justify your answer.

Solution:

(a) Uncomputable. We will reduce `TestHalt` to `PrintsHW`(P,x).

```
TestHalt(P, x):
  P'(x):
    run P(x) while suppressing print statements
    print("Hello World!")

  return PrintsHW(P', x)
```

If `PrintsHW` exists, `TestHalt` must also exist by this reduction. Since `TestHalt` cannot exist, `PrintsHW` cannot exist.

(b) Uncomputable. We will reduce `TestHalt` to `PrintsHWByK`(P,x,k).

```
TestHalt(P, x):
    P'(x):
        run P(x) while suppressing print statements
        print("Hello World!")
```

```
    return PrintsHWByK(P', x, 2)
```

Here, we notice that `P'` has only two lines (or at most `len(P) + 1` lines, depending on how this is implemented), and we print "Hello World!" by the last line of `P'` if and only if `P(x)` halts.

Alternatively, we can reduce `PrintsHW`(P,x) from part (a) to this program `PrintsHWByK`(P,x,k):

```
PrintsHW(P, x):
  for i in range(len(P)):
    if PrintsHWByK(P, x, i):
      return true
  return false
```

Note that we technically need to iterate through all the lines here, since there may be large jumps within the code of `P`; this means that we may for example jump from line 1 to line 100 and back to line 2 to print "Hello World!", but `PrintsHWByK`$(P,x,100)$ will return false, since we first reach line 100 without printing "Hello World!".

(c) Computable. You can simply run the program until k steps are executed. If P has printed "Hello World!" by then, return true. Else, return false.

The reason that part (b) is uncomputable while part (c) is computable is that it's not possible to determine if we ever execute a specific line because this depends on the logic of the program, but the number of computer instructions can be counted.

3 Countability and the Halting Problem

Note 11
Note 12

Using methods from countability, we will prove the Halting Problem is undecidable.

(a) What is a reasonable representation for a computer program? Using this definition, show that the set of all programs are countable. *(Hint: Machine Code)*

(b) The Halting Problem only considers programs which take a finite length input. Show that the set of all finite-length inputs is countable.

(c) Assume that you have a program that tells you whether or not a given program halts on a specific input. Since the set of all programs and the set of all inputs are countable, we can enumerate them and construct the following table.

	x_1	x_2	x_3	x_4	$\cdots$
p_1	H	L	H	L	$\cdots$
p_2	L	L	L	H	$\cdots$
p_3	H	L	H	L	$\cdots$
p_4	L	H	L	L	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

An H (resp. L) in the ith row and jth column means that program p_i halts (resp. loops) on input x_j. Now write a program that is not within the set of programs in the table above.

(d) Find a contradiction in part a and part c to show that the halting problem can't be solved.

Solution:

(a) As in discussion and lecture, we represent a computer programs with a set of finite-length strings (which, in turn, can be represented by a set of finite length binary strings). The set of finite length binary strings are countably infinite. Therefore the set of all programs is countable.

(b) Notice that all inputs can also be represented by a set of finite length binary strings. The set of finite length binary strings are countably infinite, as proved in Note 11. Therefore the set of all inputs is countable.

(c) For the sake of deriving a contradiction in part (d), we will use the following program:

procedure P'(x_j)
  if $P_j(x_j)$ halts **then**
    loop
  else
    halt
  end if
end procedure

(d) If the program you wrote in part c) exists, it must occur somewhere in our complete list of programs, P_n. This cannot be. Say that P_n has source code x_j (i.e. its source code corresponds to column j). What is the (i, j)th entry of the table? If it's H, then $P_n(x_j)$ should loop forever, by construction; if it's L, then $P_n(x_j)$ should halt. In either case, we have a contradiction.

4 N-Bit

Note 12

Let N-Bit be a program that takes in a program P and an input x and outputs "true" if the program ever directly assigns a variable to an n-bit number and "false" otherwise. Can N-Bit exist? If so, describe the procedure, and if not, prove that it cannot exist. (Hint: model off of the proof that TestHalt cannot exist)

Solution: No. Assume that NBit exists. Now let's define Foil in this way:

```
Foil(P):
        if Nbit(P, P) == true:
                return
        else:
                x = 2^n
                return
```

Then, in the same way as Turing and Halt, when we run Foil(Foil) we get a contradiction in the program and therefore NBit cannot exist.