NUS National University of Singapore

School of Computing

Computability and Efficiency

(The Halting Problem)

Video 6.5b

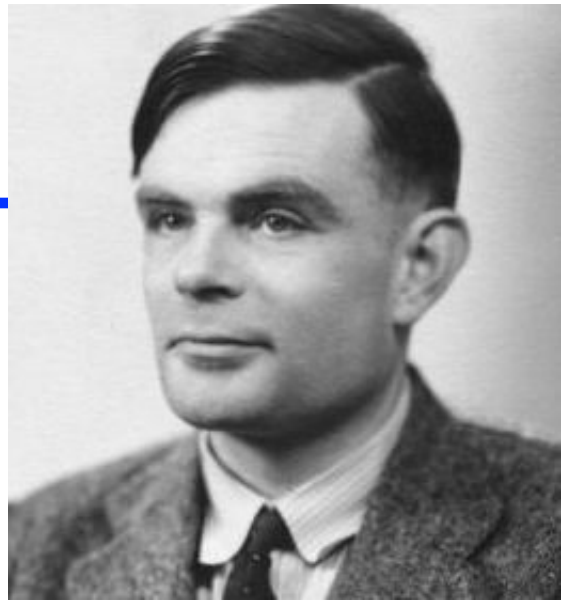
Hon Wai Leong

Department of Computer Science
National University of Singapore

Email, FB: leonghw@comp.nus.edu.sg

Halting Problem is not computable
(…and it's cool to know why too)

Computability



NOT Everything is Computable!

Computability

Theorem: (Turing, 1936)

The halting problem is not computable.

Here, we show this result using the *same problem* that Turing used back in 1936.

But the proof will be just a bit different.

The Halting Problem (definition)

Halting Problem:

Input: *Any program* P and *any data* D,

Todo: Does program P running on data D halt or not?

$$\text{Halt-Checker}(P,D) = \begin{cases} YES & \text{if } P(D) \text{ halts} \\ NO & \text{if } P(D) \text{ does not halt} \end{cases}$$

Two quick reminders…

Halt-Checker is an algorithm

So, it must halt after a finite number of steps!

Can Halt-Checker just "run P on D" & wait?

Case 1: "P running on D" halts. Answer YES ☺

Case 2: "P running on D" does not halt?

Then Halt-Checker will itself be stuck. ☹

No-brainer solution does NOT work!

In general, how to know if program *P running on data D* will halt or not?

(Not easy – needs insights on the algorithm)

This program (algorithm) halts (1)

Problem-1: Algorithm to Compute the sum of (1 + 2 + 3 + … + 99 + 100)

ALGORITHM BAD-Sum-to-Hundred;

1. Let Sum ← 0 ;
2. Sum ← Sum + 1
3. Sum ← Sum + 2
4. Sum ← Sum + 3

   … … …

100. Sum ← Sum + 99
101. Sum ← Sum + 100
102. Return value of Sum as result
103. End

This algorithm (program) BAD-Sum-to-Hundred clearly *halts*.
(after 103 steps)

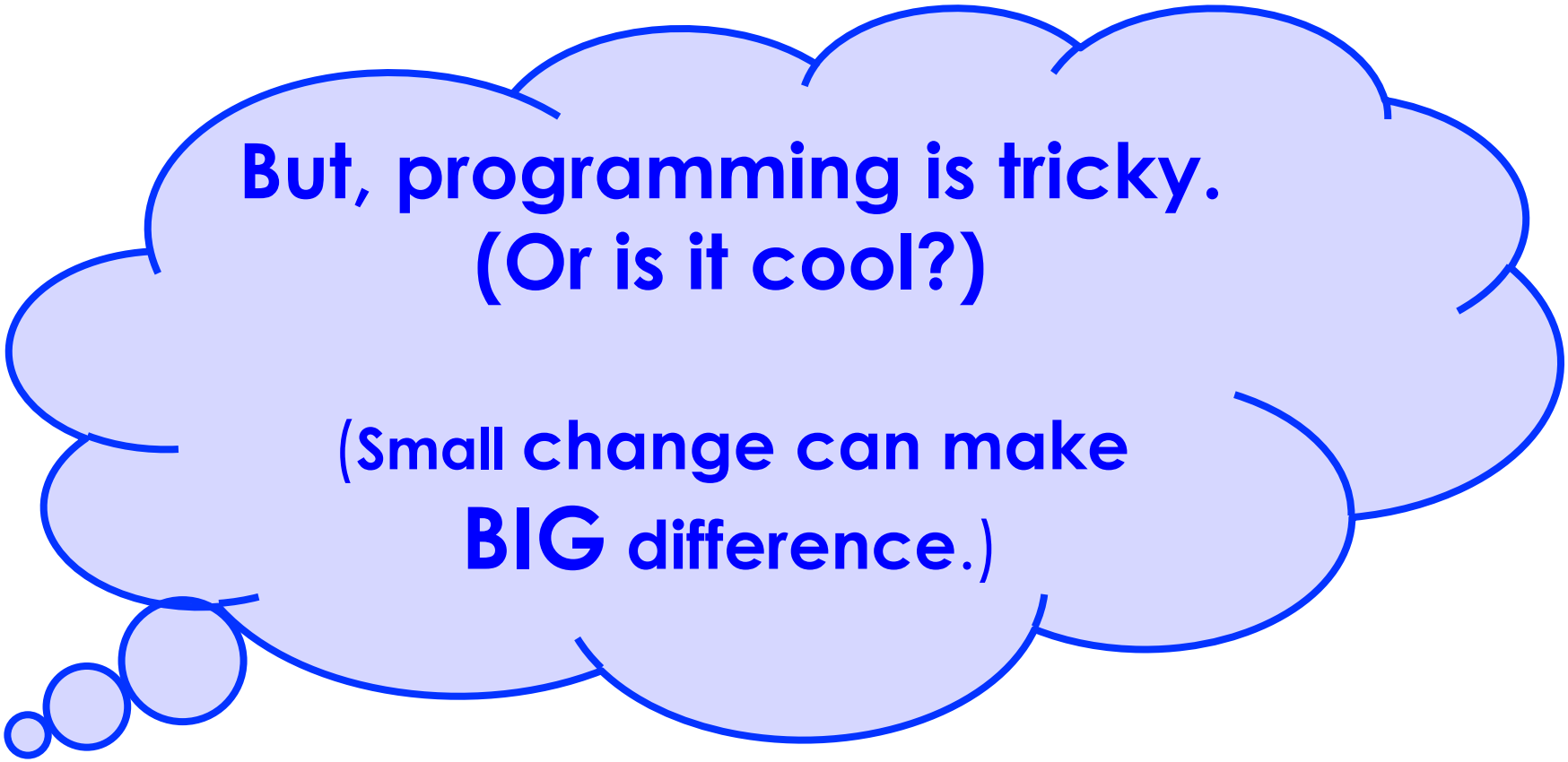
What about Sum-v2(n)?

Algorithm Sum-v2(n)

ALGORITHM Sum-v2(n)

1. Let Sum ← 0 ;
2. Let k ← 1 ;
3. While (k ≤ n) repeat Steps 4-6
4. Sum ← Sum + k
5. k ← k + 1
6. end-of-while-block;
7. Print out the value of Sum
8. End

This algorithm Sum-v2(n) also *halts*.
(after ($4n$+5) steps)


But, programming is tricky.
(Or is it cool?)
(Small change can make
BIG difference.)

Small change to Sum-v2(n)

Algorithm BAD-Sum-v2(n)

1. Let Sum ← 0 ;
2. Let k ← 1 ;
3. While (k ≤ n) repeat Steps 4-6
4. Sum ← Sum + k
5. [redacted]
6. end-of-while-block;
7. Print out the value of Sum
8. End

Suppose Step 5 got deleted, by accident

What happens to the algorithm?

Does BAD-Sum-v2(n) halt?

Algorithm BAD-Sum-v2(n)

1. Let Sum ← 0 ;
2. Let k ← 1 ;
3. While (k ≤ n) repeat Steps 4-6
4. Sum ← Sum + k
5. end-of-while-block;
6. Print out the value of Sum
7. End

k is not changed.

(k ≤ n) is always TRUE (when n ≥ 1)

This algorithm **does NOT halts.**

Your computer will just freezes …

TOI working hard, app freezes

What about this program? (1)

ALGORITHM Odd-Primes-Theorem;

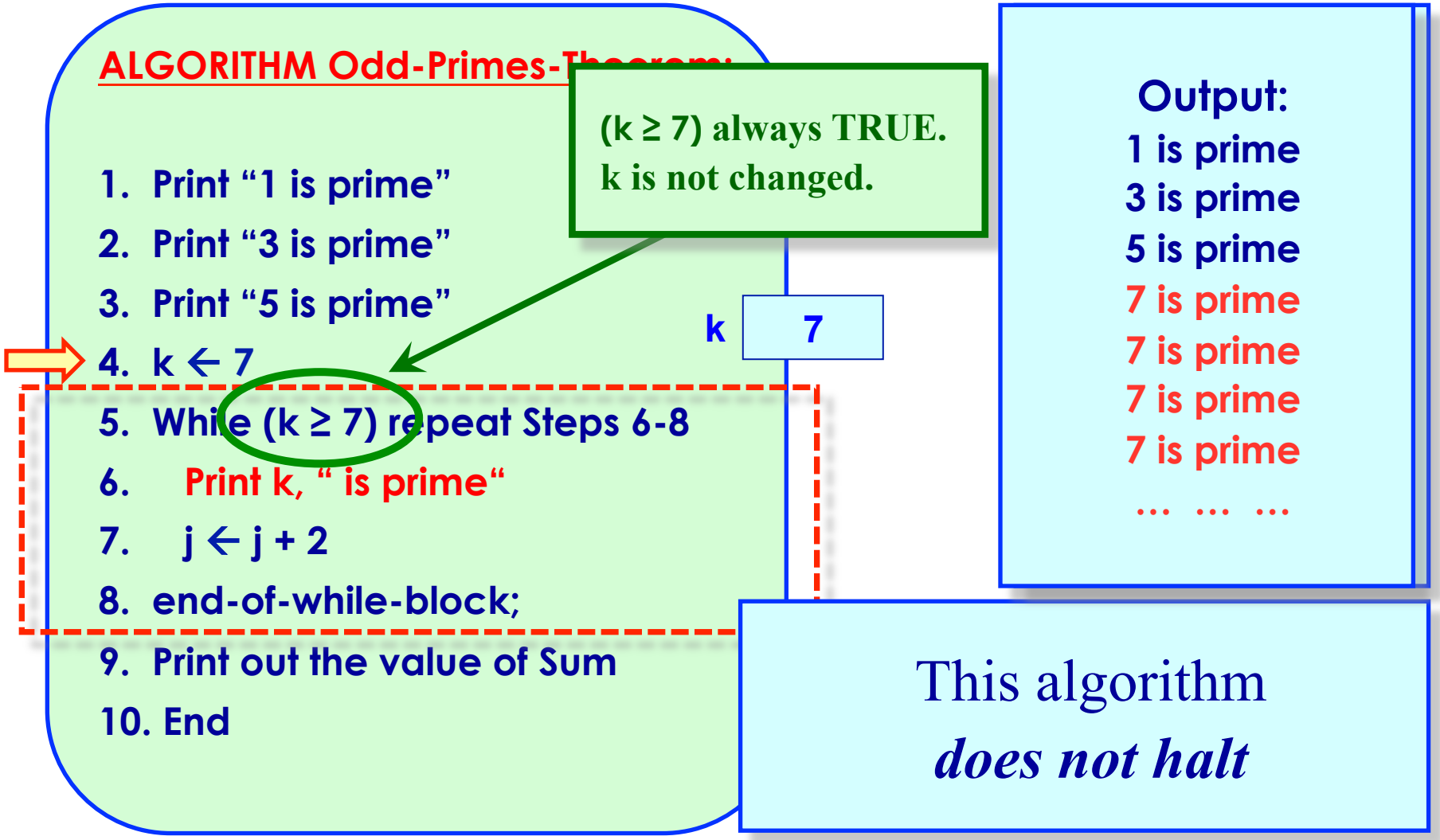
1. Print "1 is prime"
2. Print "3 is prime"
3. Print "5 is prime"
4. k ← 7
5. While (k ≥ 7) repeat Steps 6-8
6. Print k, " is prime"
7. j ← j + 2
8. end-of-while-block;
9. Print out the value of Sum
10. End

k | 7

Output (Step 1-3):

1 is prime
3 is prime
5 is prime

What about this program? (2)



The CS prof's dilemma

CS prof's Dilemma:

Given a complex program (algorithm) *P*,
it is very difficult to tell if it will halt
when it is run on data *D*.

But CS profs have to grade many
students' programs.

Let's see…

CS-Prof grading HW-programs (1)

1. CS prof assigns programming-HW.

2. Students work & submit programs.

P1, P2, P3,…

has secret test-data X, Y

3. CS prof needs to grade all the student programs (P1, P2, P3,…) on test-data X, Y.

CS-Prof grading HW-programs (2)

Programs
P1, P2, P3, …

test-data
X, Y

Grading Process:
* Takes student program P
* Runs P on X (call it P(X))
* Runs P on Y
* Assigns Marks for P.

Starts Grading….

* P1(X), P1(Y) give correct answers. Finishes in < 1sec

* P2(X) give correct answers. P2(Y) give wrong answers.

CS-Prof grading HW-programs (2)

More Grading….

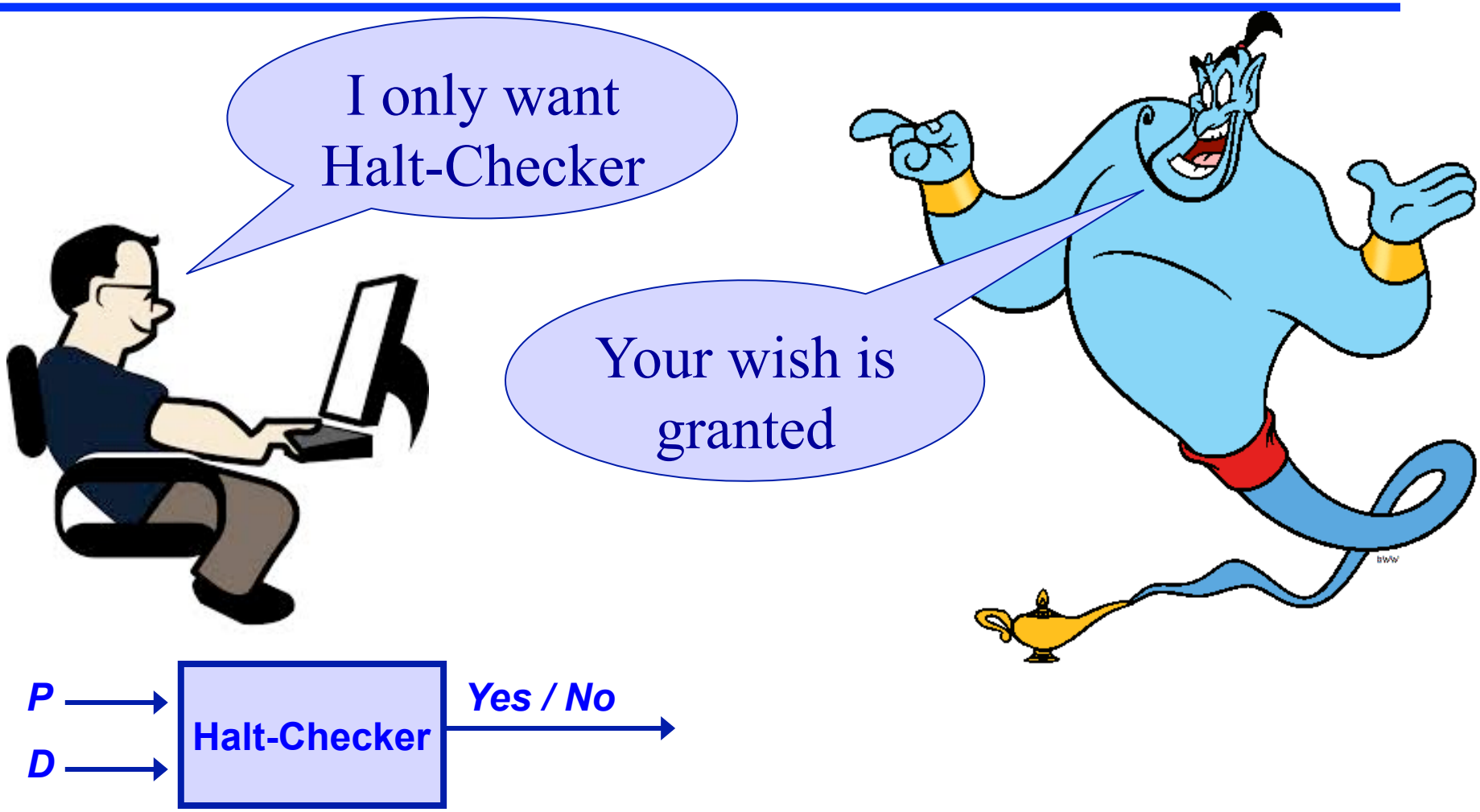
* P3(X) takes a long time, 2.5hrs.
  Gave wrong answer. ☹

* P3(Y) taking much longer…
  prof waited 48 hours! Still nothing…☹

* In the meantime, prof finishes grading
  all the rest of the other students' program. ☺

CS-prof dilemma:
Should prof "kill" the program P3(Y)?) ☹
Is that fair to student?
- what if 11 days needed by P3(Y)?

CS-prof seeks *ancient* helper…



What if CS-Prof has Halt-Checker?

Grading Process (with Halt-Checker):

* Takes student program P
* **If** (**Halt-Checker(P, X) = YES**)
  then run P(X)
* Repeat for data Y.
* Assigns Marks for P.

P(X) will halt

Halt-Checker will solve CS-prof dilemma!

prof skips all those cases that will NOT halt;

back to The Halting Problem

Halting Problem:

Input: *Any program* P and *any data* D,

Todo: Does program P halt when run on data D

$$\text{Solve-Halt}(P,D) = \begin{cases} Yes & \text{if } P(D) \text{ halts} \\ No & \text{if } P(D) \text{ does not halt} \end{cases}$$

Result (Alan Turing, 1936)

Halting Problem:

Input: *Any program P* and *any data D*,

Todo: Does program *P* running on data *D* halt?

Theorem: (Turing, 1936)
Halting problem is not computable.

Algorithm Halt-Checker(*P*, *D*) does not exist.

There is no algorithm Halt-Checker(*P*, *D*) that will take as input any program *P* and any data *D*, and in a finite number of steps, will answer if "program *P* running on data *D*" will halt or not halt.

We ARE going to prove this result!

(If you understand it ALL, go amaze your CS friends)

Overview of the Proof

Use proof by contradiction.

First, **assume program Halt-Checker(P,D) exists**.

Second, go through a sequence of logical-steps, and obtain a contradiction.

we build SuperSolve(P,D)

This implies that original assumption must be false.
(i.e., **Halt-Checker(P,D) does not exist**)

Proof: (1)

1. Assume program Halt-Checker(P,D) exists
2. Build new program SuperSolve(P,D)

```
PROGRAM SuperSolve(P,D)
1.  If  (Halt-Checker(P,D) = NO)
2.     then Stop                  // SuperSolve halts here
3.     else  goto Step 5
4.  endif
5.  Goto Step 5;                  // infinite loop here!
```

SuperSolve is also a program.
It makes use of Halt-Checker(P,D)

Proof: (2)

2b. Analyze Super-Solve(P,D)

```
PROGRAM SuperSolve(P,D)
1. If  (Halt-Checker(P,D) = NO)
2.     then Stop (or End)  // SuperSolve halts here
3.     Else  goto Step 5
4. endif
5. Goto Step 5;                  // infinite loop here!
```

Fact 1: Suppose "P running on D" does not halt. Then, in Step 1, Halt-Checker(P,D) outputs NO then in Step 2, SuperSolve halts.

Proof: (3)

2b. Analyze Super-Solve(P,D) …continued

PROGRAM SuperSolve(P,D)

1. **If (Halt-Checker(P,D) = NO)**
2. **then Stop (or End) // SuperSolve halts here**
3. **Else goto Step 5**
4. **endif**
5. **Goto Step 5; // infinite loop here!**

Fact 2: Suppose "P running on D" halts.
Then, in Step 1, Halt-Checker(P,D) outputs YES
then in Step 3,5 SuperSolve runs into
infinite loop. So SuperSolve does not halt.

2c. Summary of results (Fact 1 & Fact 2)

Fact 1: Suppose "P running on D" does not halt.
Then, in Step 1, Halt-Checker(P,D) outputs NO
then it goes to Step 2. **And SuperSolve halts**.

Fact 2: Suppose "P running on D" halts.
Then, in Step 1, Halt-Checker(P,D) outputs YES
then in Step 3,5 SuperSolve runs into infinite loop.
Then SuperSolve does not halt.

Facts 1 & 2 applies to a*ll program P* and *data D*,
SuperSolve is also a program.
So, let's set P=**SuperSolve**

2d'. Set P=**SuperSolve** in **Fact 1**

Fact 1: **Suppose "SuperSolve on D" does not halt.**

Then, in Step 1, Halt-Checker(**SuperSolve**,D) = NO

then it goes to Step 2. **And SuperSolve halts**.

So, if "SuperSolve running on D" does not halt, then we have a CONTRADICTION.

2d". Set P=**SuperSolve** in **Fact 2**

Fact 2: Suppose "**SuperSolve** running on D" halts.

Then, in Step 1, Halt-Checker(**SuperSolve**,D) = YES

then in Step 3,5 SuperSolve runs into
infinite loop. So SuperSolve does not halt.

So, if "SuperSolve running on D" halts,
then we have a CONTRADICTION.

Overview of the Proof

Use proof by contradiction

First, **assume program Halt-Checker(P,D) exists**.

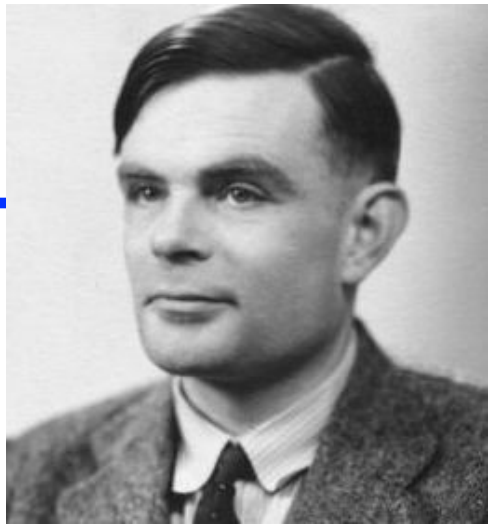
Second, go through a sequence of logical-steps, and obtain a contradiction.

Contradiction

we build SuperSolve(P,D)

This implies that original assumption must be false.
(i.e., **Halt-Checker(P,D) does not exist**)

Non-Computability



Theorem: (Turing, 1936)

Halting Problem is not computable.

Other non-computable problems:

Program Equivalence:
Given any two progam *P*1 and *P*2, we want to know if *P*1 and *P*2 always produce the same output (for all possible input data *D*)?

All Powerful Virus-Detector:
Given *any* program *P*, can we know if *P* is a virus.

Does this proof "look" familiar?

PQ: (Step 4) Looking Back.

Q: Did this proof "look" familiar to you?

A: Does it remind you of our friend Barber-Q?

In this proof, we created program SuperSolve, and we sent SuperSolve as input to SuperSolve.

Don't it remind you of "Self-Reference"?

Finally, just for fun….

PROGRAM Goto-Paris;

1. k ← 1
2. While (k ≥ 0) repeat Steps 3-4
3. Print "I Love GEQ1000. Thank you"
4. end-of-while-block
5. Print "Everyone in GEQ1000 goes to Paris"
6. End

Q1: Who goes to Paris?

Q2: Will Program Goto-Paris halt?

(End of video 6.5b)

If you want to contact me,

Email: leonghw@comp.nus.edu.sg

School of Computing