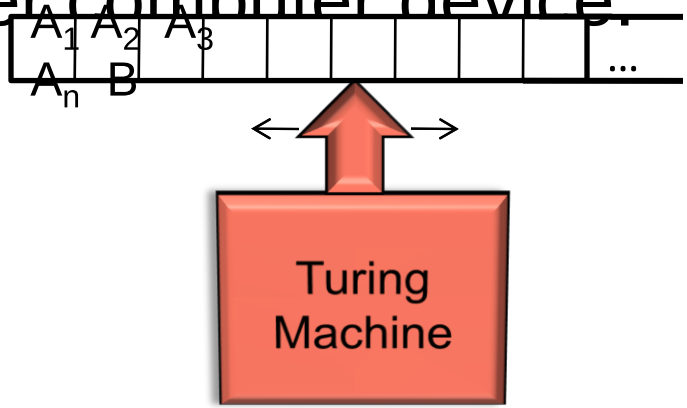
The Halting Problem

Overview

- Review TMs
- Parts of a TM
- Description of a TM
- Intro to Halting Problem
- Can a TM accept a TM as input?
- The Halting Problem Proof
- The Halting Problem is not possible in C
- UTMs and the TM in the Halting Problem
- References

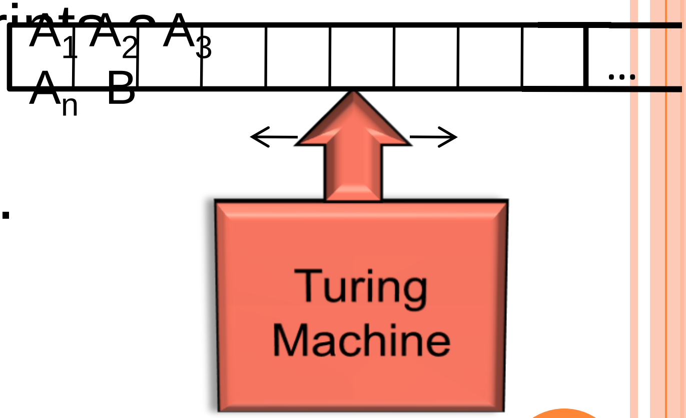
Turing Machines

- TMs finite, finite description.
- Model computation, and sophisticated methods.
- Theoretical model of a computing machine.
- As powerful as any other computer device.
- Has many properties…



PARTS OF A TM

- Semi-infinite input tape, containing an input word (string).
- Tape made of individual cells.
- Cells hold a symbol from the tape alphabet Γ.
- Read-write head reads then prints a symbol.
- Then head shifts one cell left or right.
- TM changes state internally.



TM Description
7 Tuple, M = (Q, Σ, Γ, δ, q_o, B, q_{accept})

- Q [finite set of states]
- Γ [gamma, the tape alphabet]
- B [the blank symbol, $B \in \Gamma$]
- Σ [sigma, the input alphabet]
- δ [delta, the transition function]
- q_o [initial state, $q_o \in Q$]
- q_{accept} [accept state]
- q_{reject} [reject state]

Limits to TMs

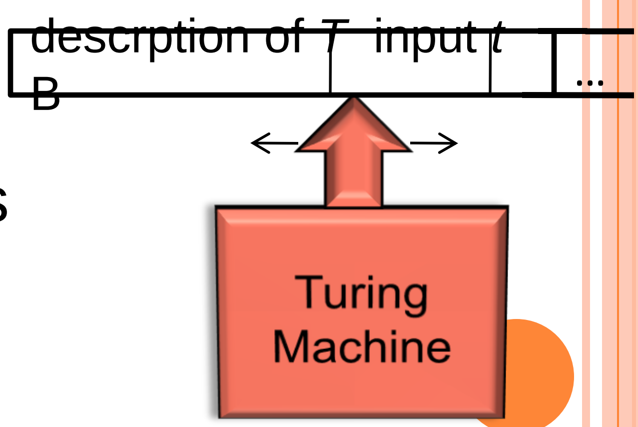
- There are limits to the power of TMs.
- A TM continues until it reaches accept state, or reject state where it will halt.
- If it never reaches one, then it continues computing forever.
- There exists problems that TMs cannot solve.
- These problems contain no effective procedure and no recursive computation exists.
- The problems unsolvable by TMs are also unsolvable by any equivalent formal programming systems.

INTRO TO THE HALTING PROBLEM

- The best known problem that is unsolvable by a TM is the Halting Problem.
- "Given an arbitrary Turing Machine *T* as input and equally arbitrary tape *t*, decide whether *T* halts on *t*."
- Basically TM that takes a TM, *T* as its input, and simulates the *T* running on input *t*, and returns or decides whether or not *T* halts on *t*.
- Can a TM accept a TM as input? (important to understand)
- 3 Examples.

CAN A TM ACCEPT A TM AS INPUT? EXAMPLE 1.

- Consider a Universal Turing Machine.
- UTMs represent the set of all possible TMs, and all possible effective procedures.
- UTMs take input in the form (*dT, t*).
- UTMs mimics the action of an arbitrary TM, *T* by reading its description off the tape, and simulates its behavior on *t*.
- Produces the same result as *T*.
- Simple TMs can also take descriptions of other TM as input.



Can a TM accept a TM as input? Example 2.

- TMs can be encoded as words, (strings) for other TMs.
- M = (Q, Σ, Γ, δ, q_o, B, q_{accept}) 7-tuples, only 4 are important.
- Represent finite set of states Q = {q_o, q_1, …} as a string in binary using unary conversion (*n+1* ones represent *n*).
- Represent Γ alphabet, 0, 1, move left, move right as a string of different size blocks of ones.
- Represent current state and next state transitions as a string using unary conversion.
- Use 0s as delimiters between strings.
- These 4 strings together make one string, the description of *T*.

CAN A PROGRAM ACCEPT A PROGRAM AS INPUT? EXAMPLE 3.

- Yes as a string, consider the valid C program.

```c
void main () {
    int i = 0;
    int a;
    scanf("%d", a); //read in the value of a
    while (i > a) { //if a is less than 0 loop forever
        i = a + 1;  //do the same computation over and over
    }               //if a is greater than 0 do not enter
}                   //end
```

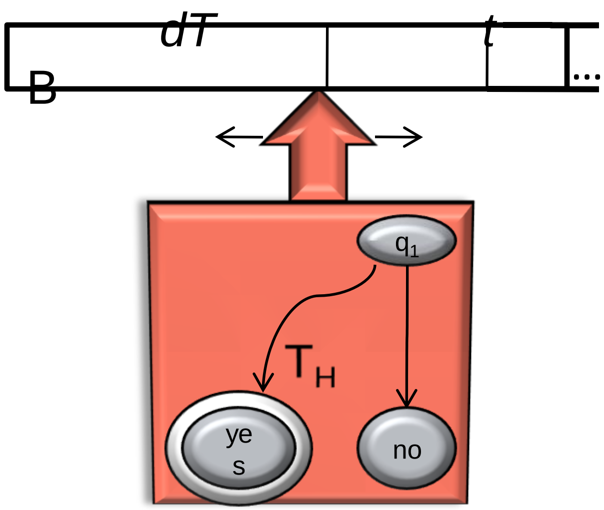
- The st
  input for another program.

- Once compiled, this is translated to machine language, then trar

```c
char *program = "void main () { int i = 0; int a; scanf("%d", &a);
                while (i > a) { i = a + 1; } }";
```

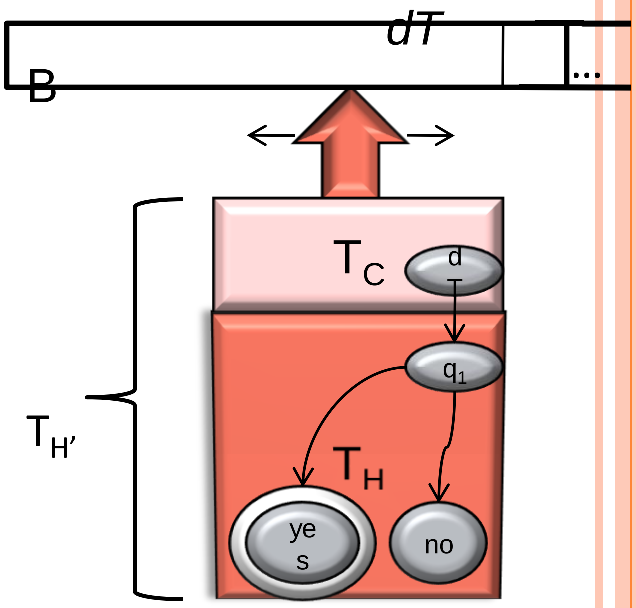
MACHINE *T* AS INPUT AND EQUALLY ARBITRARY TAPE *T*, DECIDE WHETHER *T* HALTS ON *T*."

- Formulate a proof, suppose such a machine does exist, call it T_H.
- Let *t* be input for *T*.
- Let *T* be encoded as a description for T_H.
- If *T* accepts and halts on *t*, then T_H will give an equivalent result and transfer to the halting "yes" state.
- If *T* does not halt on *t*, then T_H will transfer to the halting "no" state.
- If T_H exists, then we can construct another machine $T_{H'}$ by modifying T_H.



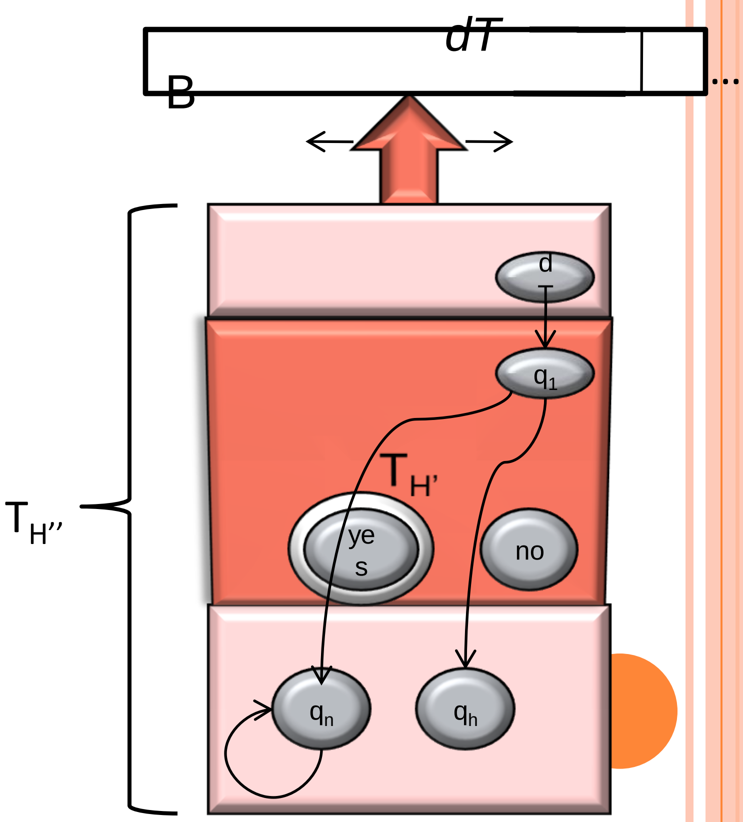
CONSTRUCT A NEW MACHINE $T_{H'}$

- Add another machine T_c (or some extra code) that makes a copy of *dT* and hands it to T_H's initial state.
- Alter T_H so that it decides if *T* halts on *dT* rather than *t*.
- T_H's only job is to decide if *T* halts on *dT*.
- If $T_{H'}$ exists, then we can construct another machine by modifying $T_{H'}$.



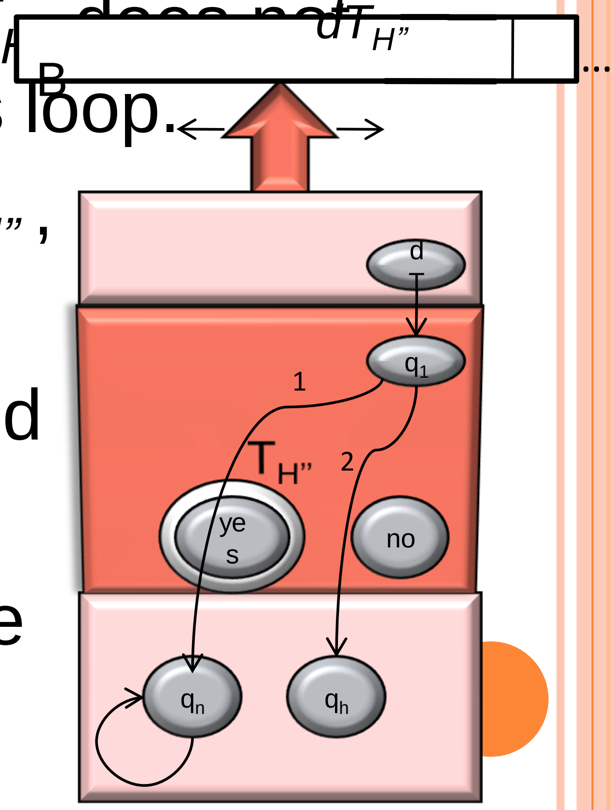
CONSTRUCT A NEW MACHINE $T_{H''}$

- Alter $T_{H'}$'s two halting transitions so that the *yes* and *no* state are diverted to two new states.
- The *yes* transition goes from q_1 to q_n, once in q_n it will never halt (infinite loop).
- The *no* transition goes from q_1 to q_h a halting state.



The Halting Problem

- If $T_{H''}$ exists, then we can input its own description $dT_{H''}$.
- Case 1: If $T_{H''}$ halts on $dT_{H''}$, then $T_{H''}$ does not halt on $dT_{H''}$ because of an endless loop.
- Case 2: If $T_{H''}$ does not halt on $dT_{H''}$, then $T_{H''}$ does halt on $dT_{H''}$.
- This contradicts that T_H ever existed in the first place.
- The Halting Problem is not solvable by any TM.



THE HALTING PROBLEM IS NOT POSSIBLE IN C .

- Assume a Halts() function exists. Input the c program from earlier into the function.

```
char *program = "void main () { int i = 0; int a; scanf("%d", &a);
                 while (i > a) { i = a + 1; } }";
```

- Imag

```
// Halts(P, I) == 1 if string P is a valid C program
//                        that eventually halts when reading
//                        input I.
//              == 0 otherwise

int Halts(char *P; char *I);
// <Insert source code of Halts here.>
// <Imagine halts functions like a compiler.>
```

- If Halts e

```
Halts(program, "1"); //returns 1
Halts(program, "-1"); //returns 0
```

THE HALTING PROBLEM IS NOT POSSIBLE IN C.

- Observe the new program in C. Save the program as diagonal.c
- Run diagonal and add its own source code as input.
- Halts(diagonal, diagonal) results in two cases.
- Returns 0, then diagonal loops forever, but this can only happen if Halts returns 1.
- Returns 1, then diagonal halts, but this can only happen if Halts returns 0.
- This contradiction means the Halts() function cannot exist.

```
// Halts(P, I) == 1 if string P is a valid C program
//                      that eventually halts when reading
//                      input I.
//             == 0 otherwise
  void main () {
  //read all of the input stream
  //mscanf is guarenteed to return
  char *program;
  mscanf("%s", &program); //input buffer

      if( Halts( program , program ) == 1 ) {
          while(1){
          //infinite loop
          }
      }
  }
```

Difference between UTMs and the TM in the Halting Problem.

- It's true that UTMs can simulate the behavior of any arbitrary TM *T* on its input *t* (*including itself*), and get the same result as *T*.
- Whether *T* halts and accepts, or halts and rejects, or runs infinitely a UTM will do the same.
- But a UTM or any TM cannot decide, or return a result that says if an arbitrary *T* will halt on an arbitrary *t*.
- The code for such a machine cannot exist because if it did, by the definition of the machine itself it should accept it's own code and not contradict itself.

Questions

- How is a TM converted into input for another TM?
- Why can’t we code Halts function in C?

References

- Dewdney, A. K. The New Turing Omnibus. 2001. New York. Chapter 59 “The Halting Problem.”
- Greenlaw, R., Hoover, H. James. Fundamentals of Theory of Computation. Morgan Kaufmann Publishers, Inc. 1998. San Francisco, California. Chapter 1 “Some Computing Puzzles.”
- Homer, S., Selman, Alan L. Computability and Complexity Theory. Texts in Computer Science. 2001 Springer-Verlag New York, Inc. Chapter 1 “Introduction to Computability,” and Chapter 3 “Undecidability.”
- Stanford Encyclopedia of Philosophy. Feb. 01, 2008. <http://plato.stanford.edu/entries/turing-machine/>