Introduction to Turing Machines and Complexity

So far we've been presenting several models of computing devices. Finite automata are good for devices with a small amount of memory and relatively simple control. Pushdown automata are good for devices with unlimited memory with a stack. However we have shown limitations of these models for even simple tasks. This makes them too restrictive for general purpose computers.

In contrast, the Turing Machine, first proposed by Alan Turing in 1936, is a much more powerful model. It is essentially a finite automaton but with an unlimited and unrestricted memory and is a more accurate model of a general purpose computer since it can do everything a general purpose computer can do (although slower). Nevertheless, there are problems that Turing machines can't solve; therefore a real computer can't solve them either.

This is where we make a change in direction. We will start to examine problems that are at the threshold and beyond the theoretical limits of what is possible to compute using computers today. We will examine the following issues with the help of TM's:

1. We use the simplicity of the TM model to prove formally that there are specific problems (i.e. languages) that the TM cannot solve.
   - Three classes: "Recursive" = TM can accept the strings in the language and determine if a string is not in the language. Sometimes these are called "decidable".
   - "recursively enumerable" = TM can accept the strings in the language but cannot tell for certain that a string is not in the language. Sometimes these are called "partially decidable".
   - "non-RE" = no TM can even recognize the members of the language. These are "non decidable."

2. We then look at problems (languages) that do have TM's that accept them and always halt; i.e., they not only recognize the strings in the language, but they tell us when they are sure the string is not in the language.

   - The classes P and NP are those languages recognizable by deterministic and nondeterministic TM's, respectively, that halt within a time that is some polynomial in the input. Polynomial is as close as we can get, because real computers and different models of (deterministic) TM's can differ in their running time by a polynomial function, e.g., a problem might take $O(n^2)$ time on a real computer and $O(n^6)$ time on a TM.

3. NP-complete problems: These are in a sense the "hardest" problems in NP. These problems correspond to languages that are recognizable by a nondeterministic TM. However, we will also be able to show that in polynomial time we can reduce any NP-complete problem to any other problem in NP. This means that if we could prove an NP Complete problem to be solvable in polynomial time, then P = NP.

4. Some specific problems that are NP-complete: satisfiability of boolean (propositional logic) formulas, traveling salesman, etc.

Intuitive Argument – A Problem Computers Cannot Solve : Undecidable Problems

Given a C program (or a program in any programming language, really) that prints "hello, world" is there another program that can test if a program given as input prints "hello, world"?

This is tougher than it may sound at first glance. For some programs it is easy to determine if it prints hello world. Here is perhaps the simplest:

```
#include "stdio.h"
void main()
{
    printf("hello, world\n");
}
```

It would be fairly easy to write a program to test to see if another program consisting solely of printf statements will output "hello, world". But what we want is a program that can take **any arbitrary program** and determine if it prints "hello, world". This is much more difficult. Consider the following program:

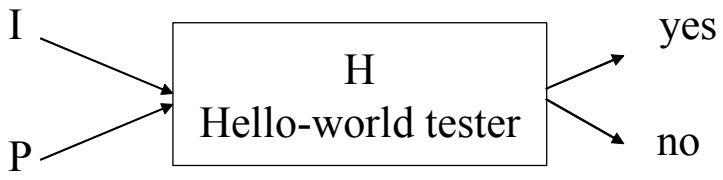
```
#include "stdio.h"
#define e 3
#define g (e/e)
#define h ((g+e)/2)
#define f (e-g-h)
#define j (e*e-g)
#define k (j-h)
#define l(x) tab2[x]/h
#define m(n,a) ((n&(a))==(a))
long tab1[]={ 989L,5L,26L,0L,88319L,123L,0L,9367L };
int tab2[]={ 4,6,10,14,22,26,34,38,46,58,62,74,82,86 };

main(m1,s) char *s; {
    int a,b,c,d,o[k],n=(int)s;
    if(m1==1){ char b[2*j+f-g]; main(l(h+e)+h+e,b); printf(b); }
    else switch(m1-=h){
        case f: a=(b=(c=(d=g)<<g)<<g)<<g;
            return(m(n,a|c)|m(n,b)|m(n,a|d)|m(n,c|d));
        case h:
for(a=f;a<j;++a)if(tab1[a]&&!(tab1[a]%((long)l(n))))return(a);
        case g:
            if(n<h)return(g);if(n<j){n-=g;c='D';o[f]=h;o[g]=f;}
            else{c='\r'-'\b';n-=j-g;o[f]=o[g]=g;}
            if((b=n)>=e)for(b=g<<g;b<n;++b)o[b]=o[b-h]+o[b-g]+c;
            return(o[b-g]%n+k-h);
        default: if(m1-=e) main(m1-g+e+h,s+g); else *(s+g)=f;
            for(*s=a=f;a<e;) *s=(*s<<e)|main(h+a++,(char *)m1);
        }
}
```

The program above, when compiled with a C compiler will actually print “hello, world”. It is an example of obfuscated C code (generally not a good example of how you should program!)

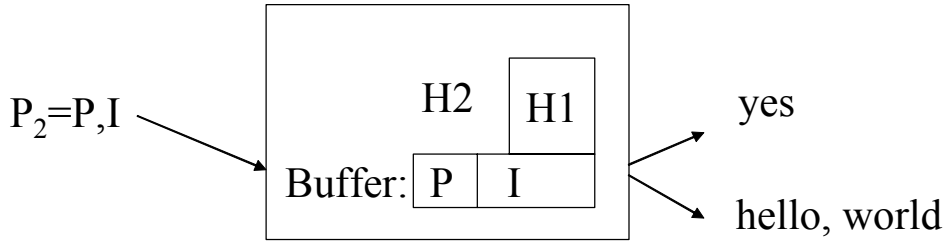
When we start to look at the problem of creating a program that can determine if any arbitrary program prints “hello world” we see this program can be very difficult to create indeed. In fact, we can prove there is no C program to solve that problem (called **undecidable**) by supposing that there were such a program H, the “hello-world-tester."

H takes as input a program P and an input file I for that program, and tells whether P, with input I, prints “hello world" (by which we mean it does so as the first 13 characters) by outputting “yes” if it does, and “no” if it does not.

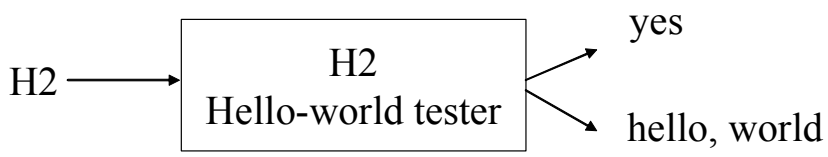


Next we modify H to a new program H1 that acts like H, but when H prints no, H1 prints “hello, world.”. To do this, we need to find where “no” is printed and change the printf statement to “hello world” instead:

Next modify H1 to H2 . The program H2 takes only one input, P_2, instead of both P and I. To do this, the new input P_2 must include the data input I and the program P. The program P and data input I are all stored in a buffer in program H2. H2 then simulates H1, but whenever H1 reads input, H2 feeds the input from the buffered copy. H2 can maintain two index pointers into the buffered data to know what current data and code should be read next:



However, H2 cannot exist. If it did, what would H2(H2) do?
That is, we give H2 as input to itself:



If H2 (H2) =yes, then H2 given H2 as input evidently does not print "hello, world.". It is printing "yes". If we go back to the original program H, then H(H2) outputs yes if H2 prints "hello, world". But H2 is not printing "hello, world" it is printing "yes" instead.

We have a similar contradiction if H2(H2) = "hello, world". But if H2 prints "hello, world" then back with H the output should be "yes". But if the output was "yes" then we would not get the output of "hello, world".

This situation is paradoxical and we conclude that H2 cannot exist. As a result, H1 cannot exist and H cannot exist. Therefore we have contradicted the assumption that H exists and no program H can tell whether or not another arbitrary program P prints "hello world".

Reducing one problem to another

Once we have a single problem known to be undecidable we can determine that other problems are also undecidable by **reducing** a known undecidable problem to the new problem. We will use this same idea later when we talk about proving problems to be NP-Complete.

To use this idea, we must take a problem we know to be undecidable. Call this problem U. Given a new problem, P, if U can be reduced to P so that P can be used to solve U, then P must also be undecidable.

It is important to show that the new problem P can be used to solve the undecidable problem U, and not vice-versa. If we show that our new problem can be solved by the undecidable problem, then it merely shows that something impossible for computers to do can solve our new problem. But it doesn't say anything about our new problem. We might just be using a really hard solution to an easy problem. But if we can show the other direction, that P can solve U, then P must be at least as hard as U, which we already know to be undecidable.

Consider the problem: Does program Q ever call function foo?
We can prove this problem is undecidable.

Just as we saw with the 'hello world' problem, it is easy to write a program that can determine if some programs call function foo. For example, a program with no function called foo obviously does not invoke function foo. But we could have a program that contains lots of control logic to determine whether or not function foo is invoked. This general case is much harder, and in fact undecidable. To prove this we use the reduction technique for the "hello world" problem:

1. Rename the function "foo" in program Q and all calls to that function.
2. Add a function "foo" that does nothing and is not called.
3. Modify the program to remember the first 12 characters that it prints, storing them in array A
4. Modify the program so that whenever it executes any output statement, it checks the array A to see if the 12 characters written are "hello, world" and if so, invokes function foo.

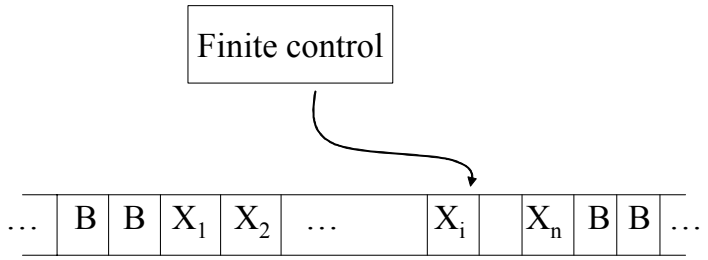
If the final program prints "hello, world" then it must also invoke function foo. Similarly, if the program does not print "hello, world" then it does not invoke foo.

Let's say that we have a program F-Test that can determine if a program calls foo. If we run F-Test on the modified program above, not only can it determine if a program calls foo, it can also determine if the program prints "hello, world". Therefore it is also capable of determining the "hello, world" problem. But we showed that this problem is undecidable, so our program F-Test must be undecidable as well.

We'll re-visit problem reduction when we come to talk about NP-Complete problems.

Turing Machines

The Turing machine was described back in 1936, well before the days of modern computers. Nevertheless, they have remained a popular model for what is possible to compute on today's systems. Most new advances in computing also operate under the model of the TM, which means that they may compute faster, but are subject to the same limitations of computers today. A TM may be visualized by the figure below. It consists of a finite control (i.e. a finite state automaton) that is connected to an infinite tape



The tape consists of cells where each cell holds a symbol from the tape alphabet. Initially the input consists of a finite-length string of symbols and is placed on the tape. To the

left of the input and to the right of the input, extending to infinity, are placed blanks. The tape head is initially positioned at the leftmost cell holding the input.

In one move the TM will:

1. Change state, which may be the same as the current state
2. Write a tape symbol in the current cell, which may be the same as the current symbol
3. Move the tape head left or right one cell

Formally, the Turing Machine is denoted by the 7-tuple:

$$M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$$

Where:

Q = finite states of the control
Σ = finite set of input symbols, which is a subset of Γ below
Γ = finite set of tape symbols
δ = transition function. $\delta(q,X)$ are a state and tape symbol X.
The output is the triple, (p, Y, D)
Where p = next state, Y = new symbol written on the tape, D = direction to move the tape head
q_0= start state for finite control
B = blank symbol. This symbol is in Γ but not in Σ.
F = set of final or accepting states of Q.

Example: let's make a TM that recognizes the language $L = \{ w\#w \mid w \in (0,1)^* \}$. That is, we have a language separated by a # symbol with the same string on both sides.

Here is a strategy we can employ to create the Turing machine:

1. Scan the input to make sure it contains a single # symbol. If not, reject.
2. Starting with the leftmost symbol, remember it and write an X into its cell. Move to the right, skipping over any 0's or 1's until we reach a #. Continue scanning to the first non-# symbol. If this symbol matches the original leftmost symbol, then write a # into the cell. Otherwise, reject.
3. Move the head back to the leftmost symbol that is not X.
4. If this symbol is not #, then repeat at step 2. Otherwise, scan to the right. If all symbols are # until we hit a blank, then accept. Otherwise, reject.

Basically, we are zig-zagging across the tape, matching corresponding symbols on each side of the # symbol. When all symbols have been checked and none remain, we accept the input.

Typically we will describe TM's in this informal fashion. The formal description gets quite long and tedious. Nevertheless, we will give a formal description for this particular problem.

We can use a table format or a transition diagram format. In the transition diagram format, a transition is denoted by:

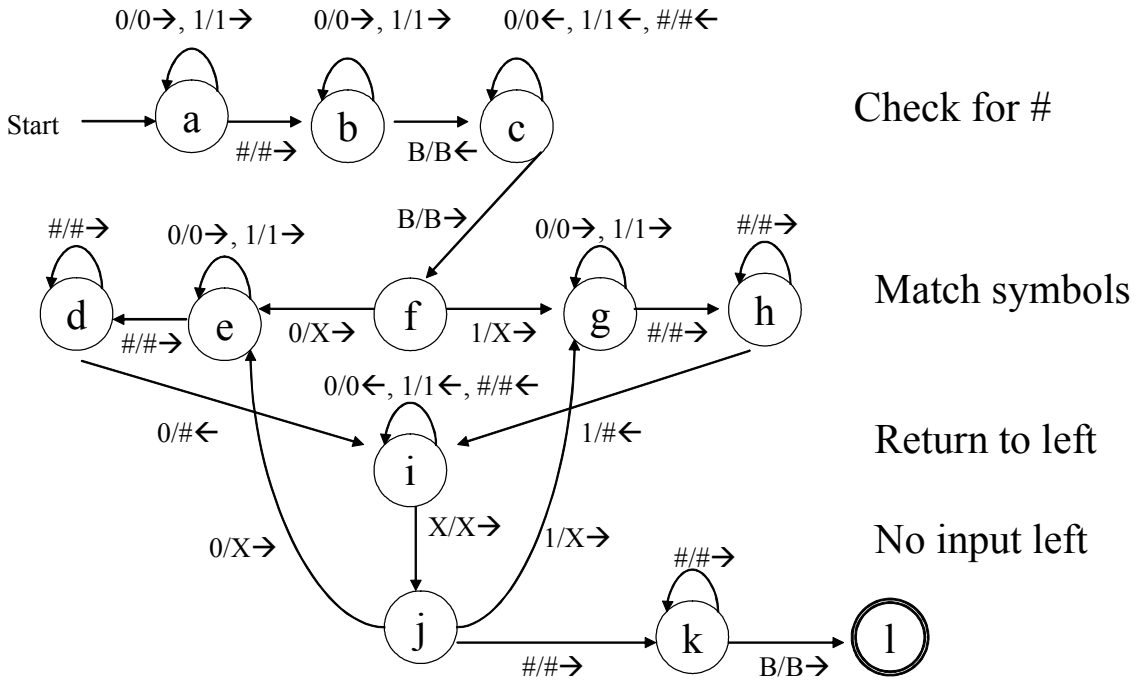
Input symbol / Symbol-To-Write Direction to Move

For example:

0 / 1 →

Means take this transition if the input is 0, and replace the cell with a 1 and then move to the right.

Here is the TM that accepts language L. As is usual, the TM "dies" if any unspecified input is received.



Instantaneous Descriptions for Turing Machines

Sometimes it is useful to describe what a TM does in terms of its ID (instantaneous description), just as we did with the PDA. In the ID, we show all non-blank cells in the tape with the current state in the finite control as a "pointer" to what cell the head is over.

When we move to another state, we use the turnstile symbol ⊢ to denote the move. As before, to denote zero or many moves, we can use $\vdash^*$.

For example, for the above TM on the input 10#10 we can describe our processing as follows:

Ba10#10B ⊢ B1a0#10B ⊢ B10a#10B ⊢ B10#b10B ⊢
B10#b10B ⊢ B10#1b0B ⊢ B10#10bB ⊢ B10#1c0B $\vdash^*$
cB10#10B ⊢ Bf10#10B
$\vdash^*$ BXX#XXBl

In this example I showed the blanks that border the input symbols since they are used in the turing machine to define the borders of our input.

Turing Machines and Halting

One way for a TM to accept input is to end in a final state. Another way is acceptance by halting. We say that a TM halts if it enters a state q, scanning a tape symbol X, and there is no move in this situation; i.e. $\delta(q,X)$ is undefined.

Note that this definition of halting was not used in the transition diagram for the TM we described earlier; instead that TM died on unspecified input. However, it is possible to modify the prior example so that there is no unspecified input except for our accepting state. An equivalent TM that halts exists for a TM that accepts input via final state.

In general, we assume that a TM always halts when it is in an accepting state.

Unfortunately, it is not always possible to require that a TM halts even if it does not accept the input. Such languages are called *recursive*. Turing machines that always halt, regardless of accepting or not accepting, are good models of algorithms for decidable problems.

More on Turing Machines

There are a number of extensions or variations that we can make to the basic Turing machine. While these extensions can be useful to prove a particular theorem, they do not add anything extra that the basic Turing machine can't already compute.

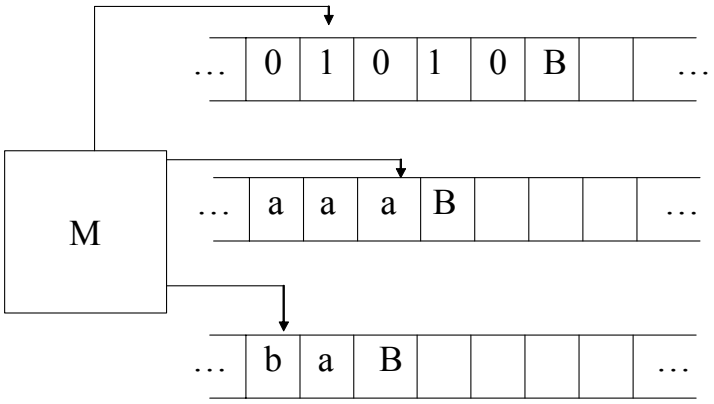
As a simple example, consider a variation to the Turing machine where we have the option of staying put instead of forcing the tape head to move left or right by one cell. This new machine may be more convenient to represent certain languages, but it doesn't add to the power of the model. In the old model, we could replace each "stay put" move in the new machine with two transitions, one that moves right and one that moves left, to get the same behavior.

Multitape Turing Machines

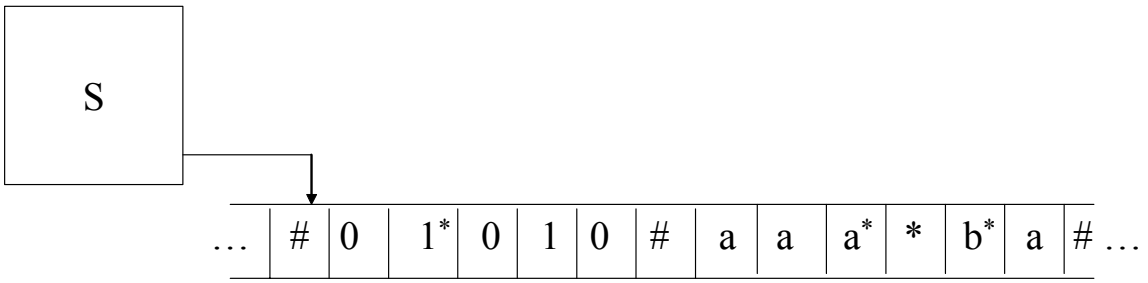
A multitape Turing machine is like an ordinary TM but it has several tapes instead of one tape. Initially the input starts on tape 1 and the other tapes are blank. The transition function is changed to allow for reading, writing, and moving the heads on all the tapes simultaneously. This means we could read on multiples tape and move in different directions on each tape as well as write a different symbol on each tape, all in one move.

Theorem: A multitape TM is equivalent in power to an ordinary TM. Recall that two TM's are equivalent if they recognize the same language. We can show how to convert a multitape TM, M, to a single tape TM, S:

Say that M has k tapes. Then create TM S so that it simulates the effect of having k tapes by interleaving each of the k tapes information on its single tape. This uses a new symbol # as a delimiter to separate the contents of each tape. In addition to the contents of the tapes, S keeps track of the location of the heads. This is done by writing a tape symbol with a * to mark the place where the head on the tape would be. The tape symbol with the * are new tape symbols that don't exist with M. The finite control must have the proper logic to distinguish say, x* and x and realize both refer to the same thing, but one is the current tape symbol.



We can construct an equivalent single tape machine S using the delimiter as shown below to encode the data from all k tapes from M:



To simulate a single move on M, S must scan its tape from the first # symbol Then we must find the symbol with the *. This indicates the current input symbol for that

particular tape on M. We then process that input, translating the symbol with the * to the symbol without the *. This process is repeated for each virtual tape until all tapes are completed.

If at any point S moves one of the virtual tape heads onto a #, then this action signifies that M has moved the corresponding head onto the previously unread blank portion of that tape. To accommodate this situation, S writes a blank symbol on this tape cell and shifts the tape contents to the rightmost # by one, adds a new #, and then continues.

Nondeterministic Turing Machines

Just like we constructed NFA's for DFA's, we can do the same thing with a TM. To turn a regular TM into a nondeterministic TM, we simply allow nondeterminism for each move. Each time we make a nondeterministic move, you can think of this as a branch or "fork" to two simultaneously running machines. Each machine gets a copy of the entire tape. If any one of these machines ends up in an accepting state, then the input is accepted.

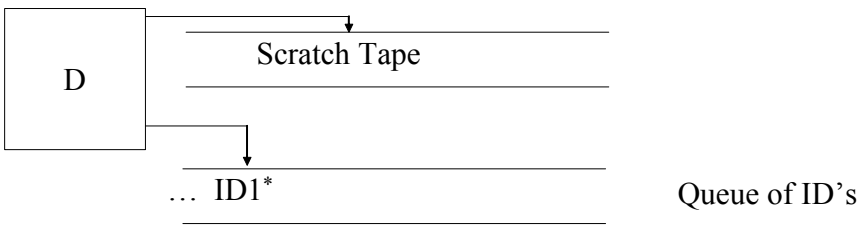
Although powerful, nondeterminism does not affect the power of the TM model:

Theorem: Every nondeterministic TM has an equivalent deterministic TM.

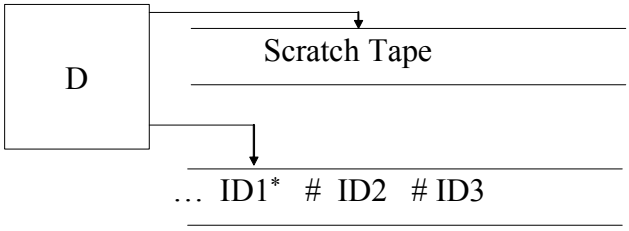
We can prove this theorem by simulating any nondeterministic TM, N, with a deterministic TM, D. The basic idea is to visualize N as a tree with branches whenever we fork off to two (or more) simultaneous machines. We then use D to try all possible branches of N, sequentially. If D ever finds the accept state on one of these branches, then D accepts. It is quite likely that D will never terminate in the event of a loop if there is no accepting state.

It is crucial that this search be done in a breadth-first rather than depth-first manner. An individual branch may extend to infinity, and if we start down this branch then D will be stuck forever when some other branch may accept the input.

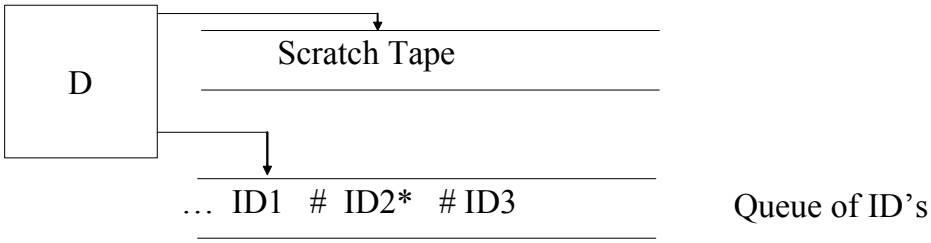
We can simulate M on D by a tape with a queue of ID's and a scratch tape for temporary storage. Each ID contains all the moves we have made from one state to the next, for one "branch" of the nondeterministic tree. From the previous theorem, we can make as many multiple tapes as we like and this can is still equivalent to a single tape machine. Initially, D looks like the following:



ID1 is the sequence of moves we make from the start state. The * indicates that this is the current ID we are executing. We make a move on the TM. If this move results in a "fork" by following nondeterministic paths, then we create a new ID and copy it to the end of the queue using the scratch tape. For example, say that in ID1 we have two nondeterministic moves, resulting in ID2 and ID3:



We use the special # symbol to separate each ID on the tape. After we're done with a single move in ID1, which may result in increasing the length of ID1 and storing it back to the tape, we move on to ID2:



Here, the * indicates which ID we are working on in the queue. If any one of these states is accepting in an ID, then the machine quits and accepts. If we ever reach the last ID, then we repeat back with the first ID.

Note that although the constructed deterministic TM is equivalent to accepting the same language as a nondeterministic TM, the deterministic TM might take exponentially more time than the nondeterministic TM. It is unknown if this exponential slowdown is necessary. We'll come back to this in the discussion of P vs. NP.

Theorem: Since any deterministic Turing Machine is also nondeterministic (there just happens to be no nondeterministic moves), there exists a nondeterministic TM for every deterministic TM.

In the text but skipping here: We can also create TM's with multiple stacks, and enumerable machines (counters).

Equivalence of TM's and Modern Computers

In one sense, a real computer has a finite number of states, and thus is **weaker** than a TM. But, we can postulate an infinite supply of tapes, disks, or some peripheral storage device to simulate an infinite TM tape. Additionally, we can assume there is a human operator to mount disks, keep them stacked neatly on the sides of the computer, etc.

To show the equivalence we need to show that we can simulate a TM on a computer, and that we can simulate a computer on a TM.

1. Simulate a TM on a Computer

This direction is fairly easy. Given a computer with a modern programming language, certainly, we can write a computer program that emulates the finite control of the TM. The only issue remains the infinite tape. Our program must map cells in the tape to storage locations in a disk. When the disk becomes full, we must be able to map to a different disk in the stack of disks mounted by the human operator.

2. Simulate a Computer on a TM

In this exercise the simulation is performed at the level of stored instructions and accessing words of main memory.

- TM has one tape that holds all the used memory locations and their contents.
- Other TM tapes hold the instruction counter, memory address, computer input file, and scratch data.
- The computer's instruction cycle is simulated by:
  1. Find the word indicated by the instruction counter on the memory tape.
  2. Examine the instruction code (a finite set of options), and get the contents of any memory words mentioned in the instruction, using the scratch tape.
  3. Perform the instruction, changing any words' values as needed, and adding new address-value pairs to the memory tape, if needed.

This indicates that anything a computer can do, a TM can do, and vice versa. Although it would be slow to simulate a computer on a TM, we can show that the difference in speed is polynomial. Each step done on the computer can be completed in $O(n^2)$ steps on the TM (see book for details of proof).

While slow, this is key information if we wish to make an analogy to modern computers. Anything that we can prove using Turing machines translates to modern computers with a polynomial time transformation.

Whenever we talk about defining algorithms to solve problems, we can equivalently talk about how to construct a TM to solve the problem. If a TM cannot be built to solve a particular problem, then it means our modern computer cannot solve the problem either.

Decidability

We have already seen an example of an undecidable problem, that of creating a program capable of determining if another program outputs "hello, world". This is analogous to another famous problem, the halting problem.

The halting problem is, is it possible to tell if another program will halt or not? This is easy for some programs. But for others it is not – in fact, it is impossible to solve for an arbitrary input program. We saw this earlier with the program that tests for the output "hello world". If we change this problem to "test for a program that halts or not" then we can use the same argument with the hello-world-tester to prove that the Halting problem is not decidable.

Let's look at a few other languages that are not decidable. First we need to cover some math.

Countability and Diagonalization

The notion of countability was described by Georg Cantor in 1873. If we have two infinite sets, how can we tell whether one is larger than the other? Obviously we can't start counting with each element or we will be counting forever. Cantor's solution is to make a **correspondence** to the set of natural numbers.

A correspondence is a function f: A→B that is one-to-one from A to B. Every element of A maps to a unique element of B, and each element of B has a unique element of A mapping to it.

Example: The set of natural numbers, N = { 1, 2, 3, 4, … }.
The set of even numbers, E = { 2, 4, 6, … }
It might seem that N is bigger than E, since we have values in N that are not in E (by one measure, we have twice as many.) However, using Cantor's definition of size both sets have the same size:

N	E = F(N)
1	2
2	4
3	6
etc.	

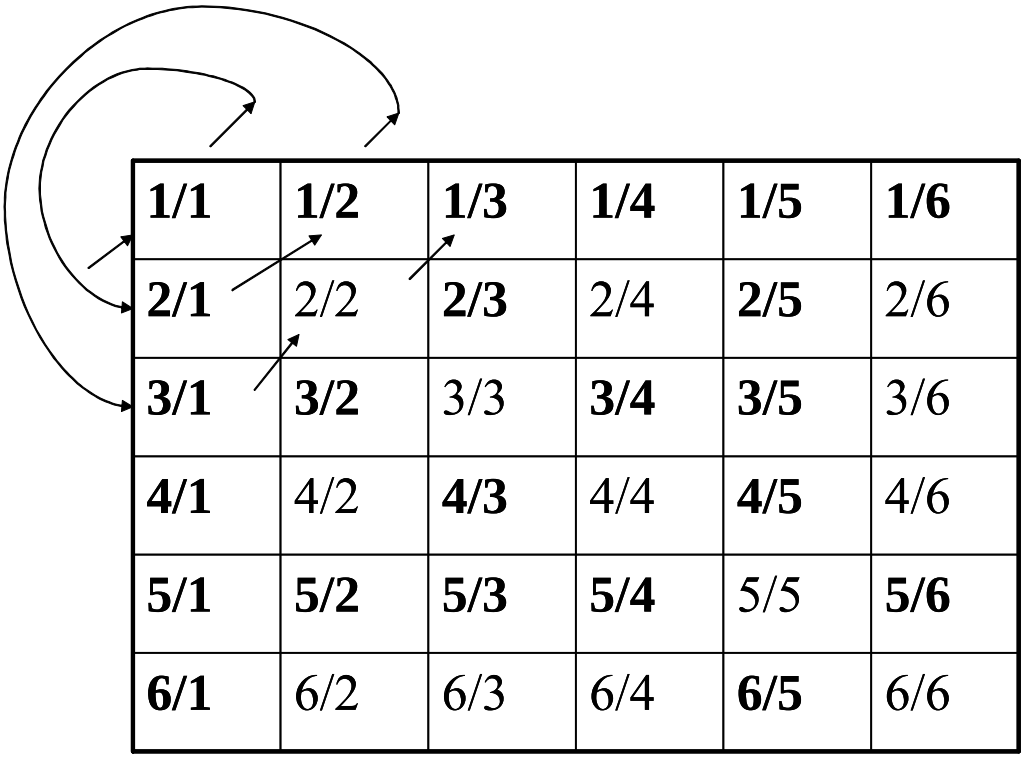
For every number in N, there is a corresponding value in E.

Definition: A set is **countable** if it is finite or if it has the same size as the natural numbers.

For example, as we saw above, E is countable.

Example: The set of positive rational numbers, Q, is countable. That is,
$Q = \{ m/n \mid m, n \in N \}$.

To show that this is countable, we need to make a 1:1 correspondence between the rational numbers and the natural numbers. We must make sure that each rational number is paired with one and only one natural number. Consider the mapping as shown in the matrix below:



1/1	**1/2**	**1/3**	**1/4**	**1/5**	**1/6**
2/1	2/2	**2/3**	2/4	**2/5**	2/6
3/1	**3/2**	3/3	**3/4**	**3/5**	3/6
4/1	4/2	**4/3**	4/4	**4/5**	4/6
5/1	**5/2**	**5/3**	**5/4**	5/5	**5/6**
6/1	6/2	6/3	6/4	**6/5**	6/6

. . .

If we started in the first row and just worked our way to the right, we could assign each rational number to a natural number:

1/1 → 1
1/2 → 2
1/3 → 3

etc.

However, we would never make an assignment for values like 2/1.
The solution is the traverse the diagonals of the matrix. We assign values as:

1/1 → 1
2/1 → 2
1/2 → 3
3/1 → 4
2/2 → skipped
1/3 → 5

etc.

Notice how we had to skip any elements that would cause a repeat. Continuing in this way, we can obtain a list for all elements of Q and therefore Q is countable.

Uncountable Sets

Since we have seen infinite sets that are countable, it might seem like any infinite set is countable. However, this is not the case.

Example: The set of real numbers, R, is not countable.

Suppose that R is countable. Then there is a correspondence between members of R and members of N. The following table shows some hypothetical correspondences:

N	R
1	3.14159…
2	55.555…
3	0.12345…
4	0.50000…
…	

Given such a table, we can construct a value x that is in R but that has no pairing with a member in N.

To construct x, we ensure it has a digit that is different from all values listed in the table. We can do this by starting with the first fractional digit of 3.14159. This is the digit 1. So we pick something different, say we pick 4. Then we move to the second value. The second fractional digit of 55.555 is 5. So we pick something different, say 6. The third fractional digit of 0.12345 is 3. So we pick something different, say 1.

We can continue in this way, to construct x = 0.4612 …

The value x is in R. However, we know that x has no corresponding value in N because it differs from n in N by the nth fractional digit. Since x has no corresponding value in N, the set of real numbers is not countable.

This result is important because it tells us something our TM's and computers cannot compute. It is impossible to exactly compute the real numbers – we must settle for something else, e.g. a less precise answer or computation.

Corollary: There exist languages that are not recognizable by a Turing Machine.

First, the set of all Turing machines is countable. We can show this by first observing that the set of all strings Σ^* is countable, for a finite alphabet Σ. With only finitely many strings of each length, we may form a list of Σ^* by writing down all strings of length 0, all strings of length 1, all strings of length 2, etc.

The set of all TM's is countable because each TM may be encoded by a string s. This string encodes the finite control of the TM. If we omit those strings that are not valid TM's, then we can obtain a list of all TM's.

To show that the set of languages is not countable, observe that the set of all infinite binary sequences is uncountable. The proof for this is identical to the proof we used to show that the set of real numbers is uncountable.

The set of all languages has a correspondence to the set of all infinite binary sequences. (For the alphabet of {0,1} they are the same. For languages with more than two symbols, we use multiple bits to represent the symbols). Therefore, the set of all languages is also uncountable and we conclude there are languages that are not recognized by any Turing machine.

In fact, there are many more languages that are not recognized by TM's than languages that are recognized by TM's. Fortunately, most of the time we don't care about these other languages, but are only interested in ones that TM's can recognize.

The above property of enumerability is one reason why we call languages recognizable by TM's to be *recursively enumerable*. The "recursion" part is historical, from using recursion to implement many of these problems (and therefore meaning that this problem is decidable, or recognizable by a TM).