Module-5
TURING MACHINE

Problems that Computer Cannot Solve

Consider a simple C program with a simple print statement:

```
main()
{
    printf("hello, world\n");
}
```

It is easy to discover that this program prints **hello, world** and terminates.

However, we may have another program that might print **hello, world** as shown below. It takes an input n, and looks for positive integer solutions to the equation $x^n+y^n=z^n$. If it finds one, it prints **hello, world.** If it never finds integers x, y, and z to satisfy the equation, then it continues searching forever and never prints **hello, world.**

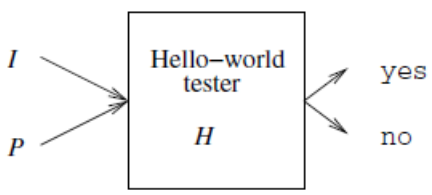
```
int exp(int i, n)
/* computes i to the power n */
{
    int ans, j;
    ans = 1;
    for (j=1; j<=n; j++) ans *= i;
    return(ans);
}

main ()
{
    int n, total, x, y, z;
    scanf("%d", &n);
    total = 3;
    while (1) {
        for (x=1; x<=total-2; x++)
            for (y=1; y<=total-x-1; y++) {
                z = total - x - y;
                if (exp(x,n) + exp(y,n) == exp(z,n))
                    printf("hello, world\n");
            }
        total++;
    }
}
```

If the value of n that the program reads is 2, then it will eventually find combinations of integers such as `total` $= 12$, $x = 3$, $y = 4$, and $z = 5$, for which $x^n + y^n = z^n$. Thus, for input 2, the program *does* print `hello, world`.

However, for any integer $n > 2$, the program will never find a triple of positive integers to satisfy $x^n + y^n = z^n$, and thus will fail to print `hello, world`. Interestingly, until a few years ago, it was not known whether or not this program would print `hello, world` for some large integer n. The claim that it would not, i.e., that there are no integer solutions to the equation $x^n + y^n = z^n$ if $n > 2$, was made by Fermat 300 years ago, but no proof was found until quite recently. This statement is often referred to as "Fermat's last theorem."

- **Hypothetical tester:** If a problem has an algorithm like H that always tells correctly whether an instance of the problem has answer "yes" or "no" then the problem is said to be "decidable". Otherwise, the problem is "undecidable".



A hypothetical program H that is a hello-world detector

The **hypothetical tester** is a conceptual tool introduced in the study of undecidable problems to illustrate why certain problems cannot be solved by any algorithm. In this context, the "hypothetical tester" is designed to determine whether a given program, when executed with specific input, will output a particular result—in this case, whether it prints **"hello, world"** as its first output.

This program, denoted as H, takes two inputs:

1. A program P (the code to be tested).
2. An input I (the input to the program P).

The goal of H is to decide if the program P with input I will produce the output **"hello, world"** as the first set of characters it prints.

Expected Behavior of H:

- If P with input I prints "hello, world" as its first output, H should return **"yes"**.
- Otherwise, it should return **"no"**.

The challenge arises because:

- A program P may take an **infinite amount of time** to decide its output. For instance:
- If P is searching for a mathematical solution (like Fermat's Last Theorem) before it prints anything, H cannot predict whether it will eventually find a solution and print "hello, world" or loop indefinitely.
- Even for programs that terminate, analyzing the behavior of every possible program P with every possible input I is **non-trivial and undecidable**.

The hypothetical tester is a classic example used to demonstrate:

1. **Limits of Computation**: Some problems, even simple-sounding ones, are fundamentally unsolvable by any algorithm.
2. **Undecidability**: The inability to solve a problem algorithmically is a critical concept in theoretical computer science and is at the core of problems like the Halting Problem.

The **Halting Problem** is a fundamental concept in computer science and computational theory. It addresses whether it's possible to design an algorithm that can determine, for any given program and input, whether the program will eventually stop running (halt) or continue running forever.

Reducing one problem to another: A problem that cannot be solved by computer is called undecidable. A **reduction technique** is used to prove the **undecidability of** a halting problem. It is sufficient to show that if we could solve the new problem, then we could use that solution to solve a problem we already know is undecidable. The strategy is suggested in the figure given below. The technique is called the reduction of P1 to P2.

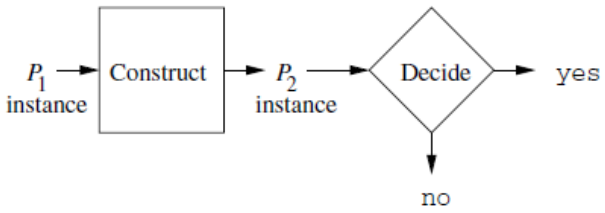


Figure 8.7: If we could solve problem P_2, then we could use its solution to solve problem P_1

Using this technique, a problem P1 is reducible to problem P2 if a solution to the problem P2 can be used to solve the problem P1. Thus,

- if P1 is reducible to P2 and P2 is decidable, then P1 is decidable
- If P1 is reducible to P2 and P2 is undecidable, then P1 is undecidable.

What is Turing Machine?

In 1936, **Alan M Turing** proposed the Turing machine as a model of any possible computation. This model is computer like, rather than program like, even though true electronic, or even electromechanical computers were several years in the future.

A Turing machine is a computational model, like **Finite Automata (FA)**, **Pushdown automata (PDA)**, which works on unrestricted grammar. The Turing machine is the most powerful computation model when compared with FA and PDA. Instead of using stack as in PDA, the Turing Machine (TM) uses the tape to store the symbols.

Church-Turing thesis

Any algorithmic procedure that can be carried out by a human or a computer, can also be carried out by a Turing machine. Now it is universally accepted by computer scientists that TM is a Mathematical model of an algorithm. TM has an algorithm and an algorithm has a TM. **If there is an algorithm problem is decidable, TM solves that problem. This is known as Church-Turing thesis.**

Notation of Turing Machine:

(Q. Explain the Turing Machine model working principle. (4M))

The Turing machine consists of a finite control, which can be in any of a finite set of states. There is a tape divided into squares or cells, each cell can hold any one of a finite number of symbols.

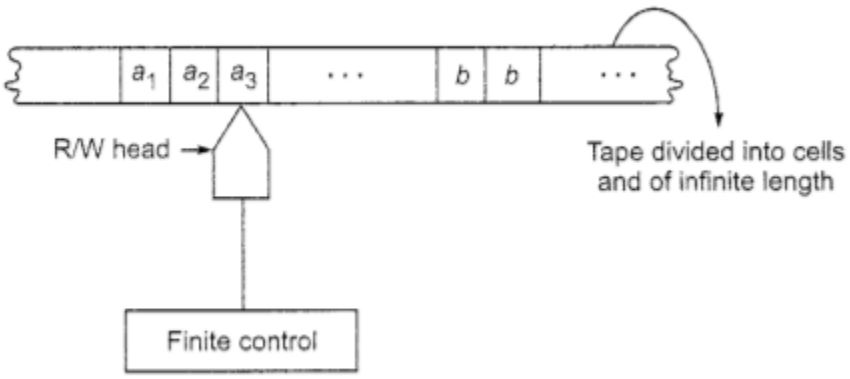


Fig. 9.1 Turing machine model.

- Initially, the input, which is a finite length string of symbols chosen from the input alphabet, is placed on the tape. All other tape cells, extending infinitely to the left and right, initially hold a special symbol called the blank. The blank is a tape symbol, but not an input symbol, and there may be other tape symbols besides the input symbols and the blank, as well.

- There is a tape head that is always positioned at one of the tape cells. The Turing machine is said to be scanning that cell. Initially, the tape head is at the leftmost cell that holds the input.

- A move of the Turing machine is a function of the state of the finite control and the tape symbol scanned. In one move, the Turing machine will,

  a) Change state: The next state optionally may be the same as the current state.
  b) Write a tape symbol in the cell scanned. This tape symbol replaces whatever symbol was in that cell. Optionally, the symbol written may be the same as the symbol currently there.
  c) Move the tape head left or right. In our formalism we require a move, and do not allow the head to remain stationary. This

restriction does not constrain what a Turing machine can compute, since any sequence of moves with a stationary head could be condensed, along with the next tape head move, into a single state change, a new tape symbol, and a move left or right.

Definition of Turing Machine:

(Q. Define Turing machine-2M)

A Turing machine M is a 7-tuple, namely **(Q, Σ, Γ, δ, qo, B, F)**, where,

- Q: is the finite nonempty set of states.
- Γ is a finite nonempty set of tape symbols,
- B ∈ Γ is the blank.
- Σ is a nonempty set of input symbols and is a subset of Γ and B ∉ Σ.
- δ is the transition function.

$$\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}.$$

The arguments of δ (q, X) are a state q and a tape symbol X. The value of δ(q, X) is a triple (p, Y, D) where:

  - p is the next state, in Q.
  - Y is the symbol in Γ, written in the cell being scanned, replacing whatever symbol was there.
  - D is the direction, either L or R, standing for "left" or "right" specifying the direction in which the R/W head moves.

- qo ∈Q is the initial state, and
- F ⊆ Q is the set of final or accepting states.

Instantaneous Description (IDs) of Turing Machine:

In Turing Machine, the instantaneous description is defined on the whole string and the current state of the machine.

Thus, we shall use the string $X_1X_2,\ldots\ldots X_{i-1}\ \mathbf{q}X_iX_{i+1}\ldots\ldots X_n$ to represent an ID in which,

1. q is the state of the Turing machine.
2. The tape head is scanning the ith symbol from the left.
3. $X_1X_2,\ldots X_n$ is the portion of the tape between the leftmost and the right most nonblank. As an exception, if the head is to the left of the leftmost nonblank or to the right of the rightmost nonblank, then some prefix or suffix of $X_1X_2,\ldots X_n$ will be blank.

- We use $\vdash_M$ to designate a move of a Turing machine M from one ID to another.
- If $\delta(q, X_i) = (p, Y, L)$, then:

  $X_1X_2 \cdots X_{i-1}qX_iX_{i+1} \cdots X_n \vdash_M$

  $X_1X_2 \cdots X_{i-2}pX_{i-1}Y\,X_{i+1} \cdots X_n$
- If $\delta(q, X_i) = (p, Y, R)$, then:

  $X_1X_2 \cdots X_{i-1}qX_iX_{i+1} \cdots X_n \vdash_M$

  $X_1X_2 \cdots X_{i-1}Y\,pX_{i+1} \cdots X_n$

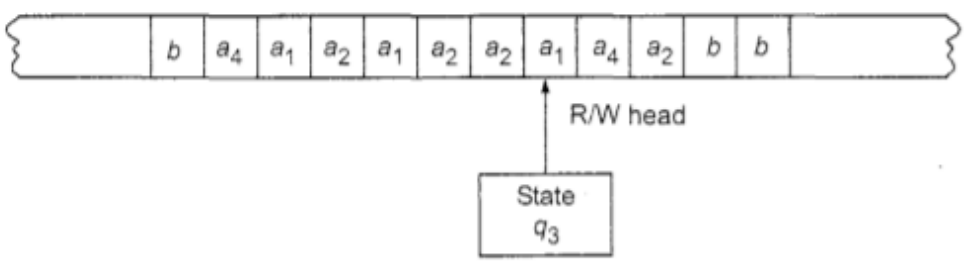


Fig. 9.2 A snapshot of Turing machine.

Thus, the ID is as given in Fig. 9.3.

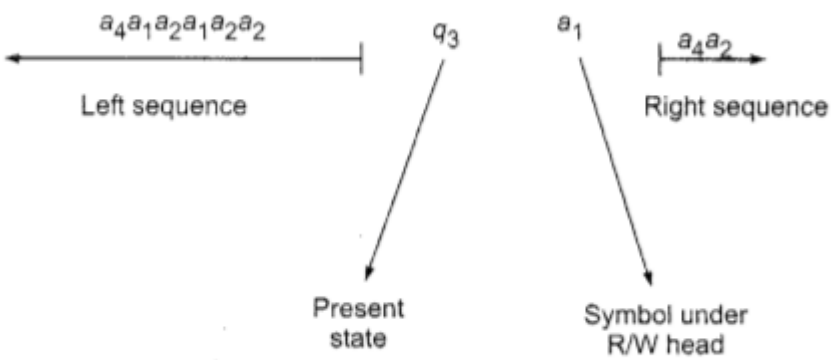


Fig. 9.3 Representation of ID.

Designing Turing Machine:

1. **Let us design a Turing machine and see how it behaves on a typical input. The TM we construct will accept the language L={0^n 1^n:n>=1}**

Initially, it is given a finite sequence of 0's and 1's on its tape, preceded and followed by an infinity of blanks. Alternately, the TM will change a 0 to an X and then a 1 to a Y, until all 0's and 1's has been matched.

The formal specification of the TM M is,

$$M = (\{q_0, q_1, q_2, q_3, q_4\}, \{0, 1\}, \{0, 1, X, Y, B\}, \delta, q_0, B, \{q_4\})$$

where δ is given by the table in Fig. 8.9.

State	Symbol 0	1	X	Y	B
q_0	(q_1, X, R)	–	–	(q_3, Y, R)	–
q_1	$(q_1, 0, R)$	(q_2, Y, L)	–	(q_1, Y, R)	–
q_2	$(q_2, 0, L)$	–	(q_0, X, R)	(q_2, Y, L)	–
q_3	–	–	–	(q_3, Y, R)	(q_4, B, R)
q_4	–	–	–	–	–

Figure 8.9: A Turing machine to accept $\{0^n 1^n \mid n \geq 1\}$

This table is called as transition table.

Transition Diagrams for Turing Machines:

A transition diagram for TM consists of a set of nodes corresponding to the states of the TM. An arc from state q to state p is labeled by one or more items of the form X/Y, D where X and Y are tape symbols, and D is a direction, either L or R. That is, whenever δ(q,X)=(p, Y, D), we find the label X/Y, D on the arc from q to p.

Let $L = \{0^n 1^n : n \geq 1\}$, which is a CFL. We will design a Turing machine M accepting L. Let

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

where $Q = \{q_0, q_1, \ldots, q_4\}$, $\Sigma = \{0, 1\}$, $\Gamma = \{0, 1, X, Y, B\}$, $F = \{q_4\}$, and δ is shown in the following figure.

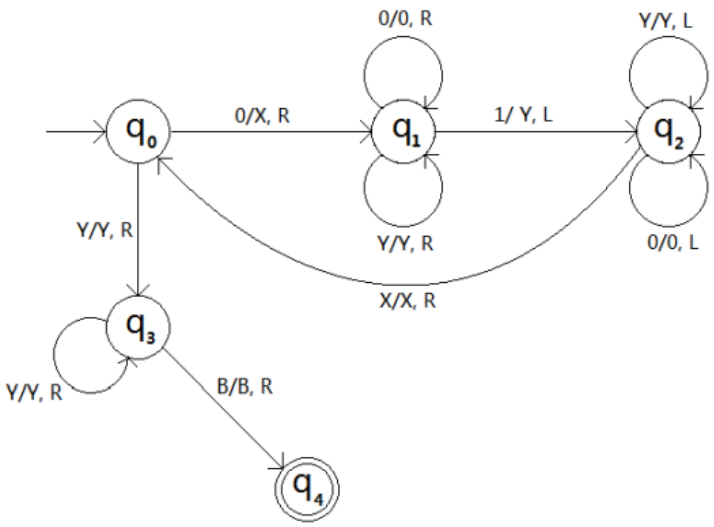


Figure 3: Turing machine accepts $\{0^n 1^n : n \geq 1\}$

Let show the IDs for the input strings 0011 and 0010.

Here is an example of an accepting computation by M. Its input is 0011. Initially, M is in state q_0, scanning the first 0, i.e., M's initial ID is $q_0 0011$. The entire sequence of moves of M is:

$$q_0 0011 \vdash X q_1 011 \vdash X 0 q_1 11 \vdash X q_2 0 Y 1 \vdash q_2 X 0 Y 1 \vdash$$
$$X q_0 0 Y 1 \vdash X X q_1 Y 1 \vdash X X Y q_1 1 \vdash X X q_2 Y Y \vdash X q_2 X Y Y \vdash$$
$$X X q_0 Y Y \vdash X X Y q_3 Y \vdash X X Y Y q_3 B \vdash X X Y Y B q_4 B$$

For another example, consider what M does on the input 0010, which is not in the language accepted.

$$q_0 0010 \vdash X q_1 010 \vdash X 0 q_1 10 \vdash X q_2 0 Y 0 \vdash q_2 X 0 Y 0 \vdash$$
$$X q_0 0 Y 0 \vdash X X q_1 Y 0 \vdash X X Y q_1 0 \vdash X X Y 0 q_1 B$$

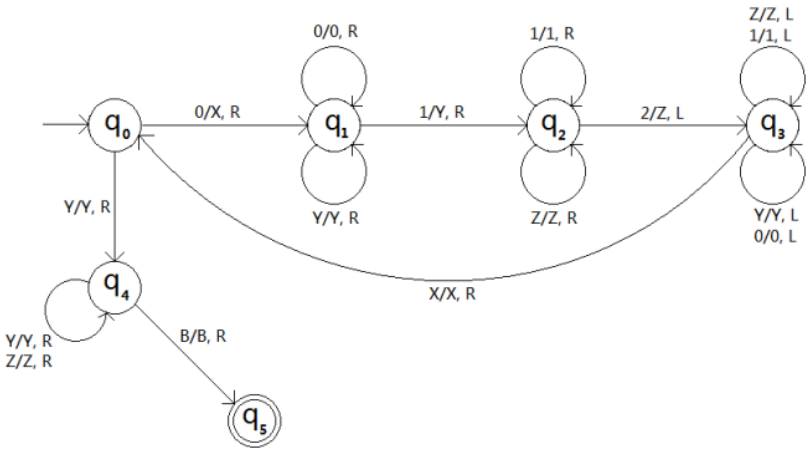
The behavior of M on 0010 resembles the behavior on 0011, until in ID $X X Y q_1 0$ M scans the final 0 for the first time. M must move right, staying in state q_1, which takes it to the ID $X X Y 0 q_1 B$. However, in state q_1 M has no move on tape symbol B; thus M dies and does not accept its input. □

Example 2:

Construct a T.M. to accept the language L= {0^n 1^n 2^n: n>=1}.

Ans:

The Turing machine works similar with the above example: it marks the first 0 as X, the first 1 as Y, the first 2 as Z; then it goes back to the second 0, and marks it as X, the second 2 as Y, the second 2 as Z, and so on.

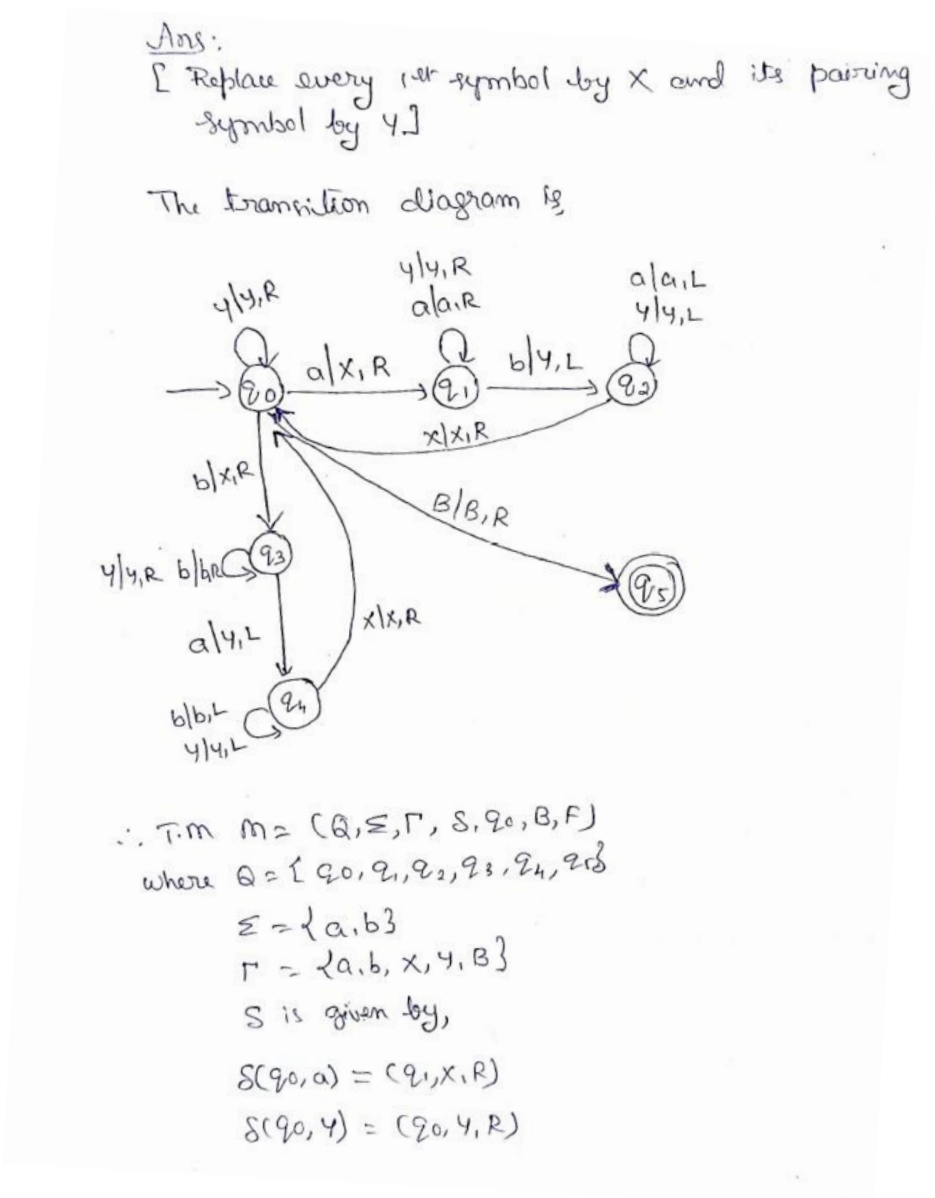


Example 3:

Obtain a TM to accept a string of a's and b's such that $n_a(w)=n_b(w)$.

Ans:

The general procedure: When TM is in initial state q0, first symbol of the input string can be either a or b. Replace the first symbol a (or b) by X and search for next symbol b (or a) and replace it by symbol Y. Repeat this until blank symbol B is scanned.



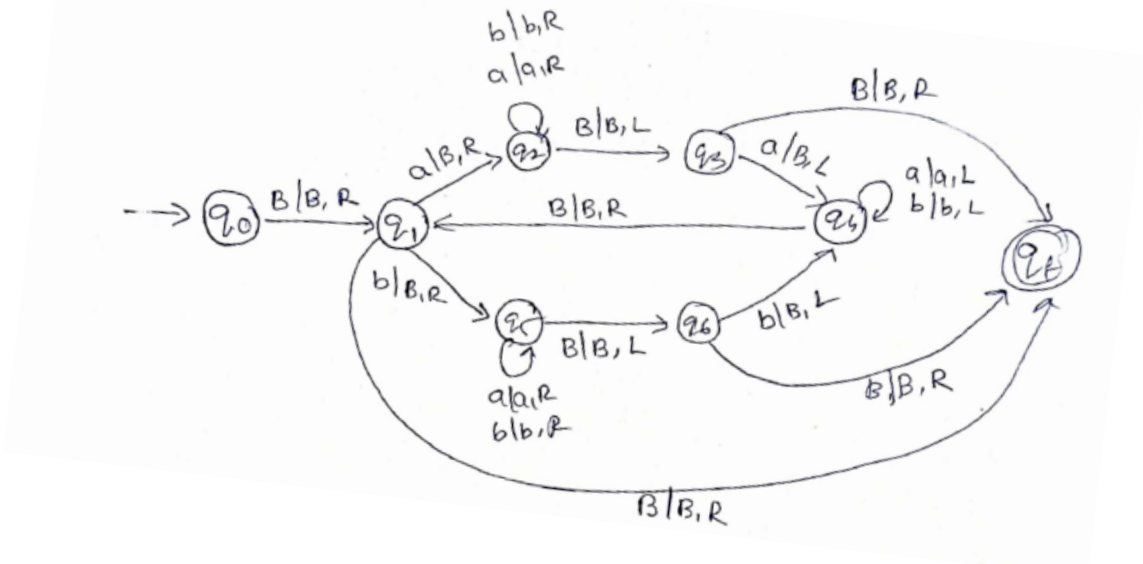
$\delta(q_0, a) = (q_1, a, R)$
$\delta(q_1, Y) = (q_1, Y, R)$
$\delta(q_1, b) = (q_2, Y, L)$
$\delta(q_2, a) = (q_2, a, L)$
$\delta(q_2, Y) = (q_2, Y, L)$
$\delta(q_2, X) = (q_0, X, R)$
$\delta(q_0, b) = (q_3, X, R)$
$\delta(q_3, b) = (q_3, b, R)$
$\delta(q_3, Y) = (q_3, Y, R)$
$\delta(q_3, a) = (q_4, Y, L)$
$\delta(q_4, b) = (q_4, b, L)$
$\delta(q_4, Y) = (q_4, Y, L)$
$\delta(q_4, X) = (q_0, X, R)$
$\delta(q_0, B) = (q_5, B, R)$

q_0 is the initial state
$B \in T$ is the blank symbol.
$F = \{q_5\}$

Example 4: Obtain a TM to accept a palindrome constisting of a's and b's of any length.

Ans: General idea is to place the given string in between blank symbols. For string to be palindrome, its first and last, second and last but one and so oon should match. If we found match, replaced matched symbols by blank symbol B.

The transistion diagram is,



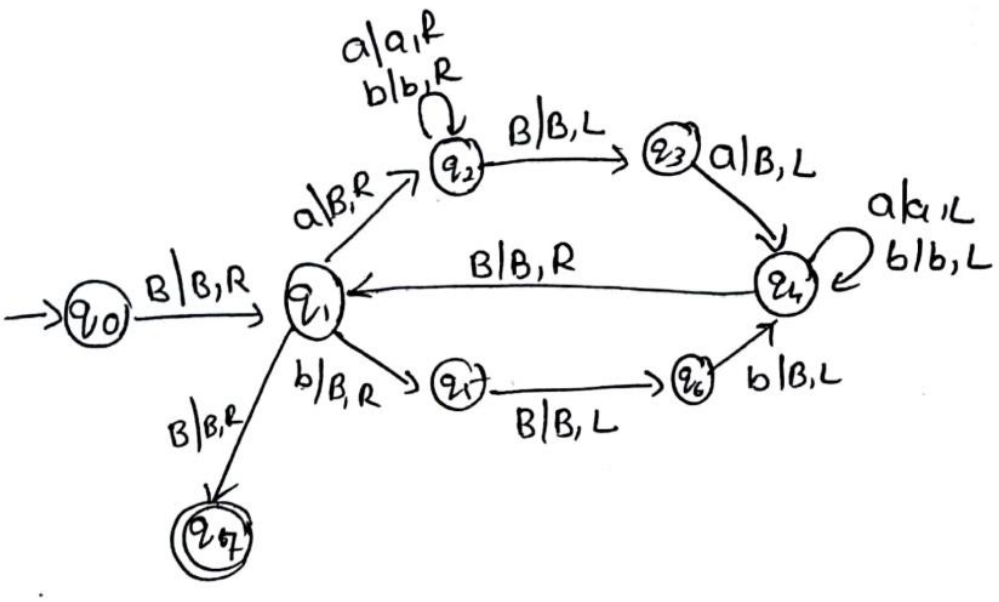
therefore, M=(Q, ∑, Γ,δ, q0, B, F) where,

Q={q0,q1,q2,q3,q4,q5,q6,qf}
∑={a,b}
Γ={a,b,B}
δ is shown in the transition diagram
q0 is the initial state
B is the blank symbol
F={qf}

Example 5: Obtain a TM to accept the language L={ww^R: w∈ (a+b)*}

Ans: ww^R is also a palindrome problem only, but length of the palindrome must be even. Hence TM is same as previous but no transition required from q3 and q6 towards the final state.

Therefore, the transition diagram is,



Example 6: Let x and y are two positive integers. Obtain a turing machine to perform x+y. (Addition operation on TM)

Ans: Let us represent the numbers on the tape using unary representation 1'. Let 0 be the separator.

Ex: x=5 can be represented as 11111 and y=3 can be represented as 111

Then 5+3 can be represented on the tape together with 0 as a separated,

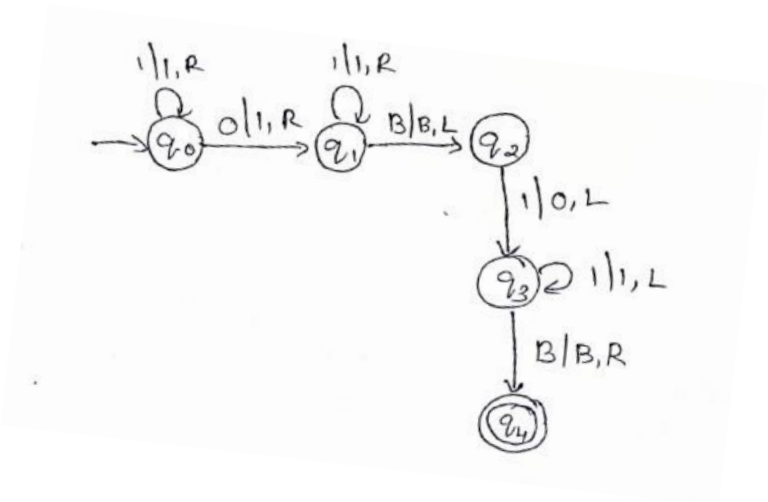
1 1 1 1 1 0 1 1 1

Then output should be of the form:

1 1 1 1 1 1 1 1 0

ic, read all 1's until we encounter the 0. Then change 0 to 1 and keep moving right until blank symbol is encounter. Once found move left

and change 1 to 0 and keep moving right read sum from leftmost blank symbol.
The transition diagram is,



Example 7: Monus operation or propoer subtraction on TM.

Let us,

show how a Turing machine might compute the function $\dot{-}$, which is called *monus* or *proper subtraction* and is defined by $m \dot{-} n = \max(m - n, 0)$. That is, $m \dot{-} n$ is $m - n$ if $m \geq n$ and 0 if $m < n$.

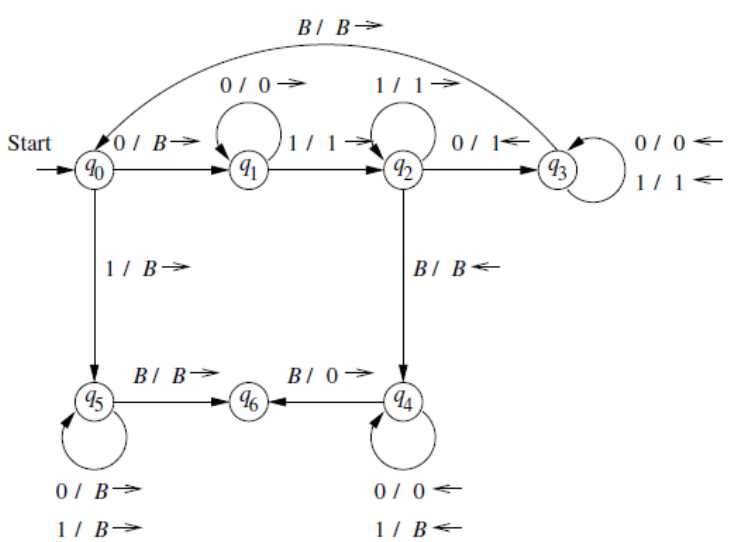
TM, M will start with a tape consisting of 0^m 1 0^n surrounded by blanks.
M halts with 0^{m-n} on its tape, surrounded by blanks.
Ic, for example 5-3 can be represented as,

B000001000B

The result 5-3=2 will be represented on the tape as,

BBBBB00BBBB

The transition diagram is,



Here the symbol→ indicates R(right)
and ← indicates L (left)

The Language of a Turing Machine

In TM the input string is placed on the tape, and the tape head begins at the leftmost input symbol. If the TM eventually enters an accepting state, then the input is accepted, and otherwise not.

More formally, let M= (Q, Σ, Γ, δ, qo, B, F) be a Turing machine. Then L(M) is the set of strings w in ∑* such that $q_0 w \vdash^* \alpha p \beta$ for some state p in F and any tape strings α and β.

The set of languages we can accept using a Turing machine is often called the **recursively enumerable languages or RE** languages.

Turing Machines and Halting Problem:

There is another notion of "acceptance" that is commonly used for Turing machines "acceptance by halting". We say a TM halts if it enters a state q, scanning a tape symbol X, and there is no move in this situation i_e, δ(q,X) is undefined.

DECIDABILITY

- The term **decidability** defined using Turing machines. When a Turing machine reaches a final state, it halts.
- We can also say that a Turing machine M halts when M reaches a state q and a current symbol 'a' to be scanned so that δ(q, a) is undefined.
- There are TMs that never halt on some inputs in any one of these ways. So, we make a distinction between the languages accepted by a TM that halts on all input strings and a TM that never halts on some input strings.

Definition of Recursively Enumerable Language:

"A **language L**⊆Σ* is recursively enumerable if there exists a TM M, such that L= L(M)".

Definition of Recursive Language:

A **language L**⊆Σ* is recursive if there exists some TM M that satisfies the following two conditions.

i) If w ϵ L then M accepts w (that is, reaches an accepting state on processing w) and halts.

ii) If w ∉L then M eventually halts without reaching an accepting state

Definition Decidable: A problem with two answers (Yes/No) is decidable if the corresponding language is recursive. In this case, the language L is also called **decidable**.

Definition Undecidable: A problem/language is **undecidable** if it is not decidable.

Programming Techniques for Turing Machines

The goal is to give a sense of how a Turing machine can be used to compute in a manner not unlike that of a conventional computer. Eventually, we want to convince you that a TM is exactly as powerful as a conventional computer.

8.3.1 Storage in the State

We can use the finite control not only to represent a position in the "program" of the Turing machine, but to hold a finite amount of data. Figure 8.13 suggests this technique (as well as another idea: multiple tracks). There, we see the finite control consisting of not only a "control" state q, but three data elements A, B, and C. The technique requires no extension to the TM model; we merely think of the state as a tuple. In the case of Fig. 8.13, we should think of the state as $[q, A, B, C]$. Regarding states this way allows us to describe transitions in a more systematic way, often making the strategy behind the TM program more transparent.

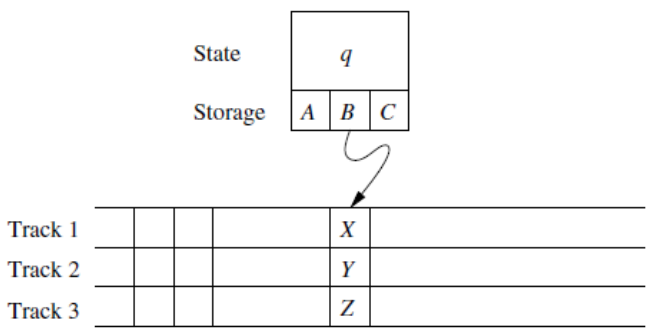


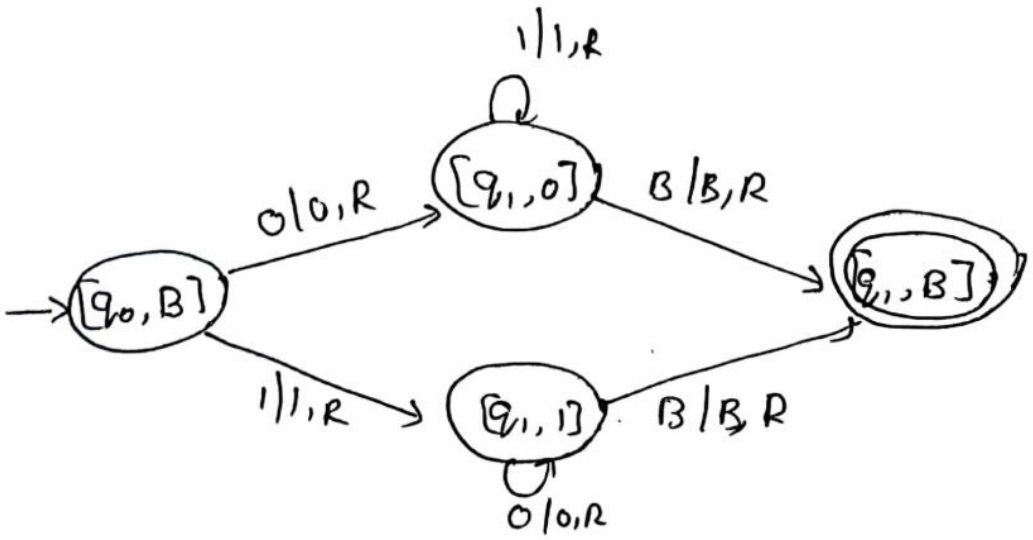
Figure 8.13: A Turing machine viewed as having finite-control storage and multiple tracks

Example 8.6: We shall design a TM

$$M = (Q, \{0,1\}, \{0,1,B\}, \delta, [q_0, B], B, \{[q_1, B]\})$$

that remembers in its finite control the first symbol (0 or 1) that it sees, and checks that it does not appear elsewhere on its input. Thus, M accepts the language $\mathbf{01}^*+\mathbf{10}^*$. Accepting regular languages such as this one does not stress the ability of Turing machines, but it will serve as a simple demonstration.

The transition diagram is,



Multiple Tracks

Another useful "trick" is to think of the tape of a Turing machine as composed of several tracks. Each track can hold one symbol, and the tape alphabet of the TM consists of tuples, with one component for each "track". Thus, for instance, the cell scanned by the tape head in Fig. 8.13 contains the symbol [X, Y, Z]. Like the technique of storage in the finite control, using multiple tracks does not extend what the Turing machine can do.

Extensions to the Basic Turing Machine:

a) Multi-tape Turing Machines

(Q. Write a short note on Multitape TM. (06M))

A multi-tape TM is as shown below. The device has a finite control "state", and some finite number of tapes. Each tape is divided into cells, and each cell can hold any symbol of the finite tape alphabet.

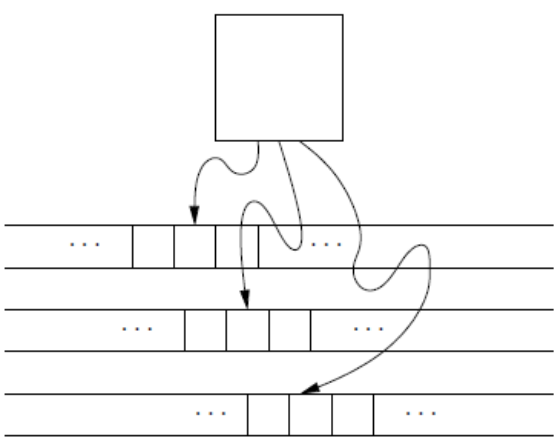


Figure 8.16: A multitape Turing machine

As in the single tape TM, the set of tape symbols includes a blank, and has a subset called the input symbols, of which the blank is not a

member. The set of states includes an initial state and some accepting states. Initially,

- The input, a finite sequence of input symbols, is placed on the first tape.
- All other cells of all the tapes hold the blank.
- The finite control is in the initial state.
- The head of the first tape is at the left end of the input
- All other tape heads are at some arbitrary cell. Since tapes other than the first tape are completely blank, it does not matter where the head is placed initially; all cells of these tapes, "look" the same.

A move of the multitape TM depends on the state and the symbol scanned by each of the tape heads. In one move, the multitape TM does the following:

- The control enters a new state, which could be the same as the previous state.
- On each tape, a new tape symbol is written on the cell scanned. Any of these symbols may be the same as the symbol previously there.
- Each of the tape heads makes a move, which can be either left, right, or stationary. The heads move independently, so different heads may move in different directions, and some may not move at all.

Equivalence of one-tape TM and Multi-tape TM:

(Q. Prove that every language accepted by a multi-tape TM is recursively enumerable.5M)

Theorem 8.9: Every language accepted by a multitape TM is recursively enumerable.

PROOF: The proof is suggested by Fig. 8.17. Suppose language L is accepted by a k-tape TM M. We simulate M with a one-tape TM N whose tape we think of as having $2k$ tracks. Half these tracks hold the tapes of M, and the other half of the tracks each hold only a single marker that indicates where the head for the corresponding tape of M is currently located. Figure 8.17 assumes $k = 2$. The second and fourth tracks hold the contents of the first and second tapes of M, track 1 holds the position of the head of tape 1, and track 3 holds the position of the second tape head.

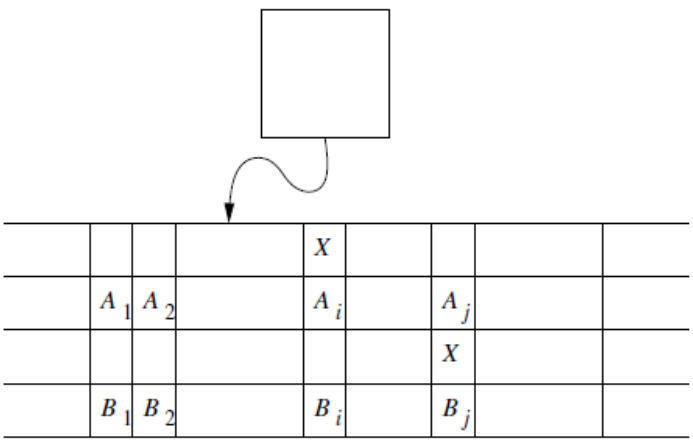


Figure 8.17: Simulation of a two-tape Turing machine by a one-tape Turing machine

To simulate a move of M, N's head must visit the k head markers. So that N not get lost, it must remember how many head markers are to its left at all times; that count is stored as a component of N's finite control. After visiting each head marker and storing the scanned symbol in a component of its finite control, N knows what tape symbols are being scanned by each of M's heads. N also knows the state of M, which it stores in N's own finite control. Thus, N knows what move M will make.

N now revisits each of the head markers on its tape, changes the symbol in the track representing the corresponding tapes of M, and moves the head markers left or right, if necessary. Finally, N changes the state of M as recorded in its own finite control. At this point, N has simulated one move of M.

We select as N's accepting states all those states that record M's state as one of the accepting states of M. Thus, whenever the simulated M accepts, N also accepts, and N does not accept otherwise. □

b) NONDETERMINISTIC TURING MACHINES
(Q. Write a short note on Nondeterministic TM)

In the case of standard Turing δ *(q1, a)* was defined for some elements of Q x Γ as an element of Q x Γ x {L, *R}*. Now we extend the definition of δ. In a nondeterministic TM, δ *(q1, a)* is defined as a subset of Q x Γ x {L *R}*.

Definition: A nondeterministic Turing machine is a 7-tuple (Q, Σ, Γ, δ, qo, B, F), where,

- Q is a finite nonempty set of states.
- Γ is a finite nonempty set of tape symbols,
- B ∈ Γ is the blank.

- Σ is a nonempty set of input symbols and is a subset of Γ and B ∉ Σ.
- qo ∈Q is the initial state, and
- F ⊆ Q is the set of final states.
- δ is a partial function from Q x Γ into the power set of Q x Γ x *{L. R}*.

Undecidability

We then divide problems that can be solved by a Turing machine into two: classes, those that have an algorithm ic, a Turing machine that halts whether or not it accepts its inputs, and those that are only solved by Turing machines that may run forever on inputs they do not accept.

A Language That Is Not Recursively Enumerable:

- A Language L is recursively enumerable (RE) if L=L(M) for some TM M. A language is recursively enumerable if there exists a Turing machine that will enumerate all the strings in the language or accept strings in the language but may not halt for strings not in the language.
- A language *L* is recursive iff exists a Turing machine *M* such that
  i) $L = L(M)$ and
  ii) *M* always halt (even if it does not accept).

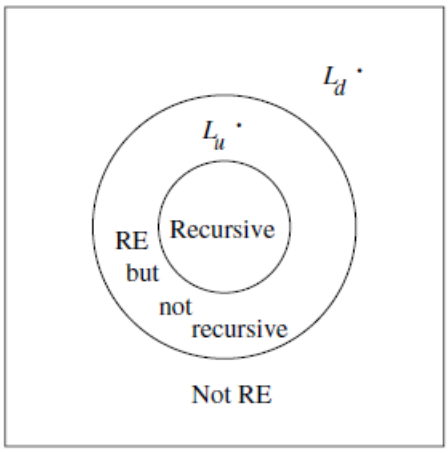
Definition: A language *L* is **undecidable** iff *L* is not recursive.

Note: The term "recursive" as synonym for "decidable" comes from Mathematics (prior to computers).

Non-Recursively Enumerable (Non-RE) Languages
Definition: A language is non-recursively enumerable if no Turing machine exists that can enumerate or accept all strings in the language. In other words, it is not possible to construct an algorithm to semi-decide membership.

- These are the most complex and "unsolvable" problems.
- The language and its complement are both non-recursively enumerable.

Following figure suggests the relationship among three classes of languages:



1. The recursive languages
2. The languages that are recursively enumerable but not recursive.
3. The non-recursively-enumerable(non-RE) languages.

   Recursive ⊆ RE ⊆ Non-RE.

Undecidability often involves proving that certain problems cannot be solved by any algorithm (or equivalently, any Turing machine).

- To analyze such problems, we need a formal way to represent them as inputs to Turing machines. Binary coding provides a way to encode:
  - Strings in a language.
  - Descriptions of Turing machines.
  - Complex decision problems.

Example: The Halting Problem

- The Halting Problem asks whether a Turing machine halts on a given input.
- To analyze it, both the Turing machine (its states and transitions) and the input string are encoded into a single binary string. This encoding allows us to reason about whether a universal Turing machine (UTM) can solve the problem.

Codes for Turing Machines:

Let us devise a binary code for Turing machines so that each TM with input alphabet {0, 1} may be thought of as a binary string.

Code for Turing Machine

To represent a TM, $M = (Q, \{0,1\}, \Gamma, \delta, q_1, B, F)$ as a binary string, we need to assign integers to the states, tape symbols and directions.

- Assume $Q = \{q_1, q_2, \ldots, q_k\}$ are states for some k. Let q_1 is the initial state. (we will not use q_0, as numbering starts from 1).
- Assume $\Gamma = \{X_1, X_2, \ldots X_m\}$ for some m.

  Let $X_1 \Rightarrow 0$
  $X_2 \Rightarrow 1$
  $X_3 \Rightarrow B$
- Assume directions of tape-head as,
  $D_1 \Rightarrow L$
  $D_2 \Rightarrow R$

Suppose T.M's transition rule is,
$\delta(q_i, X_j) = (q_k, X_l, D_m)$ for some integers i, j, k, l & m.

The code is : $\boxed{0^i 1 0^j 1 0^k 1 0^l 1 0^m}$ here 1 acts as separators.

Then the code for entire TM is:
$C_1 \,11\, C_2 \,11\, \ldots 11\, C_n$

Example :

Let $M = (\{q_1, q_2, q_3\}, \{0,1\}, \{0,1,B\}, \delta, q_1, B, \{q_2\})$
(Q, Σ, Γ, F)

where $\delta = \{ \delta(q_1, 1) = (q_3, 0, R)$
$\delta(q_3, 0) = (q_1, 1, R)$
$\delta(q_3, 1) = (q_2, 0, R)$
$\delta(q_3, B) = (q_3, 1, L) \}$

Let us convert individual δ into binary code as follows:

- $\delta(q_1, 1) = (q_3, 0, R) \Rightarrow \delta(q_1, X_2) = (q_3, X_1, D_2)$
  $\because X_2 \Rightarrow 1$
  $D_2 \Rightarrow R$
  $X_1 \Rightarrow 0$
  $X_3 \Rightarrow B$

$\therefore$ code $\boxed{C_1 = 0^1 1 0^2 1 0^3 1 0^1 1 0^2}$

- $\delta(q_3, 0) = (q_1, 1, R) \Rightarrow \delta(q_3, X_1) = (q_1, X_2, D_2)$

  $\therefore$ code $C_2 = 0^3 1 0^1 1 0^1 1 0^2 1 0^2$

- $\delta(q_3, 1) = (q_2, 0, R) \Rightarrow \delta(q_3, X_2) = (q_2, X_1, D_2)$

  $\therefore$ code $C_3 = 0^3 1 0^2 1 0^2 1 0^1 1 0^2$

- $\delta(q_3, B) = (q_3, 1, L) \Rightarrow \delta(q_3, X_3) = (q_3, X_2, D_1)$

  $\therefore$ code $C_4 = 0^3 1 0^3 1 0^3 1 0^2 1 0^1$

$\therefore$ The code for entire T.M is,

$C_1 11 C_2 11 C_3 11 C_4$

i.e. 0100100010100111000101010100100111
0001001001010011000100010001000010

Complements of Recursive and RE languages:

(Q. Prove that complement of recursive language is recursive)

Theorem: If L is a recursive language, so is $\overline{L}$

PROOF: Let $L = L(M)$ for some TM M that always halts. We construct a TM $\overline{M}$ such that $\overline{L} = L(\overline{M})$ by the construction suggested in Fig. 9.3. That is, $\overline{M}$ behaves just like M. However, M is modified as follows to create $\overline{M}$:

1. The accepting states of M are made nonaccepting states of $\overline{M}$ with no transitions; i.e., in these states $\overline{M}$ will halt without accepting.
2. $\overline{M}$ has a new accepting state r; there are no transitions from r.
3. For each combination of a nonaccepting state of M and a tape symbol of M such that M has no transition (i.e., M halts without accepting), add a transition to the accepting state r.

Since M is guaranteed to halt, we know that $\overline{M}$ is also guaranteed to halt. Moreover, $\overline{M}$ accepts exactly those strings that M does not accept. Thus $\overline{M}$ accepts $\overline{L}$. □

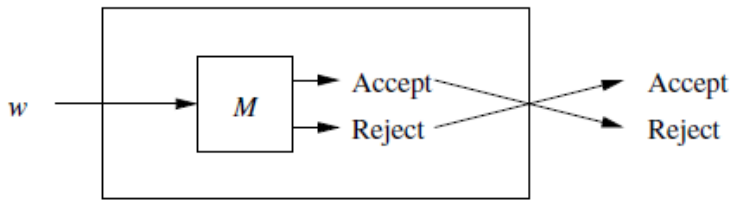


Figure 9.3: Construction of a TM accepting the complement of a recursive language

Q. Prove that complement of recursively enumerable language is recursive.

Theorem 9.4: If both a language L and its complement are RE, then L is recursive. Note that then by Theorem 9.3, $\overline{L}$ is recursive as well.

PROOF: The proof is suggested by Fig. 9.4. Let $L = L(M_1)$ and $\overline{L} = L(M_2)$. Both M_1 and M_2 are simulated in parallel by a TM M. We can make M a two-tape TM, and then convert it to a one-tape TM, to make the simulation easy and obvious. One tape of M simulates the tape of M_1, while the other tape of M simulates the tape of M_2. The states of M_1 and M_2 are each components of the state of M.

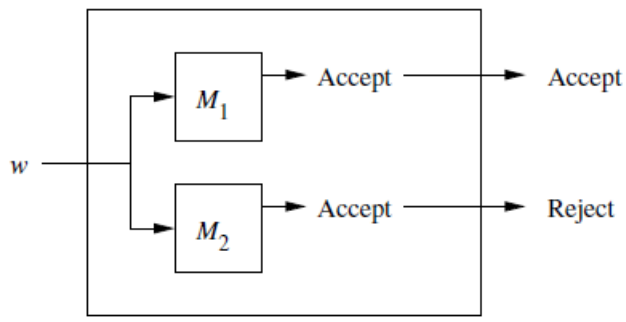


Figure 9.4: Simulation of two TM's accepting a language and its complement

If input w to M is in L, then M_1 will eventually accept. If so, M accepts and halts. If w is not in L, then it is in $\overline{L}$, so M_2 will eventually accept. When M_2 accepts, M halts without accepting. Thus, on all inputs, M halts, and $L(M)$ is exactly L. Since M always halts, and $L(M) = L$, we conclude that L is recursive. □

The Universal Language:

A Turing machine could be used to simulate a computer that had been loaded with an arbitrary program. That is to say, a single TM can be used as a "stored program computer", taking its program as well as its data from one or more tapes on which input is placed.

We define L_u, the universal language, to be the set of binary strings that encode, a pair (M,w), where M is a TM with, the binary input alphabet, and w is a string in (0+1)* such that w is in L(M).

That is, L_u is the set of strings representing a TM and an input accepted by that TM. We shall show that there is a TM U, often called the **universal Turing machine**, such that L_u=L(U).

It is easiest to describe U as a multi-tape Turing machine. In the case of U, the transitions of M are stored initially on the first tape, along with the string w. A second tape will be used to hold the simulated tape of M, using the same format as for the code of M. That is, tape symbol Xi of M will be represented by 0^i and tape symbols will be separated by single 1's. The third tape of U holds the state of M, with state q_i represented by i 0's. A sketch of U is in Fig. 9.5

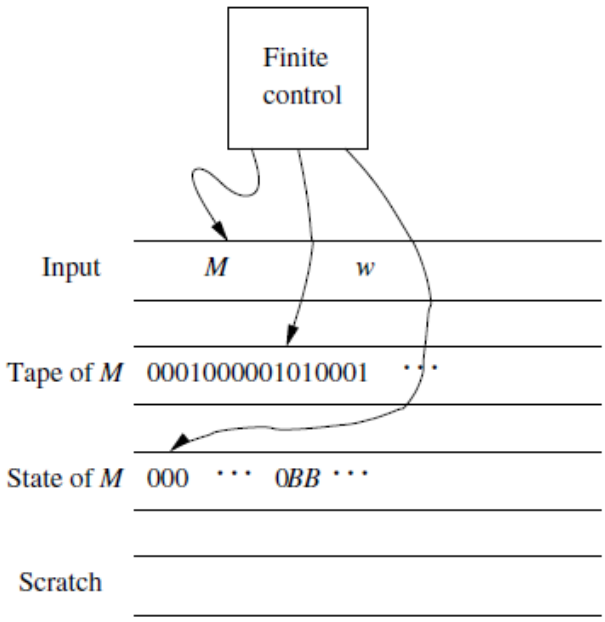


Figure 9.5: Organization of a universal Turing machine

The operation of U can be summarized as follows:

1. Examine the input to make sure that the code for M is a legitimate code for some TM. If not, U halts without accepting. Since invalid codes are assumed to represent the TM with no moves, and such a TM accepts no inputs, this action is correct.

2. Initialize the second tape to contain the input w, in its encoded form. That is, for each 0 of w, place 10 on the second tape, and for each 1 of w, place 100 there. Note that the blanks on the simulated tape of M, which are represented by 1000, will not actually appear on that tape; all cells beyond those used for w will hold the blank of U. However, U knows that, should it look for a simulated symbol of M and find its own blank, it must replace that blank by the sequence 1000 to simulate the blank of M.

3. Place 0, the start state of M, on the third tape, and move the head of U's second tape to the first simulated cell.

4. To simulate a move of M, U searches on its first tape for a transition $0^i10^j10^k10^l10^m$, such that 0^i is the state on tape 3, and 0^j is the tape

symbol of M that begins at the position on tape 2 scanned by U. This transition is the one M would next make. U should:

(a) Change the contents of tape 3 to 0^k; that is, simulate the state change of M. To do so, U first changes all the 0's on tape 3 to blanks, and then copies 0^k from tape 1 to tape 3.

(b) Replace 0^j on tape 2 by 0^l; that is, change the tape symbol of M. If more or less space is needed (i.e., $i \neq l$), use the scratch tape and the shifting-over technique of Section 8.6.2 to manage the spacing.

(c) Move the head on tape 2 to the position of the next 1 to the left or right, respectively, depending on whether $m = 1$ (move left) or $m = 2$ (move right). Thus, U simulates the move of M to the left or to the right.

5. If M has no transition that matches the simulated state and tape symbol, then in (4), no transition will be found. Thus, M halts in the simulated configuration, and U must do likewise.

6. If M enters its accepting state, then U accepts.

In this manner, U simulates M on w. U accepts the coded pair (M, w) if and only if M accepts w.

****