Design a Turing machine to add two given integers.

Solution:

Assume that \( m \) and \( n \) are positive integers. Let us represent the input as \( 0^m B 0^n \).

If the separating \( B \) is removed and \( 0 \)'s come together we have the required output, \( m + n \) is unary.

(i) The separating \( B \) is replaced by a 0.
(ii) The rightmost 0 is erased i.e., replaced by \( B \).

Let us define \( M = (\{q_0, q_1, q_2, q_3, q_4\}, \{0\}, \{0, B\}, \delta, q_0, \{q_4\}) \). \( \delta \) is defined by Table shown below.

<table>
<thead>
<tr>
<th>State</th>
<th>Tape Symbol</th>
<th>0</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>( q_0 )</td>
<td>((q_0, 0, R))</td>
<td>((q_1, 0, R))</td>
<td></td>
</tr>
<tr>
<td>( q_1 )</td>
<td>((q_1, 0, R))</td>
<td>((q_2, B, L))</td>
<td></td>
</tr>
<tr>
<td>( q_2 )</td>
<td>((q_3, B, L))</td>
<td>—</td>
<td></td>
</tr>
<tr>
<td>( q_3 )</td>
<td>((q_3, 0, L))</td>
<td>((q_4, B, R))</td>
<td></td>
</tr>
</tbody>
</table>

\( M \) starts from ID \( q_0 0^m B 0^n \), moves right until seeking the blank \( B \). \( M \) changes state to \( q_1 \). On reaching the right end, it reverts, replaces the rightmost 0 by \( B \). It moves left until it reaches the beginning of the input string. It halts at the final state \( q_4 \).

Some unsolvable Problems are as follows:
(i) Does a given Turing machine \( M \) halts on all input?
(ii) Does Turing machine \( M \) halt for any input?
(iii) Is the language \( L(M) \) finite?
(iv) Does \( L(M) \) contain a string of length \( k \), for some given \( k \)?
(v) Do two Turing machines \( M_1 \) and \( M_2 \) accept the same language?

It is very obvious that if there is no algorithm that decides, for an arbitrary given Turing machine \( M \) and input string \( w \), whether or not \( M \) accepts \( w \). These problems for which no algorithms exist are called “UNDECIDABLE” or “UNSOLVABLE”.

Code for Turing Machine:
Our next goal is to devise a binary code for Turing machines so that each TM with input alphabet \( \{0, 1\} \) may be thought of as a binary string. Since we just saw how to enumerate the binary strings, we shall then have an identification of the Turing machines with the integers, and we can talk about "the \( i \)th Turing machine, \( M_i \)." To represent a TM \( M = (Q, \{0, 1\}, \Gamma, \delta, q_1, B, F) \) as a binary string, we must first assign integers to the states, tape symbols, and directions \( L \) and \( R \).

- We shall assume the states are \( q_1, q_2, \ldots, q_r \) for some \( r \). The start state will always be \( q_1 \), and \( q_2 \) will be the only accepting state. Note that, since we may assume the TM halts whenever it enters an accepting state, there is never any need for more than one accepting state.

- We shall assume the tape symbols are \( X_1, X_2, \ldots, X_s \) for some \( s \). \( X_1 \) always will be the symbol 0, \( X_2 \) will be 1, and \( X_3 \) will be \( B \), the blank. However, other tape symbols can be assigned to the remaining integers arbitrarily.

- We shall refer to direction \( L \) as \( D_1 \) and direction \( R \) as \( D_2 \).

Since each TM \( M \) can have integers assigned to its states and tape symbols in many different orders, there will be more than one encoding of the typical TM. However, that fact is unimportant in what follows, since we shall show that no encoding can represent a TM \( M \) such that \( L(M) = L_d \).

Once we have established an integer to represent each state, symbol, and direction, we can encode the transition function \( \delta \). Suppose one transition rule is \( \delta(q_i, X_j) = (q_k, X_l, D_m) \), for some integers \( i, j, k, l, \) and \( m \). We shall code this rule by the string \( 0^i10^j10^k10^l10^m \). Notice that, since all of \( i, j, k, l, \) and \( m \) are at least one, there are no occurrences of two or more consecutive 1's within the code for a single transition.

A code for the entire TM \( M \) consists of all the codes for the transitions, in some order, separated by pairs of 1's:

\[ C_11C_211 \cdots C_{n-1}11C_n \]

where each of the \( C \)'s is the code for one transition of \( M \).
Diagonalization language:

- The language \( L_d \), the **diagonalization language**, is the set of strings \( w \) such that \( w \) is not in \( L(M_i) \).

That is, \( L_d \) consists of all strings \( w \) such that the TM \( M \) whose code is \( w \) does not accept when given \( w \) as input.

The reason \( L_d \) is called a "diagonalization" language can be seen if we consider Fig. 9.1. This table tells for all \( i \) and \( j \), whether the TM \( M_i \) accepts input string \( w_j \); 1 means "yes it does" and 0 means "no it doesn't." We may think of the \( i \)th row as the **characteristic vector** for the language \( L(M_i) \); that is, the 1's in this row indicate the strings that are members of this language.

\[
\begin{array}{cccccc}
 & 1 & 2 & 3 & 4 & \cdots \\
1 & 0 & 1 & 1 & 0 & \cdots \\
2 & 1 & 1 & 0 & 0 & \cdots \\
3 & 0 & 0 & 1 & 1 & \cdots \\
4 & 0 & 1 & 0 & 1 & \cdots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \ddots \\
\end{array}
\]

Diagonal

This table represents language acceptable by Turing machine

The diagonal values tell whether \( M_i \) accepts \( w_i \). To construct \( L_d \), we complement the diagonal. For instance, if Fig. 9.1 were the correct table, then the complemented diagonal would begin 1, 0, 0, 0, \ldots. Thus, \( L_d \) would contain \( w_1 = \epsilon \), not contain \( w_2 \) through \( w_4 \), which are 0, 1, and 00, and so on.

The trick of complementing the diagonal to construct the characteristic vector of a language that cannot be the language that appears in any row, is called **diagonalization**. It works because the complement of the diagonal is
Proof that $L_d$ is not recursively enumerable:

**Theorem 9.2:** $L_d$ is not a recursively enumerable language. That is, there is no Turing machine that accepts $L_d$.

**Proof:** Suppose $L_d$ were $L(M)$ for some TM $M$. Since $L_d$ is a language over alphabet $\{0, 1\}$, $M$ would be in the list of Turing machines we have constructed, since it includes all TM's with input alphabet $\{0, 1\}$. Thus, there is at least one code for $M$, say $i$; that is, $M = M_i$.

Now, ask if $w_i$ is in $L_d$.

- If $w_i$ is in $L_d$, then $M_i$ accepts $w_i$. But then, by definition of $L_d$, $w_i$ is not in $L_d$, because $L_d$ contains only those $w_j$ such that $M_j$ does not accept $w_j$.

- Similarly, if $w_i$ is not in $L_d$, then $M_i$ does not accept $w_i$, Thus, by definition of $L_d$, $w_i$ is in $L_d$.

Since $w_i$ can neither be in $L_d$ nor fail to be in $L_d$, we conclude that there is a contradiction of our assumption that $M$ exists. That is, $L_d$ is not a recursively enumerable language. \(\Box\)

Recursive Languages:

We call a language $L$ **recursive** if $L = L(M)$ for some Turing machine $M$ such that:

1. If $w$ is in $L$, then $M$ accepts (and therefore halts).

2. If $w$ is not in $L$, then $M$ eventually halts, although it never enters an accepting state.

A TM of this type corresponds to our informal notion of an “algorithm,” a well-defined sequence of steps that always finishes and produces an answer. If we think of the language $L$ as a “problem,” as will be the case frequently, then problem $L$ is called **decidable** if it is a recursive language, and it is called **undecidable** if it is not a recursive language.
**Theorem 9.3**: 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 $\overline{L}$. \(\square\)
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 $M$.

![Diagram of two TMs accepting a language and its complement](image)

**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. \(\square\)

Universal Language:

We define $L_u$, the *universal language*, to be the set of binary strings that encode, in the notation of Section 9.1.2, 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 $L_u = L(U)$. Since the input to $U$ is a binary string, $U$ is in fact some $M_i$ in the list of binary-input Turing machines we developed in
Undecidability of Universal Language:

**Theorem 9.6**: $L_u$ is RE but not recursive.

**Proof**: We just proved in Section 9.2.3 that $L_u$ is RE. Suppose $L_u$ were recursive. Then by Theorem 9.3, $\overline{L_u}$, the complement of $L_u$, would also be recursive. However, if we have a TM $M$ to accept $\overline{L_u}$, then we can construct a TM to accept $L_d$ (by a method explained below). Since we already know that $L_d$ is not RE, we have a contradiction of our assumption that $L_u$ is recursive.

![Diagram](image)

**Figure 9.6: Reduction of $L_d$ to $\overline{L_u}$**

Suppose $L(M) = \overline{L_u}$. As suggested by Fig. 9.6, we can modify TM $M$ into a TM $M'$ that accepts $L_d$ as follows.

1. Given string $w$ on its input, $M'$ changes the input to $w111w$. You may, as an exercise, write a TM program to do this step on a single tape. However, an easy argument that it can be done is to use a second tape to copy $w$, and then convert the two-tape TM to a one-tape TM.

2. $M'$ simulates $M$ on the new input. If $w$ is $w_i$ in our enumeration, then $M'$ determines whether $M_i$ accepts $w_i$. Since $M$ accepts $\overline{L_u}$, it will accept if and only if $M_i$ does not accept $w_i$; i.e., $w_i$ is in $L_d$.

Thus, $M'$ accepts $w$ if and only if $w$ is in $L_d$. Since we know $M'$ cannot exist by Theorem 9.2, we conclude that $L_u$ is not recursive. $\square$
Class p-problem solvable in polynomial time:

A Turing machine $M$ is said to be of time complexity $T(n)$ [or to have “running time $T(n)$”] if whenever $M$ is given an input $w$ of length $n$, $M$ halts after making at most $T(n)$ moves, regardless of whether or not $M$ accepts. This definition applies to any function $T(n)$, such as $T(n) = 50n^2$ or $T(n) = 3^n + 5n^4$; we shall be interested predominantly in the case where $T(n)$ is a polynomial in $n$. We say a language $L$ is in class $\mathcal{P}$ if there is some polynomial $T(n)$ such that $L = L(M)$ for some deterministic TM $M$ of time complexity $T(n)$.

Non deterministic polynomial time:

A nondeterministic TM that never makes more than $p(n)$ moves in any sequence of choices for some polynomial $p$ is said to be non polynomial time NTM.

- NP is the set of languages that are accepted by polynomial time NTM’s
- Many problems are in NP but appear not to be in $\mathcal{P}$
- One of the great mathematical questions of our age: is there anything in NP that is not in $\mathcal{P}$?

NP-complete problems:

If we cannot resolve the “$p=\text{np}$” question, we can at least demonstrate that certain problems in NP are the hardest, in the sense that if any one of them were in $\mathcal{P}$, then $\mathcal{P}=\text{NP}$.

- These are called NP-complete.
- Intellectual leverage: Each NP-complete problem’s apparent difficulty reinforces the belief that they are all hard.

Methods for proving NP-Complete problems:

- Polynomial time reduction (PTR): Take time that is some polynomial in the input size to convert instances of one problem to instances of another.
- If $P_1$ PTR to $P_2$ and $P_2$ is in $\mathcal{P}$ then so is $P_1$.
- Start by showing every problem in NP has a PTR to Satisfiability of Boolean formula.
- Then, more problems can be proven NP complete by showing that SAT PTRs to them directly or indirectly.