There are some "programming techniques" when designing Turing machines.

1. **Storing a tape symbol in the finite control**. We can build a TM whose states are pairs $[q, X]$ where q is an original state, and X is a tape symbol. The first component is used as before, the second component can be used in remembering a particular tape symbol.

There are some "programming techniques" when designing Turing machines.

1. **Storing a tape symbol in the finite control**. We can build a TM whose states are pairs $[q, X]$ where q is an original state, and X is a tape symbol. The first component is used as before, the second component can be used in remembering a particular tape symbol.

We used this technique in the sample Turing machine that recognizes palindromes: The first letter of the input was stored in the state (state q_a or q_b), and the machine moved to the end of the input and verified that the letter there was identical to the first letter.

Example. A TM that recognizes the language

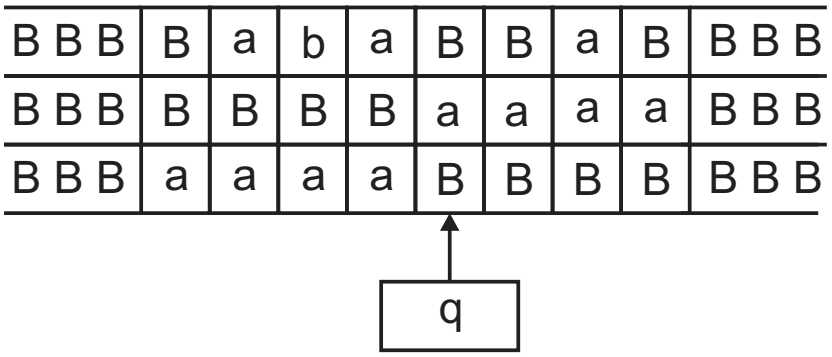
$$L = ab^* + ba^*$$

The machine reads the first symbol, remembers it in the finite control, and checks that the same symbol does not appear anywhere else in the input word.

$$\begin{aligned}
\delta(q_0, a) &= ([q, a], a, R) \\
\delta(q_0, b) &= ([q, b], b, R) \\
\delta([q, a], b) &= ([q, a], b, R) \\
\delta([q, b], a) &= ([q, b], a, R) \\
\delta([q, a], B) &= (q_F, B, R) \\
\delta([q, b], B) &= (q_F, B, R)
\end{aligned}$$

(Compare this with a finite automaton that recognizes L.)

2. **Multiple tracks**. Sometimes it is useful to imagine that the tape consists of multiple "tracks". We can store different intermediate information on different tracks:



The tape alphabet is then

$$\Gamma_1 \times \Gamma_2 \times \cdots \times \Gamma_n$$

where Γ_i is the alphabet of symbols on the i'th track.

3. **Checking off symbols**. This is equivalent to having a second track where we can place blank B or symbol $\surd$. The tick mark can be conveniently used in remembering which letters of the input have been already processed. It is useful in recognizing languages where we have to count letters, for example.

3. **Checking off symbols**. This is equivalent to having a second track where we can place blank B or symbol $\surd$. The tick mark can be conveniently used in remembering which letters of the input have been already processed. It is useful in recognizing languages where we have to count letters, for example.

Example. Let

$$L = \{ww \mid w \in (a+b)^*\}.$$

We first use the tick mark to find the center of the input word: Mark alternatively the first and last unmarked letters, one-by-one. The last letter to be marked is in the center. So we know where the second w should start.

Using the "Storage in the finite control" technique, check that the letters in the first half and the second half are identical.

Example. A sketch of a TM with a **three track** tape that recognizes

$$L = \{a^p \mid p \text{ is a prime number } \}$$

Initially the input is written on the first track and the other two tracks contain B's. (This means we identify a with $[a, B, B]$ and B with $[B, B, B]$.)

The machine operates as follows. It starts by placing two a's on the second track. Then it repeats the following:

1. Copy the content of the first track to the third track.
2. Subtract the number on the second track from the third track as many times as possible. If the third track becomes empty, halt in a non-accepting state. (The number on the first track was divisible by the number on the second track.)
3. Increase the number on the second track by one. If the number becomes the same as the number on the first track halt in an accepting state. Else go back to step 1.

4. **Shifting over**. This means adding an new cell at the current location of the tape. This is done by shifting all symbols one position to the right by scanning the tape from the current position to the right, remembering the content of the previous cell in the finite control, and writing it to the next cell on the right. Once the rightmost non-blank symbol is reached the machine can return to the new vacant cell that was introduced.

(In order to recognize the rightmost non-blank symbol, it is convenient to introduce an end-of-tape symbol that is written in the first cell after the last non-blank symbol.)

5. **Subroutines**. We can use subroutines in TM in the same way as they are used in normal programming languages. A subroutine uses its own set of states, including its own "initial state" q_0' and a return state q_r.

To call a subroutine, the calling TM simply changes the state to q_0', and makes sure the read-write head is positioned on the leftmost symbol of the "input" to the subroutine.

Constructing TM to perform specific tasks can be quite complicated. Even to recognize some simple languages may require many states and complicated constructions. However, TM are powerful enough to be able to simulate any computer program.

The claim that Turing machines can compute everything that is computable using any model of computation is known as **Church's thesis**. Since the thesis talks about *any* model of computation, it can never be proved. But there is strong evidence supporting the thesis: So far, TM have been able to simulate any other model of computing anyone has invented (λ-calculus, Markov-algorithms, Post systems, recursive functions etc.)

Example. Let us see how a Turing machine would simulate a conventional computer architecture. The tape contains all data the computer has in its memory.

The data can be organized for example in such a way that the word v_i in the memory location i of the computer is stored on the tape as

$$\#0^i * v_i\#$$

The contents of the registers of the CPU are stored on their own tracks on the tape.

To execute the next instruction, the TM finds the memory location addressed by the **Program Counter** register. In order to do that it goes through all memory locations one by one and – using the tick marks – counts if the address i is the same as the content of the Program Counter register. When it finds the correct memory location i, it reads the **instruction** v_i and memorizes it in the finite control. (The computer has only finitely many different instructions.)

To each instruction corresponds its own **subroutine**. To simulate the instruction, the TM can use the same tick marking to find any required memory locations, and then execute the particular task. The task may be adding the content of a memory location to some register, for example.

Adding two numbers can be easily implemented (especially if we represent all number in the unary format so that number n is represented as the word a^n).

To write a word to the memory may require shifting all cells on the right hand side of the memory location, but we know how to do that.

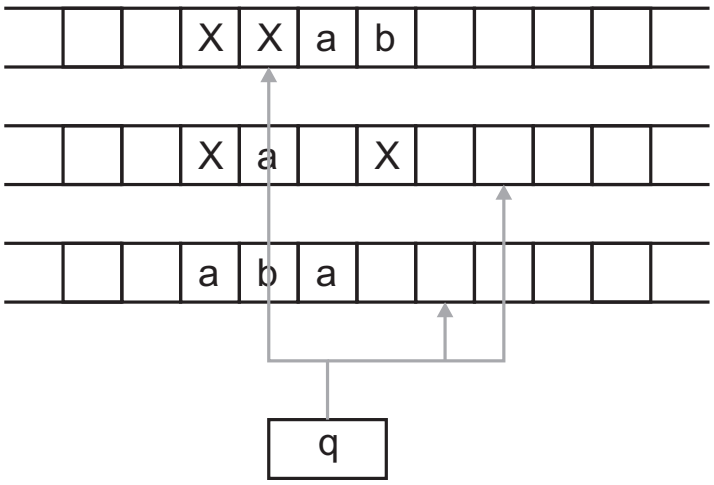
Modifications of Turing machines

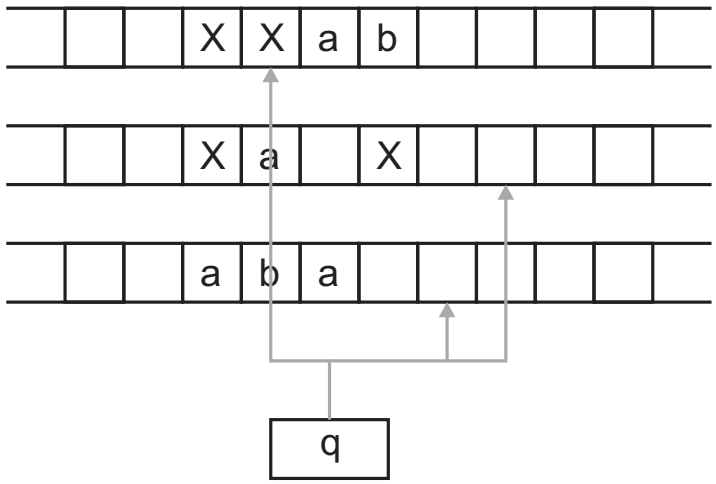
The following modifications do **not** make Turing machines more powerful. The modified TMs accept exactly the same family of r.e. languages as our basic model.

Modifications of Turing machines

The following modifications do **not** make Turing machines more powerful. The modified TMs accept exactly the same family of r.e. languages as our basic model.

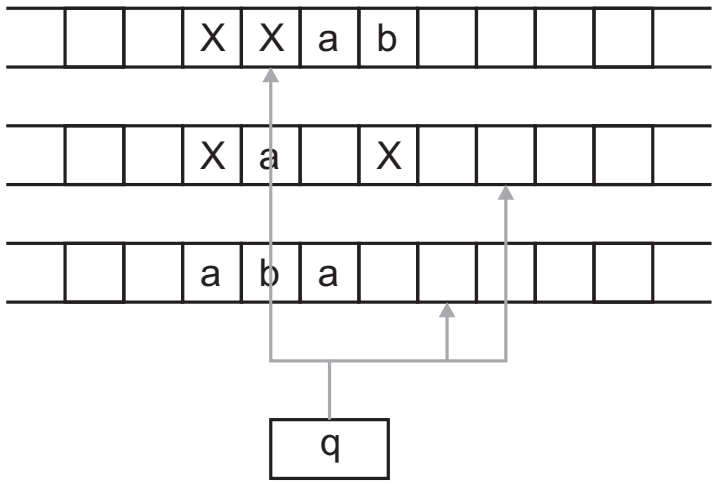
1. **Multiple tapes**. We can allow the TM to have more than one tape. Each tape has its own independent R/W head. This is different from one tape TM with multiple tracks, since the R/W heads of different tapes can now be at different positions.





Depending on the state of the finite control and the current tape symbols on all tapes the machine can

- change the state,
- overwrite the currently scanned symbols on all tapes, and
- move each R/W head to left or right independently of each other.



Formally, the transition function δ is now a (partial) function from

$$(Q \setminus \{f\}) \times \Gamma^n$$

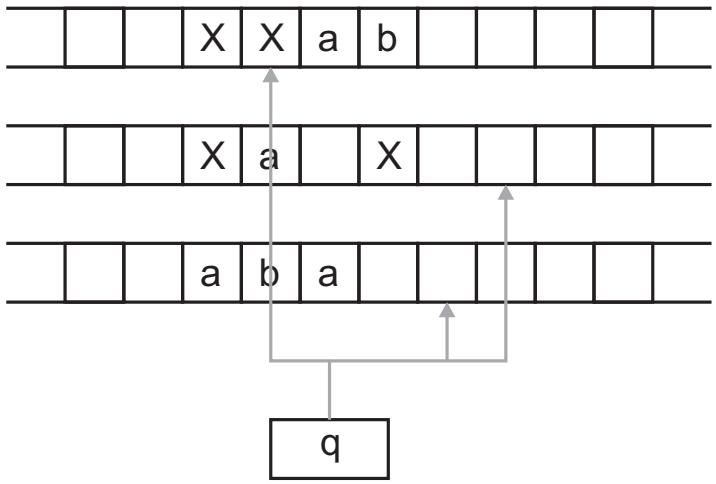
to

$$Q \times \Gamma^n \times \{L, R\}^n$$

where n is the number of tapes. A transition

$$\delta(q, X_1, X_2, \ldots, X_n) = (p, Y_1, Y_2, \ldots, Y_n, d_1, d_2, \ldots, d_n),$$

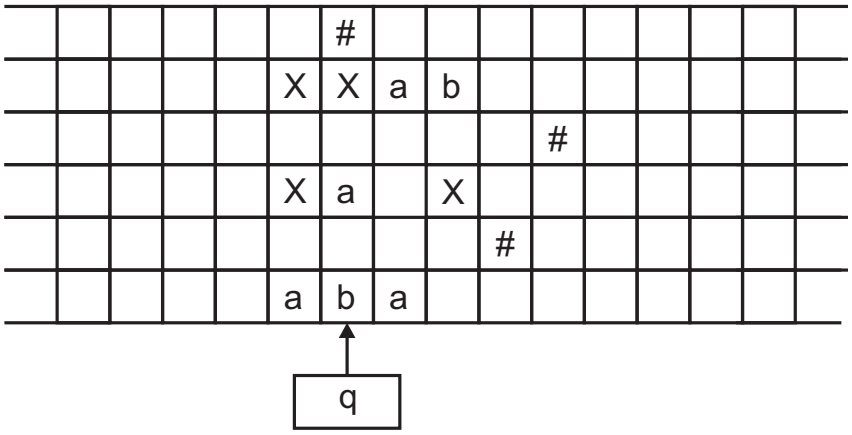
means that the machine, in state q, reading symbols X_1, $X_2, \ldots X_n$ from the n tapes, changes its state to p, writes symbols Y_1,$Y_2, \ldots Y_n$ on the tapes, and moves the first, second, third, etc. R/W head to directions indicated by d_1, $d_2, \ldots d_n$, respectively.



Initially, the input is written on tape number one, and all other tapes are blank. A word is accepted iff the machine enters a final state.

Simulating a multitape TM using a single tape.

The single tape will have two tracks for every tape of M: One track is identical to M's corresponding tape; The other one contains a single # indicating the position of the R/W head on that tape.



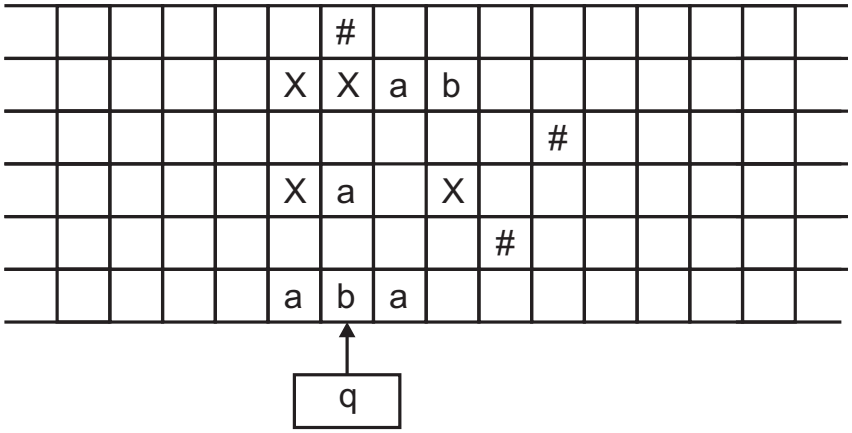
To simulate one move of the multitape machine M, we scan the tape from left to right, remembering in the finite control the tape symbols below #'s.

Once we have encountered all #'s, the machine can figure out the new state p and the action taken on each tape. During another sweep over the tape, the machine can execute the instruction by writing the required symbols and moving the #'s on the tape left or right.

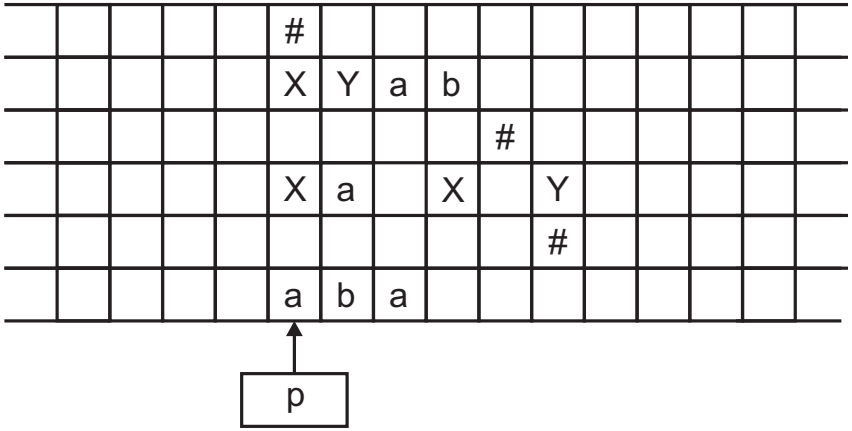
Example. If

$$\delta(q, X, B, B) = (p, Y, Y, B, L, L, R)$$

the simulating machine will turn



into



by a simulation loop that sweeps back-and-forth between the extremal #'s.

Note that simulating one step of the multitape machine requires scanning back-and-forth over the input, so the one-tape machine will be much slower. But all that matters is that the machines accept exactly same words.

3. **Nondeterministic Turing machines**. We can also introduce nondeterminism. Now δ is a function from

$$(Q \setminus \{f\}) \times \Gamma$$

to **subsets** of

$$Q \times \Gamma \times \{L, R\}.$$

When scanning tape symbol X in state q, the machine can execute any instruction from the set $\delta(q, X)$. If $\delta(q, X)$ is empty, the machine halts.

A word w is accepted if and only if there exists a computation that takes the initial ID

$$q_0 \; w$$

into a ID where the state is a final state.

(Note that there may be other computations that halt in a non-final state, or do not halt at all, but the word is accepted as long as there exists at least one accepting computation.)

Example. Recognizing the language

$$L = \{ww \mid w \in (a+b)^*\}$$

is made easier using non-determinism. We do not need to find the center point of the input word first: we may guess where the center point is. If in the end all letters have been erased we know that the guess was correct and the word is in the language.

Example. It is easy to construct a non-deterministic TM that recognizes the language

$$\{w\#u \mid w, u \in (a+b)^* \text{ and } w \text{ is a subword of } u \}.$$

The machine first makes a non-deterministic guess where w starts inside u. Then it verifies the correctness of the guess by erasing symbols in w and u, one by one. If the process ends successfully and the whole w gets erased, the word is in the language!

Simulating a non-deterministic TM M by a deterministic one.

Let r be the maximum size of the sets $\delta(q, X)$ for any q and X. In other words, there are always at most r possible choices in the nondeterministic machine.

The deterministic TM uses three tapes (which we know can be converted into a one-tape version.)

- Tape 1 contains the input, and that tape never changes.
- On tape 2 we generate words over the alphabet

  $$\{1, 2, \ldots, r\}$$

  in some predetermined order.

  (For example, start with the shortest words, and move up to longer and longer words. Words of equal length are generated in the lexicographical order. In other words, count integers 1,2,3,... in base r.)

  We use the word generated on the second tape to make a selection among non-deterministic choices for each move.
- For each word generated on tape 2 we simulate M on tape 3 using the content on tape 2 to select among possible non-deterministic choices.

More precisely: Let x be the word generated on tape 2. We copy the input from tape 1 to tape 3, and simulate the nondeterministic TM on tape 3.

At the i'th simulation step we use the i'th letter of x to choose which nondeterministic alternative to use.

In any of the following cases we give up, generate the next word on the second tape, and repeat the process:

- If the next r-ary digit on the second tape is greater than the number of choices in the current ID of the machine we are simulating, or
- If we consumed all letters from the second tape.

If on the other hand the machine halts in a final state, the simulating machine accepts the word.

The machine we constructed is deterministic. It is also equivalent to the original nondeterministic machine M:

$\Longrightarrow$ If the input w is accepted by M, there exists a sequence of choices that leads to an accepting calculation. Sooner or later that sequence of choices gets generated on tape 2. Once that happens, we get an accepting computation on tape 3, and the word w is accepted.

$\Longleftarrow$ If $w \notin L(M)$, there does not exist a sequence of choices that leads to a final state. Therefore, our deterministic machine never halts: it keeps generating longer and longer words on tape 2 never halting.

The deterministic TM in effect tries all possible non-deterministic choices one after the other until (if ever) it finds an accepting computation.

Note that the simulating deterministic machine M' is **much slower** than the original non-deterministic machine M. If M accepts word w in n moves, it may take over r^n moves by M' to accept the same word.

It is a celebrated open problem whether the deterministic machine needs to be so much slower.

Note that the simulating deterministic machine M' is **much slower** than the original non-deterministic machine M. If M accepts word w in n moves, it may take over r^n moves by M' to accept the same word.

It is a celebrated open problem whether the deterministic machine needs to be so much slower.

Denote by **P** the family of languages that are recognized by some deterministic TM in polynomial time. In other words, L is in P if there exists a Turing machine M and a polynomial p such that M recognizes L, and for every $w \in L$ the accepting computation of w by M uses at most $p(|w|)$ moves.

Analogously, let us denote by **NP** the family of languages that are recognized by some non-deterministic TM in polynomial time. (That means, for every $w \in L$ there exists an accepting computation for w that uses at most $p(|w|)$ moves.)

It is a famous open problem whether

$$\mathbf{P} \stackrel{?}{=} \mathbf{NP}$$

i.e., whether there are languages that are recognized non-deterministically in polynomial time, but recognizing them deterministically requires super-polynomial time.

It is generally assumed that

$$P \neq NP$$

but no proof is known.

This problem has enormous practical importance because there are many important computational problems that are known to be in NP, but nobody has been able to find an efficient deterministic algorithm for them.

Closure properties

Theorem.

(1) The family of recursive languages is effectively closed under complementation. (If L is recursive, so is its complement $\Sigma^* \setminus L$.)

(2) The families of recursive and recursively enumerable languages are effectively closed under union and intersection. (If L_1 and L_2 are recursive or r.e., so are $L_1 \cup L_2$ and $L_1 \cap L_2$.)

(3) A language $L \subseteq \Sigma^*$ is recursive if and only if both L and its complement $\Sigma^* \setminus L$ are recursively enumerable.

We'll have negative results later (after we learn how to prove that a language is not recursive or r.e.) In particular: the family of r.e. language is **not** closed under complementation.

Proof.

(1) Recursive languages are closed under complementation.

Let L be a recursive language. This means it is recognized by some Turing machine

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

that halts on every input.

(1) Recursive languages are closed under complementation.

Let L be a recursive language. This means it is recognized by some Turing machine

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

that halts on every input.

Let us build a Turing machine

$$M' = (Q \cup \{f'\}, \Sigma, \Gamma, \delta', q_0, B, f')$$

that recognizes the complement of L:

M' is identical to M except that M' has a new final state f', and whenever $\delta(q, X)$ is not defined for $q \neq f$ we add the transition $\delta'(q, X) = (f', X, R)$.

- If $w \in L(M)$ then both M and M' halt in state f. That state is not the final state of M', so M' does not accept w, and $w \notin L(M')$.
- If $w \notin L(M)$ then M halts in a non-final state q. Therefore M' continues one more move and enters its final state f'. So $w \in L(M')$.

Clearly M' halts on every input.

(2) Recursive and r.e. families are closed under union and intersection.

Let L_1 and L_2 be recognized by TMs M_1 and M_2, respectively. In the recursive cases we also assume the corresponding TM halts on all inputs.

(2) Recursive and r.e. families are closed under union and intersection.

Let L_1 and L_2 be recognized by TMs M_1 and M_2, respectively. In the recursive cases we also assume the corresponding TM halts on all inputs.

Idea 1. We can construct a TM M that executes M_1 and M_2 **sequentially** on the same input w.

- **Intersection**: M accepts iff both M_1 and M_2 accept.
- **Union**: M accepts iff M_1 or M_2 accepts.

This works well for the intersections, and for the union of recursive languages.

However, this sequential approach does not work for the union of r.e. languages: It could be that M_1 does not halt with the input w while M_2 accepts w. Machine M should accept w, but the execution of M_1 never halts so M never gets into executing M_2.

(2) Recursive and r.e. families are closed under union and intersection.

Let L_1 and L_2 be recognized by TMs M_1 and M_2, respectively. In the recursive cases we also assume the corresponding TM halts on all inputs.

Idea 2. Execute M_1 and M_2 **in parallel** on the same input w. We can easily do this using two tapes: execute M_1 on tape 1 and M_2 on tape 2.

More precisely, M

- Copies the input w from tape 1 to tape 2,
- Using states in $Q_1 \times Q_2$ simulates simultaneously step-by-step TM M_1 on tape 1 and M_2 on tape 2.

More precisely, M

- Copies the input w from tape 1 to tape 2,
- Using states in $Q_1 \times Q_2$ simulates simultaneously step-by-step TM M_1 on tape 1 and M_2 on tape 2.

In the case of the union operation:

- As soon as either simulation enters its final state, M enters its final state and the input is accepted.
- If one of the simulations halts in a non-final state, that simulation enters and maintains a "wait" state while the other simulation continues.
- Once both simulations reach "wait" state then M halts in a non-final state.

Now it is clear that M accepts w iff M_1 or M_2 accepts w. Moreover, if M_1 and M_2 halt on all inputs then M has this same property.

We know how to convert the two-tape TM into a single tape TM. (And this conversion preserves the property of halting on all inputs.)

In the case of the intersection operation:

- If one of the simulations enter its final state then that simulation maintains that state while the other simulation continues.
- If one of the simulations halts in a non-final state, that simulation enters and maintains a "wait" state while the other simulation continues.
- Once both simulations halt: if both reached the final state then M enters its final state and the input is accepted. Otherwise, M halts in a non-final state.

Now it is clear that M accepts w iff M_1 and M_2 accept w. Moreover, if M_1 and M_2 halt on all inputs then M has this same property.

(3) A language $L \subseteq \Sigma^*$ is recursive if and only if both L and its complement $\Sigma^* \setminus L$ are recursively enumerable.

(3) A language $L \subseteq \Sigma^*$ is recursive if and only if both L and its complement $\Sigma^* \setminus L$ are recursively enumerable.

$\Longrightarrow$ If L is recursive then by (1) its complement $\Sigma^* \setminus L$ is also recursive. Every recursive language is also recursively enumerable.

(3) A language $L \subseteq \Sigma^*$ is recursive if and only if both L and its complement $\Sigma^* \setminus L$ are recursively enumerable.

$\Longleftarrow$ Assume that L is recognized by TM M_1, and that its complement $\Sigma^* \setminus L$ is recognized by TM M_2. We may assume that the machines do not halt unless they reach their final state. (Introduce a new infinite loop state for the cases when the machines halt in non-final states.)

As in (2) above, we construct a TM M that executes M_1 and M_2 in parallel on the same input w using two tapes. Note that precisely one of the two simulations will halt. If M_1 halts then M enters the final state, if M_2 halts then M halts in a non-final state.

Clearly M halts on every input, and it accepts w if and only if M_1 accepts w.