So, w_j can not belong to L_d, but it also cannot not belong to L_d. This is obviously impossible. Therefore, the assumption that there exists a Turing Machine M_j that recognises L_d leads us to contradiction. Which means that this assumption must be false and no Turing Machine from our sequence of Turing Machines can accept L_d. Since this sequence contains all the Turing Machines over the input alphabet $\{0, 1\}$, this means that no Turing Machine can recognise L_d. This means that L_d is not recognisable (or partially decidable). With this, we have proven our first crucial result about computability, which is summarised in the following theorem.

Theorem 12.1: Language that is not Partially Decidable/Recognisable

Let
$$L_d = \{w_i | M_i \text{ does not accept } w_i\},$$
where w_i is the i-th string in the ordering of the strings over $\{0, 1\}$ and M_i is the Turing Machine with the encoding w_i. Then L_d is not partially decidable.

12.4 Universal Turing Machine and A Language That is Not Totally Decidable

In the previous section, we have found a language that is not partially decidable or recognisable. This means that the class of languages that are not recognisable is not empty. We also know that the class of languages that are totally decidable is not empty - there are many many languages for which Turing Machines exist that halt on every input and that accept exactly that language. The only class of the languages that we have considered, and for which we still haven't concluded whether it is empty or not is the class of paritally decidable but not totally decidable languages. As a reminder, a language is in this class if there exists a Turing Machine that recognises it, but no Turing Machine that recognises it halts on all inputs. We know certainly that every totally decidable language is also partially decidable, but we don't know whether there are any partially decidable languages that are not totally decidable. Therefore, we don't know whether the class of partially decidable and the class of totally decidable languages are really different. In this section, we are going to prove that they, indeed, are different.

Our strategy for achieving that goal will be to first construct a Universal Turing Machine, which will be a Turing Machine that accepts arbitrary other Turing Machine and a string, and simulates the operation of that Turing Machine on that string. We are then going to consider the Universal Language, the language of all encodings $\langle M, w\rangle$ of a Turing Machine M and a string w, such that M accepts w. We are going to show that the Universal Turing Machine accepts the Universal Language, therefore that language is partially decidable. Then we are going to take a slight detour and use the concept of the Universal Turing Machine to provide simple proofs of some important properties of partially decidable and totally decidable languages. Finally, we are going to use some of these properties to show that no Turing Machine can decide the Universal Language, which is to say that no Turing Machine can halt on all inputs and accept exactly the strings from the Universal Language. Thus, the Universal Language is partially decidable but not totally decidable.

In most of the literature, the languages that are not totally decidable are called *undecidable*. We will also sometimes use this terminology. Note that we do not make an assumption that the undecidable language is even partially decidable. All we know is that it is not totally decidable. Also, if language L is accepted by Turing Machine M that halts on all inputs, we will say that M *decides* L.

12.4.1 Universal Turing Machine and Universal Language

As we said above, the Universal Turing Machine is a regular Turing Machine that is able to simulate an arbitrary Turing Machine M over the language $\{0, 1\}$ on an arbitrary string over $\{0, 1\}$. We can assume that M has only three tape symbols, 0, 1 and B, that the starting state is q_1 and the only accepting state is q_2. We have seen in Section 12.2 how such a Turing Machine can be encoded as a binary string. If we concatenate an arbitrary string w to the encoding of M, we get a binary encoding of a Turing Machine M and a string w. We will denote this encoding by $\langle M, w\rangle$. We can then construct the Universal Turing Machine U that will run the Turing Machine M on input string w.

For simplicity, we will assume that U has four tapes (Figure 12.1). We have seen that adding additional tapes does not increase the power of the model - exactly the same class of languages is accepted by single-tape Turing Machines as with multi-tape ones. The first tape holds the input $\langle M, w\rangle$. The second tape simulates the tape of M. The third tape holds the state of M. The fourth tape is an auxiliary scratch tape, used to help with simulation. The fourth tape is not strictly necessary, but makes our life a bit easier. At start, the second tape will contain the string w, and the third tape will contain the string 0, encoding the state q_1 of M. This is shown in Figure 12.1. The encoding of M is given at the start of the Input Tape, with 111 at the start of it and 111 at the end of it, followed by the string $w = 01011$. The tape simulating the tape of M contains just w, while the third tape contains the encoding of the start state of M, which is q_1, encoded by 0. The scratch tape contains just B's.

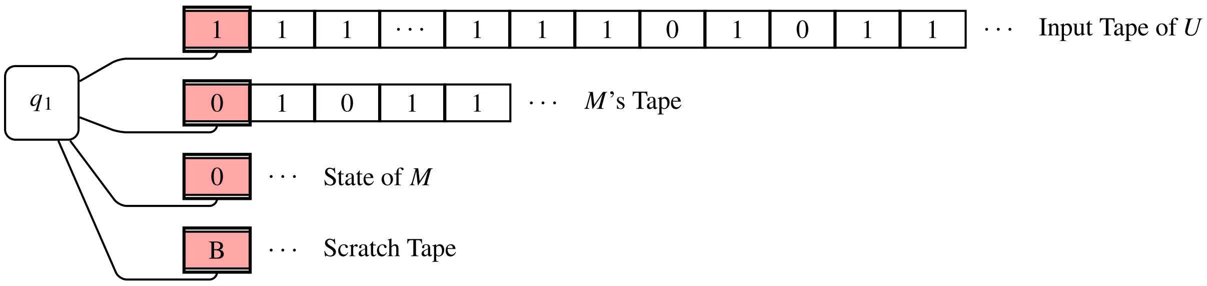

Figure 12.1: Universal Turing Machine U

Given encoding of $\langle M, w\rangle$ as an input, U operates in the following way. First, it scans its input tape to make sure that it contains a valid encoding of a Turing Machine that has moves. That is, it has to make sure that it starts with 111, which is followed by a sequence of the strings of type $0^i10^j10^k10^l10^m$ (encoding the move $\delta(q_i, X_j) = (q_k, X_l, D_m)$ of M), and that no two such entries exist where i and j are the same. The last check makes sure that the encoding is of a deterministic Turing Machine. U can use the scratch tape for performing this check. If the Input Tape does not contain encoding of a Turing Machine with moves, U halts immediately and the input $\langle M, w\rangle$ is rejected. If the Input Tape contains encoding of a Turing Machine with moves, U initialises the M's Tape and the tape for State of M to the appropriate inital values, and then simulates the computation of M on w. This can be done by copying the encoding of the current state of M to the scratch tape, adding 1 and then adding to that the encoding of the symbol currently scanned in the M's Tape. This gives, for our example, the string 010 on the scratch tape. U can then go over all the transitions of M on its Input Tape, and locate the encoding of transition that start with the string on its scratch tape, as this is the transition for the current state of M and the currently scanned tape symbol of M. If the required transition of M does not exist, it means that M should halt, so U also halts in a non-accepting state and the string $\langle M, w\rangle$ is rejected. If the required transition of M exists, U then scans the rest of th encoding of this transition, determining the symbol to overwrite the currently scanned symbol on M's tape with, direction to which to move the head of that tape and the new state. The encoding of the new state is written on the State of M tape. Then U simulates the next move of M, This process continues until either there is no next move for M, in which case $\langle M, w\rangle$ is rejected, or until 00 is written on the State of M tape of U, which signalises that M enters the accepting state. This makes U halt too and accept the input $\langle M, w\rangle$.

It is easy to see that U accepts the string $\langle M, w\rangle$ exactly when M accepts w. The language accepted by the Universal Turing Machine is called the *Universal Language* and denoted by L_u.

Definition 12.1: Universal Language L_u

Universal Language, denoted by L_u is the language comprising all the encodings of a Turing Machine M with the input alphabet $\{0, 1\}$ and a string w over the input alphabet $\{0, 1\}$ such that M accepts w.

$$L_u = \{\langle M, w\rangle | M \text{ accepts } w\}.$$

12.4.2 Properties of Partially Decidable and Totally Decidable Languages

Construction of the Univeral Turing Machine U represents one of the most important milestones in the Theory of Computation. Up until now, we have only looked at Turing Machines for specific tasks - for example, for recognising specific languages such as $\{0^n1^n | n \geq 0\}$ or for adding two integers. These Turing Machines are akin to pocket calculators or arcade machines. They execute a fixed program on a provided input, and cannot be used for any other purpose. A pocket calculator accepts as an input an arithmetic expressions and returns the value of that expression. Different pocket calculators support different arithmetic operations, but in principle they are all the same. They cannot be used for any other purpose than calculating the value of an arithmetic expression. On the other hand, we see personal computers as much more general devices, the devices that in themselves do not do anything useful, and the sole purpose of which is to run (almost) arbitrary programs, such as word processors, editors, compilers, instant messangers, web browsers and games. We would be far less excited about computers if they were useful for only one particular task (as it was the case in the dawn of the age of computing).

All that changes with the Universal Turing Machine. The Universal Turing Machine is able to simulate an arbitrary other Turing Machine (albeit, in the form that we introduced it, restricted to the input alphabet $\{0, 1\}$) on an arbitrary input. Therefore, we can see the Universal Turing Machine as a computer that accepts the encodings of programs and inputs to them, and executes these programs on these inputs. This is where it becomes more obvious why Turing Machines are good models for computers, as opposed to, for example, Finite Automata. It is not possible to design a finite automaton that will read the encoding of an arbitrary other finite automaton and an input string, simulate that finite automaton on that string and accept if the simulated finite automaton accepts. This idea of a computer accepting different programs and executing them seems obvious to us, but it was quite a revolutionary concept at a time and transformed the use of computers completely.

Coming back to our theory, the Universal Turing Machine shows that we can build a Turing Machine to simulate another Turing Machine. This has a great impact, as it allows us to consider a concept of *reducibility*. For our purposes here, we will look at reducibility in terms of reducing a problem of whether an aribtrary string w_1 belongs to the language L_1 to the problem of whether some string w_2 to which w_1 can be transformed belongs to some other language L_2. Let us denote a problem of whether an arbitrary string w_1 belongs to the language L_1 as P_1, and the problem of whether an arbitrary string w_2 belongs to the language L_2 as P_2. Reducing P_1 to P_2 means giving a method to transform every string w_1 to some string w_2 such that w_1 belongs to L_1 exactly when w_2 belongs to L_2. If P_1 can be reduced to P_2, this means that P_2 is at least as hard as P_1, because P_2 is a more general problem. It also means that if we know how to solve P_2, we also know how to solve P_1. Suppose, for example that P_1 is determining whether a string w_1 is of the form 0^n1^n and that P_2 is determining whether a string w_2 has the same number of 0's and 1's. If we can solve P_2, then we can also solve P_1 by merely testing whether w_1 is of the form $\mathbf{0}^*\mathbf{1}^*$ and then using the method for solving P_2 to check whether w has the same number of 0's and 1's.

In terms of Turing Machines, if deciding whether a string w_1 belongs to the language L_1 can be reduced to deciding whether a string w_2 belongs to the language L_2, and if there exists a Turing Machine M_2 that decides on latter problem (that is, the Turing Machine that halts in an accepting state for every string in L_2 and halts in a non-accepting state for every string not in L_2), then we can construct a Turing Machine M_1 that will first transform its input w_1 to w_2, then simulate M_2 on w_2 and, based on the result of this, decide whether to accept w_1 or not. In other words, if L_2 is (totally) decidable, then L_1 is also (totally) decidable. On the other hand, if we know that L_1 is not totally decidable, then L_2 cannot be totally decidable either, because if it was, we could have used the described procedure to build a Turing Machine that will decide L_1, which we know is not possible.

We can now prove some important results.

Lemma 12.2: $\overline{L}$ Is Decidable if L is Decidable

If L is totally decidable, then the complement of L, $\overline{L}$, is totally decidable too.

Proof. Suppose that L is totally decidable. Then there exists a Turing Machine M that will accept every string from L and reject every string not in L. We can then build a Turing Machine M' that will, for its input string w, simulate M on w. If M accepts w, M' rejects w and vice versa - if M rejects w, then M' accepts w. We know that M halts on every string w, so we also know that M' will halt on every input string. Therefore, M' halts on all inputs and it accepts exactly the strings that M rejects, which are all strings from $\overline{L}$. Therefore, $\overline{L}$ is totally decidable. This construction is summarised in Figure 12.2. □

Note that we cannot say the same for partially decidable languages. If language L is partially decidable, we cannot say whether $\overline{L}$ is partially decidable.

Applying this result to the diagonal language L_d, for which we know that it isn't even partially decidable, tells us that $\overline{L_d}$ is not toally decidable. If $\overline{L_d}$ was totally decidable, then L_d as its complement would also be totally decidable, which we know is not true. Therefore, the following corollary holds.

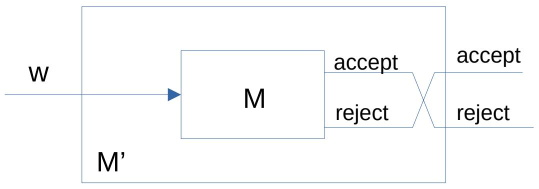

Figure 12.2: Turing Machine M' that decides on $\overline{L}$.

Corollary 12.3

$\overline{L_d} = \{w_i | M_i \text{ accepts } w_i\}$ is not totally decidable.

Lemma 12.4

If both L and $\overline{L}$ are partially decidable, then L is totally decidable.

Proof. Let both L and $\overline{L}$ be partially decidable. Then there is a Turing Machine M that accepts L and Turing Machine M' that accepts $\overline{L}$. Note that we don't know what happens with the strings not in L when M is simulated on them - M might or might not halt on them. The same holds for the strings in L and M'. However, we know that for any string, either M or M' (but not both) will halt in an accepting state, because every string belongs to either L or $\overline{L}$ and L and $\overline{L}$ do not have strings in common. Therefore, if we build a Turing Machine M'' that simulates M and M' in parallel on an input string w, and halts as soon as either M or M' halts in an accepting state, we know that M'' will halt for any w. If M halts on w in an accepting state, then M'' accepts w. If M' halts on w in an accepting state, then M'' rejects w. Therefore, M'' halts for any input string, and accepts only the strings from L. Therefore, L is totally decidable. This construction is summarised in Figure 12.3. □

Note that in the construction above, we cannot simulate first M and then M' on a string w, because we don't know whether M will ever halt. Thus, the simulation needs to proceed in parallel. This can be achieved easily if we generate a sequence of pairs $(1,1),(2,1),(1,2),(2,2),(1,3),(3,3)\ dots$ and then for each pair $(1,i)$ simulate M on w for i steps, and for every par $(2,i)$ simulate M' on w for i steps. Since either M or M' halts on w in an accepting state, for some pair (k,l) one of M or M' will halt in an accepting state, and therefore M'' will halt too.

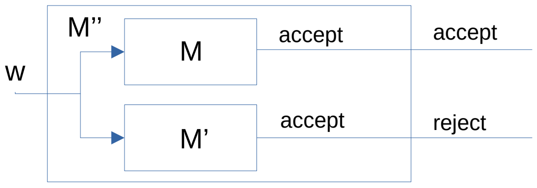

Figure 12.3: Turing Machine M'' that decides on L.

This lemma has an important consequence, in that if we know that a language L is partially decidable, but not totally decidable, we know that $\overline{L}$ cannot be partially decidable, because if it was, then L would also be totally decidable. This results will allow us to find many more examples of languages that are not partially decidable, because as soon as we find any language that is partially decidable, but not totally decidable, we will know that its complement is not partially decidable.

Lemma 12.5

If language L is partially decidable but not totally decidable, then $\overline{L}$ is not partially decidable.

12.4.3 Undecidability of the Universal Language

We know that L_u, the Universal Language, is partially decidable. But is it totally decidable? It turns out that it is not, as we will prove in the following Lemma.

Lemma 12.6: L_u is not totally decidable

L_u is not totally decidable.

Proof. By Corollary 12.3, we know that $\overline{L_d} = \{w_i | M_i \text{ accepts } w_i\}$ is not totally decidable. Let us suppose that L_u is totally decidable. Then there exists a Turing Machine N such that for any Turing Machine M and any input string w, N halts on input $\langle M, w \rangle$, accepts w if M accepts w and rejects otherwise. But then we can build a Turing Machine N' that will take a string w_i for any i, concatenate it to itself (obtaining $w_i w_i$), and then run the Turing Machine N on it. Note that the "left" part of $w_i w_i$ is encoding of a Turing Machine M_i, and the right part is w_i. Therefore $w_i w_i$ is $\langle M_i, w_i \rangle$. We know that N will definitely halt on $\langle M_i, w_i \rangle$ and that it will accept if M_i accepts w_i, and reject otherwise. Let N' accept w_i if N accepts $w_i w_i = \langle M_i, w_i \rangle$. Then N' halts on all inputs and accepts exactly the strings from $\overline{L_d}$. This means that $\overline{L_d}$ is totally decidable, contradicting Corollary 12.3. Thus, the assumption that N' that decides L_u exists leads us to contradiction. Therefore, L_u cannot be totally decidable. The construction of this proof is summarised in Figure 12.4, with D being a Turing Machine that takes a binary string as an input and duplicates it on its tape. □

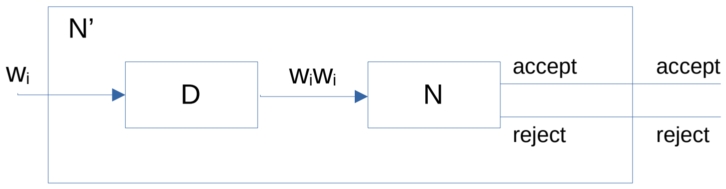

Figure 12.4: Turing Machine N' that decides on L_u.

With this example, we now have our first example of a language that is partially decidable, but not totally decidable. This proves that the classes of partially decidable and totally decidable languages are, indeed, different and that the class of partially decidable languages is a proper superset of the class of totally decidable languages. Also, it can easily be proven that every regular language is totally decidable. Therefore, we can represent the relation between the classes of languages that we have introduced so far with the diagram in Figure 12.5.

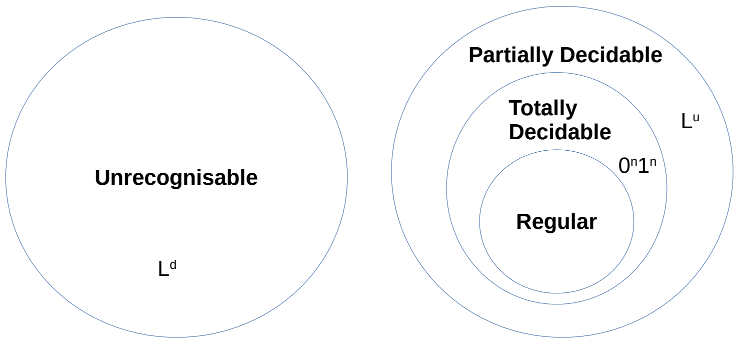

Figure 12.5: Relationship Between the Classes of Languages.

12.4.4 The Halting Problem

We will now introduce the famous problem known as the Halting Problem. This is the problem/language that Alan Turing proved to be undecidable in his original paper in 1936. In some literature this is the first undecidable problem (that is at the same time also partially decidable) that is introduced.

Definition 12.2: Halting Problem/Language

The Halting Language, L_{halt}, is the language

$$L_{\text{halt}} = \{\langle M, w\rangle | M \text{ halts on input } w\}.$$

Notice again the ambiguity between the "language" and the "problem" that we used. The Halting Problem is, for a given Turing Machine M and a given string w, both of which are parameters to the problem, to decide whether M halts on w. The Halting Language is the actual language described above. Therefore, the Halting Problem is deciding for a given Turing Machine M and a given string w, whether $\langle M, w\rangle$ belongs to the Halting Language.

Theorem 12.7: Halting Problem is Undecidable

L_{halt} is partially decidable, but not totally decidable.

Proof. It is obvious that L_{halt} is partially decidable. For a given string $\langle M, w\rangle$, we can simply use the Universal Turing Machine to simulate M on w. If the Universal Turing Machine halts (regardless of whether it accepts or rejects the string), $\langle M, w\rangle$ is accepted. It is obvious that in this way only the strings from the language L_{halt} get accepted.

Assume that L_{halt} is totally decidable. This means there is a Turing Machine M_{halt} that decides on L_{halt}. But then we can construct a Turing Machine M_u that decides on the Universal Language L_u in the following way. For an input $\langle M, w\rangle$, let M_u first run M_{halt} on $\langle M, w\rangle$. If it accepts, this means that M halts on w. M_u then runs the Universal Turing Machine on $\langle M, w\rangle$, and we know that it will halt and therefore either accept or reject $\langle M, w\rangle$. If M_{halt} rejects $\langle M, w\rangle$, that means that M does not halt on w, and therefore M_u rejects $\langle M, w\rangle$. Therefore, M_U always halts and accepts only the strings of form $\langle M, w\rangle$, where M accepts w. This means that M_u accepts exactly the Universal Language L_u. This makes L_u totally decidable, which we know is not true. Therefore, the assumption that L_{halt} is totally decidable leads us to contradiction. So, L_{halt} is not totally decidable. The construction of M_u is show in Figure 12.6. □

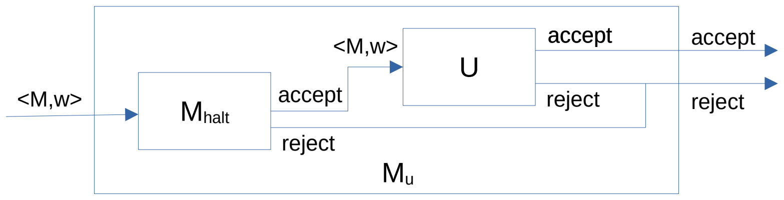

Figure 12.6: M_u That Would Accept L_u if L_{halt} Was Decidable.

Since L_{halt} is partially decidable, but not totally decidable, by Lemma 12.5 we know that $\overline{L_{\text{halt}}}$ cannot be even partially decidable. Therefore, the following corollary holds.

Corollary 12.8

The complement of the Halting Language,

$$\overline{L_{\text{halt}}} = \{\langle M, w\rangle | M \text{ does not halt on } w\}$$

is not partially decidable.

Undecidability of the Halting Problem, as well as that of the Universal Language, also has interpretation in more tangible terms of computer programs, since they are equivalent (in terms of decidability) to Turing Machines. When translated into terms of computer programs, undecidability of the Halting Problem means that we cannot write a program in any programming language that will take an arbitrary other program and an input to it, and return `true` if that program terminates and `false` otherwise. It is worth emphasising that we are not talking here about determining for some particular program whether it ever terminates or not, but we are talking about having an *analyser* program that will receive as inputs an arbitrary program and an input to that program and decide whether that program terminates on that input. Taking, for example, the program

```
int x=2, y=3;
return x+y;
```

and the empty list of parameters as the input, the output would need to be `true`. However, for the program

```
while (1);
return 23;
```

and the empty list of parameters the output would be `false`. Of course, we can write a program that will return `true` for all the programs that do terminate - this program merely needs to run its input program with the given inputs, and wait for the program to finish. If the input program terminates, then our analyser program will return `true`. But we have no way to tell if some program runs for a very long time whether it will eventually terminate (after, for example, 5 billion years), or it will never terminate. Therefore, this analyser program does not *decide* on the termination for each instance of a program and an input.

Very important observation here is that we do not claim that the analyser program that we want just currently doesn't exist. We claim something much stronger - that the analyser program *cannot* possibly exist on any current or future computer.

12.5 Examples of Undecidable Problems

In this section, we will give further examples of undecidable problems/languages. Each example that we give will be of a language that is partially decidable, but not totally decidable. For each such language L that we show to be partially but not totally decidable, the language $\overline{L}$ will give us another example of a language that is not even partially decidable.

12.5.1 Post's Correspondence Problem

Definition 12.3: Post's Correspondence Problem

Given two sequences of k strings, $(w_1, w_2, \ldots, w_k)$ and $(x_1, x_2, \ldots, x_k)$, find out if there exists a concatenation of corresponding strings from each list to form the same string.

We call the required concatenation a *solution* to the problem. We are, therefore, looking for an algorithm that will return "yes" if the solution exists, and "no" otherwise. For example, let $w_1 = 1, w_2 = 10111, w_3 = 10$ and $x_1 = 111, x_2 = 10, x_3 = 0$, then a solution in this case is

$$w_2w_1w_1w_3 = x_2x_1x_1x_3 = 101111110.$$

and our algorithm should return "yes". On the other hand, let $w_1 = 10, w_2 = 011, w_3 = 101$ and $x_1 = 101, x_2 = 11, x_3 = 011$. Then it is not hard to see that there is no solution. If there was a solution, it would have to start with w_1 and x_1, since these two are the only strings at the same position in their respective sequences where one is a prefix of the other. Now we have one string starting with 10 and the other starting with 101. Therefore, there is the last 1 in x_1 that is unmached in the concatenation of the strings w_1, w_2, w_3. Therfore, the next element in the concatenation has to be either w_1 and x_1 or w_3 and x_3. Assume it is w_1. Then on one side we have a concatenation $w_1w_1 = 1010$ and on the other $x_1x_1 = 101101$. These two concatenations do not match, therefore w_1w_1 (and x_1x_1) cannot be the start of a solution. Assume, instead, that w_3 and x_3 are the next strings in the concatenation. Then $w_1w_3 = 10101$ and $x_1x_3 = 101011$. There is a mismatch again. Therefore, for this input, there is no solution and our algorithm should return "no".

When seen as a language recognition problem, the Post's Correspondence Problem can be seen as a language

$$L_{\text{pcp}} = \{\langle (w_1, w_2, \ldots, w_k), (x_1, x_2, \ldots, x_k)\rangle \mid \text{there is a solution for } w_1, w_2, \ldots, w_k \text{ and } x_1, x_2, \ldots, x_k\},$$

and we want to determine the decidability of the language L_{pcp}. $\langle (w_1, w_2, \ldots, w_k), (x_1, x_2, \ldots, x_k)\rangle$ is some encoding of the sequences of strings $(w_1, w_2, \ldots, w_k)$ and $(x_1, x_2, \ldots, x_k)$ as binary strings.

It can be shown that if L_{pcp} is totally decidable, then so is L_{halt}, which we know is false. Therefore, L_{pcp} is not totally decidable. On the other hand, it is definitely partially decidable, since for any two input