# POST CORRESPONDENCE PROBLEM

The undecidability of the string is determined with the help of Post's Correspondence Problem (PCP). Let us define the PCP.

"The Post's correspondence problem consists of two lists of string that are of equal length over the input. The two lists are A = $w_1, w_2, w_3, \ldots, w_n$ and B = $x_1, x_2, x_3, \ldots x_n$ then there exists a non empty set of integers $i_1, i_2, i_3, \ldots$, in such that,
$w_1, w_2, w_3, \ldots w_n = x_1, x_2, x_3, \ldots x_n$"

To solve the post correspondence problem we try all the combinations of $i_1, i_2, i_3, \ldots$, in to find the w1 = x1 then we say that PCP has a solution.

Example 1:

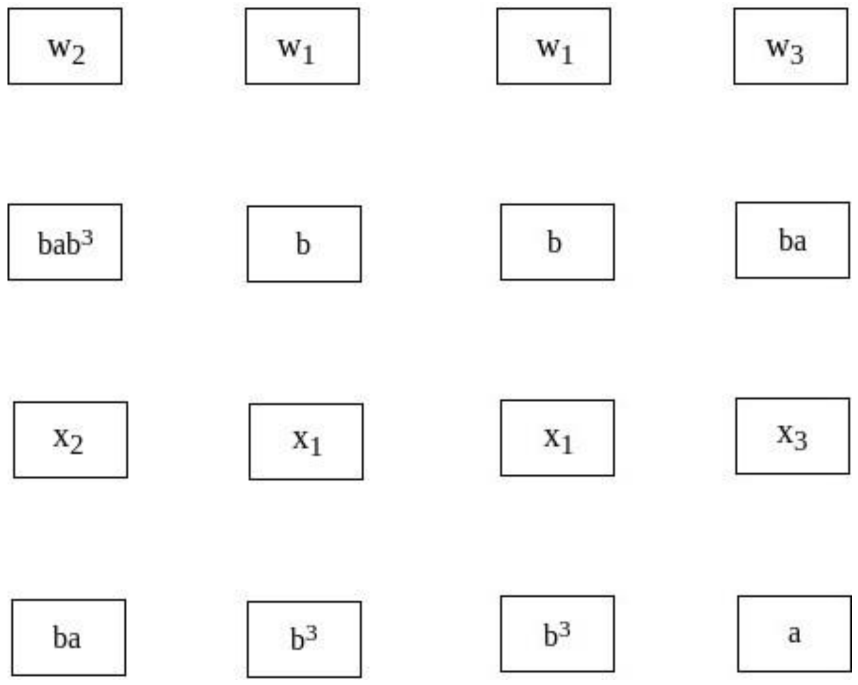
Consider the correspondence system as given below

A = (b, $bab^3$, ba) and B = ($b^3$, ba, a). The input set is $\sum = \{0, 1\}$. Find the solution.

**Solution:**

A solution is 2, 1, 1, 3. That means $w_2w_1w_1w_3 = x_2x_1x_1x_3$

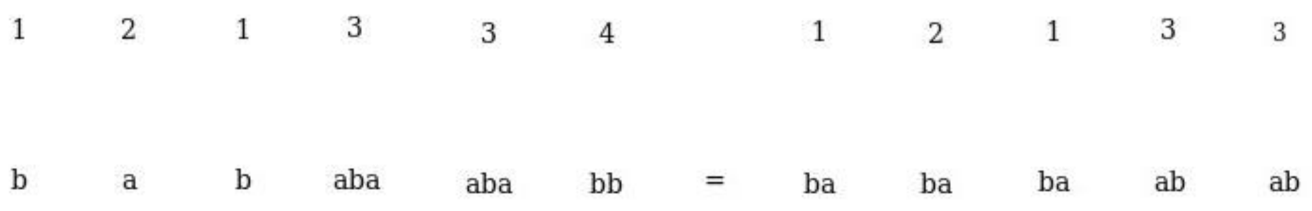
The constructed string from both lists is $bab^3b^3a$.



Example 2:

Does PCP with two lists x = (b, a, aba, bb) and y = (ba, ba, ab, b) have a solution?

**Solution:** Now we have to find out such a sequence that strings formed by x and y are identical. Such a sequence is 1, 2, 1, 3, 3, 4. Hence from x and y list



| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 1 | 3 | 3 | 4 | | 1 | 2 | 1 | 3 | 3 |
| b | a | b | aba | aba | bb | = | ba | ba | ba | ab | ab |

Example 3:

Obtain the solution for the following system of posts correspondence problem. A = {100, 0, 1}, B = {1, 100, 00}

**Solution:** Consider the sequence 1, 3, 2. The string obtained from A = babababb. The string obtained from B = bababbbb. These two strings are not equal. Thus if we try various combination from both the sets to find the unique sequence, we could not get such a sequence. Hence there is no solution for this system.

Example 4:

Obtain the solution for the following system of posts correspondence problem, X = {100, 0, 1}, Y = {1, 100, 00}.

**Solution:** The solution is 1, 3, 1, 1, 3, 2, 2. The string is

X1X3X1X1X3X2X2 = 100 + 1 + 100 + 100 + 1 + 0 + 0 = 1001100100100
Y1Y3Y1Y1Y3Y2Y2 = 1 + 00 + 1 + 1 + 00 + 100 + 100 = 1001100100100

# RECURSIVE ENUMERABLE LANGUAGES

## Recursive Enumerable (RE) or Type -0 Language

RE languages or type-0 languages are generated by type-0 grammars. An RE language can be accepted or recognized by Turing machine which means it will enter into final state for the strings of language and may or may not enter into rejecting state for the strings which are not part of the language. It means TM can loop forever for the strings which are not a part of the language. RE languages are also called as Turing recognizable languages.

## Recursive Language (REC)

A recursive language (subset of RE) can be decided by Turing machine which means it will enter into final state for the strings of language and rejecting state for the strings which are not part of the language. e.g.; L= {$a^nb^nc^n$|n>=1} is recursive because we can construct a turing machine which will move to final state if the string is of the form $a^nb^nc^n$ else move to non-final state. So the TM will always halt in this case. REC languages are also called as Turing decidable languages. The relationship between RE and REC languages can be shown in Figure 1.

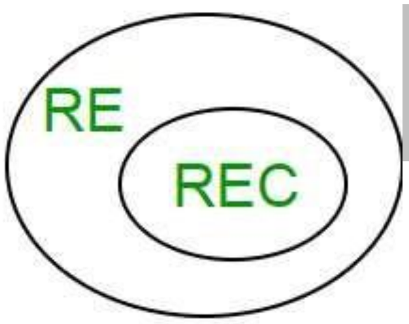


Figure 1

## Closure Properties of Recursive Languages

- **Union**: If L1 and If L2 are two recursive languages, their union L1∪L2 will also be recursive because if TM halts for L1 and halts for L2, it will also halt for L1∪L2.
- **Concatenation:** If L1 and If L2 are two recursive languages, their concatenation L1.L2 will also be recursive. For Example:
- L1= {$a^nb^nc^n$|n>=0}
- L2= {$d^me^mf^m$|m>=0}
- L3= L1.L2
- = {$a^nb^nc^nd^m e^mf^m$|m>=0 and n>=0} is also recursive.

L1 says n no. of a's followed by n no. of b's followed by n no. of c's. L2 says m no. of d's followed by m no. of e's followed by m no. of f's. Their concatenation first matches no. of a's, b's and c's and then matches no. of d's, e's and f's. So it can be decided by TM.

- **Kleene Closure:** If L1is recursive, its kleene closure L1* will also be recursive. For Example:
  L1= {$a^nb^nc^n$|n>=0}
  L1*= { $a^nb^nc^n$||n>=0}* is also recursive.
- **Intersection and complement**: If L1 and If L2 are two recursive languages, their intersection L1 ∩ L2 will also be recursive. For Example:
- L1= {$a^nb^nc^n$dm|n>=0 and m>=0}
- L2= {$a^nb^nc^nd^n$|n>=0 and m>=0}
- L3=L1 ∩ L2
- = { $a^nb^nc^nd^n$ |n>=0} will be recursive.

L1 says n no. of a's followed by n no. of b's followed by n no. of c's and then any no. of d's.

L2 says any no. of a's followed by n no. of b's followed by n no. of c's followed by n no. of d's. Their intersection says n no. of a's followed by n no. of b's followed by n no. of c's followed by n no. of d's. So it can be decided by turing machine, hence recursive.

Similarly, complement of recursive language L1 which is ∑*-L1, will also be recursive.

## P, NP AND NP-COMPLETE PROBLEMS

Problems that can be solved in polynomial time are called ***tractable***, and problems that cannot be solved in polynomial time are called ***intractable***.

There are several **reasons for intractability**.

- **First**, we **cannot solve** arbitrary instances of intractable problems in a reasonable amount of time unless such **instances are verysmall**.

**Second**, although there might be a huge difference between the running times in *O(p(n))* for polynomials of **drastically different degrees**. where p(n) is a polynomial of the problem's input sizen.

- **Third**, polynomial functions possess many convenient properties; in particular, both the sum and composition of two polynomials are **always polynomialstoo**.
- **Fourth**, the choice of this class has led to a development of an extensive theory called ***computational complexity***.

**Definition: Class *P*** is a class of decision problems that can be solved in polynomial time by deterministic algorithms. This class of problems is called ***polynomial class***.

- Problems that can be solved in polynomial time as the set that computer science theoreticians call **P**. A more formal definition includes in P only **decision problems**, which are problems with **yes/no**answers.
- TheclassofdecisionproblemsthataresolvableinO($p$($n$))**polynomialtime**,where$p$($n$)is a polynomial of problem's input size $n$

  **Examples:**
  - Searching
  - Elementuniqueness
  - Graph connectivity
  - Graph acyclicity
  - Primality testing (finally proved in2002)
- **The restriction of P** to decision problems can be justified by the followingreasons.
  - First, it is sensible to **exclude problems not solvable in polynomial time** because of their exponentially large output. e.g., generating subsets of a given set or all the permutations of n distinctitems.
  - Second, **many important problems that are not decision problems** in

their most natural formulation can be reduced to a series of decision problems that are easier to study. For example, instead of asking about the minimum number of colors needed to color the vertices of a graph so that no two adjacent vertices are colored the same color. Coloring of the graph's vertices with no more than m colors for m = 1, 2, (The latter is called the **m-coloringproblem**.)

- So, every decision problem can not be solved in polynomial time. Some **decision** problems cannot be solved at all by any algorithm. Such problems are called **undecidable**, as opposed to **decidable** problems that can be solved by an algorithm (**Halting problem**).

- **Non polynomial-time algorithm:** There are many important problems, however, for which no polynomial-time algorithm has been found.
  - ***Hamiltonian circuit problem*:** Determine whether a given graph has a Hamiltonian circuit—a path that starts and ends at the same vertex and passes through all the other vertices exactly once.
  - ***Traveling salesman problem*:** Find the shortest tour through n cities with known positive integer distances between them (find the shortest Hamiltonian circuit in a complete graph with positive integer r weights).
  - ***Knapsack problem*:** Find the most valuable subset of n items of given positive integer weights and values that fit into a knapsack of a given positive integer capacity.
  - ***Partition problem*:** Given n positive integers, determine whether it is possible to partition them into two disjoint subsets with the same sum.
  - ***Bin-packing problem*:** Given n items whose sizes are positive rational numbers not larger than 1, put them into the smallest number of bins of size1.
  - ***Graph-coloring problem*:** For a given graph, find its chromatic number, which is the smallest number of colors that need to be assigned to the graph's vertices so that no two adjacent vertices are assigned the same color.
  - ***Integer linear programming problem*:** Find the maximum (or minimum) value of a linear function of several integer-valued variables subject to a finite set of constraints in the form of linear equalities and inequalities.

**Definition: A nondeterministic algorithm** is a two-stage procedure that takes as its input an instance I of a decision problem and does the following.

1. **Nondeterministic ("guessing") stage:** An arbitrary string S is generated that can be thought of as a candidate solution to the giveninstance.
2. **Deterministic ("verification") stage:** A deterministic algorithm takes both I and S as its input and outputs yes if S represents a solution to instance I. (If S is not a solution to instance I , the algorithm either returns no or is allowed not to halt atall.)

Finally, a nondeterministic algorithm is said to be ***nondeterministic polynomial*** if the time efficiency of its verification stage is polynomial.

**Definition: Class *NP*** is the class of decision problems that can be solved by nondeterministic polynomial algorithms. This class of problems is called ***nondeterministic polynomial***.

Most decision problems are in NP. First of all, this class includes all the problems in P:

$$\mathbf{P} \subseteq \mathbf{NP}$$

This is true because, if a problem is in P, we can use the deterministic polynomial time algorithm that solves it in the verification-stage of a nondeterministic algorithm that simply ignores string S generated in its nondeterministic ("guessing") stage. But NP also contains the Hamiltonian circuit problem, the partition problem, decision versions of the traveling salesman, the knapsack, graph coloring, and many hundreds of other difficult combinatorial optimization. The halting problem, on the other hand, is among the rare examples of decision problems that are known not to be in NP.

Note that P = NP would imply that each of many hundreds of difficult combinatorial decision problems can be solved by a polynomial-time algorithm.

**Definition:** A decision problem *D*1 is said to be ***polynomially reducible*** to a decision problem *D*2, if there exists a function *t* that transforms instances of *D*1 to instances of *D*2 such that:

1. *t* maps all yes instances of *D*1 to yes instances of *D*2 and all no instances of *D*1 to no instances of*D*2.
2. *t* is computable by a polynomial time algorithm.

This definition immediately implies that if a problem D1 is polynomially reducible to some problemD2 that can be solved in polynomial time, then problem D1 can also be solved in polynomial time

**Definition:** A decision problem *D* is said to be ***NP-complete*** if it is hard as any problem in NP.

1. It belongs to class *NP*
2. Every problem in *NP* is polynomially reducible to*D*

The fact that closely related decision problems are polynomially reducible to each other is not very surprising. For example, let us prove that the Hamiltonian circuit problem is polynomially reducible to the decision version of the traveling salesman problem.

**Theorem: A decision problem is said to be *NP-complete* if it is hard as any problem in NP.**

**Proof:** Let us prove that the Hamiltonian circuit problem is polynomially reducible to the decision version of the traveling salesman problem.

We can map a graph G of a given instance of the Hamiltonian circuit problem to a complete weighted graph G′ representing an instance of the traveling salesman problem by assigning 1 as the weight to each edge in G and adding an edge of weight 2 between any pair of nonadjacent vertices in G. As the upper bound *m* on the Hamiltonian circuit length, we take *m* = *n*, where *n* is the number of vertices in G (and G′ ). Obviously, this transformation can be done in polynomialtime.

Let G be a yes instance of the Hamiltonian circuit problem. Then G has a Hamiltonian circuit, and its image in G′ will have length n, making the image a yes instance of the decision traveling salesman problem.

Conversely, if we have a Hamiltonian circuit of the length not larger than n in G′, then its length must be exactly n and hence the circuit must be made up of edges present in G, making the inverse image of the yes instance of the decision traveling salesman problem be a yes instance of the Hamiltonian circuitproblem.

This completes the proof.

**Theorem: State and prove Cook's theorem.**

Prove that CNF-sat is *NP*-complete.

Satisfiability of boolean formula for three conjuctive normal form is NP-Complete.

*NP* problems obtained by polynomial-time reductions from a *NP*-complete problem **Proof:** The notion of *NP*-completeness requires, however, polynomial reducibility of *all* problems in *NP*, both known and unknown, to the problem in question. Given the bewildering variety of decision problems, it is nothing short of amazing that specific examples of *NP*-complete problems have been actually found.

Nevertheless, this mathematical feat was accomplished independently by Stephen Cook in the United States and Leonid Levin in the former Soviet Union. In his 1971 paper, Cook [Coo71] showed that the so-called ***CNF-satisfiability problem*** is *NP*complete.

| $_1$ | $_2$ | $_3$ | $\overline{\ }_1$ | $\overline{\ }_2$ | $\overline{\ }_3$ | $_1\overline{V}_2\overline{V}_3$ | $\overline{\ }_1V_2$ | $\overline{\ }_1\overline{V}_2\overline{V}_3$ | $({}_1\overline{V}{}_2\overline{V}{}_3)\blacktriangle(\overline{\ }_1V{}_2)\blacktriangle(\overline{\ }_1\overline{V}{}_2\overline{V}{}_3)$ |
|---|---|---|---|---|---|---|---|---|---|
| T | T | T | F | F | F | T | T | F | F |
| **T** | **T** | **F** | F | F | T | T | T | T | **T** |
| T | F | T | F | T | F | T | F | T | F |
| T | F | F | F | T | T | T | F | T | F |
| F | T | T | T | F | F | F | T | T | F |
| F | T | F | T | F | T | T | T | T | T |
| F | F | T | T | T | F | T | T | T | T |
| F | F | F | T | T | T | T | T | T | T |

The CNF-satisfiability problem deals with boolean expressions. Each boolean expression can be represented in conjunctive normal form, such as the following expression involving three boolean variables $x_1$, $x_2$, and $x_3$ and their negations denoted $\overline{\ }_1$, $\overline{\ }_2$, and $\overline{\ }_3$respectively:

$$(x_1 \vee \bar{x}_2 \vee \bar{x}_3) \& (\bar{x}_1 \vee x_2) \& (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)$$

The CNF-satisfiability problem asks whether or not one can assign values *true* and *false* to variables of a given boolean expression in its CNF form to make the entire expression *true*. (It is easy to see that this can be done for the above formula: if $x_1$ = *true*, $x_2$ = *true*, and $x_3$ = *false*, the entire expression is *true*.)

Since the Cook-Levin discovery of the first known *NP*-complete problems, computer scientists have found many hundreds, if not thousands, of other examples. In particular, the well- known problems (or their decision versions) mentioned above—Hamiltonian circuit, traveling salesman, partition, bin packing, and graph coloring—are all *NP*-complete. It is known, however, that if *P* != *NP* there must exist *NP* problems that neither are in *P* nor are *NP*-complete.

Showing that a decision problem is *NP*-complete can be done in two steps.

1. First, one needs to show that the problem in question is in *NP*; i.e., a randomly generated string can be checked in polynomial time to determine whether or not it represents a solution to the problem. Typically, this step is easy.

2. The second step is to show that every problem in NP is reducible to the problem in question in polynomial time. Because of the transitivity of polynomial reduction, this step can be done by showing that a known NP-complete problem can be transformed to the problem in question in polynomial ime.

The definition of *NP*-completeness immediately implies that if there exists a deterministic polynomial-time algorithm for just one *NP*-complete problem, then every problem in *NP* can be solved in polynomial time by a deterministic algorithm, and hence *P* = *NP*.

# UNDECIDABLE PROBLEM ABOUT TURING MACHINE

The reduction is used to prove whether given language is desirable or not. In this section, we will understand the concept of reduction first and then we will see an important theorem in this regard.

Reduction

Reduction is a technique in which if a problem P1 is reduced to a problem P2 then any solution of P2 solves P1. In general, if we have an algorithm to convert an instance of a problem P1 to an instance of a problem P2 that have the same answer then it is called as P1 reduced P2. Hence if P1 is not recursive then P2 is also not recursive. Similarly, if P1 is not recursively enumerable then P2 also is not recursively enumerable.

**Theorem:** if P1 is reduced to P2

1. If P1 is undecidable, then P2 is also undecidable.
2. If P1 is non-RE, then P2 is also non-RE.

**Proof:**

1. Consider an instance w of P1. Then construct an algorithm such that the algorithm takes instance w as input and converts it into another instance x of P2. Then apply that algorithm to check whether x is in P2. If the algorithm answer 'yes' then that means x is in P2, similarly we can also say that w is in P1. Since we have obtained P2 after reduction of P1. Similarly if algorithm answer 'no' then x is not in P2, that also means w is not in P1. This proves that if P1 is undecidable, then P1 is also undecidable.
2. We assume that P1 is non-RE but P2 is RE. Now construct an algorithm to reduce P1 to P2, but by this algorithm, P2 will be recognized. That means there will be a Turing machine that says 'yes' if the input is P2 but may or may not halt for the input which is not in P2. As we know that one can convert an instance of w in P1 to an instance x in P2. Then apply a TM to check whether x is in P2. If x is accepted that also means w is accepted. This procedure describes a TM whose language is P1 if w is in P1 then x is also in P2 and if w is not in P1 then x is also not in P2. This proves that if P1 is non-RE then P2 is also non-RE.

Empty and non empty languages:
There are two types of languages empty and non empty language. $L_e$t $L_e$ denotes an empty language, and $L_{ne}$ denotes non empty language. $L_e$t w be a binary string, and Mi be a TM. If $L(M_j) = \Phi$ then Mi

does not accept input then w is in $L_e$. Similarly, if $L(M_j)$ is not the empty language, then w is in $L_{ne}$. Thus we can say that

$L_e = \{M \mid L(M) = \Phi\}$

$L_{ne} = \{M \mid L(M) \neq \Phi\}$