Regular VS context free

- Beside some technical definition, regular languages are subset of context-free languages (CFL). Regular languages are recognized by Finite Automata, whereas context free languages are recognized by Push Down Automata. For regular languages you may write Regular Expressions, for context free language you may write Context Free Grammar. Regular languages are closed under concatenation, union, Kleen star, complement and intersection where as CFL is closed under concatenation, union, Kleen star.

- Let's start with the formal definition of grammar. A grammar is a precise description of a formal language, ie., it describes what possible sequence of symbols/strings constitute valid words or sentences in that language, but doesn't describe their semantics.   
  
A grammar G <N, Σ ,P, S> consists of the following components:

1. A finite set N of non terminal symbols or variables.
2. A finite set Σ of terminal symbols that are disjoint from N.
3. A finite set P of production rules of the form   
   (Σ U N)\* N (Σ U N)\* -> (Σ U N)\* where \* is the Kleene star operator and U denotes the set union. Each production rule maps from one string of symbols to another where the left hand side contains at least one non terminal symbol.
4. A distinguished start symbol S ∈ N.

A language is said to be a ***regular language*** if it is generated by a *regular grammar*. A grammar is said to be *regular*if it's either *right-linear* or *left-linear.*   
Specifically, a grammar  G <N, Σ ,P, S> is said to be right-linear if each of its production rules is either of the form A -> xB or of the form A -> x, where A and B are non terminal symbols in N and x is a string of terminal symbols in   
Σ\*. Similarly, it is left-linear if each of its production rules is either of the form A -> Bx or of the form A -> x, where A and B are non terminal symbols in N and x is a string of terminal symbols in Σ\*.  
  
A language is said to be ***context-free*** if it is generated by a *context-free grammar*. A grammar G <N, Σ, P, S> is context-free if the production rules are of the form N -> (N U Σ)\*.  
  
Unlike regular grammars, the right hand side of the production rules in context free grammars are unrestricted and can be any combination of terminals and non terminals. Regular languages are subsets of context free languages.

CFG is ambiguous whereas RG is unambiguous.

Finite automata

Finite Automata(FA) is the simplest machine to recognize patterns.

A Finite Automata consists of the following :

Q : Finite set of states.

∑ : set of Input Symbols.

q : Initial state.

F : set of Final States.

δ : Transition Function.

Formal specification of machine is  
{ Q, ∑, q, F, δ }.

FA is characterized into two types:

**1) Deterministic Finite Automata (DFA)**

DFA consists of 5 tuples {Q, ∑, q, F, δ}.

Q : set of all states.

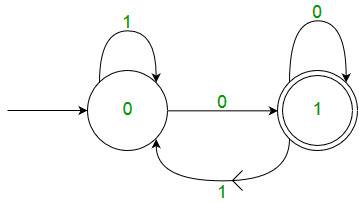
∑ : set of input symbols. ( Symbols which machine takes as input )

q : Initial state. ( Starting state of a machine )

F : set of final state.

δ : Transition Function, defined as δ : Q X ∑ --> Q.

In a DFA, for a particular input character, the machine goes to one state only. A transition function is defined on every state for every input symbol. Also in DFA null (or ε) move is not allowed, i.e., DFA cannot change state without any input character.

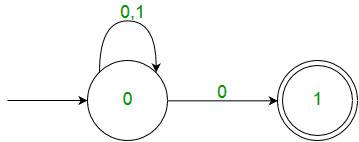
For example, below DFA with ∑ = {0, 1} accepts all strings ending with 0.  
[](https://www.geeksforgeeks.org/wp-content/uploads/Finite_automata_introduction_1.jpg)  
One important thing to note is, ***there can be many possible DFAs for a pattern***. A DFA with minimum number of states is generally preferred.

**2) Nondeterministic Finite Automata(NFA)**  
NFA is similar to DFA except following additional features:  
1. Null (or ε) move is allowed i.e., it can move forward without reading symbols.  
2. Ability to transmit to any number of states for a particular input.  
However, these above features don’t add any power to NFA. If we compare both in terms of power, both are equivalent.

Due to above additional features, NFA has a different transition function, rest is same as DFA.

δ: Transition Function

δ:  Q X (∑ U ϵ ) --> 2 ^ Q.

As you can see in transition function is for any input including null (or ε), NFA can go to any state number of states.  
For example, below is a NFA for above problem  
[](https://www.geeksforgeeks.org/wp-content/uploads/Finite_automata_introduction_2.jpg)  
One important thing to note is, ***in NFA, if any path for an input string leads to a final state, then the input string accepted***. For example, in above NFA, there are multiple paths for input string “00”. Since, one of the paths leads to a final state, “00” is accepted by above NFA.

**Some Important Points:**  
1. Every DFA is NFA but not vice versa.  
2. Both NFA and DFA have same power and each NFA can be translated into a DFA.  
3. There can be multiple final states in both DFA and NFA.  
3. NFA is more of a theoretical concept.  
4. DFA is used in Lexical Analysis in Compiler

Arden's Theorem

In order to find out a regular expression of a Finite Automaton, we use Arden’s Theorem along with the properties of regular expressions.

***Statement*** −

Let **P** and **Q** be two regular expressions.

If **P** does not contain null string, then **R = Q + RP** has a unique solution that is **R = QP\***

**Proof** −

R = Q + (Q + RP)P [After putting the value R = Q + RP]

= Q + QP + RPP

When we put the value of **R** recursively again and again, we get the following equation −

R = Q + QP + QP2 + QP3…..

R = Q (ε + P + P2 + P3 + …. )

R = QP\* [As P\* represents (ε + P + P2 + P3 + ….) ]

Hence, proved.

Assumptions for Applying Arden’s Theorem

* The transition diagram must not have NULL transitions
* It must have only one initial state

Method

**Step 1** − Create equations as the following form for all the states of the DFA having n states with initial state q1.

q1 = q1R11 + q2R21 + … + qnRn1 + ε

q2 = q1R12 + q2R22 + … + qnRn2

…………………………

…………………………

…………………………

…………………………

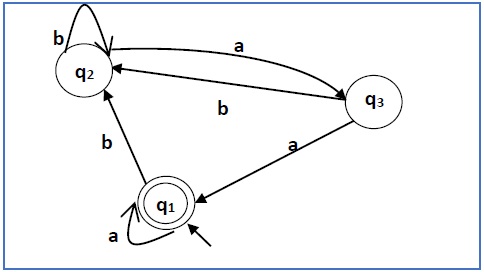
qn = q1R1n + q2R2n + … + qnRnn

**Rij** represents the set of labels of edges from **qi** to **qj**, if no such edge exists, then **Rij = ∅**

**Step 2** − Solve these equations to get the equation for the final state in terms of **Rij**

**Problem**

Construct a regular expression corresponding to the automata given below −



**Solution** −

Here the initial state and final state is **q1**.

The equations for the three states q1, q2, and q3 are as follows −

q1 = q1a + q3a + ε (ε move is because q1 is the initial state0

q2 = q1b + q2b + q3b

q3 = q2a

Now, we will solve these three equations −

q2 = q1b + q2b + q3b

= q1b + q2b + (q2a)b (Substituting value of q3)

= q1b + q2(b + ab)

= q1b (b + ab)\* (Applying Arden’s Theorem)

q1 = q1a + q3a + ε

= q1a + q2aa + ε (Substituting value of q3)

= q1a + q1b(b + ab\*)aa + ε (Substituting value of q2)

= q1(a + b(b + ab)\*aa) + ε

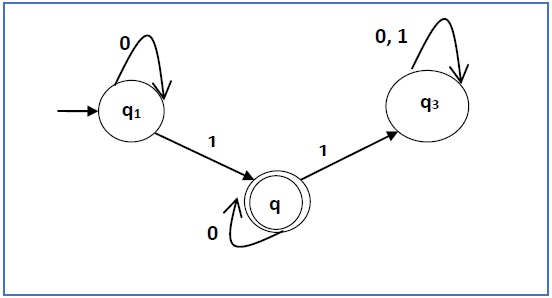
= ε (a+ b(b + ab)\*aa)\*

= (a + b(b + ab)\*aa)\*

Hence, the regular expression is (a + b(b + ab)\*aa)\*.

**Problem**

Construct a regular expression corresponding to the automata given below −



**Solution** −

Here the initial state is q1 and the final state is q2

Now we write down the equations −

q1 = q10 + ε

q2 = q11 + q20

q3 = q21 + q30 + q31

Now, we will solve these three equations −

q1 = ε0\* [As, εR = R]

So, q1 = 0\*

q2 = 0\*1 + q20

So, q2 = 0\*1(0)\* [By Arden’s theorem]

Hence, the regular expression is 0\*10\*.

Pumping lemma for regular language

Theorem

Let L be a regular language. Then there exists a constant **‘c’** such that for every string **w** in **L** −

**|w| ≥ c**

We can break **w** into three strings, **w = xyz**, such that −

* |y| > 0
* |xy| ≤ c
* For all k ≥ 0, the string xykz is also in L.

Applications of Pumping Lemma

Pumping Lemma is to be applied to show that certain languages are not regular. It should never be used to show a language is regular.

* If **L** is regular, it satisfies Pumping Lemma.
* If **L** does not satisfy Pumping Lemma, it is non-regular.

Method to prove that a language L is not regular

* At first, we have to assume that **L** is regular.
* So, the pumping lemma should hold for **L**.
* Use the pumping lemma to obtain a contradiction −
  + Select **w** such that **|w| ≥ c**
  + Select **y** such that **|y| ≥ 1**
  + Select **x** such that **|xy| ≤ c**
  + Assign the remaining string to **z.**
  + Select **k** such that the resulting string is not in **L.**

**Hence L is not regular.**

**Problem**

Prove that **L = {aibi | i ≥ 0}** is not regular.

***Solution*** −

* At first, we assume that **L** is regular and n is the number of states.
* Let w = *anbn*. Thus |w| = 2n ≥ n.
* By pumping lemma, let w = xyz, where |xy| ≤ n.
* Let x = ap, y = aq, and z = arbn, where p + q + r = n, p ≠ 0, q ≠ 0, r ≠ 0. Thus |y| ≠ 0.
* Let k = 2. Then xy2z = apa2qarbn.
* Number of as = (p + 2q + r) = (p + q + r) + q = n + q
* Hence, xy2z = an+q bn. Since q ≠ 0, xy2z is not of the form anbn.
* Thus, xy2z is not in L. Hence L is not regular.

Pumping Lemma for context free languages

If **L** is a context-free language, there is a pumping length **p** such that any string **w ∈ L** of length **≥ p** can be written as **w = uvxyz**, where **vy ≠ ε**, **|vxy| ≤ p**, and for all **i ≥ 0, uvixyiz ∈ L**.

## Applications of Pumping Lemma

Pumping lemma is used to check whether a grammar is context free or not. Let us take an example and show how it is checked.

### Problem

Find out whether the language **L = {xnynzn | n ≥ 1}** is context free or not.

### Solution

Let **L** is context free. Then, **L** must satisfy pumping lemma.

At first, choose a number **n** of the pumping lemma. Then, take z as 0n1n2n.

Break **z** into **uvwxy,** where

**|vwx| ≤ n and vx ≠ ε.**

Hence **vwx** cannot involve both 0s and 2s, since the last 0 and the first 2 are at least (n+1) positions apart. There are two cases −

**Case 1** − **vwx** has no 2s. Then **vx** has only 0s and 1s. Then **uwy**, which would have to be in **L**, has **n** 2s, but fewer than **n** 0s or 1s.

**Case 2** − **vwx** has no 0s.

Here contradiction occurs.

Hence, **L** is not a context-free language.

Closure properties of regular languages

**1. Closure under Union**

If L and M are regular languages, so is L UM.

Proof : Let L and M be the languages of regular expressions R and S, respectively.

Then R+S is a regular expression whose language is L U M

**2. Closure under Concatenation and Kleene Closure**

The same idea can be applied using Kleeneclosure :

RS is a regular expression whose language is LM.

R\* is a regular expression whose language is L\*.

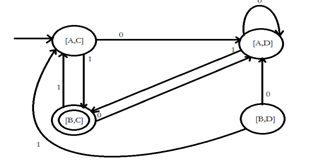
**3. Closure under intersection**

If L and M are regular languages, so is L ∩ M

Proof : Let A and B be two DFA’s whose regular languages are L and M respectively.

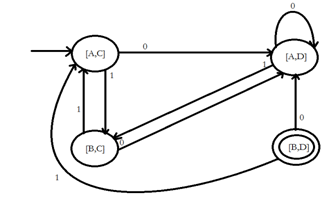
Now, construct C, the product automation of A and B. Make the final states of C be the pairs consisting of final states of both A and B.





**4. Closure under Difference**

If L and M area regular languages, so is L-M, which means all the strings that are in L , but not in M. Proof : Let A and B be two DFA’s whose regular languages are L and M respectively. Now, construct C, the product automation of A and B. Make the final states of C be the pairs consisting of final states of A, but not of B. The DFA’s A-B and C-D remain unchanged, but the final DFA varies as follows:



**5. Closure under Concatenation**

The complementof a language L (with respect to an alphabet Σ such that Σ\* contains L) isΣ\* – L Since Σ\* is surely regular, the complement of a regular language is always regular

**6. Closure under Reversal**

Given language L, LR is the set of strings whose reversal is in L

L = {0, 01, 100}; LR = {0, 10, 001}

Basis: If E is a symbol a, ε, or ∅, then ER = E.

Induction: If E is

* F+G, then ER = FR + GR.
* FG, then ER = GRFR
* F*, then ER = (FR)*

Let E = 01\* + 10\*.

ER=(01∗+10∗)R=(01∗)R+(10∗)RER=(01∗+10∗)R=(01∗)R+(10∗)R

=(1∗)R0R+(0∗)R1R=(1∗)R0R+(0∗)R1R

=(1R)∗0+(0R)∗1=(1R)∗0+(0R)∗1

=1∗0+0∗1=1∗0+0∗1

**7. Closure under homomorphism**

Definition of homomorphism:

A homomorphism on an alphabet is a function that gives a string for each symbol in that alphabet.

**Closure property:**

If L is a regular language, and h is a homomorphism on its alphabet, then h(L) = {h(w) | w is in L} is also a regular language.

Proof: Let E be a regular expression for L.

Apply h to each symbol in E.

Language of resulting RE is h(L)

Example:

Let h(0) = ab; h(1) = ε.

Let L be the language of regular expression 01\* + 10\*.

Then h(L) is the language of regular expression abε\* + ε(ab)\*

abε\* + ε(ab)\* can be simplified.

ε\* = ε, so abε\* = abε.

ε is the identity under concatenation.

That is, εE = Eε= E for any RE E.

Thus, abε\* + ε(ab)\* = abε + ε(ab)\*

= ab + (ab)\*.

Finally, L(ab) is contained in L((ab)*), so a RE for h(L) is (ab)*.

**Converting grammar to Chomsky Normal Form (CNF)**

A context free grammar (CFG) is in Chomsky Normal Form (CNF) if all production rules satisfy one of the following conditions:

* A non-terminal generating a terminal (e.g.; X->x)
* A non-terminal generating two non-terminals (e.g.; X->YZ)
* Start symbol generating ε. (e.g.; S-> ε)

Consider the following grammars,

G1 = {S->a, S->AZ, A->a, Z->z}

G2 = {S->a, S->aZ, Z->a}

The grammar G1 is in CNF as production rules satisfy the rules specified for CNF. However, the grammar G2 is not in CNF as the production rule S->aZ contains terminal followed by non-terminal which does not satisfy the rules specified for CNF.

**Note –**

* For a given grammar, there can be more than one CNF.
* CNF produces the same language as generated by CFG.
* CNF is used as a preprocessing step for many algorithms for CFG like CYK(membership algo), bottom-up parsers etc.
* For generating string w of length ‘n’ requires ‘2n-1’ production or steps in CNF.
* Any Context free Grammar that do not have ε in it’s language has an equivalent CNF.

**How to convert CFG to CNF?**

**Step 1.** Eliminate start symbol from RHS.  
If start symbol S is at the RHS of any production in the grammar, create a new production as:  
S0->S  
where S0 is the new start symbol.

**Step 2.** Eliminate null, unit and useless productions.  
If CFG contains null, unit or useless production rules, eliminate them.

**Step 3.** Eliminate terminals from RHS if they exist with other terminals or non-terminals. e.g,; production rule X->xY can be decomposed as:  
X->ZY  
Z->x

**Step 4.** Eliminate RHS with more than two non-terminals.  
e.g,; production rule X->XYZ can be decomposed as:  
X->PZ  
P->XY

P and NP problems

In Computer Science, many problems are solved where the objective is to maximize or minimize some values, whereas in other problems we try to find whether there is a solution or not. Hence, the problems can be categorized as follows −

Optimization Problem

Optimization problems are those for which the objective is to maximize or minimize some values. For example,

* Finding the minimum number of colors needed to color a given graph.
* Finding the shortest path between two vertices in a graph.

Decision Problem

There are many problems for which the answer is a Yes or a No. These types of problems are known as **decision problems**. For example,

* Whether a given graph can be colored by only 4-colors.
* Finding Hamiltonian cycle in a graph is not a decision problem, whereas checking a graph is Hamiltonian or not is a decision problem.

What is Language?

Every decision problem can have only two answers, yes or no. Hence, a decision problem may belong to a language if it provides an answer ‘yes’ for a specific input. A language is the totality of inputs for which the answer is Yes. Most of the algorithms discussed in the previous chapters are **polynomial time algorithms**.

For input size ***n***, if worst-case time complexity of an algorithm is ***O(nk)***, where ***k*** is a constant, the algorithm is a polynomial time algorithm.

Algorithms such as Matrix Chain Multiplication, Single Source Shortest Path, All Pair Shortest Path, Minimum Spanning Tree, etc. run in polynomial time. However there are many problems, such as traveling salesperson, optimal graph coloring, Hamiltonian cycles, finding the longest path in a graph, and satisfying a Boolean formula, for which no polynomial time algorithms is known. These problems belong to an interesting class of problems, called the **NP-Complete** problems, whose status is unknown.

In this context, we can categorize the problems as follows −

P-Class

The class P consists of those problems that are solvable in polynomial time, i.e. these problems can be solved in time ***O(nk)*** in worst-case, where **k** is constant.

These problems are called **tractable**, while others are called **intractable or superpolynomial**.

Formally, an algorithm is polynomial time algorithm, if there exists a polynomial ***p(n)*** such that the algorithm can solve any instance of size **n** in a time ***O(p(n))***.

Problem requiring ***Ω(n50)*** time to solve are essentially intractable for large ***n***. Most known polynomial time algorithm run in time ***O(nk)*** for fairly low value of ***k***.

The advantages in considering the class of polynomial-time algorithms is that all reasonable **deterministic single processor model of computation** can be simulated on each other with at most a polynomial slow-d

NP-Class

The class NP consists of those problems that are verifiable in polynomial time. NP is the class of decision problems for which it is easy to check the correctness of a claimed answer, with the aid of a little extra information. Hence, we aren’t asking for a way to find a solution, but only to verify that an alleged solution really is correct.

Every problem in this class can be solved in exponential time using exhaustive search.

P versus NP

Every decision problem that is solvable by a deterministic polynomial time algorithm is also solvable by a polynomial time non-deterministic algorithm.

All problems in P can be solved with polynomial time algorithms, whereas all problems in *NP - P* are intractable.

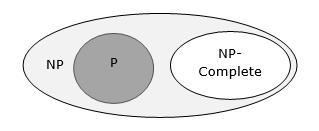
It is not known whether ***P = NP***. However, many problems are known in NP with the property that if they belong to P, then it can be proved that P = NP.

If ***P ≠ NP***, there are problems in NP that are neither in P nor in NP-Complete.

The problem belongs to class **P** if it’s easy to find a solution for the problem. The problem belongs to **NP**, if it’s easy to check a solution that may have been very tedious to find

NP hard and NP complete

A problem is in the class NPC if it is in NP and is as **hard** as any problem in NP. A problem is **NP-hard** if all problems in NP are polynomial time reducible to it, even though it may not be in NP itself.



If a polynomial time algorithm exists for any of these problems, all problems in NP would be polynomial time solvable. These problems are called **NP-complete**. The phenomenon of NP-completeness is important for both theoretical and practical reasons.

Definition of NP-Completeness

A language **B** is ***NP-complete*** if it satisfies two conditions

* **B** is in NP
* Every **A** in NP is polynomial time reducible to **B**.

If a language satisfies the second property, but not necessarily the first one, the language **B** is known as **NP-Hard**. Informally, a search problem **B** is **NP-Hard** if there exists some **NP-Complete** problem **A** that Turing reduces to **B**.

The problem in NP-Hard cannot be solved in polynomial time, until **P = NP**. If a problem is proved to be NPC, there is no need to waste time on trying to find an efficient algorithm for it. Instead, we can focus on design approximation algorithm.

NP-Complete Problems

Following are some NP-Complete problems, for which no polynomial time algorithm is known.

* Determining whether a graph has a Hamiltonian cycle
* Determining whether a Boolean formula is satisfiable, etc

# NP-hard

Intuitively, these are the problems that are at least as hard as the NP-complete problems. Note that NP-hard problems do not have to be in NP, and they do not have to be decision problems.

The precise definition here is that a problem *X* is NP-hard, if there is an NP-complete problem *Y*, such that *Y* is reducible to *X* in polynomial time.

But since any NP-complete problem can be reduced to any other NP-complete problem in polynomial time, all NP-complete problems can be reduced to any NP-hard problem in polynomial time. Then, if there is a solution to one NP-hard problem in polynomial time, there is a solution to all NP problems in polynomial time.

**Example**

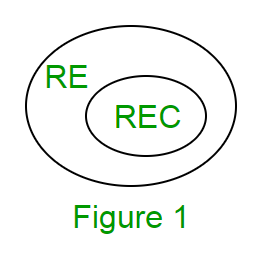
The halting problem is an NP-hard problem. This is the problem that given a program P and input I, will it halt? This is a decision problem but it is not in NP. It is clear that any NP-complete problem can be reduced to this one. As another example, any NP-complete problem is NP-hard.

# Recursive and 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= {anbncn|n>=1} is recursive because we can construct a turing machine which will move to final state if the string is of the form anbncn 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.  


**Difference**

The main difference is that in recursively enumerable language the machine halts for input strings which are in language L. but for input strings which are not in L, it may halt or may not halt.

When we come to recursive language it always halt whether it is accepted by the machine or not. if it accepted it reaches at (q accept) and halt. and if not accepted by the machine it directly reach (q halt)

**Recursive Language:**

Let L be a language over an input and if TM 'T' (Turing machine T) excepts every woed in L and rejects every word of L' it is called as recursive language.

**Example:**

1. String ends with '101'
2. Number of 'a'= number of 'b'

Accept(T) = L

Reject(T) = L'

Loop(T) = ϕϕ

ϕ = nullϕ = null

**Recursive enumeration language:**

Let L be a langauage over an input, and if TM 'T' excepts every word of L and rejects or loops every word in L' then it is called recursive enumeration language.

**Example:**

anbnanbn

Accept(T) = L

Reject(T) + Loop(T) = L'

Unisversal turing machine

A Universal Turing machine is a Turing machine that can run any algorithm but still uses an infinite paper tape as its storage or output medium. How the algorithm is described to the Universal Turing Machine or run is not specified; this is an abstract concept, not a real physical machine. Just as a Turing machine can solve any computing problem, a Universal Turing machine can be ‘programmed’ to do the same. This was conceived when Computer Science was still very much in its infancy and forms a very distant foundation for the programmable computers and devices we see today.

In [computer science](https://en.wikipedia.org/wiki/Computer_science), a **universal Turing machine** (**UTM**) is a [Turing machine](https://en.wikipedia.org/wiki/Turing_machine) that can simulate an arbitrary Turing machine on arbitrary input. The universal machine essentially achieves this by reading both the description of the machine to be simulated as well as the input thereof from its own tape.

To understand this definition, you need to know that the alphabet and the finite state machine are considered to be a part of the machine, while the tape contents isn’t.

So if two turning machines differ only by tape contents they are the same machine running on different inputs.

The universal machine is thus an alphabet and a FSM, which can be used to simulate any other machine by changing only the initial state of the tape.

# Theory of Computation | Decidability and Undecidability

Identifying languages (or problems\*) as decidable, undecidable or partially decidable is a very common question in GATE. With correct knowledge and ample experience, this question becomes very easy to solve.

Lets start with some definitions:-

**Decidable language** -A decision problem P is said to be decidable (i.e., have an algorithm)  
if the language L of all yes instances to P is decidable.  
Example- (I) (Acceptance problem for DFA) Given a DFA does it accept a given  
word?

(II) (Emptiness problem for DFA) Given a DFA does it accept any word?

(III) (Equivalence problem for DFA) Given two DFAs, do they accept the  
same language?

**Undecidable language** -– A decision problem P is said to be undecidable if the language L of all  
yes instances to P is not decidable or a language is undecidable if it is not decidable. An undecidable language maybe a partially decidable language or something else but not decidable. If a language is not even partially decidable , then there exists no Turing machine for that language.

**Partially decidable or Semi-Decidable Language** -– A decision problem P is said to be semi-decidable (i.e., have a semi-algorithm) if the language L of all yes instances to P is RE. A language ‘L’ is partially decidable if ‘L’ is a RE but not REC language.

**Recursive language(REC)** – A language ‘L’ is said to be recursive if there exists a Turing machine which will accept all the strings in ‘L’ and reject all the strings not in ‘L’. The Turing machine will halt every time and give an answer(accepted or rejected) for each and every string input. A language ‘L’ is decidable if it is a recursive language. All decidable languages are recursive languages and vice-versa.

**Recursively enumerable language(RE)** – A language ‘L’ is said to be a recursively enumerable language if there exists a Turing machine which will accept (and therefore halt) for all the input strings which are in ‘L’ but may or may not halt for all input strings which are not in ‘L’. By definition , all REC languages are also RE languages but not all RE languages are REC languages.

# MATHEMATICAL INDUCTION

[The principle of mathematical induction](https://www.themathpage.com/aPreCalc/mathematical-induction.htm#principle)

THE NATURAL NUMBERS are the counting numbers:  1, 2, 3, 4, etc. Mathematical induction is a technique for proving a statement -- a theorem, or a formula -- that is asserted about *every* natural number.

By "every", or "all," natural numbers, we mean any one that we name.

For example,

1 + 2 + 3 + .  .  .  + *n* = ½*n*(*n* + 1).

This asserts that the sum of consecutive numbers from 1 to *n* is given by the formula on the right.  We want to prove that this will be true for *n* = 1, *n* = 2, *n* = 3, and so on.  Now we can test the formula for any *given* number, say *n* = 3:

1 + 2 + 3 = ½**·** 3**·** 4 = 6

-- which is true.  It is also true for *n* = 4:

1 + 2 + 3 + 4 = ½**·** 4**·** 5 = 10.

But how are we to prove this rule for *every* value of *n*?

The method of proof is the following. We show that *if* the statement -- the rule -- is true for any specific number *k* (e.g. 104), then it will also be true for its successor, *k* + 1 (e.g. 105). We then show that the statement will be true for 1. It then follows that the statement will be true for 2. Therefore it will be true for 3. It will be true for any natural number we name.

This is called the principle of mathematical induction.

If

1. When a statement is true for natural number n=k, then it will also be true for its successor n=k+1

2. The statement is true for n=1

Then the statement will be true for every natural numbers n.