Countability, Computability and Decidability

We saw the machines, languages and grammar and we can also see that each smaller language is a proper subset of the circle that encapsulates it. To see if CSL is a proper subset of Recursive language, we have mathematical proofs but no concrete example of a language. Same goes for when we need to show that Recursive language is a proper subset of Recursively Enumerable Languages.

Now a general question we may ask: Is Turing Machine capable of accepting any language? Or for any language is there a Turing machine accepting it?

This is tough to answer directly, but we are going to use Set Theory for an indirect proof.

We are going to take a set of all the languages and another set of all the Turing machines. Both will be infinite, but we are going to prove that the set of TMs will have a smaller cardinality than the set of languages. Therefore, not all of the languages will have an associated TM.

Set → Finite

→ Infinite → Countable

→ un-countable

Countable sets: A set 'S' is said to be countable if all the elements of the set can be put in one-to-one correspondence with the set of natural numbers. The natural number corresponding to the elements of the set in question is called *index* of the element.

Uncountable sets: A set is uncountable, if it is infinite and not countable.

Example of countable: Set of all even numbers, set of all odd numbers.

Example of uncountable: Set of all real numbers

Alternative definition of countability

A set is said to be countable if there exists an enumeration method using which all the elements of the set can be generated and for any particular element, it takes only finite number of steps to generate it. The finite number of steps taken to generate the element can be as its index and hence mapping into natural numbers.

Let's take some examples:

Example 1: $\frac{p}{q} : p, q \in Z_f$ or set of all quotients p/q where p and q belong to set of all positive integers.

Set: {1/1, 1/3, 1/5, 1/6, ….} now we need to find enumeration method.

If we take 1/1, ½, 1/3, ¼ … etc then 2/1 will never be able to reach. Wrong enumeration

Reciprocal will also be invalid enumeration.

Go in the increasing order of p and q and arrange them.

1/1, ½, 2/1, 1/3, 2/2, 3/1, … then we can give index.

Therefore, this set is countable.

Example 2: $\Sigma^*: \Sigma = \{a, b\}$ or set of all the strings over a finite alphabet.

Σ = {ε, a, b, aa, bb, aabb, ….}

Find the enumeration method: Follow the dictionary order? Wrong enumeration. 'b' will never come

Enumeration method: Arrange in increasing length.

So this set is countable as an element can be reached in finite number of steps.

1. *Note: every infinite set of strings from finite set of alphabets is countable.*
2. *Note: This method of arranging in alphabetical order sorting in the increasing string length sorting is called* ***Proper Order.***

Property: Every subset of a countable set is either finite or countable.

Now we know that Σ^* is countable. A language is a subset of Σ^*. Therefore, every language is countable. It can be finite or countable.

Example 3: Set of all Turing machines are countable. (Important)

Let's take $\Sigma = \{0,1\}$ then $\Sigma^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, \dots\}$ and we know that every Turing machine can be encoded as a string of 0's and 1's (Universal Turing Machine)

1. Now Σ^* is countable.
2. Therefore, let's say set of all Turing Machines is S. Then $S \subseteq \Sigma^*$
3. Every subset of countable set either finite or countable.

Therefore, using 1, 2, and 3, the set of all TMs S is countable.

Implications of the fact that set of all Turing machines is countable

1. Since the set of Turing machines are countable, the set of all Recursively Enumerable Languages is also countable.
2. Using 1 all Recursive Languages are countable.
3. Using 2: all Context Sensitive Languages are countable
4. Using 3: all Context Free Languages are countable.
5. Using 4: all Regular Languages are countable.

Using 2, 3, 4, 5 above we can say that the set of all LBA, set of all PDA (deterministic and non-deterministic), and set of all FA (deterministic and non-deterministic) are countable.

Diagonalization method to prove that set of all languages are uncountable: We have $\Sigma = \{a, b\}$, Σ^* is countable. Σ^* is the set of all possible strings on 'a' and 'b'. Subset of Σ^* is a language. Now the power set of Σ^* is the set of all the subsets of Σ^* which will be 2^{Σ^*}. Let's see if 2^{Σ^*} is countable or not.

Statement: If a set 'S' is countably infinite, 2^S is uncountable.
We have to prove above statement.

Σ^* = {ε, a, b, aa, ab, ba, bb, aaa, …}

L = {a, aa, bb} Representation in Σ^* = {0, 1, 0, 1, 0, 0, 1, 0, …}

So, I can represent every language as a string of 0's and 1's from Σ^*.

Assumption : Let us assume that 2^{Σ^*} is countable.

Work: If any set S is countable, it has an index and depending on that index I can put all the elements in a table which looks like this.

Index	Representation of an element in Σ^*
1	0100101…
2	1010010…
3	0101010…
4	0101010…
5	1010001…
6	0111101…
7	1001010…

Now if an element belongs to 2^{Σ^*} then that element should be in a table like above corresponding to some index value in $\mathbb{N}$.

Now we have to look at the diagonal values of the representation of elements in the above table (marked in red). The values are: 0001000…

Diagonal Values: 0001000

Now we need to complement the values: 1110111…

Now these complemented values will differ with all of the languages in at least one bit. These bit (0 and 1) are taken from the list of languages. The diagonal elements when complemented will give a new language which will always be different from all the languages in this table. Therefore we can't reach this new language through an enumeration or can't give a natural numbered one-to-one correspondence to it.

Thus, our assumption is wrong and the set 2^{Σ^*} is not countable.

Verdict: Set 2^{Σ^*} is not countable or uncountable infinite set.

Now we already saw that the set of all Turing machines is countable. This means that the cardinality of that set is less than the cardinality of the set 2^{Σ^*}.

Therefore, for all the languages possible, there will not be a one-to-one Turing machine map possible. This means that there must be some languages which are not Recursively Enumerable.

Let's see some properties, may be asked (remember them):

1. If S1 and S2 are countable sets, then S1 U S2 is also countable and S1xS2 is also countable. (run the enumeration process of S1 and S2 alternatively to get enumeration for S1 U S2. Therefore, this statement is correct, this can be extended to finite number of sets S1 S2 S3..Sn).

2. The Cartesian product of finite number of countable sets is countable
   We can enumerate on S1xS2 using the indices values.
   Ex: S1 = {a1, a2, a3, …} (it is countable (given) so there are indices like 1, 2, 3… for the elements) 1 2 3
   S2 = {b1, b2, b3,… }(it is countable (given) so there are indices like 1, 2, 3… for the elements) 1 2 3

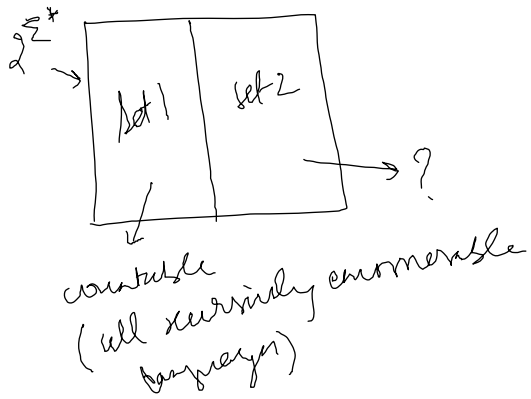
   Now S1xS2 is countable when we have index value for each of the element of this new set. So I'm gonna add the indices and see which elements can come in this new set.

   for a sum of 2 indices possible: (1, 1) : ordered pairs possible (a1, b1)
   for a sum of 3 indices possible: (1,2), (2,1): ordered pairs possible: (a1, b2) (a2, b1) etc.

   S1xS2 = {(a1,b1), (a1, b2), (b2, a1),… } (which is countable)

3. The set of all languages that are not recursively enumerable is uncountable.

   We know that set 2^{Σ^*} is uncountable. Now let's divide this set into languages which are RE and languages which are not like below.



Now we know that Set 1 is countable because Turing machines are countable. Let us assume set 2 is countable. Then it means:

$$S_1 \cup S_2 = 2^{\Sigma^*}$$

This means 2^{Σ^*} should be countable is S1 and S2 are both countable. Therefore, out assumption is wrong.

⇨ S2 is not countable. (Proved)

Computability and Decidability

Computability and decidability are same in the sense, both of them talk about if there exists an algorithm or not. Before going ahead, let's see what an algorithm is and what is the difference between an algorithm and a Turing machine.

An algorithm is a Turing machine that is supposed to halt. So for a problem if there is an algorithm it is as good as saying that for that problem there is a Turing machine that is supposed to halt. But if there is a Turing machine it doesn't mean that is an algorithm. A TM that is an algorithm is supposed to halt after finite number of steps. Therefore, only Halting TMs are considered as algorithms.

Now we know that TMs are countable. HTMs are subset of TMs. Therefore, HTMs are countable. And this Algorithms are also countable.

Now, set of all languages is uncountable, every problem can be compared to a language. This means that the number of problems will be uncountable and number of algorithms will be countable. Therefore, there are some problems for which there will be no algorithm.

Computability and Decidability are very identical but with a minor difference.

Computability:

Let us take a function $f(n) = n^2 + 1$. The function will have a domain and a range. If I give this function some value 'n', it is going to give out a value by computing $n^2 + 1$.

If there is an algorithm to compute the function, such a function is called computable. An algorithm is a HTM we know that, so we can say:

If there exists a Turing Machine and we give an input (n) to its tape and it gives the output $f(n)$ on the same tape and then it halts for every input 'n' given. Such a function is called computable.

Definition: If there is a function defined on a domain and there is a Turing Machine which will produce the output on the tape given the input on the tape and halts for every input chosen from the domain. Such a function is computable.

Decidability: Let us take a problem (a statement which has true or false as the answer).

Example : Problem: If 'n' is prime number

Domain: Set of all natural numbers (chosen by us)

Answer to be determined: Yes or No

Note: Therefore, the only difference between Computability and Decidability is that we talk about the function in computability and we talk about the problem in Decidability.

Definition: If there is a problem for which there exists an algorithm/halting Turing machine and this TM takes anything from the domain and decides if the problem has Yes or No as the answer followed by halting. Then that problem is decidable.

Note: A large domain can be collectively undecidable but an instance of the set will always be decidable. ***A common domain may be decidable or undecidable*****. Because there will be a Turing machine that will say Yes or No for a particular instance but may not find such decidability for a large domain.**

Example: Problem: Given grammar 'G' is ambiguous

Domain: Set of all Context Free grammar.

This problem is undecidable. But if we take an instance of domain, let's say, Grammar 'G1' then it is a decidable problem.

In TOC you will get questions on decidability. Let's see some of them. We use an indirect method to answer decidability. To say a problem is undecidable or decidable, we talk about reducibility.

Let's say there is a problem P_1 and there is a conversion algorithm or HTM that converts it to P_2. Now if P_2 has an "algorithm" or halting Turing machine then P_1 is decidable. This conversion of one problem to another problem is called *reducibility.*

1. Therefore, if P_2 is decidable, P_1 is decidable.
2. If it is already proved that problem P1 is undecidable, then the problem P2 must be undecidable.

Starting point for the theory of undecidability is Halting Problem.

Intuition: Take the halting problem -> reduce it to some problem P1 P2 P3 etc. All of them will be undecidable.

TURING MACHINE HALTING PROBLEM

This is the most interesting problem in computer science. This is the basic point which is used to prove that other problems are undecidable.

Statement: *Given the description of a Turing machine 'm' and an input 'w', does 'm' when started with 'w' as its input eventually halts?*

The undecidability of the above statement is proved using the theorem below.

Theorem: *If the halting problem were decidable, then every Recursively Enumerable language would be Recursive. Therefore, the halting problem is undecidable.*

It means that if I take a TM 'm' and string 'w' and give this (m, w) to an algorithm which says that 'm' halts on 'w' (i.e. halting problem is decidable) then the TM 'm' on 'w' should give Yes or No when the membership algorithm is tested on 'm' and 'w'. This means that there will always be a concrete Yes or No answer from all of the TM in the set of all TMs. Therefore, all the languages which are Recursively Enumerable will become Recursive. But there exists a proof that atleast one Recursively Enumerable language is not recursive. A contradiction has occurred. So, the halting problem is not decidable.

SOME UNDECIDABLE PROBLEMS BASED ON TURING MACHINE HALTING PROBLEM

Let's see some problems which can be converted from Halting Problem of Turing Machine. Based on that we can say that the problem is undecidable or not.

Problem 1: The state entry problem is given a Turing machine M, a state $q \in Q$ and $w \in \Sigma^*$. Decide whether or not the state 'q' is ever entered when M is applied to 'w'.

Proof: Using this statement I can make the halting problem decidable, so this statement or problem is undecidable.

Explanation: Let's say we have a Turing machine. A TM halts on a dead configuration. So for every dead configuration state I can transition that state to a new state 'q' which will be the new dead configuration. Therefore, this problem becomes decidable. But if this is the case, then the general halting problem becomes decidable too. Therefore, this problem is *undecidable* by contradiction.

Problem 2: Given a TM 'M', whether or not M halts if started with a blank tape.

Answer: This problem is undecidable which can be proved by reducing the halting problem to this problem.

Note: Almost any problem related to Recursively Enumerable languages is undecidable. The reason is the unavailability of membership algorithm.

(We will see the undecidable RE based questions in a table later).

POST CORRESPONDENCE PROBLEM

In context free grammars we have few problems like ambiguity problem which is undecidable. So, directly taking that halting problem and reducing it to see decidability of an ambiguity problem is somewhat complex. So, people have taken the halting problem and converted it to the *post correspondence problem.* This post correspondence problem is converted to ambiguity problem. Using this it is possible to show that the ambiguity problem of context free languages is undecidable. This becomes a kind of an intermediate step.

Problem Statement: Given two sequences of 'n' strings on some alphabet Σ, sat A = $w_1 w_2 \dots w_n$ and B = $v_1 v_2 \dots v_n$, we say there exists a PC-solution for pair (A, B) if there is a non-empty sequence of integers i, j, …k, such that

$$w_i w_j \dots w_k = v_i v_j \dots v_k$$

PC problem is to devise an algorithm that will tell us for ant (A,B), whether or not there exist a PC-solution.

THIS PROBLEM IS UNDECIDABLE and can be extended to ambiguity problem of context free grammars.

COMPLEXITY CLASSES

We learnt that a problem is decidable if there exists an algorithm/Turing machine for it. Now, when I say "algorithm" we are not actually interested in all the algorithms. There will be some algorithms that will

not be applicable as programs in practice as they take lots of resources like CPU time and memory. We are interested in finding out the algorithm that is feasible.

We have learnt previously that modifications to Turing Machines does not increase the power. Therefore the set of languages accepted by them remains same. But, some modifications decrease the time complexity of the calculation. Example will be a Turing machine with one tape and a Multi-tape Turing machine when accepting a language: $L = \{a^n b^n : n \geq 1\}$. A TM with one tape will accept the language in $O(n^2)$ and a multi-tape TM will accept the language in $O(n)$.

We know that all the problems can't have a Turing Machine. Within the set of problems that can be solved, we divide them into two classes depending on if the problem has a deterministic Turing machine or a Non-deterministic Turing machine as it's acceptor.

Deterministic Turing machine problems fall in ***P-class*** and non-deterministic Turing machine problems fall in ***NP-class.*** We are interested in P-class problems generally. If we have an NP-class problem, we don't use the algorithm in practice. Instead we use heuristics for that.

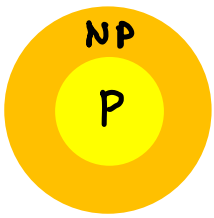
P-class: The set of all languages/problems that are accepted by some deterministic Turing machine in polynomial time.

This deterministic TM can use any number of tapes (any number of data structures) in a practical computer.

Polynomial time: $O(n^k)$, k can be any number

NP-class: The set of all languages/problems that are accepted by some non-deterministic Turing machine in polynomial time.

Note: For every problem that is solved by deterministic Turing machine can be solved by non-deterministic Turing machine. Therefore, we can draw the following:



Now the question comes, is P = NP (open question right now). We know that all the Turing Machines are equivalent in power. But when a non-deterministic Turing machine is converted to an equivalent deterministic then the language/problem will maybe accepted by it in *exponential* time instead of polynomial time. Both the machines are not calculating the same problem in same time.

DECIDABILITY PROBLEMS TABLE

Problem	**RL**	**DCFL**	**CFL**	**CSL**	**Rec**	**REL**
Does $w \in L$? (membership problem)	**D**	**D**	**D**	**D**	**D**	**UD**
Is $L = \phi$? (Emptiness problem)	**D**	**D**	**D**	**UD**	**UD**	**UD**
Is $L_1 = L_2$? (Equality Problem)	**D**	**UD**	**UD**	**UD**	**UD**	**UD**
Is $L = \Sigma^*$? (Completeness problem)	**D**	**UD**	**UD**	**UD**	**UD**	**UD**
Is $L_1 \subseteq L_2$ (Subset problem)	**D**	**UD**	**UD**	**UD**	**UD**	**UD**
Is $L_1 \cap L_2 = \phi$	**D**	**UD**	**UD**	**UD**	**UD**	**UD**
Is L finite or not? (finiteness problem)	**D**	**D**	**D**	**UD**	**UD**	**UD**
Is complement of 'L' a language of same type or not?	**D**	**D**	**UD**	**D**	**D**	**UD**
Is intersection of two languages of same type?	**D**	**UD**	**UD**	**UD**	**UD**	**UD**
Is L regular language?	**D**	**D**	**UD**	**UD**	**UD**	**UD**

1. For regular languages everything is decidable
2. For Recursively enumerable languages everything is undecidable.
3. For recursive language membership and complement is decidable
4. Memorize CFL (DCFL and CSL are not asked).

PROPERTIES OF CONTEXT FREE LANGUAGES

If the language is context free definitely there will be a context free grammar.

Union: If L_1 and L_2 are two context free languages, then their union $L_1 \cup L_2$ is also a context free language.

If S_1 is the start symbol for CFG_1 which generates language L_1 and S_2 is the start symbol for CFG_2 which generates language L_2, then the start symbol for context free grammar that generates $L_1 \cup L_2$ will look like:

$$S \rightarrow S_1 \,/\, S_2$$

Concatenation: If L_1 and L_2 are two context free languages, then their concatenation $L_1 \cdot L_2$ is also a context free language.

If S_1 is the start symbol for CFG_1 which generates language L_1 and S_2 is the start symbol for CFG_2 which generates language L_2, then the start symbol for context free grammar that generates $L_1 \cdot L_2$ will look like:

$$S \rightarrow S_1 \cdot S_2$$

Closure: If L_1 is a context free language then L_1^* is also a context free language.

If S_1 is the start symbol for CFG_1 which generates language L_1 then for language L_1^* the start symbol for context free grammar will look like:

$$S \rightarrow S_1 S \; / \; \varepsilon$$

Therefore, context free languages are closed under union, concatenation and Kleene closure.

Intersection: If L_1 and L_2 are two context free languages, then their intersection $L_1 \cap L_2$ is NOT always a context free grammar.

Proof by counter example:

$$L_1 = \{a^n b^n c^m | n, m \geq 0\}$$

$$L_2 = \{a^m b^n c^n | m, n \geq 0\}$$

$$L_1 \cap L_2 = \{a^n b^n c^n | n \geq 0\}$$

Language L_1 and L_2 are context free but the language obtained through their intersection is not context free.

Complement: If L_1 is a context free language then the its complement $\overline{L_1}$ may not always be context free.

Proof by contradiction:

We know that language $L_1 \cap L_2$ is not context free.

Now we can write: $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$

Assume: Complement of a context free language is context free.

With this assumption $\overline{L_1}$ and $\overline{L_2}$ are context free and we know that union of two context free languages is context free, therefore, $\overline{\overline{L_1} \cup \overline{L_2}}$ should be context free always. But we know that $L_1 \cap L_2$ is not context free always and $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$

- ⇨ Our assumption is wrong.
- ⇨ Hence, context free languages are not closed under complement.

Context free languages are not closed under intersection and complement.

DECIDABLE PROBLEMS ON CONTEXT FREE LANGUAGES

Membership Problem (Decidable): In context free languages membership problem is decidable. This means that given a string we can decide if it belongs to the language or not. CYK is the membership algorithm used.

Emptiness Problem (Decidable): Given a context free language L, is it empty or not?

How to decide: Take the CFG for the language and simplify it. In the simplified grammar if you found that the start symbol S is becoming useless, this means that we cannot generate anything and hence the language will be empty.

Finiteness Problem (Decidable): Given a context free language L, is it finite or not?

How to decide: Take the CFG for the language and simplify it and in the resulting grammar you try to draw the dependency graph.

Example: Let's say we got the following simplified language:

$$(1)\quad S \rightarrow AB$$

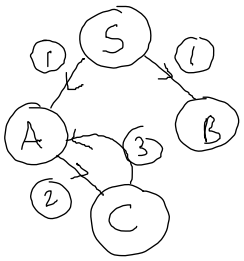
$$(2)\quad A \rightarrow aC \;/\; a$$

$$(3)\quad C \rightarrow aA \;/\; b$$

$$(4)\quad B \rightarrow a$$

The dependency graph will have the variables as nodes:

Dependency graph:



Now, if there is a cycle in dependency graph, then we can say that the language is infinite otherwise it is finite. This is how we are going to *decide* the Finiteness Problem in context free language.

PROPERTIES OF REGULAR LANGUAGES

1. **Union:** Regular languages are closed under union.
   Proof: Let's say two languages L_1 and L_2 are regular. Then their union $L_1 \cup L_2$ is also regular as the regular expressions, say R_1 and R_2 for the languages L_1 and L_2 can take representation by $R_1 + R_2$ for language L1 U L2.

2. **Concatenation:** Regular languages are closed under concatenation.

Proof: Let's say two languages L_1 and L_2 are regular. Then their concatenation $L_1 . L_2$ is also regular as the regular expressions, say R_1 and R_2 for the languages L_1 and L_2 can take representation $R_1.R_2$ for the language $L_1 . L_2$.

3. **Kleene Closure:** Regular languages are closed under Kleene Closure *.
   Proof: Let's say a languages L_1 is regular. Then, it's Kleene Closure L_1* is also regular as the regular expression, say R_1 for the language L_1 can take representation R_1* for the language L_1* and R_1* will also be a regular expression.

4. **Complement:** Regular languages are closed under complementation.
   Proof: Let L1 be a regular language. Then $\overline{L_1} = \Sigma^* - L_1$. For L1 we will have a DFA and for $\overline{L_1}$ we will have a complement of the DFA which is also a DFA. And since DFAs are acceptors of the Regular Languages, therefore the complement $\overline{L_1}$ is also regular.

5. **Intersection:** Regular languages are closed under intersection.
   Proof: We can write $L_l \cap L_2$ as $\overline{\overline{L_1} \cup \overline{L_2}}$ and using the complement and union properties discussed above we can prove that $L_l \cap L_2$ is also regular. (*This is called proof by construction).*

6. **Difference:** Regular languages are closed under difference
   Proof: I can again use proof by construction to say that $L_1 - L_2 = L_1 \cap \overline{L_2}$ and use previous properties to conclude that $L_1 - L_2$ is also a regular language.

7. **Reversal:** Regular languages are closed under reversal.
   Proof: Take the DFA of the language L, reverse it. The reversed FA is going to take L^R as its language. Since we got a FA, the language it accepts is also regular.

8. **Homomorphism:** Regular languages are closed under homomorphism.
   - A homomorphism is a way to compare two groups for structural similarities. It is a function between two groups which preserves the group structure in each group. It's a tool to compare two groups for similarities.

9. **Inverse Homomorphism:** Regular languages are closed under inverse homomorphism
   The inverse of homomorphic image of a language L is:
   $$h^{-1}(L) = \left\{\frac{x}{h(x)}\right\} \; is \; in \; L$$
   For a string w, $h + (w) = \left\{\frac{x}{h(x)} \in L\right\}$

10. **Right Quotient:** Regular languages are closed under right quotient.
    Let L_1 and L_2 be languages on the same alphabet, then, the right quotient of L_1 on L_2 is defined as:
    $$\frac{L_1}{L_2} = \{x : xy \; \epsilon \; L1 \; for \; some \; y \; \epsilon \; L2\}$$
    Example: $L_1 = \{01,001,101,0001,1101\}$, $L_2 = \{01\}$ then $\frac{L_1}{L_2} = \{\varepsilon, 0,1,00,11\}$

You don't directly perform the right quotient operation on the regular expressions. Make the language first and then divide every string in L1 with string in L2.

11. **INIT operation:** Regular languages are closed under INIT operation.
    Example: L = {a, ab, aabb}
    Init means what are all the prefixes of a string.

    Init(L) = {ε, a, ε, a, ab, ε, a, aa, aab, aabb } = { ε, a , ab, aa, aab, aabb}
    To say the regular languages are closed under INIT, you create a DFA and set every state as the final state except the dead state.

12. **Substitution:** Regular languages are closed under substitution.
    It means that if for a language L, every symbol is replaced with *other language* (not a string of other language because that is homomorphism). Then the resulting language is also a Regular Language.

    Examples: Σ = {a, b}
    f(a) = 0*
    f(b) = 01*

    L = a + b*
    f(L) = 0* + (01*)*

13. **Regular languages are NOT CLOSED under infinite union of Regular Languages. This means there exists at least one example that shows that Regular Languages are not closed under infinite union.**
    Example: L1 = $\{a^1b^1\}$, therefore L1 U L2 U L3… => L = $\{a^nb^n : n \geq 1\}$
    L2 = $\{a^2b^2\}$
    L3 = $\{a^3b^3\}$
    …

    Language L is not regular.