Sifa Ngari – 141733

Felicia Kyalo – 145602

Edinah Kamau – 145491

Automata Theory Write-up Assignment

Question One

Turing machines were initially introduced by Alan Turing in his work published in 1936-1937. These machines serve as basic conceptual tools to explore and understand the boundaries and capabilities of computation. Turing machines exhibit equivalent computational capability to any established computational model, irrespective of whether the comparison is drawn in the context of language recognition or the computation of functions.

The standard Turing machine comprises three primary elements: a finite state control (FSC) represented by a set of states, a semi-infinite tape where the initial input is located, and a read/write head positioned at the left end of the tape where the input is placed. At each step, the Turing machine assesses the symbol being read by the head and its current state. It proceeds to modify the symbol on the tape, potentially shifts to a new state, and may choose to move the head either left or right by one cell.

Turing machines are employed in the realm of automata theory and the theory of computation to examine the characteristics of algorithms and ascertain the feasibility of solving specific problems using computers. They serve as a means to simulate algorithmic behavior and evaluate the computational complexity associated with them.

According to Hopcroft and Ullman, the Turing Machine is expressed as a 7-tuple (Q, T, B, ∑, δ, q0, F) where:

- **Q** represents a finite set of states
- **T** is a finite tape alphabet
- **B** is the blank tape symbol
- **∑** is a finite input alphabet

- **δ** is a transition function which maps Q × T → Q × T × {L, R}.
- **q0** is the initial state.
- **F** is the set of final accept states.

Question Two

Multi-tape vs Single tape Turing Machines

Multi-tape Turing Machines consist of multiple tapes, each accessible through a distinct head that has ability to move independently of the other heads. On the other hand, a single-tape Turing machine utilizes a solitary infinite tape, which is partitioned into cells. Its finite control is equipped with one read/write head that points to a specific cell on the tape. To illustrate the working of the two variations of the two machines, we can consider a two-tape and one-tape Turing Machine for a certain language L.

$$\text{Let L} = \{x \circ 2 + \circ x | x \in \{0, 1\} *\}.$$

Algorithm for Two-Tape Turing Machine

1. As long as the symbol on the first tape head is either 0 or 1, duplicate it onto the second tape while continuously advancing both heads to the right.
2. Upon encountering a symbol 2, return the second tape head to the starting position.
3. While observing symbols 2 on the first tape, advance the first tape head to the right.
4. Simultaneously move both tape heads together, comparing the symbols. If a match is consistently achieved and both heads reach the end simultaneously, accept the input; otherwise, reject it.

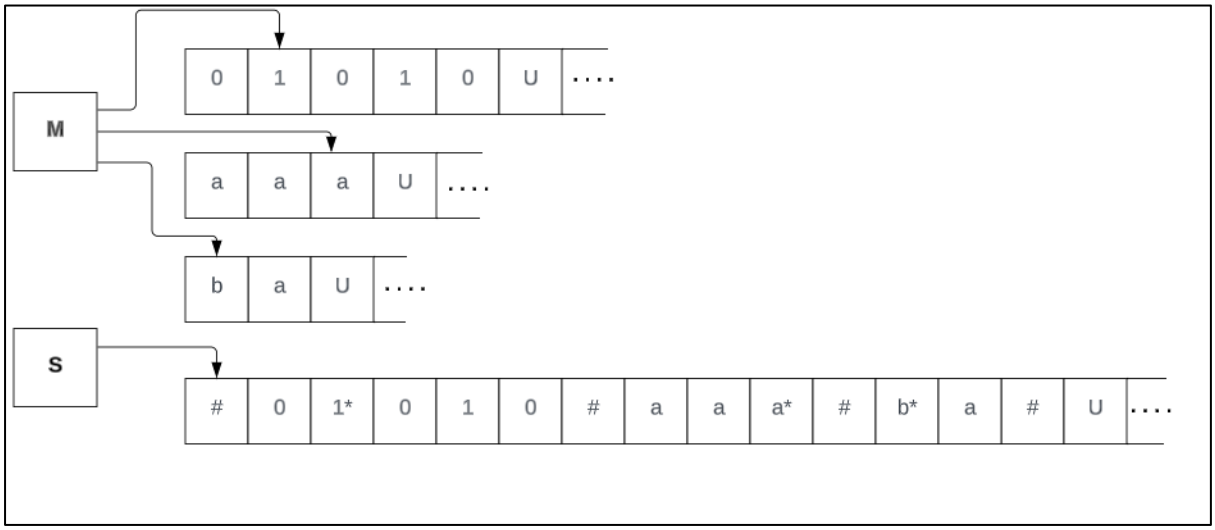
Algorithm for Single-Tape Turing Machine

1. To facilitate string comparison before and after the occurrence of the symbol 2, use a distinctive symbol to indicate the current position within the string.

2. Traverse the entire input by moving the tape head, simultaneously advancing the marker by one position to the right during each traversal while storing the read bit in the state.
3. If the strings consistently match and conclude simultaneously, the input is accepted; otherwise, it is rejected.
4. This process necessitates a total of $O(n^2)$ steps.

The time complexity of a single tape Turing Machine is equivalent to the number of steps it takes to reach a halting state. The space complexity is equivalent to the number of cells on the tape used during computation. The time and space complexity of a multi-tape Turing Machine depends on the number of steps and the maximum number of cells used across all tapes during computation. Suppose there is a function t(n) where t(n) is greater than or equal to n. In such cases, any multi-tape Turing machine that operates within t(n) time can be transformed into an equivalent single tape Turing machine that operates within $O(t(n)^2)$ time.

Diagrammatic Representation



Deterministic vs Non-deterministic Turing Machines

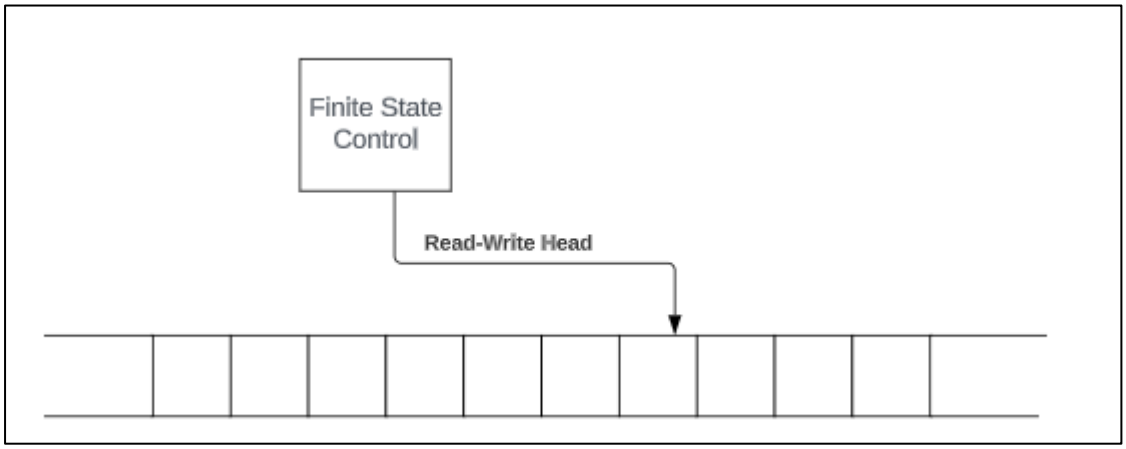
A Non-deterministic Turing machine shares similarities with a deterministic Turing machine but with one key distinction. While the transition function of a deterministic Turing machine only provides a single next step, the transition function of a Non-deterministic Turing machine offers a set of possible next steps. The computation of a deterministic Turing machine can be visualized as a linear sequence of configurations,

progressing one after the other, ultimately leading to an accept/reject state or potentially continuing indefinitely if the deterministic Turing machine enters a loop. In contrast, a Non-deterministic Turing machine's computation is represented by a configuration tree. If any branch within this tree leads to an accept state, the machine considers the input as accepted.

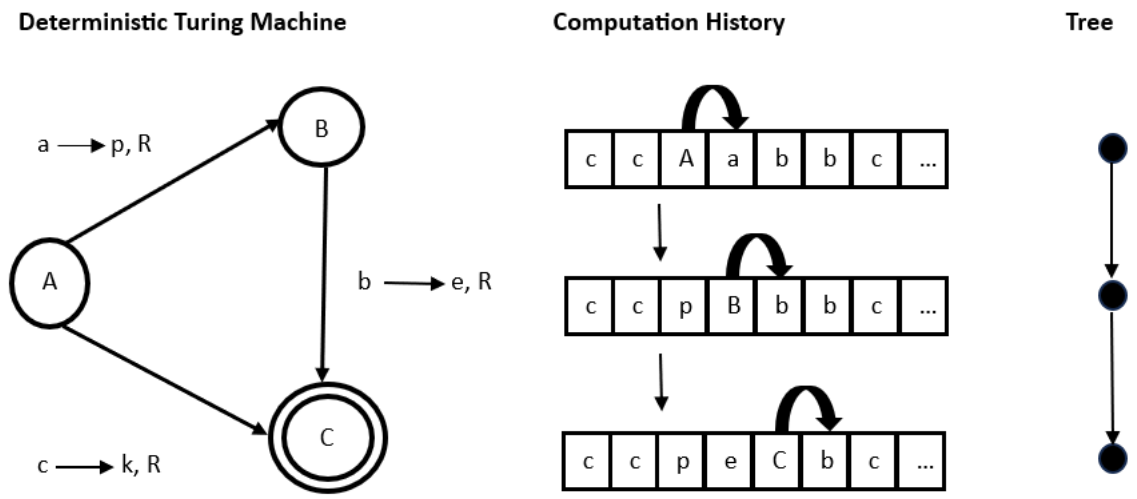
Time Complexities

Consider M, a deterministic Turing machine that halts on all inputs. The time complexity of M is represented by a function f: N → N, where f(n) denotes the maximum number of steps M takes on any input of length n. We describe M as running in time f(n). In most cases, we utilize n to denote the length of the input, denoted as |w|. On the other hand, consider R, a non-deterministic Turing machine for which all branches of its computation halt on all inputs. time complexity of R can be represented by a function f: N → N, where f(n) denotes the maximum number of steps that R utilizes on any branch of its computation for any input of length n.

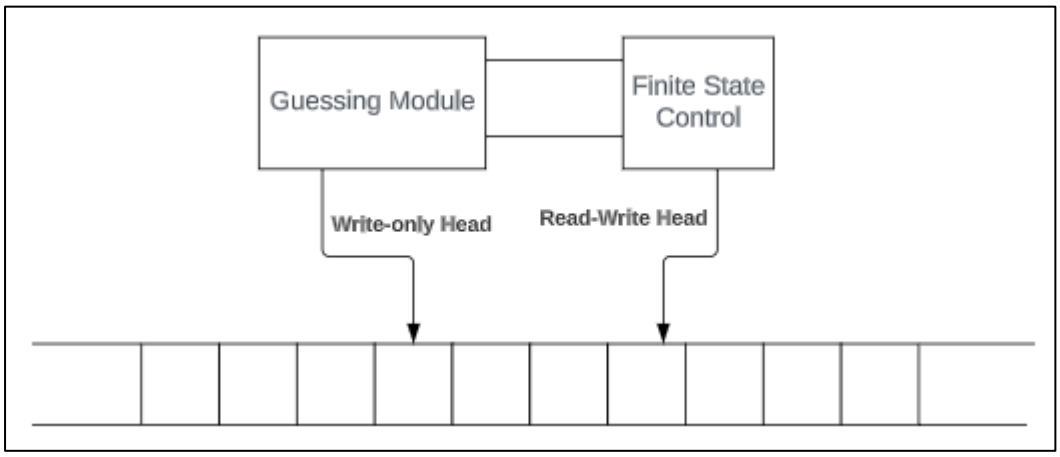
Deterministic Turing Machine



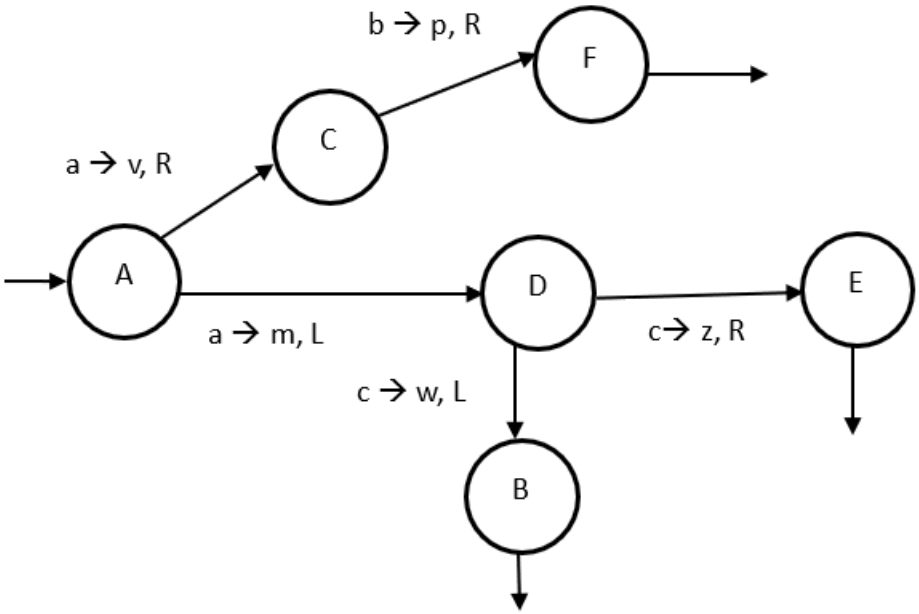
Sample of Deterministic Turing Machine transition diagram and computational history

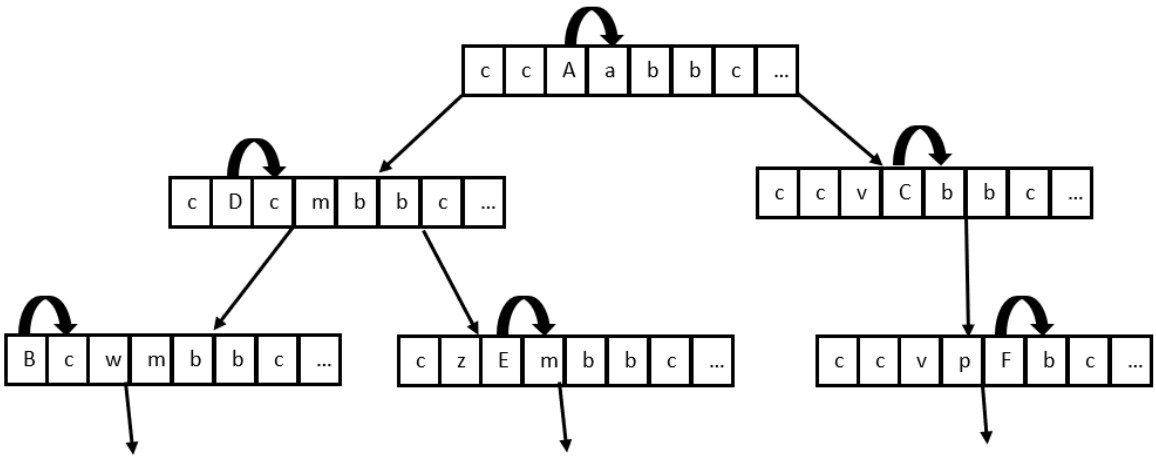


Non-Deterministic Turing Machine



Sample of Non-Deterministic Turing Machine transition diagram and computational history

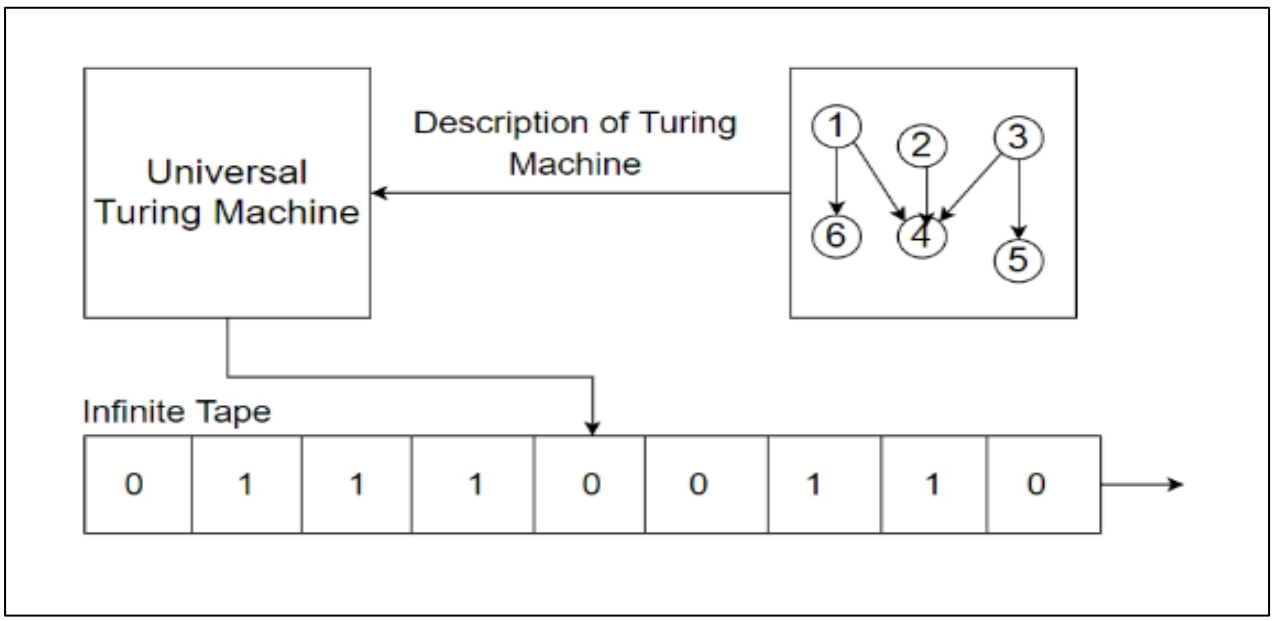




The Universal Turing Machine

A Universal Turing Machine has the ability to replicate the behavior of any given Turing machine, regardless of the specific input symbols involved. This implies that it is feasible to construct a single machine capable of performing calculations for any computable sequence. The input to a Universal Turing Machine consists of the description of machine M, which is placed on the tape and the input data for machine M. The machine can then simulate the behavior of machine M using the remaining content on the input tape. It directs the machine by altering its state based on the input, essentially functioning as a computer running another computer. As a consequence, a Universal Turing Machine has the capability to imitate the functionality of any other machine.

The existence of a universal Turing machine carries significant implications. Firstly, it enables the construction of a single computer (Turing machine) that can execute any desired algorithm (computer program). However, this universality also comes with a trade-off – certain significant computational problems are inherently unsolvable on a computer. For instance, determining whether a given computer program is a virus or not falls into this category. It is impossible to design an algorithm (Turing machine) that can effectively differentiate self-replicating programs from others.



Question 3

1. Theoretical foundation: It offers a theoretical basis for understanding the restrictions and potential of computation, establishing a framework for taking computational kinds and predictability into consideration.

2. Universal computing: It implies that a Turing machine or any computation model that is equal to it may compute any algorithmic process. This idea of universality enables researchers to examine and evaluate algorithm behavior in a consistent manner.

3. Complexity in Computation: It acts as a starting point for research into computational complexity. This makes it easier for academics to analyze the effectiveness and complexity of issue solving by categorizing problems into complexity classes like P (polynomial-time solvable problems) and NP (polynomial-time verified problems).

4. Algorithm Analysis and Design; It aids in algorithm analysis and design. It offers a measure for whether an algorithm can be put into practice and carried out by a computer. A valid algorithm is one that can be effectively calculated by a Turing computer.

5. Limits on Calculation: It aids in defining the computation's bounds. It follows that there exist unsolvable issues for which no algorithm can generate the right response given all inputs. These insolvable issues emphasize the limits of algorithmic procedures.

6. Establishes a meaningful context: Although it works with theoretical computational models, it also has applications in real-world settings. This means that, in theory, if given enough time and resources, any arithmetic assignment may be completed by a contemporary computer. The creation of programming languages, algorithms, and computer systems have all been influenced by this idea.

Question 4

Reducibility

A reduction is a computational technique that allows for the transformation of one problem into another, enabling one to utilize the solution of the second problem as a means to solve the first problem. Reducibility involves two problems, A and B. If problem A can be effectively reduced to problem B, it signifies that a solution to problem B can be leveraged to solve problem A. In other words, when A is reducible to B, the computational difficulty of solving problem A cannot surpass that of solving problem B. The concept has a number of significant applications in modern computation. For instance, reducibility can be used to test for undecidability.

The following theorems are utilized in determining undecidability via reductions:

- If problem A can be reduced to problem B, and problem B is decidable, then problem A is also decidable.
- Conversely, if problem A is undecidable and can be reduced to problem B, it follows that problem B is also undecidable.

Example of the Application of Reducibility

$HALT_{TM} = \{(M, w) \mid M \text{ is a TM and M halts on input w}\}$ is undecidable. Prove.

Applying the principle that "If A is undecidable and reducible to B, then B is undecidable," let's consider the scenario where R decides $HALT_{TM}$. We can construct another machine, S, to decide A_{TM}.

S is defined as follows:

1. Run machine R on the input (M, w).
2. If R rejects, reject the input for S.
3. If R accepts, simulate machine M on input w until it halts.
4. If M accepts, accept the input for S; if M rejects, reject the input for S.

Since A_{TM} is reducible to $HALT_{TM}$, $HALT_{TM}$ is undecidable.

P = NP

The class **NP** consists of all functions ***f*** such that, given any inputs ***x*** and ***y***, it is possible to verify, in polynomial time, whether or not ***f (x, y)*** is true.

The class **P** encompasses all functions ***f*** for which, given any input ***x***, it is possible to find a value ***y,*** in polynomial time, such that ***f (x, y)*** is true.

Polynomial time here implies that the time complexity is polynomial in terms of the number of bits used to encode the input for the given problem. The complexity class NP comprises a vast collection of practical problems. Notably, a subset of these problems is in class P, indicating that they can be effectively and efficiently solved in polynomial time. For example:

Is the Shortest Path problem in P or NP?

Given weighted graph ***G***, nodes ***s*** and ***t*** in ***G***, and value ***k***, is there a path ***p*** from ***s*** to ***t*** such ***weight(p)≤k***?

This problem belongs to the class NP because we can verify in polynomial time whether a given path ***p*** in graph ***G*** has a weight less than ***k***. However, the problem also belongs in class P because there exists a polynomial time algorithm, such as Dijkstra's algorithm or Bellman-Ford algorithm, that can effectively solve it.

Significance

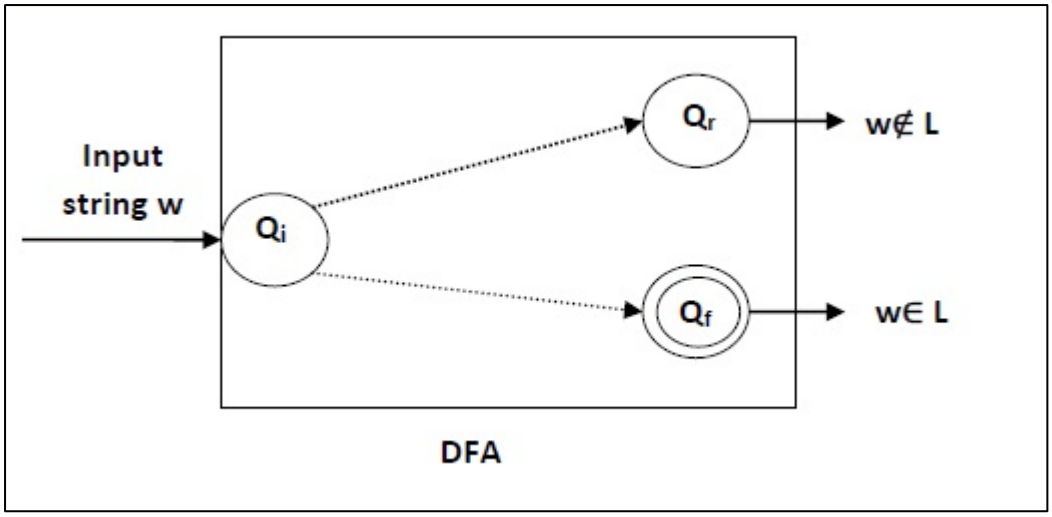
While polynomial-time algorithms have not been discovered for all the problems in NP, there is no definitive proof that such algorithms do not exist either.
The significance of this lies in the fact that many problems known to be in NP or NP-complete are problems that are of practical interest and relevance. Furthermore, the various NP-complete problems exhibit polynomial-time relationships among themselves. Therefore, if a feasible method is discovered for solving any one of these problems, it would imply the existence of feasible methods for solving all NP-complete problems

If the P=NP conjecture is proven by presenting a genuinely feasible algorithm for an NP-complete problem, the practical ramifications would be remarkable. Firstly, numerous optimization problems that hold great significance in various industries would become solvable. Secondly, the field of mathematics would undergo a transformative shift, as

computers would possess the ability to generate formal proofs for theorems that have reasonably sized proofs.

Decidability

A language is considered decidable if there exists a well-defined and efficient method to ascertain whether a given string belongs to that language or not. Essentially, a language is classified as decidable if there exists a Turing machine that can accept and halt on every input string w. For example, given a regular language **L** and string **w**, how can we check if **w** ∈ **L**? To determine this, one can utilize the deterministic finite automaton (DFA) that recognizes the language L and verify whether the input string w is accepted by the DFA.



An undecidable language refers to a language that cannot be determined by any algorithm or computer program. In simpler terms, it is not possible to create a program that can conclusively determine whether a given input string belongs to the language or not. The reason behind this is that undecidable languages possess inherent complexity, often requiring an infinite number of steps to ascertain membership. The Halting problem serves as a notable example of an undecidable language, wherein it explores the question of whether a given Turing machine will terminate on a specific input. Similarly, the Post Correspondence Problem offers another example, focusing on the inquiry of whether a given list of pairs of strings can be arranged in a specific configuration.

Decidability is important as it helps to demonstrate that certain problems are inherently unsolvable by algorithms or computer programs, regardless of their complexity or sophistication.

Dynamic Programming

Dynamic Programming is a computational methodology employed in programming that facilitates the effective resolution of a specific category of problems distinguished by the existence of overlapping subproblems and the possession of optimal substructure property. Dynamic programming harnesses the property of problems being divisible into subproblems that can be further decomposed into smaller subproblems. When these subproblems exhibit overlap, their solutions can be stored and referenced for future use. This methodology enhances the computational efficiency of the central processing unit (CPU) by avoiding redundant computations. An example of a problem that can be solved using dynamic programming is the Fibonacci series.

Fibonacci Series Algorithm

Assuming n represents the number of terms,

1. If the value of n ≤ 1, the output is 1.
2. Else, the output is the sum of the two preceding numbers.

Dynamic programming operates by storing the outcomes of subproblems to avoid recomputation when their solutions are needed. This strategy of storing subproblem values is known as memoization. By storing values in an array, we can save time by avoiding redundant computations for subproblems that have already been encountered. In the case of Fibonacci Series:

```
var m = map (0 → 0, 1 → 1)
function fib(n)
  if key n is not in map m
    m[n] = fib (n − 1) + fib (n − 2)
```

```
  return m[n]
```

Memoization in dynamic programming is a top-down approach. By reversing the direction of the algorithm, and starting from the base case and advancing towards the desired solution, dynamic programming can alternatively be executed in a bottom-up manner. This bottom-up approach avoids recursion and builds the solution iteratively. In the case of Fibonacci Series:

```
function fib(n)
  if n = 0
    return 0
  else
    var prevFib = 0, currFib = 1
    repeat n − 1 times
      var newFib = prevFib + currFib
      prevFib = currFib
      currFib = newFib
  return currentFib
```