A dark blue vertical bar runs down the left side of the page. A blue arrow points to the right from this bar, containing the date.

4/27/2023

# AUTOMATA THEORY & COMPUTABILITY

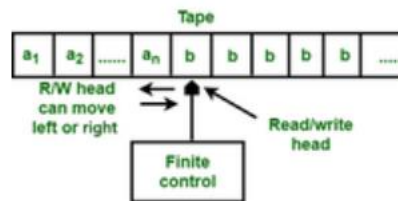
ASSIGNMENT TWO(WRITE UP)

Several thin, dark blue curved lines originate from the bottom left and sweep upwards and to the right, creating a sense of movement.

MARTIN MWANGI (SOLO STUDENT)  
137938 – ICS 3C

## Q 1 : Turing Machines

It is a mathematical model in which input is entered into cells along an infinite tape that has been separated into sections. A head reads the input tape in this device. The state of the Turing machine is kept in a state register. An input symbol is read, substituted by another symbol, its internal state is altered, and it advances from one cell to the right or left after that. The input string is accepted if the TM reaches the end state; else, it is denied.



An example of a turing machine

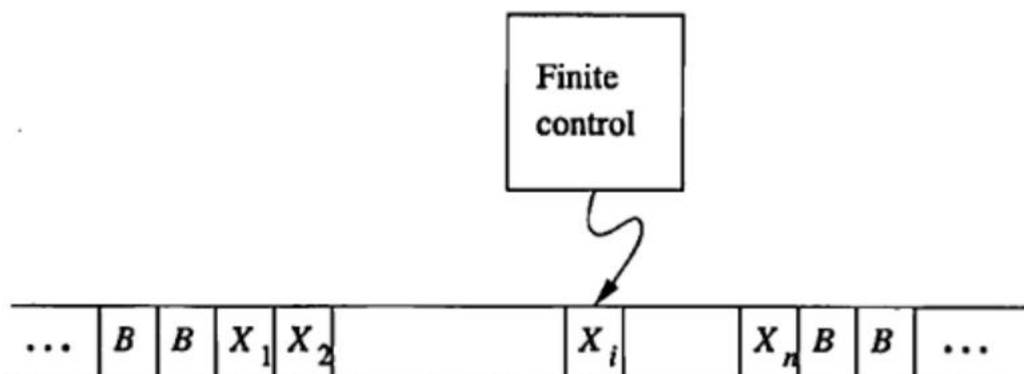
## Formal Description

Formally speaking, a TM can be represented as a 7-tuple  $(Q, \Sigma, \Gamma, \delta, q_0, B, F)$  where:

- $Q$ : The finite set of states of the finite control.
- $\Sigma$ : The finite set of input symbols.
- $\Gamma$ : The complete set of tape symbols;  $\Sigma$  is always a subset of  $\Gamma$ .
- $\delta$ : The transition function.
- $q_0$ : The start state, a member of  $Q$ , in which the finite control is found initially.
- $B$ : The blank symbol.
- $F$ : The set of final or accepting states, a subset of  $Q$ .

## Notation for the turing machine

We may visualize a turing machine as the figure shown below. The machine consists of a finite control, which can be in any of a finite set of states. There is a tape divided into squares or cells; each cell can hold any one of a finite number of symbols.



The input is initially recorded on the tape as a finite-length string of symbols selected from the input alphabet. The blank special symbol is what is initially present in every other tape cell, which runs infinitely to the left and right. The blank is not an input symbol; it is a tape symbol.

One of the tape cells has a tape head that is constantly there. They claim that cell is being scanned by the Turing machine. The tape head is initially in the input-holding cell on the left.

The tape symbol that is being read and the state of the finite control determine how the Turing machine moves. The Turing machine will in one motion;

1. Change state. The next state optionally may be the same as the current state.
2. Write a tape symbol in the cell scanned. This tape symbol replaces whatever symbol was in that cell. Optionally the symbol written may be the same as the symbol currently there.
3. Move the tape head left or right.

### **Instantaneous description(ID)**

ID of a Turing machine is a snapshot of the machine to describe the current situation

of the Turing machine. In order to describe formally what a Turing machine does, we need to develop a notation for configurations or instantaneous descriptions (ID's). Since a Turing machine, in principle, has an infinitely long tape, we might imagine that it is impossible to describe the configurations of a Turing machine concisely.

Therefore, there are an infinite number of cells in every ID's prefix and suffix that have never been visited. One of the limited input symbols or a blank must be present in each of these cells. Thus, we only display in an ID the cells that are between the leftmost and rightmost non-blanks. A specific number of blanks to the left or right of the non-blank area of the tape must also be included in the ID while the head is scanning one of the leading or trailing blanks.

In addition to representing the tape, we must represent the finite control and the tape-head position. To do so, we embed the state in the tape, and place it immediately to the left of the cell scanned.

### **The language of a Turing machine**

The input string is placed on the tape, and the tape head begins at the leftmost input symbol. If the Turing machine eventually enters an accepting state, then the input is accepted, and otherwise not. A TM accepts a language if it enters into a final state for any input string  $w$ .

A language is recursively enumerable (generated by Type-0 grammar) if it is accepted by a Turing

machine. A TM decides a language if it accepts it and enters into a rejecting state for any input not in the language. A language is recursive if it is decided by a Turing machine.

### **Types of Turing machine**

- Multi-tape turing machine
- Multi-head turing machines
- Multi-track turing machines
- Unambiguous Turing Machine
- Quantum Turing Machine
- Read-Only Turing Machine
- Probabilistic Turing Machine
- Semi-infinite turing machines
- Universal Turing Machine
- Alternating Turing machine
- Non-Deterministic Turing Machine

### **Q 2: Working of Turing Machines**

#### **a. Multi-tape vs Single tape Turing Machines.**

Multi-tape Turing machines have numerous tapes, and every tape is accessed by a different head. Independent of the other heads, each head may move. Initial input is just on tape 1, with the others being empty. At start, the input is recorded on only the first tape, leaving the remaining tapes empty. The machine then moves its heads while printing a symbol on each cassette and reading a series of symbols under its heads.

Let  $t(n)$  be a function, where  $t(n) > n$ . Then every  $t(n)$  time multi-tape Turing machine has an equivalent  $O(t^2(n))$  time single-tape Turing machine.

This means that for each step of  $(t(n))$  steps on  $M$ ,  $O(t(n))$  steps occur on  $S$ . Therefore, the running time of  $S$  is  $O(t^2(n))$ .

The time complexity of a multi-tape Turing machine and a single-tape Turing machine have a square difference.

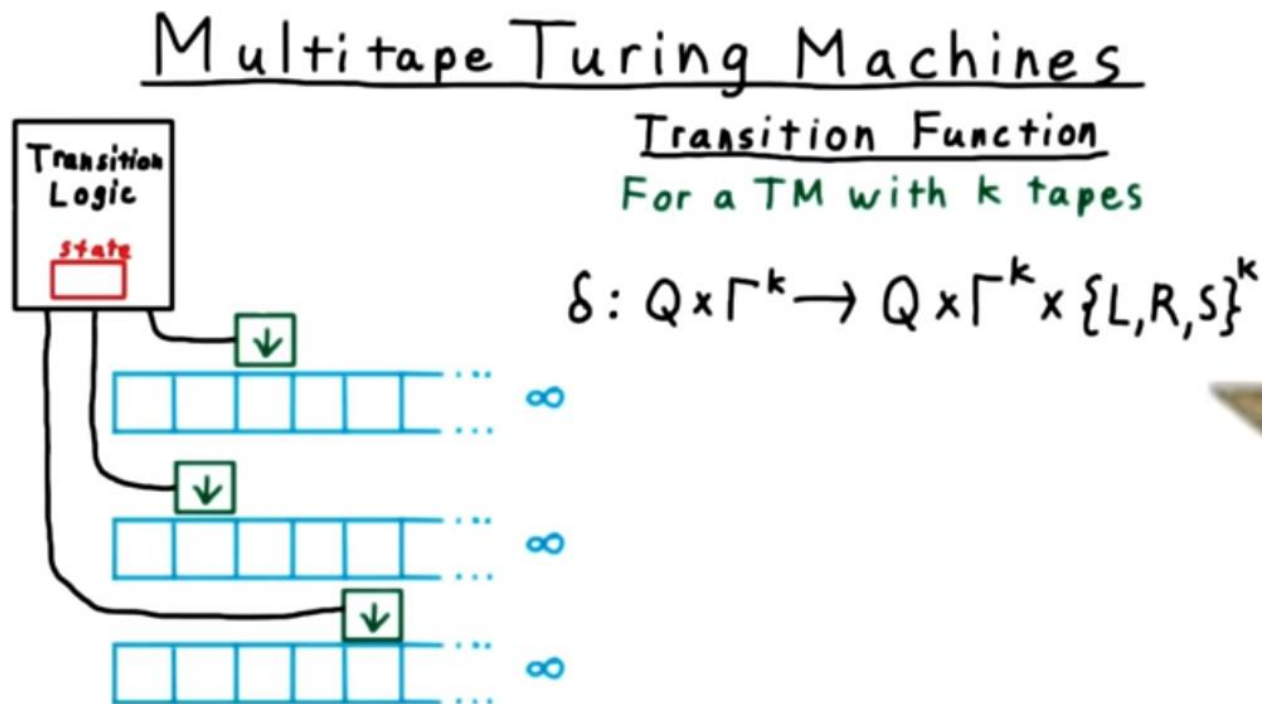


Fig 1.3. Multi-tape Turing Machine

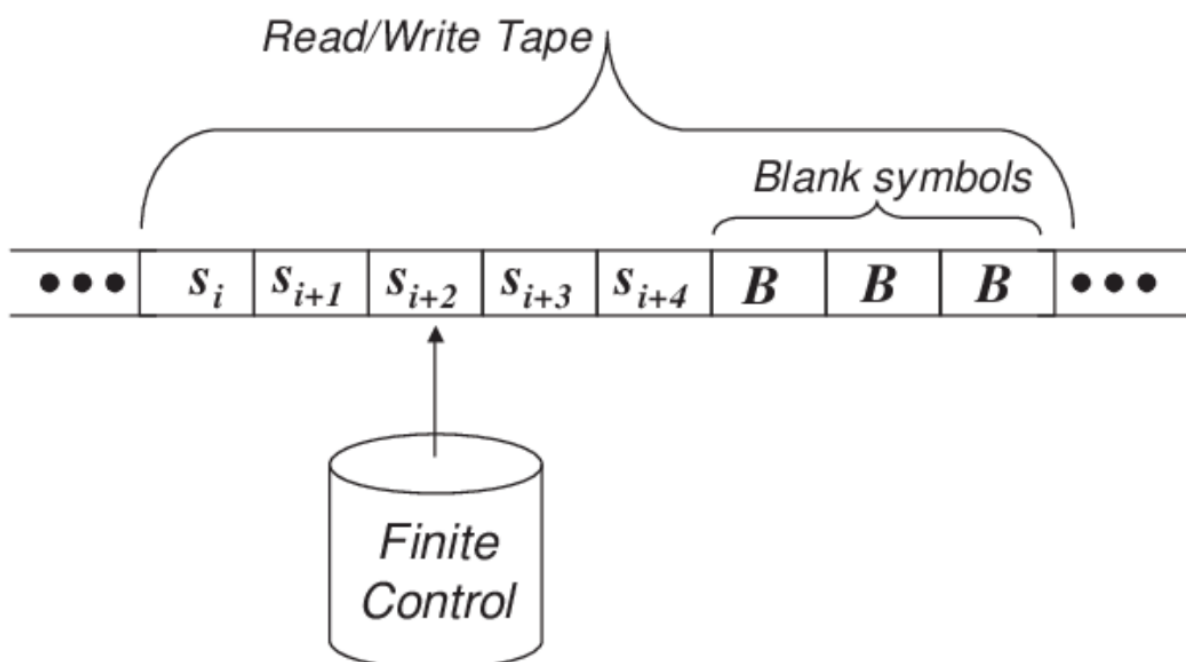


Fig 1.4. single-tape Turing Machine

b. **Deterministic vs Non-deterministic Turing Machines.**

**Deterministic:**

This machine has a finite state control, a read-write head and a two-way tape with unlimited sequence.

Program for a deterministic turing machine specifies the following

- A finite set of tape symbols (input symbols and a blank symbol)
- A finite set of states
- A transition function

In algorithmic analysis, a problem is considered to be of the P class if it can be solved in polynomial time by a deterministic one tape Turing machine.

**Non-Deterministic:**

The NDTM has a structure that is similar to the DTM, but it also includes a guessing module that is connected to a single write-only head.

The problem falls within the NP class if it can be solved in polynomial time by a nondeterministic Turing machine.

Let  $t(n)$  be a function, where  $t(n) > n$ . Then every  $t(n)$  time nondeterministic single-tape Turing machine has an equivalent  $2O(t(n))$  time deterministic single-tape Turing machine.

Skipping the proof for this, a simple yet understandable explanation for this is that the construction of a deterministic TM (D) to simulate the nondeterministic TM (N) results from searching N's computation tree.

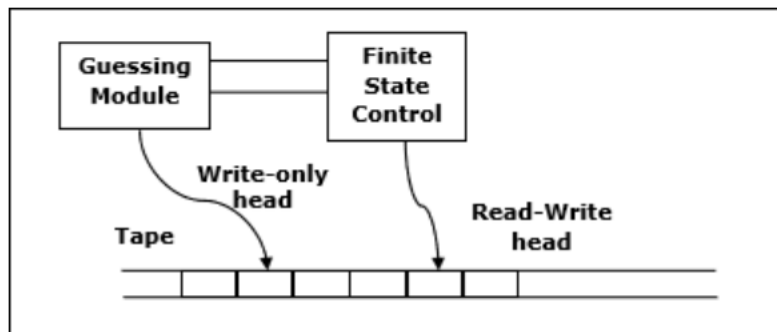
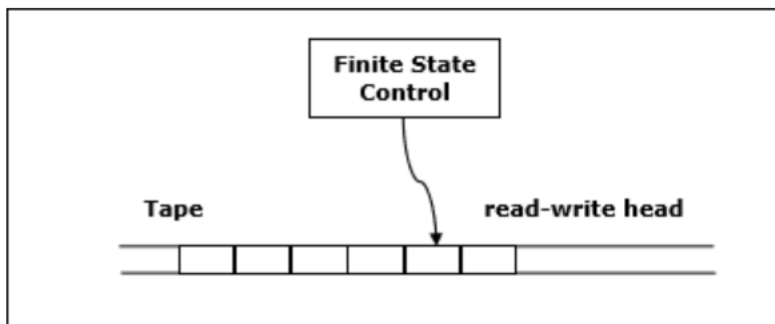
Even if a multitape TM is used, D still has the same time complexity. Because a conversion from a multitape TM to a single-tape TM at most squares the running time,

$$\sqrt{(2^{O(t(n))})} = 2^{O(\frac{1}{2}t(n))} = 2^{O(t(n))}$$

Polynomial differences in running time are considered to be small, whereas exponential differences are considered to be large. This is because of their respective growth rates.

It is for this reason that, for larger problems and larger inputs, the difference between using a single-tape TM and a multitape TM is negligible.

Furthermore, all reasonable deterministic computational models are polynomially equivalent. This means that any one of them can simulate another with only a polynomial increase in running time.

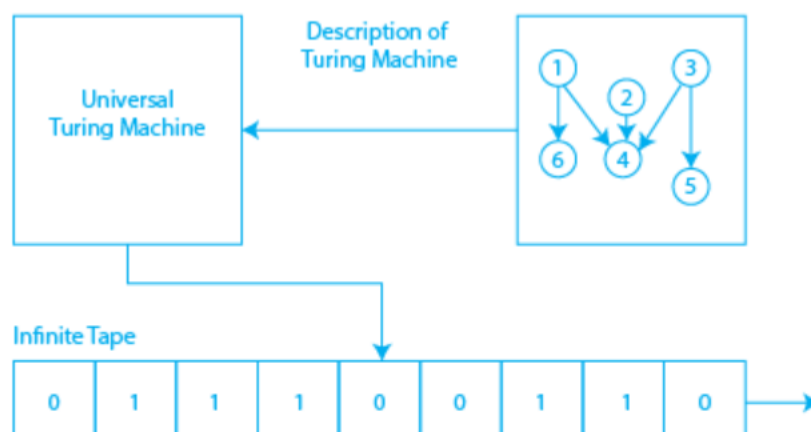


c. **The Universal Turing Machine.**

It is a type of "computer" that can carry out any algorithmic operation that a different Turing Machine is able to carry out. It does this by reading the description of another Turing machine from its input tape. The Universal Turing Machine can reproduce the states and transitions of the other Turing machine on its own tape thanks to the detailed descriptions of those states and transitions.

After reading the other Turing Machine's description, the Universal Turing Machine creates a new tape to imitate its actions. The Universal Turing Machine then reads the input tape again and uses the simulated Turing Machine to compute the required output. After writing the output to the output cassette, the Universal Turing Machine ends.

Universal Turing machine simulating has a time complexity of  $T(n)\log T(n)$ .



### **Q 3: Importance of the Church-Turing Thesis**

- The primary advantage of this thesis is that it enables us to relate formal mathematical theorems to practical computability problems. For instance, the real-world application of the thesis that the word problem for groups is Turing-undecidable is that no algorithm can resolve all instances of the word problem.
- The second advantage relates to mathematics particularly, and more especially, computability theory. A turing-machine program, or even a program in any real computing language, is almost never the first step in published demonstrations that anything is Turing-computable.

### **Q 4: Significance of the following concepts to Modern computation**

#### **a. Reducibility**

Reducibility is a way of comparing the difficulty of two problems. If there is an algorithm that can solve L by converting it into M and then solving M, then it is said that L is reducible to M.

For instance, the question of whether a graph is bipartite can be answered by asking whether it has an Eulerian circuit. This indicates that if we can figure out whether a graph has an Eulerian circuit, we can figure out whether a graph is bipartite as well.

Here is a code sample that illustrates the idea of reducibility:

```
def is_bipartite(graph):  
    if not graph:  
        return True  
    queue = [0]  
    colors = [None] * len(graph)  
    colors[0] = "red"  
    while queue:  
        vertex = queue.pop(0)  
        for neighbor in graph[vertex]:  
            if colors[neighbor] is None:  
                colors[neighbor] = "red" if colors[vertex] == "blue" else "blue"  
                queue.append(neighbor)  
            elif colors[neighbor] == colors[vertex]:  
                return False  
    return True
```



```

def has_eulerian_circuit(graph):
    in_degree = [0] * len(graph)

    for vertex in graph:
        for neighbor in graph[vertex]:
            in_degree[neighbor] += 1

    out_degree = [0] * len(graph)

    for vertex in graph:
        for neighbor in graph[vertex]:
            out_degree[vertex] += 1

    for vertex in range(len(graph)):
        if in_degree[vertex] != out_degree[vertex]:
            return False

    return True

```

The screenshot shows the Programiz Python Online Compiler interface. The main editor displays the following code:

```

1 def is_bipartite(graph):
2     if not graph:
3         return True
4     queue = [0]
5     colors = [None] * len(graph)
6     colors[0] = "red"
7     while queue:
8         vertex = queue.pop(0)
9         for neighbor in graph[vertex]:
10            if colors[neighbor] is None:
11                colors[neighbor] = "red" if colors[vertex] == "blue"
12                else "blue"
13            queue.append(neighbor)
14            elif colors[neighbor] == colors[vertex]:
15                return False
16        return True
17
18 def has_eulerian_circuit(graph):
19     in_degree = [0] * len(graph)
20     for vertex in graph:
21         for neighbor in graph[vertex]:

```

On the right side, the Shell output shows the result of running the code: `True`.

The `is_bipartite` function takes a graph as input and returns True if the graph is bipartite, False otherwise. The `has_eulerian_circuit` function takes a graph as input and returns True if the graph has an Eulerian circuit, False otherwise.

The `is_bipartite` function works by first checking if the graph is empty. If it is, then the graph is bipartite by definition. Otherwise, the function creates a queue and adds the first vertex to the queue. The function then iterates through the queue, checking if each vertex in the queue has a different color than its neighbors. If a vertex does not have a different color than its neighbors, then the graph is not bipartite. The function terminates when the queue is empty.

The `has_eulerian_circuit` function works by first counting the in-degree and out-degree of each vertex in the graph. If the in-degree of a vertex is not equal to the out-degree of the vertex, then the graph does not have an Eulerian circuit. Otherwise, the function iterates through the graph, checking if each vertex has a neighbor with the same color. If a vertex does not have a neighbor with the same color, then the graph does not have an Eulerian circuit. The function terminates when it has checked all of the vertices in the graph.

## **b. P = NP**

The significance of the  $P = NP$  problem is that it would have a major impact on the field of computer science. If  $P = NP$ , then it would mean that many problems that are currently considered to be intractable could be solved in polynomial time. This would have a major impact on fields such as cryptography, optimization, and artificial intelligence.

Here is a code sample

```
def is_prime(n):  
    if n <= 1:  
        return False  
  
    for i in range(2, int(n**0.5) + 1):  
        if n % i == 0:  
            return False  
  
    return True  
  
def is_in_set(n, set):  
    for x in set:  
        if x == n:  
            return True  
  
    return False  
  
def p_is_np():  
    set = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29}  
  
    for n in range(30, 100):  
        if not is_prime(n) and is_in_set(n, set):  
            return False  
  
    return True  
  
if __name__ == "__main__":
```

```
print(p_is_np())
```

A screenshot of a Python IDE with a dark theme. The editor shows a file named 'main.py' with 20 lines of Python code. The code defines three functions: 'is\_prime(n)' which checks if a number is prime by testing divisibility up to its square root; 'is\_in\_set(n, set)' which checks if a number is in a given set; and 'p\_is\_np()' which defines a set of numbers {2, 3, 5, 7, 11, 13, 17, 19, 23, 29} and iterates through numbers from 30 to 100, returning 'True' if a number is both prime and in the set. The IDE has a 'Run' button highlighted in blue. To the right of the code editor is a 'Shell' window showing the output 'True'.

This code sample defines a function to check if a number is prime, a function to check if a number is in a set, and a function to check if the  $P = NP$  problem is true. The code sample then runs the  $P = NP$  function on a range of numbers and prints the result.

The code sample above illustrates the difficulty of the  $P = NP$  problem. The function to check if a number is prime is relatively simple, but the function to check if the  $P = NP$  problem is true is much more complex. This is because the  $P = NP$  problem is a decision problem, which means that it must be able to answer the question "is this problem in NP?" in polynomial time. However, no one has yet been able to come up with an algorithm that can do this.

The  $P = NP$  problem is one of the most important unsolved problems in computer science. If it is ever solved, it would have a major impact on the field of computer science and many other fields, and prints the result.

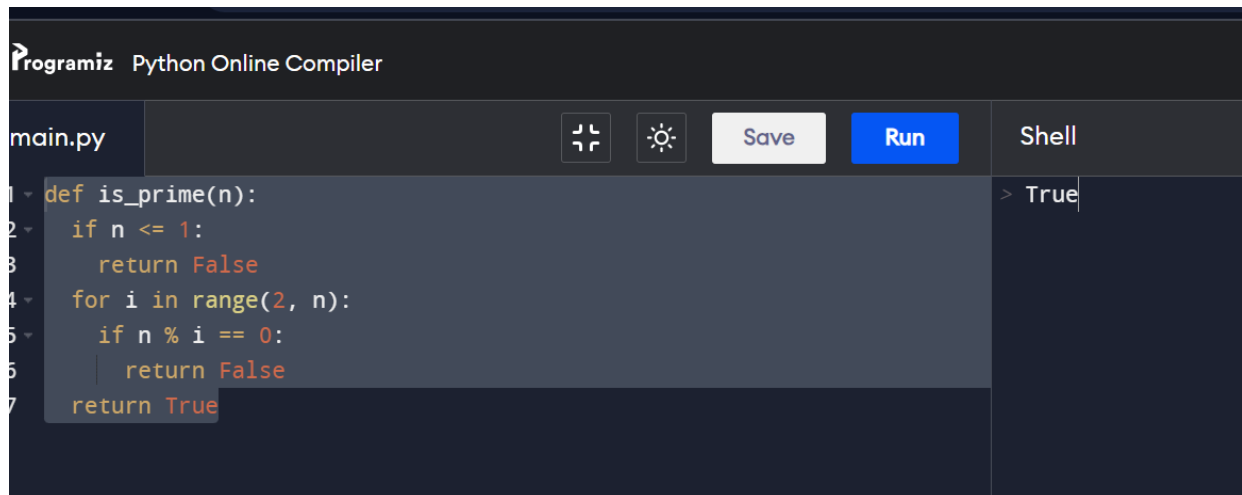
### c. Decidability

Decidability is a property of a decision problem that can be determined by an algorithm. In other words, a decision problem is decidable if there exists a Turing machine that can always halt and give a correct answer, given any input.

For example, the problem of determining whether a number is prime is decidable. The following Python code implements an algorithm for this problem:

```
def is_prime(n):  
    if n <= 1:
```

```
    return False
for i in range(2, n):
    if n % i == 0:
        return False
return True
```



The screenshot shows the Programiz Python Online Compiler interface. The editor contains the following Python code:

```
def is_prime(n):
    if n <= 1:
        return False
    for i in range(2, n):
        if n % i == 0:
            return False
    return True
```

The 'Run' button is highlighted in blue. The output shell on the right shows the result: `> True`.

#### d. Dynamic Programming

Dynamic programming is a powerful technique that can be used to solve a wide variety of problems. It is especially useful for problems that can be broken down into subproblems, and where the solutions to the subproblems can be reused. Dynamic programming is widely used in modern computation, and it is a key technique in many areas of computer science, such as artificial intelligence, operations research, and computational biology.

Here are some of the significance of dynamic programming to modern computation:

- Dynamic programming can be used to solve a wide variety of problems, including optimization problems, scheduling problems, and path-finding problems.
- Dynamic programming can be used to solve problems that are too large or complex to be solved using other methods.
- Dynamic programming can be used to find optimal solutions to problems.
- Dynamic programming can be used to find approximate solutions to problems that are too difficult to solve exactly.

Dynamic programming is a powerful tool that can be used to solve a wide variety of problems in modern computation. It is a valuable technique for anyone who works with computer science.

Here is a code sample that demonstrates how dynamic programming can be used to solve the Fibonacci

sequence problem:

```
def fibonacci(n):  
    if n == 0:  
        return 0  
    elif n == 1:  
        return 1  
    else:  
        return fibonacci(n - 1) + fibonacci(n - 2)
```

```
def main():  
    print(fibonacci(10))
```

```
if __name__ == "__main__":  
    main()
```

A screenshot of a code editor interface. The editor has a dark theme. At the top, there's a tab labeled 'main.py'. To the right of the tab are icons for a window manager (four squares), a sun/moon icon for theme toggling, and buttons for 'Save' and 'Run'. The code is written in Python and is color-coded: keywords like 'def', 'if', 'elif', 'else', 'return', and 'print' are in orange, while identifiers and literals are in white. The code is as follows:

```
1 def fibonacci(n):  
2     if n == 0:  
3         return 0  
4     elif n == 1:  
5         return 1  
6     else:  
7         return fibonacci(n - 1) + fibonacci(n - 2)  
8  
9  
10 def main():  
11     print(fibonacci(10))  
12  
13  
14 if __name__ == "__main__":  
15     main()
```

On the right side of the editor, there's a 'Shell' panel. It shows the output of the program: the number '55'. Below the output, there's a prompt character '>' followed by a vertical cursor line.

This code first checks if  $n$  is 0 or 1. If it is, then the Fibonacci number for  $n$  is simply 0 or 1, respectively. Otherwise, the code recursively calls itself to calculate the Fibonacci numbers for  $n - 1$  and  $n - 2$ . The results of these recursive calls are then added together to get the Fibonacci number for  $n$ .

This code can be improved by using dynamic programming. Dynamic programming works by storing the results of subproblems so that they do not need to be recalculated every time. In this case, we can store the Fibonacci numbers for all values of  $n$  up to  $n - 1$  in a table. Then, when we need to calculate the Fibonacci number for  $n$ , we can simply look up the value in the table.