THEORY OF COMPUTATION

Busy Beaver problem

PRIYANSHU DIXIT-BT23CSE003
SAKSHAM AGRAWAL-BT23CSE004
VARUN SINGH-BT23CSE009
YOGESH BHIVSANE-BT23CSE013
YASH VERMA-BT23CSE019

What is a Turing machine - The introduction

The **Turing Machine**, proposed by *Alan Turing* in 1936, is a mathematical model that defines the foundation of computation.

It consists of an **infinite tape**, a **read/write head**, and a **finite set of states** that guide how the machine reads symbols, writes new ones, moves left or right, and changes state.

Turing Machines help us understand **what can or cannot be computed** and lead to important questions such as whether every algorithm eventually **halts**.

The **Busy Beaver problem**, introduced by *Tibor Radó* in 1962, explores the limits of such machines.
A Busy Beaver is a special Turing Machine that writes the **maximum number of 1s** on a blank tape before halting.

For each number of states n, there exists a machine that halts after producing the most 1s possible — but for larger n, predicting whether it halts becomes **undecidable**, linking directly to the **Halting Problem**.

Our project simulates the **3-state Busy Beaver (BB3)** using **C++ programming**.

The simulation visualises how the machine:

Reads and writes symbols on a tape,
Moves its head left or right,
And halts after completing its operation.

For **three states**, the Busy Beaver is known to:
Halt after **21 steps**,
And produce **6 ones** on the tape.

This project demonstrates how simple rules can generate complex behavior and provides a visual understanding of **state transitions**, **halting conditions**, and the **limits of computation** in the Theory of Computation.

Relation Between The Halting Problem and The Busy Beaver Problem

- What the Halting Problem Says ?
  - The **Halting Problem**, proved undecidable by **Alan Turing (1936)**, states that:

    There is **no general algorithm** that can take as input a description of any program (or Turing machine) and its input, and correctly decide whether it will eventually halt or run forever.
  - In other words — there's no universal way to predict if a program stops or loops infinitely.
- What the Busy Beaver Problem Asks ?
  - The **Busy Beaver** problem asks:

    Among all halting N-state Turing machines, what is the one that runs the longest before halting (or writes the most 1s)?
  - We call this maximum number of steps $S(N)$.
- Why They're Connected
  - To compute $S(N)$,you would need to know **which machines halt and which do not**.
  - But the **Halting Problem says you can't know that in general**.
  - So:
    - To find the true $S(N)$,you must **test all N-state machines**.
    - For each machine, you'd have to decide: does it halt or loop forever?
    - Since the Halting Problem is undecidable, you **cannot algorithmically determine** halting for all machines.
    - Therefore, **there's no algorithm that can compute** $S(N)$for all N.

Busy beaver Problems

BB(1) - One State

Explanation:

- The simplest case: a Turing machine with only 1 state (plus the halt state)
- The machine can only perform 2 operations (one for reading 0, one for reading 1)
- Goal: Find the machine that writes the most 1s before halting

Solution:
Maximum steps: 1 Maximum 1s written: 1
Winning Machine Logic:

State A
Reading 0: Write 1, move Right, Halt
Reading 1: (never reached in this optimal case)

Execution trace:
Initial: ...000[0]000... Step 1: ...0001[0]00... → HALT
Result: **1 one written in 1 step**

BB (2)

Explanation:
•Two states (A, B) plus halt state
•Much more interesting! The machine can now loop before halting
•4 possible transitions to define

Maximum steps: 6 , **Maximum 1s written: 4**
Winning Machine:

State A	State B
Read 0: Write 1, Move Right, Go to B	Read 0: Write 1, Move Left, Go to A
Read 1: Write 1, Move Left, Go to B	Read 1: Write 1, Move Right, Halt

Execution trace:
Step 0: ...000[0]000... (State A) Step 1: ...0001[0]00... (State B) - wrote 1, moved R Step 2: ...000[1]100... (State A) - wrote 1, moved L Step 3: ...00[0]1100... (State B) - moved L Step 4: ...0[0]11100... (State A) - wrote 1, moved L Step 5: ...0[1]11100... (State B) - moved L Step 6: ...01111[0]0... HALT - wrote 1, moved R

Result: **4 ones written in 6 steps**

BB(3)

Three states (A, B, C) plus halt
6 transitions to define.

Complexity increases significantly

Found by Tibor Radó and Shen Lin in 1965

Solution:

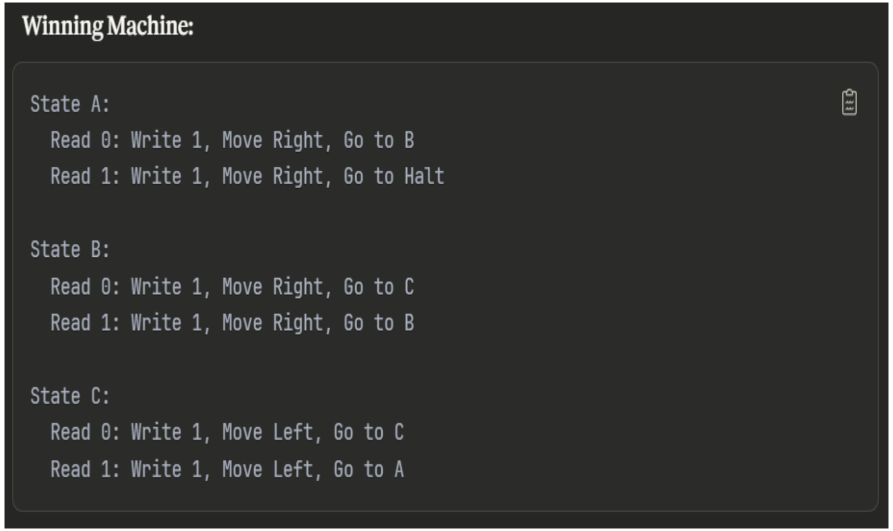
Maximum steps: 21

Maximum 1s written: 6

Key characteristics:
Creates a pattern of 6 consecutive 1s

Execution involves complex back-and-forth movement

Takes 21 steps to complete



Winning Machine:

```
State A:
  Read 0: Write 1, Move Right, Go to B
  Read 1: Write 1, Move Right, Go to Halt

State B:
  Read 0: Write 1, Move Right, Go to C
  Read 1: Write 1, Move Right, Go to B

State C:
  Read 0: Write 1, Move Left, Go to C
  Read 1: Write 1, Move Left, Go to A
```

BB(4)

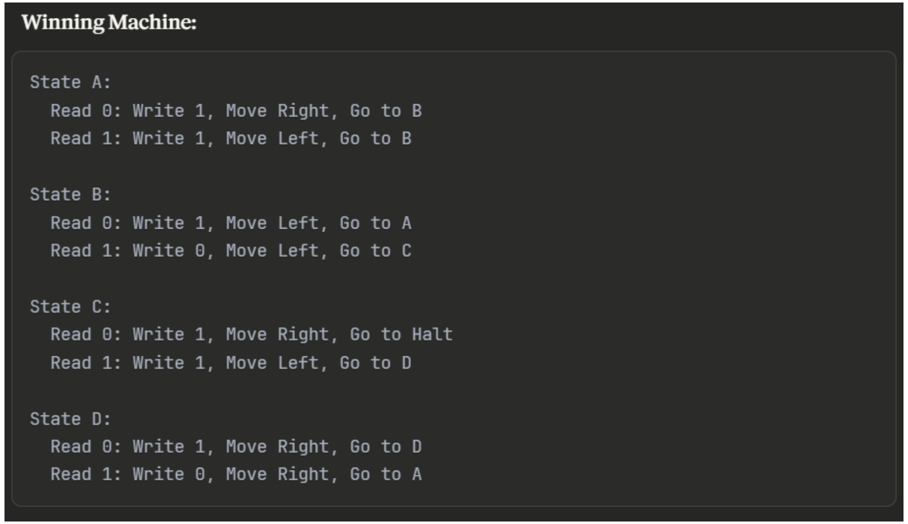
Four states (A, B, C, D) plus halt
8 transitions to define

Discovered by Allen Brady in 1983

Required computer search to find

Solution

Maximum steps: 107
Maximum 1s written: 13



Winning Machine:

```
State A:
  Read 0: Write 1, Move Right, Go to B
  Read 1: Write 1, Move Left, Go to B

State B:
  Read 0: Write 1, Move Left, Go to A
  Read 1: Write 0, Move Left, Go to C

State C:
  Read 0: Write 1, Move Right, Go to Halt
  Read 1: Write 1, Move Left, Go to D

State D:
  Read 0: Write 1, Move Right, Go to D
  Read 1: Write 0, Move Right, Go to A
```

- Runs for exactly 107 steps
- Writes 13 ones on the tape
- First case where manual analysis became extremely difficult
- Pattern involves creating, erasing, and recreating 1s

BB 5

Explanation:

- Five states (A, B, C, D, E) plus halt
- 10 transitions to define
- Discovered by Heiner Marxen and Jürgen Buntrock in 1989
- Found using sophisticated computer search algorithms

Maximum steps: 47,176,870 Maximum 1s written: 4,098

Winning Machine characteristics:
Creates an elaborate pattern over nearly 47.2 million steps
Writes 4,098 ones before halting

Massive jump in complexity from BB(4)
The execution is far too long to trace manually
Uses sophisticated looping and counting behaviors
Why this is remarkable:
BB(4) runs 107 steps → BB(5) runs 47+ million steps
This demonstrates the explosive growth of the Busy Beaver function
The machine essentially "counts" in a complex way before halting

Approaches to Tackle Complexity for N=6

- It is known that BB**(6)** is at least **7.4 × 1036543**. Research about busy beaver numbers seems stagnant as these results were discovered **using supercomputers some 30 years ago**. However, insights about this fast-growing function could provide headway in other mathematical areas.
- The **Halting Problem** makes it impossible to decide automatically whether a given machine halts. Each new “halting” machine pushes the known bound higher, but we can never be sure it’s the maximum.
- To manage complexity, breaking the problem into smaller, manageable subproblems and leveraging parallel computing are essential. Employing heuristic methods and optimization techniques can help reduce the search space. These strategies focus on enhancing efficiency without compromising solution accuracy, enabling deeper exploration of **N = 6** scenarios.
- Promising algorithms include advanced **backtracking** combined with constraint propagation, and machine learning approaches for predictive pruning. Hybrid models that integrate deterministic and probabilistic algorithms show potential to efficiently target relevant state spaces, accelerating convergence toward the solution. These methods aim to balance exploration with computational feasibility.

The Explosive Growth of BB(n)

States (n)	BB(n) Steps	Ones Written	Status
1	1	1	✓ Known
2	6	4	✓ Known
3	21	6	✓ Known
4	107	13	✓ Known
5	47,176,870	4,098	✓ Known
6	> 10^36,534	> 10^18,267	❓ Unknown

BB(n) grows faster than ANY computable function!

Simulation

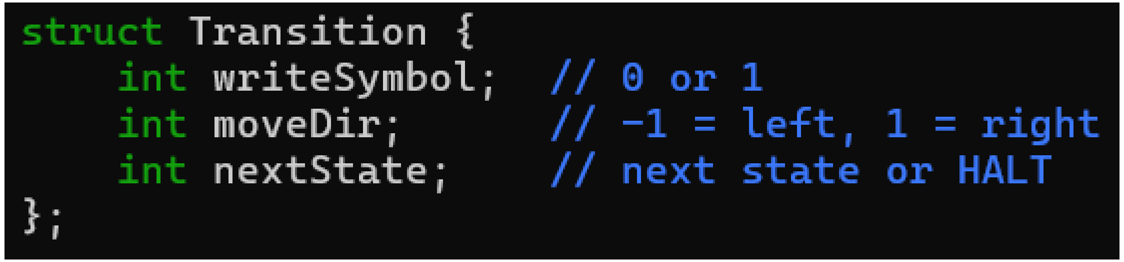
The **Busy Beaver** problem explores the limits of computation.

For a Turing Machine with n states
- BB(n) → Maximum steps before halting
- Σ(n) → Maximum number of 1’s printed before halting
- It’s impossible to compute exactly for large n, it grows faster than any computable function.

Already known Busy Beaver values with **total possible Turing machines**:
- n = 1 → **BB(1) = 1**, with **64** possible machines
- n = 2 → **BB(2) = 6**, with **20,736** possible machines
- n = 3 → **BB(3) = 21**, with **≈10^7 (10 million)** machines
- n = 4 → **BB(4) = 107**, with **≈10^{10} (10 billion)** machines

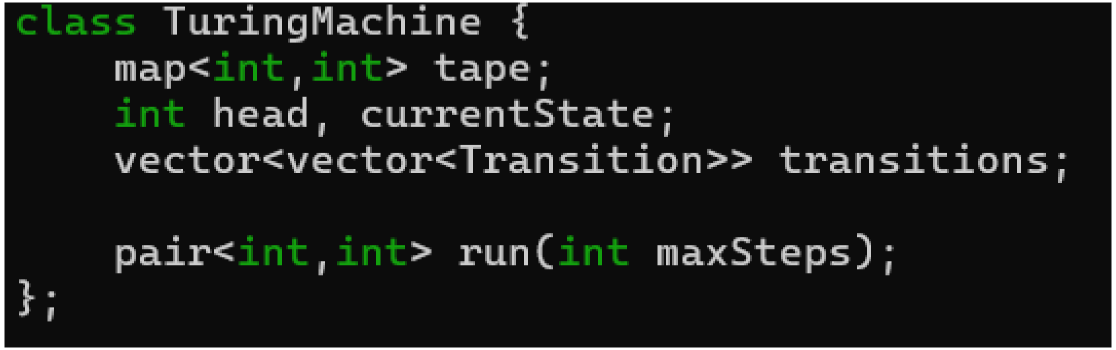
Code Structure

1. Transition Definition:



```cpp
struct Transition {
    int writeSymbol;   // 0 or 1
    int moveDir;       // -1 = left, 1 = right
    int nextState;     // next state or HALT
};
```

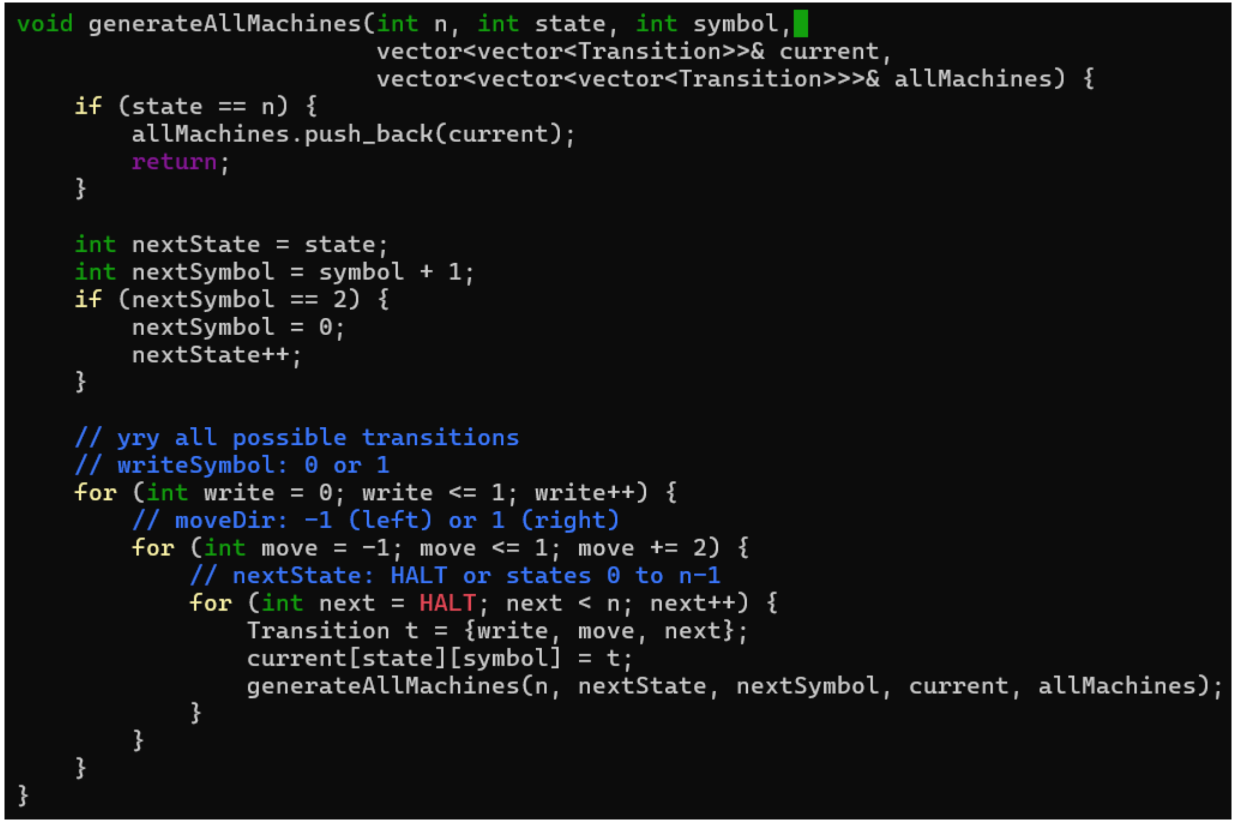
2. Turing Machine Simulator:



```cpp
class TuringMachine {
    map<int,int> tape;
    int head, currentState;
    vector<vector<Transition>> transitions;

    pair<int,int> run(int maxSteps);
};
```

Code Structure

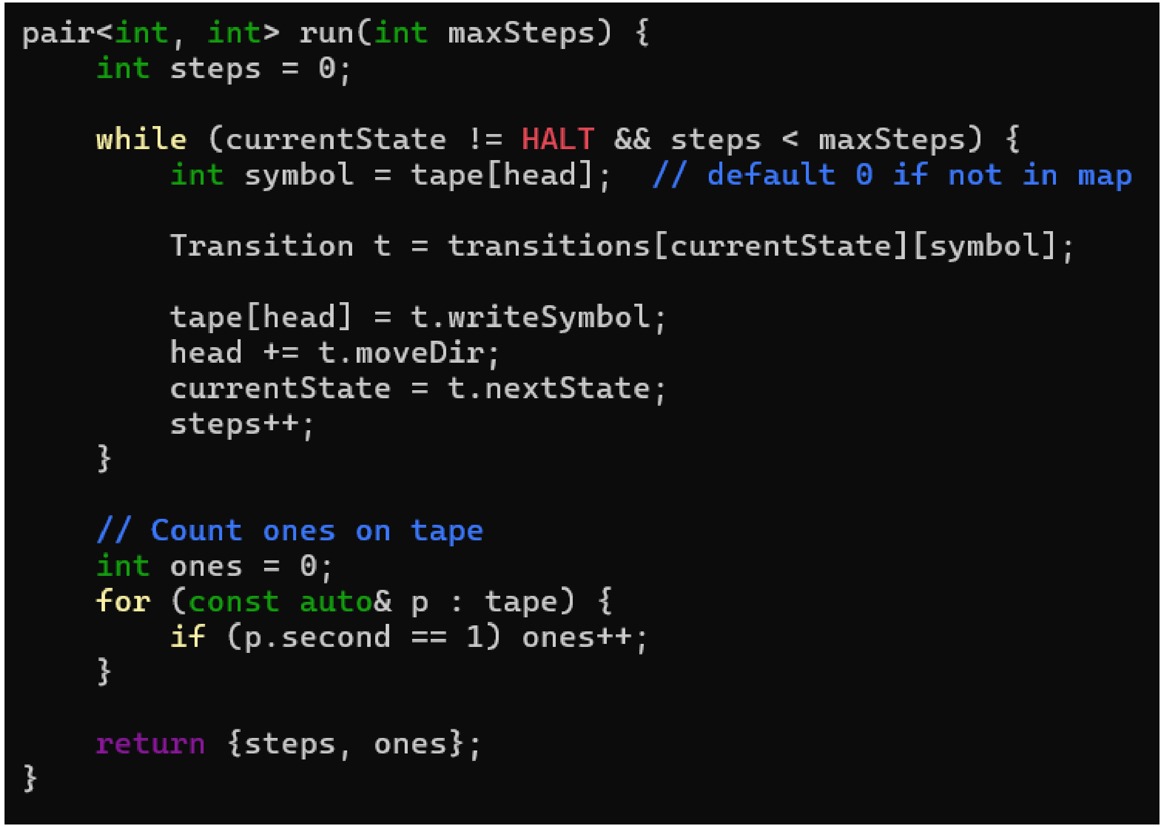
3. Machine Generation (Recursion) :



```
void generateAllMachines(int n, int state, int symbol,
                         vector<vector<Transition>>& current,
                         vector<vector<vector<Transition>>>& allMachines) {
    if (state == n) {
        allMachines.push_back(current);
        return;
    }

    int nextState = state;
    int nextSymbol = symbol + 1;
    if (nextSymbol == 2) {
        nextSymbol = 0;
        nextState++;
    }

    // yry all possible transitions
    // writeSymbol: 0 or 1
    for (int write = 0; write <= 1; write++) {
        // moveDir: -1 (left) or 1 (right)
        for (int move = -1; move <= 1; move += 2) {
            // nextState: HALT or states 0 to n-1
            for (int next = HALT; next < n; next++) {
                Transition t = {write, move, next};
                current[state][symbol] = t;
                generateAllMachines(n, nextState, nextSymbol, current, allMachines);
            }
        }
    }
}
```

Code Structure

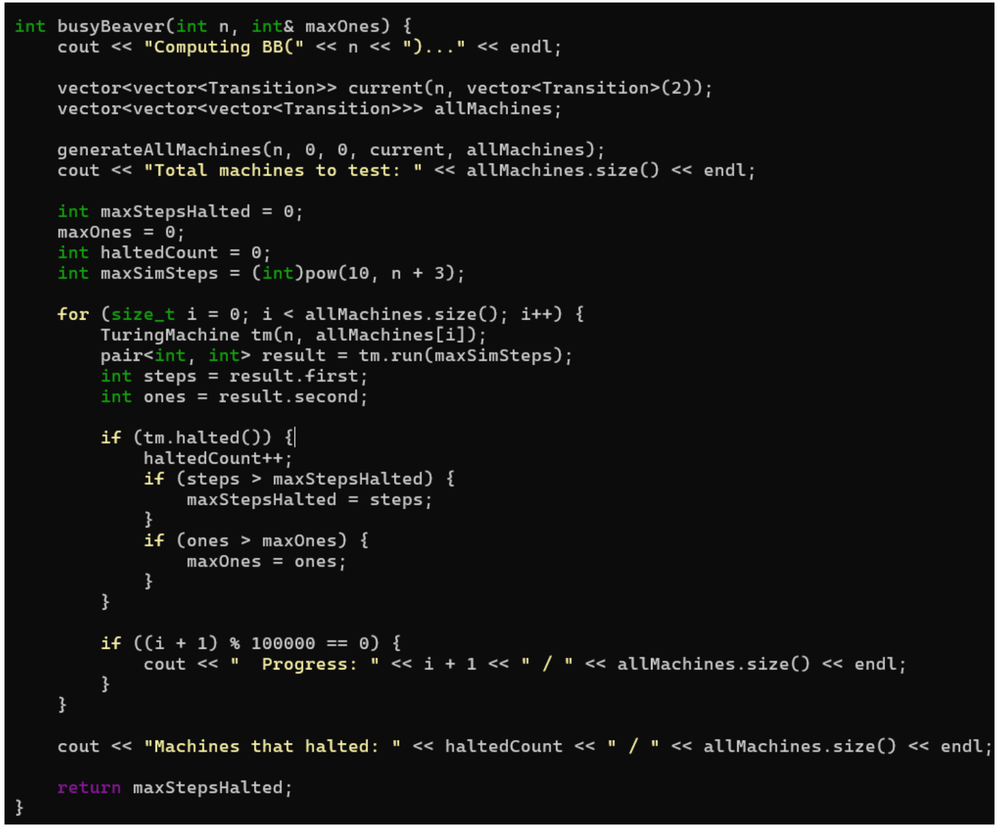
4. Core Simulation :



```cpp
pair<int, int> run(int maxSteps) {
    int steps = 0;

    while (currentState != HALT && steps < maxSteps) {
        int symbol = tape[head];  // default 0 if not in map

        Transition t = transitions[currentState][symbol];

        tape[head] = t.writeSymbol;
        head += t.moveDir;
        currentState = t.nextState;
        steps++;
    }

    // Count ones on tape
    int ones = 0;
    for (const auto& p : tape) {
        if (p.second == 1) ones++;
    }

    return {steps, ones};
}
```

Code Structure

5. Testing All Machines:



```cpp
int busyBeaver(int n, int& maxOnes) {
    cout << "Computing BB(" << n << ")..." << endl;

    vector<vector<Transition>> current(n, vector<Transition>(2));
    vector<vector<vector<Transition>>> allMachines;

    generateAllMachines(n, 0, 0, current, allMachines);
    cout << "Total machines to test: " << allMachines.size() << endl;

    int maxStepsHalted = 0;
    maxOnes = 0;
    int haltedCount = 0;
    int maxSimSteps = (int)pow(10, n + 3);

    for (size_t i = 0; i < allMachines.size(); i++) {
        TuringMachine tm(n, allMachines[i]);
        pair<int, int> result = tm.run(maxSimSteps);
        int steps = result.first;
        int ones = result.second;

        if (tm.halted()) {
            haltedCount++;
            if (steps > maxStepsHalted) {
                maxStepsHalted = steps;
            }
            if (ones > maxOnes) {
                maxOnes = ones;
            }
        }

        if ((i + 1) % 100000 == 0) {
            cout << "  Progress: " << i + 1 << " / " << allMachines.size() << endl;
        }
    }

    cout << "Machines that halted: " << haltedCount << " / " << allMachines.size() << endl;

    return maxStepsHalted;
}
```

- Each **(state, symbol)** pair has **2 × 2 × (n + 1)** possible transition rules.
- Total of 2n transitions per machine.
- Hence, **total possible machines =**

$$\left(4 * (n+1)\right)^{2n}$$