Logic Assignment-2 Report

Krishna Sai Shishir Vuppala(24b0973), Aaditya Kumar(24b0975)

September 2025

Question 1

Consider the set of 4 challenges. Let them be denoted as:

- $\mathcal{R}_1 = \{(a_1,\, b_1) \times (c_1,\, d_1)\}$
- $\mathcal{R}_2 = \{(a_2,\, b_2) \times (c_2,\, d_2)\}$
- $\mathcal{R}_3 = \{(a_3,\, b_3) \times (c_3,\, d_3)\}$
- $\mathcal{R}_4 = \{(a_4,\, b_4) \times (c_4,\, d_4)\}$

Now, let us consider all possible schema $s = (s_1, s_2, s_3, s_4)$ in their expanded form. Here, we use $*$ to denote a 'placeholder' (can be substituted for by a c_i or a d_i for an s_i). We can represent the schema as follows: Let the rule **s1** , represented as $\mathbf{s1} = \{(a_1,\, c_1), (b_1,\, d_1)\}$ now be denoted as

$$\begin{aligned} \mathbf{s_1} : a_1, c_1 \\ b_1, d_1 \end{aligned}$$

Then, we may denote the 4 rules in our schema can be represented (with the placeholder notation) as follows:

$$\begin{aligned} \mathbf{s_1} : a_1, * \\ b_1, * \end{aligned}$$

$$\begin{aligned} \mathbf{s_2} : a_1 a_2, * \\ a_1 b_2, * \\ b_1 a_2, * \\ b_1 b_2, * \end{aligned}$$

$$
\begin{aligned}
\mathbf{s_3} : a_1a_2a_3, * \\
a_1a_2b_3, * \\
a_1b_2a_3, * \\
a_1b_2b_3, * \\
b_1a_2a_3, * \\
b_1a_2b_3, * \\
b_1b_2a_3, * \\
b_1b_2b_3, *
\end{aligned}
$$

$$
\begin{aligned}
\mathbf{s_4} : a_1a_2a_3a_4, * \\
a_1a_2a_3b_4, * \\
a_1a_2b_3a_4, * \\
a_1a_2b_3b_4, * \\
a_1b_2a_3a_4, * \\
a_1b_2a_3b_4, * \\
a_1b_2b_3a_4, * \\
a_1b_2b_3b_4, * \\
b_1a_2a_3a_4, * \\
b_1a_2a_3b_4, * \\
b_1a_2b_3a_4, * \\
b_1a_2b_3b_4, * \\
b_1b_2a_3a_4, * \\
b_1b_2a_3b_4, * \\
b_1b_2b_3a_4, * \\
b_1b_2b_3b_4, *
\end{aligned}
$$

Now, we are given $\varphi = (\Omega, \Theta)$, but we are provided the choice of Θ, and the last two challenges in Ω.
Note that once (if) we have a perfect incantation schema constructed, we may construct one possible doomed schema by simply adding a non-zero offset to Θ. This is because in the Perfect Incantation Schema, we always achieve the sum Θ if we obey the rules in the schema, irrespective of the sequence of moves made by the spellcaster or Harry. Adding a non-zero offset to a Perfect Incantation Schema's Θ would result in a sum that can never be obtained if the schema is obeyed. We now proceed to construct a Perfect Incantation Schema.

Perfect Incantation Schema

Given that our roll numbers are 24B0973 and 24B0975 , we have the following:

- $\mathcal{R}_1 = \{(0,\ 9) \times (7,\ 3)\}$
- $\mathcal{R}_2 = \{(0,\ 9) \times (7,\ 5)\}$

(Without loss of generality, we can assume $\mathcal{R}_1$ to correspond to 24B0973 and $\mathcal{R}_2$ to correspond to 24B0975.)
Now, let us choose the other two challenges to be:

- $\mathcal{R}_3 = \{(1,\ 1) \times (0,\ 9)\}$
- $\mathcal{R}_4 = \{(1,\ 1) \times (0,\ 9)\}$

Then, we have the following **Perfect Incantation Schema**:

$\Omega = (\mathcal{R}_1, \mathcal{R}_2, \mathcal{R}_3, \mathcal{R}_4) \qquad \Theta = 34 \qquad s = (s_1, s_2, s_3, s_4)$

$\mathbf{s_1}$:	$\mathbf{s_2}$:	$\mathbf{s_3}$:	$\mathbf{s_4}$:
0, 7	00, 7	001, 9	0011, 9
9, 7	09, 7	001, 9	0011, 9
	90, 7	091, 9	0011, 9
	99, 7	091, 9	0011, 9
		901, 9	0911, 0
		901, 9	0911, 0
		991, 0	0911, 0
		991, 0	0911, 0
			9011, 0
			9011, 0
			9011, 0
			9011, 0
			9911, 0
			9911, 0
			9911, 0
			9911, 0

The idea behind this schema is that we know that the spellcaster has choices of 0 and 9 in the first two challenges. Now, in the next two challenges, we will remove "choice" from the spellcaster(since both choices are 1, whichever number he chooses, it doesn't matter). and if the spell caster choce zero one or two 9's in the first two challenges then we will choose two one or zero 9's in the last 2 challenges respectively to equalise it. Thus, we will end up with 9*2(since equalised), 7*2(from first two challenges from the choice of harry) and 2*1 (from the "choice" of spellcaster in last 2 challenges). So total is $2 \cdot (9 + 7 + 1) = 34$

Doomed Schema

Following from the earlier discussed construction (one of many possibilities), we have an example of a **doomed schema**:

$\Omega = (\mathcal{R}_1, \mathcal{R}_2, \mathcal{R}_3, \mathcal{R}_4)$ $\Theta = 100$ $s = (s_1, s_2, s_3, s_4)$

$\mathbf{s_1}$:	$\mathbf{s_2}$:	$\mathbf{s_3}$:	$\mathbf{s_4}$:
0, 7	00, 7	001, 9	0011, 9
9, 7	09, 7	001, 9	0011, 9
	90, 7	091, 9	0011, 9
	99, 7	091, 9	0011, 9
		901, 9	0911, 0
		901, 9	0911, 0
		991, 0	0911, 0
		991, 0	0911, 0
			9011, 0
			9011, 0
			9011, 0
			9011, 0
			9911, 0
			9911, 0
			9911, 0
			9911, 0

Question 2

The top-level idea is we're gonna loop through all possible values of all the k variables(since there are n options for each variable this should take n^k times) We will iteratively make our way from the outside to the inside on the prenex part of the formula. Suppose the formula is QxF. where F is the rest of the formula apart from the first quatifier part (Note: so F is not a wff since x is not quantified in it).Now there are two cases

- if $Q = \forall$, then for every element $a \in \{1, 2, \ldots n\}$, F[a/x] should be satisifiable. i.e. if F[a/x] is unsatisifable for any $a \in \{1, 2, \ldots n\}$ then $\forall xF$ is also unsatisifiable
- if $Q = \exists$, then for atleast one element $a \in \{1, 2, \ldots n\}$, F[a/x] should be satisifiable. i.e. if F[a/x] is satisifable for any $a \in \{1, 2, \ldots n\}$ then $\exists xF$ is also satisifiable

So since this is a recursive definition, we can model this as a recursive algorithm.

Algorithm 1 T-Logic Satisfiability Checker

```
1: function CheckSatisfiability(φ, assignment)
2:                 ▷ φ is a formula, assignment is a map from variables to values
3:     if φ is a quantifier-free matrix ψ then
4:         return EvaluateMatrix(ψ, assignment)
5:     end if
6:     Let φ be of the form Q x φ′
7:     if Q = ∀ then
8:         for each value v ∈ {0, 1, ..., n − 1} do
9:             new_assignment ← assignment ∪ {x ↦ v}
10:            if CheckSatisfiability(φ′, new_assignment) is false then
11:                return false                ▷ Found a counterexample for ∀
12:            end if
13:        end for
14:        return true                         ▷ The formula holds for all values
15:    end if
16:    if Q = ∃ then
17:        for each value v ∈ {0, 1, ..., n − 1} do
18:            new_assignment ← assignment ∪ {x ↦ v}
19:            if CheckSatisfiability(φ′, new_assignment) is true then
20:                return true                 ▷ Found a satisfying instance for ∃
21:            end if
22:        end for
23:        return false                        ▷ No value satisfied the formula
24:    end if
25: end function
```

This function returns true if the formula inputted is SAT and false if the formula is UNSAT.
So if a formula has no quantifier, then it will evaluate its satisfiability based on the formula using EVALUATEMATRIX(which is written after this). But if not, then considering the original formula as $\phi = Qx\phi'$, if the quantifier used is $\forall$, then if there is an element which makes ϕ'[a/x] UNSAT, then the original formula ϕ is also UNSAT. If there is no element which makes ϕ'[a/x] UNSAT, then the original formula is SAT.
If the quantifier used is $\exists$, then if there is an element which makes ϕ'[a/x] SAT, then the original formula ϕ is also SAT. If there is no element which makes ϕ'[a/x] SAT, then the original formula is UNSAT.
The way we check the satisfiability of ϕ'[a/x] is by recursively calling CHECK-SATISFIABILITY for ϕ' and seeing what it returns(if false UNSAT and if true SAT)

Algorithm 2 Evaluator

function EVALUATEMATRIX(ψ, assignment)
▷ Evaluate the quantifier-free matrix ψ
for all atomic formulas $P_i(t)$ in ψ **do**
Let v be the value of term t in the *assignment*
Substitute $P_i(t)$ with the boolean value of $(v \in \Lambda(P_i))$
end for
return the result of the final boolean expression ψ by analysing the parse tree.
end function

Now first let's understand the time and space complexity of EVALUATEMATRIX

Since we have to go through every single predicate, we need to visit almost every node in parse tree (except the nodes for conjunction, disjunction and implies). So O(length(ϕ)). (The efficient way to do this is to start a recursive call at the stack of the tree and then work your way down) Now, after this, evaluating the final boolean expression is just a matter of working backward through the parse tree to check SAT. Going backward basically means If there is a node whose children have a SAT value associated with them, then evaluate that part and set the value of that node to the truth value of the evaluation. i.e. if there a logical and and its children are all true, then it will also get set it true but if even one of them is false, then it gets set to false. Similarly, for logical or, implies, and negation. We can do this recursively by calling at the root node. This can also be done in O(length(ϕ)) time (since any particular node will be visited only once in the recursive stack call)

The space complexity of EVALUATEMATRIX is O(length(ϕ)height(ϕ) because we need to make a new copy of the parse tree(since we are modifying it) and also $height(\phi)$ because of recursive stack calls.
Now, let's understand the time and space complexity of CHECKSATISFIABILITY.

CLAIM: the time complexity is O(n^klength(ψ)) where k is number of quantifiers **Base Case**: if k=0 then no loop are executed and so its just the time complexity of EVALUATEMATRIX, which is O(length(ψ) as shown earlier.

Induction Hypothesis: the time complexity is O(n^klength(ψ)) where k is number of quantifiers for all $k \leq m$

Induction Step: Take k = m+1
The main time complexity is from the loop. only one of the loop(either the $\exists$ one or the $\forall$ one) will execute. In the worst case, the loop will execute n times. In each iteration of the loop, we will have one execution of CHECKSATISFIABIL-

ITY with k = m. So from the induction hypothesis, we have O(n^mlength(ψ)) executing n times. So total time complexity is, O(n^{m+1}length(ψ)). Thus, proven time complexity.
The space complexity mostly comes from the recursive stack call. Every recursive call requires us to store the previous local variables. Each assignment has at most k elements, and there will be k recursive stack calls. So the space complexity of this part is O(k^2). Combining with the space complexity of each EVALUATEMATRIX call, we get O(k^2length(ϕ)height(ϕ))

Question 3

The top-level idea of this formula is to encode picking a schema via variables and then implement addition to check if the final power is equal to the target or not for every possible route. If you want to see the final formula directly instead of the explanation then then go to the end of this question.

Encoding a Random Schema

First of all we want to somehow encode a schema in a T-logic formula somehow and then check if its perfect or not. Since the number of elements in universe is fixed and the relations are also fixed we cant use these to map out "choices". However there is not bound on the number of **variables** we can use and this is how we can introduce "choice".
Let us consider a universe of 2 elements for now.
So first of all we will have $2n$ variables each encoding the choices made by spellcaster and by Harry. so $x_1, x_2, x_3, x_4 \dots x_n$ will denote choices made by spellcaster, where $x_i = 0$ means spellcaster picked a_i and $x_i = 1$ means he picked b_i $y_1, y_2, y_3 \dots y_n$ will denote chocie made by harry where $y_i = 0$ means spellcaster picked a_i and $x_i = 1$ means he picked

Now, to denote a spellcasting plan, we can use the arrangement of variables in the prenex part as

$$\forall x_1 \exists y_1 \forall x_2 \exists y_2 \forall x_3 \exists y_3 \dots \forall x_n \exists y_n$$

Now, let's quickly justify this choice. This means that y_1 is a function of x_1 (since for each value of x_1 we need a value of y_1, Thus every value of x_1 is associated with a value of y_1 i.e. $y_1 = f(x_1)$), y_2 is a function of x_1, x_2 and so on.
This means that y_i is a function of $x_1, \dots x_i$, which is exactly what a spellcasting plan is! So this methodology is justified.

Implementing Binary Addition

Now we need to check if there is a perfect spellcasting scheme. Thus, the matrix of our formula should be true for a perfect spellcasting plan. But how will we do

this? Well, we will try to implement **binary addition** via the unary predicates.

Addition is nothing but a collection of rules(eg, if x is 1 and y is 0, then their sum is 1), so we can encode this using variables and propositions. If we have two variables, x and y, then we can introduce a variable s.

$$\forall x \forall y \exists s (P_0(x) \wedge P_0(y)) \rightarrow P_0(s)) \wedge (P_0(x) \wedge P_1(y)) \rightarrow P_1(s)) \wedge (P_1(x) \wedge P_0(y)) \rightarrow P_1(s)) \wedge (P_1(x) \wedge P_1(y)) \rightarrow P_0(s))$$

similiarly we can define a "carry" variable c too,

$$\forall x \forall y \exists c (P_0(x) \wedge P_0(y)) \rightarrow P_0(c)) \wedge (P_0(x) \wedge P_1(y)) \rightarrow P_0(c)) \wedge (P_1(x) \wedge P_0(y)) \rightarrow P_0(c)) \wedge (P_1(x) \wedge P_1(y)) \rightarrow P_1(c))$$

Now, since we are doing Binary addition, we need to store all the digits of the number and we will need variables for storing the sum. So first of all lets decide how many place digits we want.

$$L = \Theta$$

$$m = \lceil \log_2 L \rceil$$

So now we will use variables of the form $s_{i,j}$ where j ranges from 0 to $m-1$ to denote the digit position and i ranges from 0 to $2n$,(to denote the sums till the kth total(ie both both harry's and spellcaster's choice contribute to the tally) choice)
We also will have variables of the form $c_{i,j}$ where j ranges from 0 to $m-1$ to denote the digit position and i ranges from 0 to $2n+1$. These basically denote the carry generated while adding the j^{th} digit. (explain a bit more here). So Sum_i is the total accumulated power until the i^{th} total choice, and this has at max m digits, and so we have $s_{i,m-1}, s_{i,m-2} \dots s_{i,0}$ to represent it. Now we will define the following helper relations,

$$\begin{aligned} Add0(x,y,z,w) =& (P_0(x) \wedge P_0(y)) \rightarrow (P_0(z) \wedge P_0(w))) \wedge (P_0(x) \wedge P_1(y)) \rightarrow (P_1(z) \wedge P_0(w))) \\ & \wedge (P_1(x) \wedge P_0(y)) \rightarrow (P_1(z) \wedge P_0(w))) \wedge (P_1(x) \wedge P_1(y)) \rightarrow (P_0(z) \wedge P_1(w))) \end{aligned}$$

$$\begin{aligned} Add1(x,y,z,w) =& (P_0(x) \wedge P_0(y)) \rightarrow (P_1(z) \wedge P_0(w))) \wedge (P_0(x) \wedge P_1(y)) \rightarrow (P_0(z) \wedge P_1(w))) \\ & \wedge (P_1(x) \wedge P_0(y)) \rightarrow (P_0(z) \wedge P_1(w))) \wedge (P_1(x) \wedge P_1(y)) \rightarrow (P_1(z) \wedge P_1(w))) \end{aligned}$$

Basically, what these relations do is take in x and y as input which are two binary digits and then sets z as their sum and w as the carry generated. Add1 also adds an extra 1 to the digit part.
Now for completeness we will introduce a 1-variable version, when there is no carry ie for the one's place digit.

$$Add0(x,z,w) = (P_0(x) \rightarrow (P_0(z) \wedge P_0(w))) \wedge (P_1(x) \rightarrow (P_1(z) \wedge P_0(w)))$$

$$Add1(x,z,w) = (P_0(x) \rightarrow (P_1(z) \wedge P_0(w))) \wedge (P_1(x) \rightarrow (P_0(z) \wedge P_1(w)))$$

NOTE: THIS IS NOT ME INTRODUCING NEW RELATIONS TO THE STRUCTURE WE CAN JUST AS EASILY DEFINE THE FORMULA WITHOUT USING THESE SHORTCUT AND IN FACT AT THE END WE CAN SUBSTITUTE "Add1" and "Add0" WITH THEIR MEANINGS. THESE NEW REALTIONS HAVE JUST BEEN INTRODUCED TO MAKE EXPLANATION AND WRITING OF THE FINAL FORMULA EASIER.

Now the first part will be setting Sum_0 and all the initial carries to 0.

$$\theta_0 = P_0(s_{0,0}) \wedge P_0(s_{0,1}) \wedge P_0(s_{0,2}) \wedge \ldots \wedge P_0(s_{0,m-1}) \wedge P_0(c_{0,0}) \wedge P_0(c_{0,1}) \wedge P_0(c_{0,2}) \wedge \ldots \wedge P_0(c_{0,m-1})$$

Now the second part is saying that the carry generated at the $m-1th$ bit should ALWAYS be zero otherwise an overflow has occured. (We can interpret this as the total power is already more than the target and so obviously this is a failed run)

$$\theta_1 = P_0(c_{0,m-1}) \wedge P_0(c_{1,m-1}) \wedge P_0(c_{2,m-1}) \wedge \ldots \wedge P_0(c_{n,m-1})$$

Now we must generate subformulas to add numbers to the sum.
Now how to add a number to the sum? what we will do is write the number in its binary expansion(extended to m digits, so if m=5 5 would be written as 00101). and based on its binary expansion we will accordingly use Add1 and Add0 on the places to add the number. This mean if the number we want to add is x then for the position in which we have a 1 we will use Add1 with previous sum's digit of same position and carry generated from the previous digit. and where its 0 we will ue Add0 with the previous sum's digit of the same position and carry generated from the previous digit.
eg. if we have $a_1 = 9$ then in binary we put as 1001 and this can be written as

$$Add1(s_{0,m-1}, c_{0,m-2}, s_{1,m-1}, c_{0,m-1}) \wedge Add1(s_{0,m-2}, c_{0,m-3}, s_{1,m-2}, c_{0,m-2})$$
$$\ldots Add1(s_{0,1}, c_{0,1}, s_{1,1}, c_{0,2}) \wedge Add1(s_{0,0}, s_{1,0}, c_{0,1})$$

So to denote formulas subformulas like this, where the formula depends on the numbers we choose, we will use the notation $\text{add}_{x,j}$ where x is the number we are adding and i denotes the number of numbers we've added already(basically this is there so we add the)

$$add_{x,i} = Add1(s_{i,m-1}, c_{i,m-2}, s_{i+1,m-1}, c_{i,m-1}) \wedge Add0(s_{i,m-2}, c_{i,m-3}, s_{i+1,m-2}, c_{i,m-2}) \wedge \ldots$$
$$Add(1s_{0,m-1}, c_{0,m-2}, s_{1,m-1}, c_{0,m-1} \wedge Add1(s_{i,m-1}, s_{i,1}, c_{i,1})$$

where the order of Add1 and Add0's is defined based on the binary expansion of x extended to m digits. Thus, this subformula basically just adds x to Sum_{j-1} to get Sum_j
Finally, we need to check if the final sum we have is Θ or not but how to do this? let Θ have binary representation $\overline{x_{m-1} \ldots x_1 x_0}$. Then we can just do

$$P_{x_{m-1}}(s_{2n,m-1}) \wedge P_{x_{m-2}}(s_{2n,m-2}) \ldots P_{x_0}(s_{2n,0})$$

Thus, this just matches the value of Θ with Sum_{2n}.
So the final formula we get is

$$\theta_0 = P_0(s_{0,0}) \wedge P_0(s_{0,1}) \wedge P_0(s_{0,2}) \wedge \ldots \wedge P_0(s_{0,m-1}) \wedge P_0(c_{0,0}) \wedge P_0(c_{0,1}) \wedge P_0(c_{0,2}) \wedge \ldots \wedge P_0(c_{0,m-1})$$

$$\theta_1 = P_0(c_{0,m-1}) \wedge P_0(c_{1,m-1}) \wedge P_0(c_{2,m-1}) \wedge \ldots \wedge P_0(c_{n,m-1})$$

$$\theta_2 = P_{l_{m-1}}(s_{2n,m-1}) \wedge P_{l_{m-2}}(s_{2n,m-2}) \ldots P_{l_0}(s_{2n,0})$$

where $l_{m-1}, l_{m-2} \ldots$ are binary digits of Θ

$$\phi_i = (P_0(x_i) \rightarrow \text{add}_{a_i,2i-1}) \wedge (P_1(x_i) \rightarrow \text{add}_{b_i,2i-1})$$

$$\psi_i = (P_0(y_i) \rightarrow \text{add}_{c_i,2i}) \wedge (P_1(y_i) \rightarrow \text{add}_{d_i,2i})$$

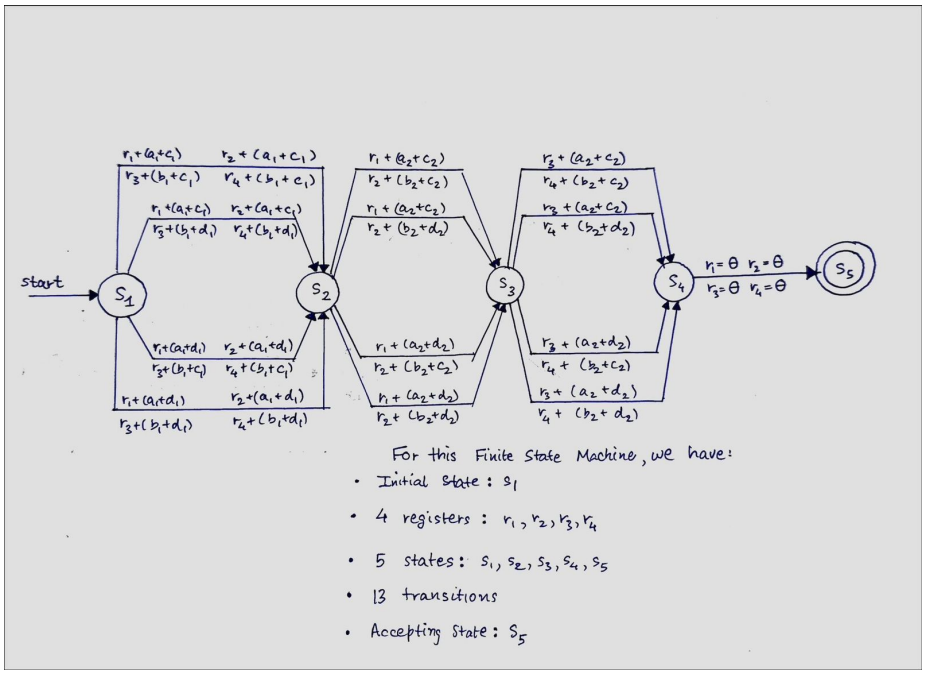
Final formula given by

$$\forall x_1 \exists y_1 \forall x_2 \exists y_2 \ldots \forall x_n \exists y_n \exists s_{0,0} \exists s_{0,1} \ldots \exists s_{2n,m-1} \exists c_{0,0} \exists c_{0,1} \ldots \exists c_{2n,m-1} \\ (\theta_0 \wedge \theta_1 \wedge \theta_2 \wedge \phi_1 \wedge \phi_2 \ldots \wedge \phi_n \wedge \psi_1 \wedge \psi_2 \ldots \wedge \psi_n)$$

Now the explanation for why this is correct was justified while constructing the formula, but again we pick a random schema using all the x_i's,y_i's and then we test out that schema by tallying up the and so if there is a perfect schema then we can make the sum equal to target power for all possible values of $x_1, x_2 \ldots x_n$ and so the matrix will be satisfiable for all possible values of this. Thus, it will only be T-Satisfiable if there exists a perfect schema.

Question 4

Ok so the main idea will be to construct an automata which can non-deterministically check all the 64 possible schemas for 2 challenges($2^2 \times 2^4 = 64$) for this we define the following automatas



I will also define it in a formal way. is is the tuple (S, R, Δ, s_1)

$$S = \{s_1, s_2, s_3, s_4, s_5\}$$

$$R = \{1, 2, 3, 4\}$$

$$s_1 \in S$$

$$\begin{aligned}
\Delta = \{&(s_1, \emptyset, (a_1 + c_1, a_1 + c_1, b_1 + c_1, b_1 + c_1), \emptyset, s_2) \\
&, (s_1, \emptyset, (a_1 + c_1, a_1 + c_1, b_1 + d_1, b_1 + d_1), \emptyset, s_2) \\
&, (s_1, \emptyset, (a_1 + d_1, a_1 + d_1, b_1 + c_1, b_1 + c_1), \emptyset, s_2) \\
&, (s_1, \emptyset, (a_1 + d_1, a_1 + d_1, b_1 + d_1, b_1 + d_1), \emptyset, s_2) \\
&, (s_2, \emptyset, (a_2 + c_2, b_2 + c_2, 0, 0), \emptyset, s_3) \\
&, (s_2, \emptyset, (a_2 + c_2, b_2 + d_2, 0, 0), \emptyset, s_3) \\
&, (s_2, \emptyset, (a_2 + d_2, b_2 + c_2, 0, 0), \emptyset, s_3) \\
&, (s_2, \emptyset, (a_2 + d_2, b_2 + d_2, 0, 0), \emptyset, s_3) \\
&, (s_3, \emptyset, (0, 0, a_2 + c_2, b_2 + c_2), \emptyset, s_4) \\
&, (s_3, \emptyset, (0, 0, a_2 + c_2, b_2 + d_2), \emptyset, s_4) \\
&, (s_3, \emptyset, (0, 0, a_2 + d_2, b_2 + c_2), \emptyset, s_4) \\
&, (s_3, \emptyset, (0, 0, a_2 + d_2, b_2 + d_2), \emptyset, s_4) \\
&, (s_4, (1 \to \Theta, 2 \to \Theta, 3 \to \Theta, 4 \to \Theta), (0, 0, 0, 0), \emptyset, s_5)\}
\end{aligned}$$

Here r_1 will represent the total sum we get when a_1 & a_2 are chosen by spellcaster, r_2 represent total sum when a_1 & b_2 are chosen by spellcaster, r_3 represents total sum when b_1 & a_2 is chosen by spellcaster and r_4 represents total sum when b_1 & b_2 chosen by spellcaster. So the automata automatically will pick one of the 64 options for schemas(4 options for s1 to s2 another 4 options for s2 to s3 and 4 options for s3 to s4 so total is 64 net choices.) if one of them is perfect, then for that set of choices, finally r_1, r_2, r_3, r_4 will all be Θ
the initial configuration is $(s_1, 0, 0, 0, 0)$ and the final goal configuration is $(s_5, \Theta, \Theta, \Theta, \Theta)$. Thus the final configuration is reachable iff there is a perfect schema.

Question 5

didnt get time to do this

Question 6

Question 1 was done by Aaditya and Shishir
Question 2 was done by Shishir
Question 3 was done by Shishir
Question 4 was done by Aaditya and Shishir