# Automated Reasoning IMC009
# Practical Assignment – Part 2

**Deadline: 20 December 2024**

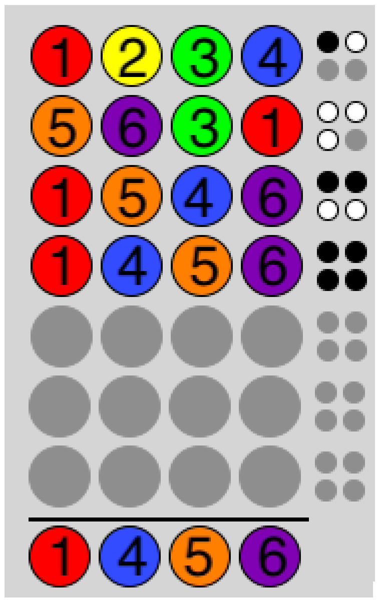
**Important** *In the report, we require an SMT formula for every program that you implement. Please carefully read the pdf file containing the assignment info before writing the report. Use the templates that we provide for all exercises.*

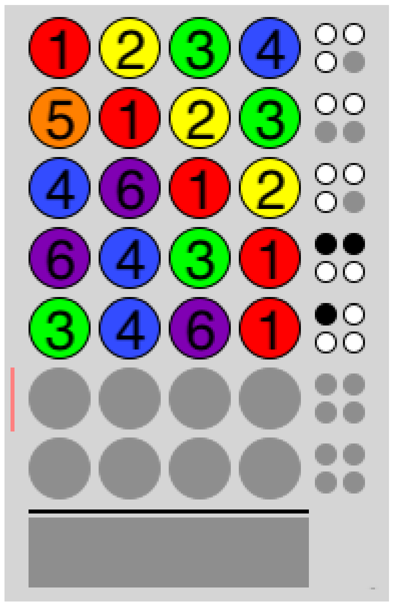
## 1 Mastermind

Mastermind is a two-person game where one person (the game master) chooses a combination of colours, and the second person (the player) has to guess the combination. The player can make a number of guesses, and for each guess, the game master informs them:

- for how many positions they chose the right colour at the right position
- for how many of the remaining positions, they chose a good colour, but at the wrong position

Two example gameplays with colours 1–6 and combinations of length 4 are given below. The black pegs next to a guess indicate the number of correct colours at the correct position; the white pegs indicate the number of correct colours at the wrong position. In the first, the player has already guessed the right combination; in the second, they can guess it in the next round (can you see what it should be?).





Of course, you can make this game much harder, by allowing longer combinations and more variety in colours. At this point, it can be quite difficult to find the solution – or even *any* combination that still satisfies the requirements. Fortunately, this is where SMT-solvers can help!

Your task will be to write a Python program that given the following information, guesses the secret combination:

- $N$: the length of the combination
- $K$: the number of colours
- a number of guesses, each along with the number of black and white pegs;

The colours are described as the numbers 1–$K$; for example the second problem above gives the following input file:

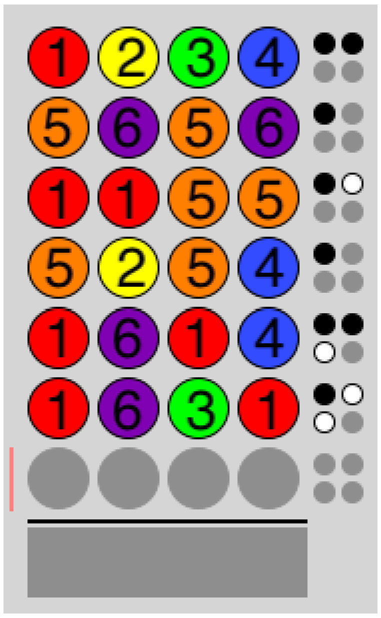
```
4 6
0 3 : 1 2 3 4
0 2 : 5 1 2 3
0 3 : 4 6 1 2
2 2 : 6 4 3 1
1 3 : 3 4 6 1
```

So $N = 4$ and $K = 6$. In the fourth guess, two of the guessed colours are at the correct position, and the other two colours still occur in the combination, but are in the wrong position.

**(a)** In the basic version of the game, *no duplicates* are allowed in the secret combination, nor in any of the guesses. For example, a combination (1,2,3,1) is *not* allowed, and will not be in any of the guesses.

Write a Python program that reads an input file describing a Mastermind puzzle (use the template we supply to avoid having to write your own parser), and outputs any duplication-free combination that might be what the game master has in mind. We have supplied a number of benchmarks for you to test your code with (input files starting with `unique_`).

**(b)** In the advanced version of the game, duplicates *are* allowed. An example of this gameplay is shown below. (The secret combination is at the bottom of the next page.)



To illustrate how the counting of the white pegs works, suppose the secret combination is (1,2,3,1,5). Then:

- If the player submits (7,1,7,7,7), the result would be one white peg (the single occurrence of colour 1 is at the wrong position).
- If the player submits (7,1,1,7,1), the result would be two white pegs (the first two occurrences of 1 are at the wrong position, the third is superfluous).
- If the player submits (7,1,1,1,1), the result would be one black peg and one white peg (one occurrence of 1 is at the correct position; of the remainder, the first is at the wrong position, while the rest is superfluous).

Write a Python program that reads an input file describing a Mastermind puzzle where the secret combination and guesses both may contain duplicates (use the template we supply to avoid having to write your own parser), and outputs any combination that might be what the game master had in mind. We have supplied a number of benchmarks for you to test your code with (input files starting with `arbitrary_`).

## Note:

In some of the supplied benchmarks, multiple correct solutions are possible. This is okay; it is your task to supply any solution that is still possible.

Depending on the quality of your implementation, some of the larger benchmarks might take a bit longer to solve. However, at least the small and medium benchmarks you should be able to do within one minute. If this is not the case, you should reconsider your encoding. (Hint: if you build an exponentially long formula, you are doing it wrong.)

The larger benchmarks can also be solved (well) within a minute, but this depends on more detailed decisions in your encoding. Solving these benchmarks is not required for a passing grade for this part of the assignment, but will be taken into account for the top scores.

P.S. the secret combinations that were not included are (6,3,4,1) for the example without duplicates, and (1,2,1,6) for the example with duplicates.

## 2 Group Project

In the Integrated Practicum for Groups course at Science Fiction University, all students following the class have to do a programming project in groups. There are a number of different projects, and all groups should have roughly the same size. The assignment of projects to students works as follows:

- There are $N$ students (named Alice, Bob, ...) and $P$ projects (A, B, C, ...). We will write *Students* for the set of students, and *Projects* for the set of projects.
- All the students make a preference, ranking all projects in the order they want to do them (ranking two projects equally is allowed). The project the student wants to do the most is assigned rank 1, the next rank 2 and so on. Hence, a preference is a function $\pi_s : \mathit{Projects} \to \{1, \ldots, P\}$ for a student $s \in \mathit{Students}$, where $P$ is the number of projects, and for instance $\pi_{\text{Alice}}(C)$ is the rank Alice has for project $C$.
- A *best possible* assignment of projects to students is chosen, where each project is assigned to either `round_down`$(N/P)$ or `round_up`$(N/P)$ students. An assignment is a function from *Students* to *Projects*. Given an assignment $A : \mathit{Students} \to \mathit{Projects}$, we define for $r \in \{1, \ldots, P\}$:

  $$N_A(r) = |\{s \in \mathit{Students} \mid \pi_s(A(s)) = r\}|$$

  which equals the number of students $s$ that, by $A$, are assigned to a project that they have ranked $r$ in their preference $\pi_s$.

  We say assignment $A$ is better than assignment $B$ if there is an $1 \leq R \leq P$ such that

  $$\begin{aligned} &N_A(r) = N_B(r) \text{ for all } R < r \leq P \\ &N_A(R) < N_B(R) \end{aligned}$$

  You may see $R$ as the worst rank where $A$ and $B$ differ, and $B$ having more students with that worst rank. This implies that $A$ has more students with better ranks than $R$.

For example: suppose there are three projects and seven students, who give the following preference:

- Alice B:1 C:1 A:3 (so she ranks B and C equally, and A below that)
- Bob C:1 B:2 A:3
- Carol B:1 C:1 A:3
- Dennis B:1 C:1 A:3
- Eliza B:1 C:2 A:3
- Fred C:1 B:2 A:3
- Gianna B:1 C:2 A:3
- Harry B:1 A:2 C:2 (so he ranks B as best, then A and C equally)

Then $A = \{$Dennis, Harry$\}$, $B = \{$Carol, Eliza, Gianna$\}$, $C = \{$Alice, Bob, Fred$\}$ is better than $A = \{$Gianna, Harry$\}$, $B = \{$Dennis, Eliza, Fred$\}$, $C = \{$Alice, Bob, Carol$\}$ because in both assignments one person gets their third choice (Dennis in the first, and Gianna in the second), but in the first assignment only one person gets their second choice (Harry), while in the second two people get their second choice (Harry and Fred). In this case, there is no assignment possible where no one gets their third choice. Of course, in a real class the problem is a bit larger and harder to oversee!

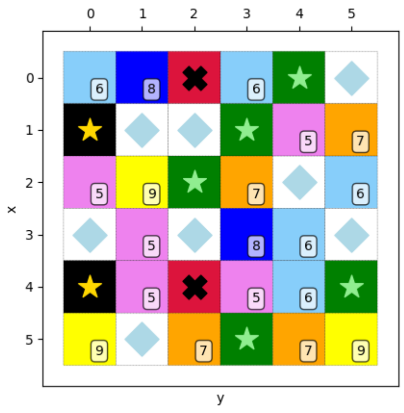
Write a program that reads the student preferences and returns a best-possible assignment. A template Python file and several challenge benchmarks are provided. For the preferences given in `ranking1.txt`, your results should include the following:

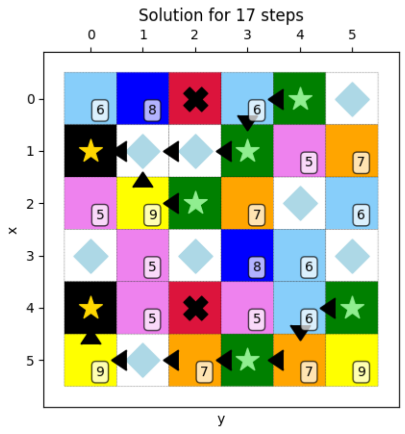
- A table that shows for each student to which project they are assigned, together with the corresponding rank.
- A table that, for each rank $r$, shows the number of students with rank $r$.

Note: if you cannot solve the exercise for all input files, you may still manually compute a solution for `ranking1.txt`. Since an iterative approach may take relatively long for the larger examples, if your program takes longer than one minute on any given benchmark, please make sure that you print progress updates between multiple calls to the SMT solver that illustrate clearly (1) that you are still making improvements, and (2) how much there is still to be done.

## 3 Robot Planning

Consider the NxN grids below.





A robot starts in one of the green cells with a star. The goal is to lead the robot to one of the black cells marked with a star in as few steps as possible. That is, we want to limit the length of the trajectory from start to target in the worst-case (over all starting positions) by a number $X$. It suffices to solve the decision problem:

> Given a value for $X$, is there a plan so that we will reach a `target` within $X$ steps from every `start` cell?

Plans are mappings from every cell to one of the possible directions $N, S, W, E$. Valid plans adhere to the following rules:

- A robot can move in one of the four cardinal directions: up (or $N$), down (or $S$), left (or $W$), right (or $E$). It will then reach the adjacent cell. It is allowed to stand still on an end point.
- Robots cannot fall off the grid. Instead, if they move towards a boundary, they will just stay where they are.
- The robot does not know where it is, but it can detect the surface type. Therefore, in every cell with the same surface type (described by a number/color), the robot must move in the same direction. The robot will not remember previous terrain types. The starts always have surface type 0, and the finish has surface type 1.
- A robot will catch fire if it visits a red `lava` cell (with a black cross, surface type 2). This makes it impossible to reach the end.
- On `sticky` patches (white with blue diamonds, surface type 4), the robot needs 7 time steps to actually finish the movement.

Write a program that solves the decision problem. Present your result in a table that, for each grid (grid_2024-A-0 , . . . , grid_2024-A-19), lists the smallest $X$ for which there is a plan so that a `target` is reached from every `start` cell. For grid_2024-A-10, show the full solution in a picture with direction-arrows (as in the example above).

**Template:** we provided two Python files: `robot.py` and `grid.py`. You only need to implement the function `solve` in `robot.py`. You may execute the file like this: `python robot.py <csv file>`. The file `grid.py` provides two classes that you should use in your solution. The `Cell` class represents a cell in the grid. The `Grid` class represents the entire grid (i.e., the set of cells and their colors), the solve function receives one grid as input. There are various helper functions you may use. Please use the following SMT notation (you do not need to explain these functions in the report).

| **Python function** | **SMT notation** |
|---|---|
| `grid.get_color(cell)` | $\mathsf{C}(x, y)$ |
| `grid.neighbours(cell, dir)` | $\mathsf{N}(x, y, d)$ |
| `grid.inv_neighbours(cell, dir)` | $\mathsf{N}^{-1}(x, y, d)$ |
| `grid.xdim` | $X$ |
| `grid.ydim` | $Y$ |
| `cell.x` | $x$ |
| `cell.y` | $y$ |

**Explanation**

$\mathsf{C}(x, y)$ gives the color (=observation) of cell $(x, y)$.

$\mathsf{N}(x, y, d)$ computes the neighbour cell in the direction $d$, for a cell $(x, y)$.

$\mathsf{N}^{-1}(x, y, d)$ computes the cell $(a, b)$ , such that $\mathsf{N}(a, b, d) = (x, y)$.

## 4 Multisets and MPO

Given a set $T$, equipped with an equality $\approx$, we define finite multisets over $T$. Intuitively, these are sets with finitely many elements from $T$ where order does not matter (like in usual sets), but the number of instances of each element does matter. Therefore, we can represent multisets over $T$ by finite lists over $T$, if we treat the lists as if the order of the list elements were irrelevant.

**Multiset equality** Given multisets $X$ and $Y$ over $T$, we can represent them as finite lists: $X = [x_1, \dots, x_n]$ and $Y = [y_1, \dots, y_m]$. We define $X \approx_{\text{mul}} Y$ if $n = m$ and there exists a bijection $\varphi : \{1, \dots, n\} \to \{1, \dots, n\}$ such that $x_{\varphi(i)} \approx y_i$, for all $1 \le i \le n$.

*Example*: if we take $T = \mathbb{N}$ and $\approx$ the usual equality on $\mathbb{N}$, then the list $[1, 1, 2, 3]$ denotes the multiset that contains two instances of 1, one instance of 2 and one instance of 3. We have $[1, 1, 2, 3] \approx_{\text{mul}} [1, 3, 2, 1]$ (we can choose $\varphi(1) = 1$, $\varphi(2) = 4$, $\varphi(3) = 3$, $\varphi(4) = 2$), but we do *not* have $[1, 1, 2, 3] \approx_{\text{mul}} [1, 2, 3, 3]$.

**(a)** We provided a template file `trs_solver.py`. Here, you will find a function, called `multiset_equality(X,Y)`, which is not yet implemented. The function should express that `X` and `Y` are multiset-equal. Your task is to implement this function.

**Multiset ordering** If, in addition, $T$ is equipped with a strict partial order $\succ$ then we define a multiset ordering as well: $[x_1, \dots, x_n] \succ_{\text{mul}} [y_1, \dots, y_m]$ if there exists a function $\chi : \{1, \dots, m\} \to \{1, \dots, n\}$ and a set $\mathsf{Strict} \subseteq \{1, \dots, n\}$ such that for all $1 \le i \le m$

- if $\chi(i) \in \mathsf{Strict}$ then $x_{\chi(i)} \succ y_i$
- if $\chi(i) \notin \mathsf{Strict}$ then $x_{\chi(i)} \approx y_i$
- if there exists $j \in \{1, \dots, m\} \setminus \{i\}$ such that $\chi(i) = \chi(j)$ then $\chi(i) \in \mathsf{Strict}$

*Example*: if we take for $\succ$ the usual order on $T = \mathbb{N}$ then we have $X = [5, 3, 1, 1, 1] \succ_{\text{mul}} [4, 3, 3, 1] = Y$: take $n = 5$, $m = 4$, $\mathsf{Strict} = \{1, 4, 5\}$, $\chi(1) = \chi(3) = \chi(4) = 1$ and $\chi(2) = 2$.

**(b)** The same question as in part **(a)**, but now for `multiset_ordering_greater(X,Y)`.

**Multiset Path Ordering** ***Note: only do this part of the assignment after the lecture that includes LPO.***

We assume some term rewriting system $\mathcal{R}$ and let $\mathcal{F}$ be the set of all function symbols that occur in $\mathcal{R}$, and $\mathcal{V}$ be the set of all variables that occur in $\mathcal{R}$. From now on, we fix $T$ to be $T(\mathcal{F}, \mathcal{V})$, the set of all terms over $\mathcal{F}$ and $\mathcal{V}$.

Assume a relation $\trianglerighteq$ on $\mathcal{F}$ such that for all $\mathtt{f}, \mathtt{g} \in \mathcal{F}$ we have $\mathtt{f} \trianglerighteq \mathtt{g}$ or $\mathtt{g} \trianglerighteq \mathtt{f}$. Furthermore, assume $\trianglerighteq$ is transitive (if $\mathtt{f} \trianglerighteq \mathtt{g} \trianglerighteq \mathtt{h}$ then $\mathtt{f} \trianglerighteq \mathtt{h}$) and reflexive (always $\mathtt{f} \trianglerighteq \mathtt{f}$). We denote $\mathtt{f} \triangleright \mathtt{g}$ if $\mathtt{f} \trianglerighteq \mathtt{g}$ and not $\mathtt{g} \trianglerighteq \mathtt{f}$. We denote $\mathtt{f} \equiv \mathtt{g}$ if $\mathtt{f} \trianglerighteq \mathtt{g}$ and $\mathtt{g} \trianglerighteq \mathtt{f}$.

We inductively define relations $\succsim$, $\succ$ and $\approx$ on $T$ as follows:

1. $s \succsim t$ if and only if $s \succ t$ or $s \approx t$
2. $s \succ t$ if and only if $s = \mathtt{f}(s_1, \dots, s_n)$ and at least one of the following holds:

**(Sub)** $s_i \succsim t$ for some $i \in \{1, \dots, n\}$

**(Copy)** $t = \mathtt{g}(t_1, \dots, t_m)$ with $\mathtt{f} \rhd \mathtt{g}$ and $s \succ t_i$ for all $i \in \{1, \dots, m\}$

**(Mul)** $t = \mathtt{g}(t_1, \dots, t_m)$ with $\mathtt{f} \equiv \mathtt{g}$ and $\{\{s_1, \dots, s_n\}\} \succ_{\text{mul}} \{\{t_1, \dots, t_m\}\}$

3. $s \approx t$ if and only if one of the following holds:

**(Var)** $s$ and $t$ are both the same variable;

**(Fun)** $s = \mathtt{f}(s_1, \dots, s_n)$ and $t = \mathtt{g}(t_1, \dots, t_n)$ and $\mathtt{f} \equiv \mathtt{g}$ and $\{\{s_1, \dots, s_n\}\} \approx_{\text{mul}} \{\{t_1, \dots, t_n\}\}$

**(c)** Your task is to implement the multiset path ordering! But don't worry, we have supplied a template, which includes a parser and the basics as discussed in the lecture.

We represent the precedence $\rhd$ as natural numbers with one variable `var_precedence(f)` for each function symbol $\mathtt{f} \in \mathcal{F}$ s.t.

$$\mathtt{f} \rhd \mathtt{g} \text{ iff } \mathtt{var\_precedence(f)} > \mathtt{var\_precedence(g)}.$$

Similar to the Lexicographical Path Ordering, we will consider every rule $\ell \to r \in \mathcal{R}$, every subterm of $\ell'$ of $\ell$, every subterm $r'$ of $r$, and every

$R \in \{$`Geq`, `Greater`, `Equal`, `GreaterSub`, `GreaterCopy`, `GreaterMul`, `EqualVar`, `EqualFun`$\}$

and create a variable $\langle l' \ R \ r' \rangle$ and a formula $\langle l' \ R \ r' \rangle \to \varphi(l', r', R)$ where $\varphi(l', r', R)$ is fully determined by $R$. Your task is to implement $\varphi(l', r', R)$ for each $R$. For example, we have already implemented these rules in the template:

$$\varphi(l', r', \mathtt{Greater}) = \langle l' \ \mathtt{GreaterSub} \ r' \rangle \vee \langle l' \ \mathtt{GreaterMul} \ r' \rangle \vee \langle l' \ \mathtt{GreaterCopy} \ r' \rangle$$

$$\varphi(l', r', \mathtt{EqualFun}) = \begin{cases} n = m \wedge \mathtt{var\_precedence(f)} = \mathtt{var\_precedence(g)} \wedge \\ \mathtt{multiset\_equality([s1,\dots,sn],[t1,\dots,tn])} \\ \text{If } l' = \mathtt{f}(s_1, \dots, s_n) \text{ and } r' = \mathtt{g}(t_1, \dots, t_m) \\ \\ \bot \text{ Otherwise} \end{cases}$$

Your task is to implement the remaining $\varphi(l', r', R)$.
After you did this, you can answer the following question by running `python trs_solver.py input.trs` and printing the true variable precedences of the model. Assume

$$\mathcal{R} = \left\{ \begin{aligned} \mathtt{f}(\mathtt{a}(x,y), \mathtt{g}(x,y), \mathtt{a}(x, \mathtt{g}(z,u))) &\to \mathtt{f}(\mathtt{a}(x,z), y, \mathtt{g}(\mathtt{g}(\mathtt{g}(y,x),x),x)) \\ \mathtt{c}(x,y,u,v) &\to \mathtt{f}(\mathtt{a}(x,y), \mathtt{f}(u,u,u), \mathtt{g}(v, \mathtt{f}(x,y,u))) \\ \mathtt{h}(\mathtt{g}(x, \mathtt{g}(u,z)), \mathtt{c}(x,y,x,z)) &\to \mathtt{a}(\mathtt{d}(x,z), u) \\ \mathtt{a}(\mathtt{f}(x,y,z), u) &\to \mathtt{h}(u, \mathtt{a}(x, \mathtt{h}(y,x))) \\ \mathtt{h}(\mathtt{d}(\mathtt{a}(x,y), \mathtt{g}(u,v)), \mathtt{a}(x,y)) &\to \mathtt{a}(\mathtt{c}(u,x,v,y), \mathtt{g}(y,x)) \\ \mathtt{f}(\mathtt{b}(x,y,z), y, x) &\to \mathtt{c}(x,x,y,x) \end{aligned} \right\}$$

So $\mathcal{F} = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{f}, \mathtt{g}, \mathtt{h}\}$ and $\mathcal{V} = \{x, y, z, u, v\}$.

Give an ordering on $\mathcal{F}$ that orients $\mathcal{R}$ with the multiset path ordering. Provide a human-readable proof for the orientation of the third rule.

**(d)** Does there exist another ordering on $\mathcal{F}$ that also orients all the rules? If so, provide this alternative ordering in your report.