Huawei UKRD University Challenge 2023

Artemiy Margaritov

Cambridge CPU Design Team, Huawei UK

1 A Tale About A Curious Robot

In a galaxy not so far away, there lived a little robot named Robo. Robo had a special job: traveling from planet to planet in a tiny spaceship. The command center controls Robo's moves in space, telling Robo which planets it should go to, one by one. Once Robo arrives on a planet, Robo reports to the command center, and the command center instructs Robo where to go next by sending a 32-bit planet ID which Robo maps to coordinates using a spaceship computer and embarks on its next voyage. The command center can ask Robo to go to a planet that Robo has already visited on its route.

As Robo zipped through space, he noticed something interesting. Sometimes, when he landed on a planet, it was a sunlit day, and sometimes it was a night and complete darkness. Robo's sensors can only detect the presence or the absence of light, so there are only two outcomes for time-of-day for Robo: Day (1) or Night (0).

The command center stores the list of planets Robo needs to visit as an ordered sequence of planet IDs. An ordered sequence of planet IDs is called a route. The command center has a powerful computer that can calculate if it will be day or night when Robo arrives on every planet while it traverses the route. The command center keeps the results received by the computer in a table where each row holds two items: a planet ID Robo needs to visit (in column 1) and time-of-day (Day or Night) which Robo will see on that planet once Robo arrives there (in column 2). An example of a Robo's route is given in Table 1.

Planet ID	**Day or Night**
5478932	NIGHT
2104549	DAY
789021	DAY
634789	NIGHT
5478932	DAY
8923401	NIGHT

Table 1: Sample Robo's route. The actual route lists 7 million visits to planets.

Now, let's warp into your challenge! Robo, with its clever space senses, has unveiled a space-time puzzle during its cosmic travels. There is a connection between the time of day Robo sees on a planet A and the time of day on previous planets that Robo visited before A. For example, for some planets A and B, if it's daytime on planet A and Robo jets over to planet B from A, guess what? It's always daytime on planet B too! Isn't space full of cosmic enigmas?

Here is a riddle for you to crack: Robo wants to play a guessing game and predict whether it'll be daytime or nighttime on the next planet once the command center tells Robo which planet

should be next. And now, brace yourselves for the cosmic twist of twists! Robo's free memory is a mere 64KiB. In addition, Robo's circuits have spare time to execute only a limited number of instructions each time when Robo takes off a planet. Everything else is reserved for Robo's navigation talents! Thus, there are only 700 computational units (we share the infrastructure for computing the computational cost) for recording and predicting the time of day for each next planet. Your mission? Create the ultimate Time-of-day Predictor Algorithms for Robo! There are three scenarios in which Robo needs your help:

Warm-up Task **Memorization Techniques**: Imagine you are Robo's programmer and you need to help Robo master the game of predicting the time of day on the next planet before Robo arrives there based on the history of patterns in the daytime Robo observed. The goal in the game is to make the highest number of correct predictions of the time of day on the next planet as Robo travels along the route given by the command center. Please don't violate the Robo's memory and compute limitations! Otherwise, Robo can go astray and get lost in space.

The command center route and the list of time of days that Robo will see is available in advance for analysis by Robo's programmer which can help to develop the best memorization policy for Robo. However, once Robo starts his trip, a programmer can't interfere with it (as Robo gets far away in space).

You need to implement C/C++ functions that Robo can use

- to record the time of day Robo observes on a planet in his restricted memory capacity and
- to use the records from memory for making predictions about the time of day on the next planet using available Robo circuits.

Hint: Robo does not necessarily need to record time-of-day in its memory on every planet: it could be beneficial to skip updates if they don't help Robo to get better in the game of predicting time-of-day on the next planet.

Note that the solutions for this task won't be scored. You don't need to submit solutions for this task. However, being a simple task, this task can be a good starting point and help to find winning solutions for the other tasks.

Task1 **Spaceship Oracle**: Robo shares his game with the spaceship computer which is Robo's good friend. The spaceship computer agrees to beep once the command center tells the next planet to go if the spaceship computer thinks that at that planet, it would be daytime when they arrive. Thus, Robo can seek suggestions from its spaceship friend, even though it doesn't know the true formula the spaceship computer is using. How can Robo incorporate the spaceship's suggestions into its memorization and prediction strategies?

Task2 **Astro Atlas**: Robo stumbles upon a cosmic treasure: a book listing all the planets, neatly divided into 1024 groups. In the book, each planet is mapped to a group ID. When Robo needs to make a prediction Robo can consult the book and read the group ID of the next planet which the command center sent Robo to. Your task is to find the best planet classifier book to give to Robo and guide Robo on how to use the group ID for prediction to increase daytime prediction accuracy.

1.1 Task 1 Requirement

The goal of Task 1 is to implement methods for

- memorizing the time-of-day Robo observes on a planet in his restricted memory capacity and
- using the records from the memory for making predictions about the time-of-day on the next planet.

The average computational cost spent on predictions of time of day must be lower than the threshold of 700 units. Section 2.1 explains how the computational cost is calculated. Section 2.4 explains how to measure the average computational cost.

Prediction algorithm evaluation criteria. Each solution is evaluated by computing the score using the following formula:

$$score = \begin{cases} 0, & \text{if } (\textit{average computational cost per planet} > 700) \\ 0, & \text{if } \textit{Robo uses more than } 64\textit{KiB at any point} \\ predictionAccuracy, & \text{otherwise} \end{cases}$$

A solution with the highest score wins. The route that will be used for evaluation is provided as part of the simulation infrastructure. Section 2.4 explains how to measure the prediction accuracy and the average computational cost of a solution. Section 2.3 explains how to avoid violating the requirement using no more than 64KiB as Robo's internal memory.

Submission format. Submissions are expected as a .tar.gz file including all relevant source files and scripts/Makefiles to evaluate the solution. The methods should be implemented in C/C++ and should support compiling to a shared library. Section 2.3 explains the required methods' interface and describes the templates available. All source files as well as a Makefile (and any other files required to build the shared library) should be a part of the submission archive. We provided a sample submission archive `task1/sampleSubmission.tar.gz` for your reference.

1.2 Task 2 Requirement

Task 2 consists of two connected sub-tasks:

- find and add a single group tag for each unique planet ID in a given route; a group tag must be an integer from the region from 0 to 1023; each unique planet ID must be associated with the same group tag on all occurances of that planet ID in the route; a group ID may be associated with any number of unique planet IDs.
- implement methods that take advantage of receiving a group tag together with planet ID:
  - memorizing the time-of-day Robo observes on a planet in his restricted memory capacity and
  - using the records from the memory for making predictions about the time of day on the next plant (given the next planet's ID and group tag).

Similarly to the restriction of Task 1, the average computational cost spent on predictions of the time of day must be lower than the threshold of 700 units. Section 2.1 explains how the computational cost is calculated. Section **??** explains how to measure the average computational cost. There are no restrictions on the usage of computation resources and memory for methods for determining group tags.

Prediction algorithm evaluation criteria. Each solution is evaluated by computing the score using the following formula:

$$score = \begin{cases} 0, & \text{if } (average\ computational\ cost\ per\ planet > 700) \\ 0, & \text{if } Robo\ uses\ more\ than\ 64KiB\ at\ any\ point \\ 0, & \text{if } if\ conditions\ on\ group\ tags\ are\ violated \\ predictionAccuracy, & \text{otherwise} \end{cases}$$

A solution with the highest score wins. The route that should be augmented with group tags and used for evaluation is provided as part of the simulation infrastructure. Section 2.4 explains how to measure the prediction accuracy and the average computational cost of a solution. Section 2.3 explains how to avoid violating the requirement using no more than 64KiB as Robo's internal memory.

Submission format. Similarly to task 1, the submission is expected as a .tar.gz archive file. The archive should include the following items:

- A modified route file named `atlas_route.txt` that includes group tags for each planet ID in the third column (where tab is used as a delimiter). The format of the expected route file is described in Section 2.7.
- A 1-page pdf file named `report.pdf` that describes the ideas/methodology used for determining the group tags. A directory name `grouptag_methods` includes all relevant code used for determining group tags for a given route. Note that this code won't be re-run as part of the evaluation. Code and the report will be used for plagiarism checks.
- The submission archive should include all relevant code and scripts to build `libTask2PredictionAlgorithm.so` shared library implementing a solution. Similarly to task 1 requirements, the methods for memorizing the route and making predictions should be implemented in C/C++. Files `PredictionAlgorithm.{hpp,cpp}` that define and declare the methods. Also, Makefile(s) for building the `libTask2PredictionAlgorithm.so` must be present in a submission archive. We provide templates for these files under the `task2/PredictionAlgorithm` directory.

2 How to Program Robo?

2.1 Task 1 Robo Simulator Basics

We share C++ source code for a simulator of Robo's voyage on a given route.

The simulator has to be linked with a shared library providing implementation of Robo's prediction algorithm. The templates for those functions can be found in the `task1/PredictingAlgorithm` directory.

The simulator iterates over a given route and queries Robo's prediction algorithm. The simulator measures prediction accuracy and returns this in its output at the end of an evaluation. In addition, the simulator also measures *average computational cost* of a prediction algorithm. The average computational cost can be found in the simulator's output at the end of an evaluation. The computational cost is calculated using the following formula:

$$\begin{aligned} computationalCost = {} & number\ of\ bitwise\ instructions \\ & + number\ additive\ instructions\ * 3 \\ & + number\ of\ multiplicative\ instructions * 7 \end{aligned} \tag{1}$$

where:

- *number of bitwise instructions* is the number of bit-manipulating instructions
- *number of additive instructions* is the number of instructions performing any addition or subtraction (including floating point operations)
- *number of multiplicative instructions* is the number of instructions performing any multiplication or addition (including floating point operations)

The average computational cost is calculated by dividing the total computation cost of all predictions by the number of visited planets:

$$averageComputationalCost = \frac{computationalCost}{totalNumberOfVisitedPlanets} \tag{2}$$

2.2 Learning Robo Simulator for Task 1

1. **Download the simulator files.** The full simulator is available on `github.com`. Installing the simulator requires 1GB of disk space. To clone the Git repository with the simulator, use the following command:

   ```
   git clone https://github.com/amargaritov/techarena23_playground.git
   ```

2. **Setup bash environment.** Before you start using the infrastructure, you need to set up certain bash environment variables. To do this, follow these steps:

   ```
   cd techarena23_playground/task1; source source.sh
   ```

3. **Setting up the compiler toolchain.** To build your solution, you will need a specific Docker image with the toolchain. The toolchain is capable of building a simulator that can measure the computational cost of a prediction algorithm. The toolchain is based on LLVM 16.0.6. In addition, unzipping the experiment route files is required. To achieve both run:

   ```
   scripts/setup_infrastructure.sh
   ```

2.3 Implementing Your Solution for Task 1

Let's have a look at what parts a Robo's predictor must have. We prepared a template of a `RoboPredictor` structure in `task1/PredictionAlgorithm/PredictionAlgorithm.hpp`:

```
struct RoboPredictor {
    struct RoboMemory;
    RoboMemory* roboMemory_ptr;

    bool predictTimeOfDayOnNextPlanet(std::uint64_t nextPlanetID,
                                      bool spaceshipComputerPrediction);

    void observeAndRecordTimeofdayOnNextPlanet(std::uint64_t nextPlanetID,
```

```
                                                    bool timeOfDayOnNextPlanet);
};
```

Explanation:

1. `struct RoboMemory:`
   (a) This structure is designed to hold all the state which Robo's memory accumulates while making predictions and observing the patterns.
   (b) You would need to complete the definition of this structure in `task1/PredictionAlgorithm/PredictionAlgorithm.cpp` by populating this structure with appropriate content.
   (c) **Requirement:** The RoboMemory structure (as well as its member objects) must allocate all member objects statically. Dynamic memory allocation is prohibited.
   (d) **Requirement:** The size of this structure must not exceed 64KiB.
2. `function predictTimeOfDayOnNextPlanet:`
   (a) This function should return a prediction for DAY (`true`) or NIGHT (`false`) on the `planetID` given as an input.
   (b) This function can consider a spaceship's suggestion given as an input.
   (c) This function can read/write from the `RoboMemory` object.
   (d) **Requirement:** This function (as well as all objects it creates) must allocate all objects statically. Dynamic memory allocation is prohibited.
3. `function observeAndRecordTimeofdayOnNextPlanet:`
   (a) This function is designed to update `RoboMemory` once a correct output for a given planet is known.
   (b) This function is always called after the `predictTimeOfDayOnNextPlanet`.
   (c) This function can read/write from the `RoboMemory` object.
   (d) **Requirement:** This function (as well as all objects it creates) must allocate all objects statically. Dynamic memory allocation is prohibited.

The methods should be implemented as a shared library named `libTask1PredictionAlgorithm.so` located in the `task1/PredictionAlgorithm` directory. That directory should contain a makefile for building the library. A template of the makefile is provided as `task1/PredictionAlgorithm/Makefile`. Please ensure that your solution follows these specifications.

2.4 Testing Your Solution for Task 1

1. **Compiling Your Prediction Algorithm.** To compile your solution into a shared library using the custom toolchain, use the following command:

   `scripts/build.sh`

This command builds your solution as a shared library `libTask1PredictionAlgorithm.so` located under `task1/PredictionAlgorithm` directory, and links the task1 simulator (the `task1/task1` executable file) with the shared library.

2. **Evaluating Your Prediction Algorithm.** To evaluate your solution using the custom toolchain, use the following command:

   ```
   scripts/evaluate.sh -r routes/route.txt
   ```

   Below is a sample prediction algorithm evaluation output of the simulator:

   ```
   Loading Robo's route for evaluation from routes/route.txt file...
   Starting evaluation of Robo's prediction algorithm...
    99% [|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||| ]
   Total number of planets visited 7000000
   Prediction accuracy 77.7%
   Number of additive instructions: 7003989 (1.000570 per planet)
   Number of multiplicative instructions: 7000000 (1.000000 per planet)
   Number of bitwise instructions: 98000000 (14.000000 per planet)
   Metric of computational cost: 168096117 (24.001709 per planet)
   ```

   There are two important metrics in the output that are used during calculating a score:

   (a) **Prediction accuracy.** This metric shows the fraction of correct predictions among all predictions.

   (b) **Metric of computational cost.** This metric shows how much computing power Robo spends on predictions. Section 2.1 explains how the simulator calculates this value.

   Other useful metrics:

   c. **Number of additive instructions.** This metric shows how many additive instructions were used while making predictions. This metric is used to compute the computational cost using the formula given in Section 2.1.

   d. **Number of multiplicative instructions.** This metric shows how many multiplicative instructions were used during making predictions. This metric is used to compute the computational cost using the formula given in Section 2.1.

   e. **Number of bitwise instructions.** This metric shows how many bitwise instructions were used during making predictions. This metric is used to compute the computational cost using the formula given in Section 2.1.

2.5 Task 2 Robo Simulator Basics

For task 2, we share a different version of Robo's voyage C++ simulator: the simulator for task 2 expects an atlas route as an input and prediction-related methods to be able to leverage the group tag information. All the files related to task 2 can be found under `task 2`.

The task 2 simulator has to be linked with a shared library providing implementation of Robo's prediction algorithm that takes into account group tags. The templates for those functions can be found in the `task2/PredictingAlgorithm` directory.

The evaluation simulator procedure and outputted metrics are similar to that of the simulator for task 1 (see Section 2.1 for more details).

Planet ID	Day or Night
5478932	NIGHT
2104549	DAY
789021	DAY
634789	NIGHT
5478932	DAY
8923401	NIGHT

Table 2: Sample Robo's route.

Planet ID	Day or Night	Planet group tag
5478932	NIGHT	7
2104549	DAY	118
789021	DAY	67
634789	NIGHT	4
5478932	DAY	7
8923401	NIGHT	67

Table 3: Sample Robo's atlas route.

2.6 Setting up Simulator for Task 2

To set up the infrastructure for task 2, please use the commands listed below. Please note that for working on task 2, sourcing `source.sh` from the `task 2` directory is required.

```
# step 1
git clone https://github.com/amargaritov/techarena23_playground.git
# step 2
cd techarena23_playground/task2; source source.sh
# step 3
scripts/setup_infrastructure.sh
```

2.7 Atlas Requirements for Task 2

An atlas route can be created from the original route by adding a third column on separated by a tab delimiter. The goal is to select group tags in a way that makes the task of predicting the time of day on the next planet easier and increases the accuracy. The planet group task must not violate the following requirements:

- only one group tag value can be associated with a planet ID throughout the whole route
- group tags is a value in a range from 0 to 1023

Table 3 demonstrates an example of an atlas route created from an ordinary sample route listed in Table 2.

2.8 Implementing Your Solution for Task 2

We prepared a template of a `RoboPredictor` structure that can leverage planet group tags in `task2/PredictionAlgorithm/PredictionAlgorithm.hpp`. The `RoboPredictor` structure for task 2 is largely similar to the ones described in Section 2.3. The only difference is that a function for predicting the time of day on the next planet can accept a group tag as an additional parameter:

```
struct RoboPredictor {

    bool predictTimeOfDayOnNextPlanet(std::uint64_t nextPlanetID,
                                      bool spaceshipComputerPrediction,
                                      int nextPlanetGroupTag); //
};
```

Note that in the simulator for task 2, the function `observeAndRecordTimeofdayOnNextPlanet` is following the description of the corresponding function from task 1 given in Section 2.3.

2.9 Testing Your Solution for Task 1

1. **Compiling Your Prediction Algorithm.** To compile your solution into a shared library using the custom toolchain, use the following command:

   ```
   scripts/build.sh
   ```

   This command builds your solution as a shared library `libTask1PredictionAlgorithm.so` located under `task1/PredictionAlgorithm` directory, and links the task1 simulator (the `task1/task1` executable file) with the shared library.

2. **Evaluating Your Prediction Algorithm.** To evaluate your solution using the custom toolchain, use the following command:

```
scripts/evaluate.sh -r routes/sample_atlas_route.txt #or
scripts/evaluate.sh -r <path to your atlas_route>
```

Note that an atlas route must be placed under the task2 directory and only relative path could be passed to the evaluate script. Note that we won't evaluate your prediction algorithm on the `sample_atlas_route.txt` but only on the `atlas_route.txt` which is included into your submission.

Congratulations! You have successfully set up and learned to use the infrastructure and implemented your solution for both Task 1 and Task 2. Follow these steps to build and evaluate your solution effectively. Good luck!