­­Part 1 of the task was relatively easy to complete, as the robot relies entirely on the light sensor to determine where to move. The raw code can be found in the appendix, but the basic algorithm is as follows:

* A global variable totalBlackTile keeps a count of the number of black tiles passed, and is updated during the course of the program
* The method moveForwardWithSpeed takes an argument movingSpeed, sets both motors to run at that speed, and continues until it detects a colour change from white to black. totalBlackTile is incremented and the robot beeps
* The main method calls moveForwardWithSpeed to move off the starting tile and onto the line of black and white tiles, then calls the turnOneMotor method to turn the robot 90­o
* While totalBlackTile is less than or equal to 15, the main method repeatedly calls moveForwardWithSpeed followed by the testBoundary method. There is also a global variable difference, which stores the difference between two colours detected by the light sensor
* when totalBlackTile reaches 15, the robot uses turnOneMotor to turn 90o to the right so it faces the end tile and is ready for part 2

One problem we faced with this part of the task was that we discovered the lighting in the testing area is not very consistent, meaning the robot’s light sensor would behave differently depending on weather, time of day and other factors. At first we were using the light sensor’s reflected light function, using the built-in getColorReflected method which returns a value from a red light between 0 and 100; taking the difference between our “current” and “previous” variables we could determine when the robot had moved from one tile to the next one of a different colour. We discovered however that it was easier and more accurate to use the getColourName method, which uses a more complex RGB light sensor to return an object representing one of eight colours, since it does a much better job of differentiating between black and white.

Another problem we found was that the robot’s wheels tended to get caught in the grooves on each side of the row of tiles, which would cause it to veer off in one direction. At first we changed the code so that the left wheel would just go slightly faster than the right (since the direction was always the same), but clearly this solution wasn’t very robust.

Eventually we came up with a method called testBoundary to solve the problem:

* testBoundary relies on the method turnOneMotorUntilMeetBoundary, which simply rotates the robot by driving a given motor until the sensor detects a colour change and returns the degree the wheel has turned by. testBoundary calls this method once for each wheel and stores the returned value in two variables. The method then takes the absolute value of the difference and uses this to correct the path of the robot by turning a certain distance in the opposite direction.

Hence part 1 is broken up into a single task, repeated 15 times and consisting of a single tile, and the path correction is performed once each time the task is performed. The task is very simple: the robot continues in a straight line until it reaches a black tile, then stops. It checks its position in the black tile relative to the edges using testBoundary, which also corrects the direction in which the robot is travelling if needed.

Part 2 of the task was harder, as we decided to use the sonar sensor to adjust the path of the robot if needed. The robot initially just turns 90o to face the end tile and moves forward for a few seconds, but after that it acts as follows:

* while the two bumpers have not hit anything (ie. while they return a value of 0), the moveForwardWithSpeed method is called followed by the scan method
* scan() simply turns the robot in each direction by calling the turn method
* turn() takes as parameters a maximum degree by which the wheels should be rotated (degreeToTurn) and a turning speed. It rotates the robot in one direction if degreeToTurn is positive and the opposite if negative, taking readings from the sonar sensor the whole time; it returns from the method when the distance it reads from the sensor gets smaller, meaning the robot will end up facing the tower on top of the end tile. The actual degree the robot turns is affected by the global variable targetDistance, which is set to the current distance to the target each time. This is to avoid accidentally reading a different target (for example a bystander’s leg); the robot will only face towards the object it has read if the difference between it and the object is less than targetDistance – meaning the robot is getting closer. scan() uses a boolean value set by turn() to determine when to stop rotating – this should result in the robot facing the end tile in all cases as it will keep moving straight forward if no object is detected.
* If the bumpers detect a collision at any point, the robot is assumed to have reached the end tile. It then continues to move forward for one second to push the tower off the tile, then the program terminates.

In theory the scan and turn methods are not actually needed, as the robot could simply drive forward until its bumpers detect a collision, but given our experience with the robot accidentally changing course in part 1 we decided it was better not to take chances.

One final thing we noticed is that the code used to turn our robot is very error-prone and hard to debug, so we decided to encapsulate the algorithms in a couple of functions (turn and turnOneMotor) to make everything easier.