City of Lights

In this project, we are interested in specifying and proving correct two programs that solve one of the programming problems of the ICPC SWERC 2018 semi-finals (Southwestern Europe Regional Contest for on-site programming for students, `https://swerc.eu/2018`). The problem, called "City of Lights", is described in details in Section 1 of this document.

This project is to be carried out using the Why3 tool (version 1.2.1), in combination with automated provers (Alt-Ergo 2.3.0, CVC4 1.7 and Z3 4.8.6). You can use other automatic provers or versions if you want, if they are freely available and recognized by Why3. You may use Coq for discharging particular proof obligations, although the project can be completed without it. The installation procedure may be found on the web page of the course at URL `https://www.lri.fr/~marche/MPRI-2-36-1/install.html`.

The project must be done individually—team work is not allowed. In order to obtain a grade for the project, you must send an e-mail to `Claude.Marche@inria.fr` and `Jean-Marie.Madiot@inria.fr`, no later than **Friday, February 7th, 2020** at 22:00 UTC+1. This e-mail should be entitled "MPRI project 2-36-1", be signed with your name, and have as attachment an archive (`zip` or `tar.gz`) storing the following items:

- The source file `lights.mlw`.
- The content of the sub-directory `lights` generated by Why3. In particular, this directory should contain session files `why3session.xml` and `why3shapes.gz`, and Coq proof scripts, if any.
- A PDF document named `report.pdf` in which you report on your work. The contents of this report counts for at least half of your grade for the project.

The report must be written in French or English, and should typically consist of 3 to 6 pages. The structure should follow the sections and the questions of the present document. For each question, detail your approach, focusing in particular on the design choices that you made regarding the implementations and specifications. In particular, loop invariants and assertions that you add should be explained in your report: what they mean and how they help to complete the proof.

A typical answer to a question or step would be: *"For this function, I propose the following implementation: [give pseudo-code]. The contract of this function is [give a copy-paste of the contract]. It captures the fact that [rephrase the contract in natural language]. To prove this code correct, I need to add extra annotations [give the loop invariants, etc.] capturing that [rephrase the annotations in english]. This invariant is initially true because [explain]. It is preserved at each iteration because [explain]. The post-condition then follows because [explain].*"

The reader of your report should be convinced at each step that the contracts are the right ones, and should be able to understand why your program is correct, e.g. why a loop invariant is initially true, why it is preserved, and why it suffices to establish the post-condition. It is legitimate to copy-paste parts of your Why3 code in the report, yet you should only copy the most relevant parts, not all of your code. In case you are not able to fully complete a definition or a proof, you should carefully describe which parts are missing and explain the problems that you faced.

In addition, your report should contain a conclusion, providing general feedback about your work: how easy or how hard was it, what were the major difficulties, was there any unexpected result, and any other information that you think is important to consider for the evaluation of the work you did.

1 The "City of Lights" Problem

We give here the exact transcript of the problem statement, extracted from the document `https://swerc.eu/2018/theme/problems/swerc.pdf`.

Paris has been called “ville lumière” (city of lights) since the 17th century. It earned this nickname in part because of the many city lights illuminating famous sites such as monuments, statues, churches, or fountains.

Those public lights in Paris are numbered from 1 to N and are all on by default. A group of hackers has gained the capability to toggle groups of lights. Every time the hackers use their program, they cause a number i (that they cannot control) to be sent to the system controlling the city lights. The lights numbered i, $2i$, $3i$, and so on (up to N) then change state instantly: lights that were on go off, and lights that were off go on.

During the night, the hackers use their programs k times. What is the greatest number of lights that are simultaneously off at the same time?

Input

The input comprises several lines, each consisting of a single integer:

- The first line contains the number N of lights.
- The second line contains the number k of uses hackers’s program.
- The next k lines contain a number i sent to the system controlling the lights.

Limits

- $1 \leq N \leq 1000000$;
- $1 \leq k \leq 100$;
- $1 \leq i \leq N$.

Output

The output should consist of a single line, whose content is an integer, the greatest number of lights that are simultaneously off at the same time

Sample Input

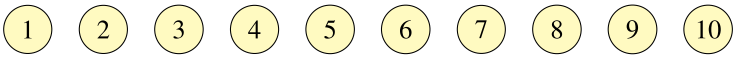
```
10
4
6
2
1
3
```

Sample Output

```
6
```

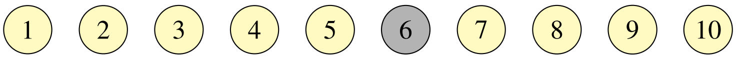
Sample Explanation

We start with a group of 10 lights which are all on.



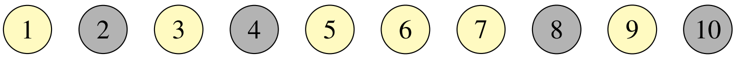
← 0 light is off

The hackers send the number 6: light 6 is toggled.



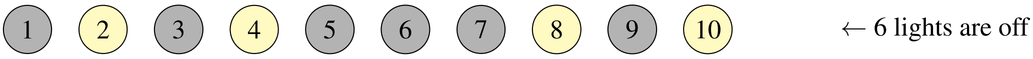
← 1 light is off

They then send the number 2: lights 2, 4, 6, 8, and 10 are toggled.

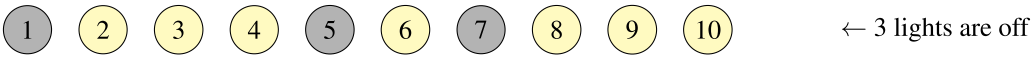


← 4 lights are off

The number 1 is then sent: all lights are toggled.



They end up sending the number 3: lights 3, 6, and 9 are toggled.



The maximum number of lights off at the same time was 6

2 Specification of the Problem

We propose to model the problem by some first-order specifications which start by defining a data type describing the full history of the sequence of turning lights off and on:

```
type lighting_configuration = {
  number_of_lights : int;
  number_of_use_of_hackers_program : int;
  chosen_light_index : int → int;
  light_on : int → int → bool;
  }
```

The fields `number_of_lights` and `number_of_use_of_hackers_program` respectively represent the constants N and k of the problem. The pure function `chosen_light_index` represents the sequence of light numbers sent by the hackers' program to the system controlling the lights, that is for each j, $1 \leq j \leq k$, (`chosen_light_index` j) denotes the light index i, $1 \leq i \leq N$, sent to the system. These first three fields thus denote abstractly the input values that are expected from the standard input on the original problem statement. The fourth field `light_on` aims at representing the state (on or off) of the lights at each step of the execution of the hackers' program: (`light_on` j l) is true if light l is on at turn j, for each j, l such that $0 \leq j \leq k$ and $1 \leq l \leq N$. The turn 0 corresponds to the initial state where all lights are on.

An object of type `lighting_configuration` does not necessarily represent a valid configuration, so we specify valid configurations via the following three predicates:

```
predicate valid_lighting_at_initial_turn (n:int) (on:int → int → bool)
(** specifies that all lights are on initially, that is at turn 0 *)

predicate valid_lighting_sequence (n j:int) (li:int → int) (on:int → int → bool)
(** for each i between 1 and j, specifies which lights are on at turn ‘i‘ in
    function of the lights on at turn ‘i-1‘ *)

predicate valid_lighting_configuration (c : lighting_configuration)
(** specifies that ‘c‘ is a fully valid configuration of lighting: the three
    first fields respect the domain of values given in the problem, and the
    lighting sequence it contains is valid *)
```

1. *In file* `lights.mlw`*, complete the definitions of the three predicates above. You may use the modulus function imported from the library* `int.EuclideanDivision`*:* (`mod` x y) *denotes* x *modulo* y*.*
2. *The constant* `sample` *given in file* `lights.mlw` *contains the configuration for the example given in the problem. Check that the goal* `sample_is_valid` *is provable by some automatic prover.*

The next step is to specify the expected answer to the problem, that is, to specify that the result of the code that we are going to write is the maximum number of lights off along all the steps. We start by defining a logic function that denotes the number of lights off in an abstract sequence:

```
function number_of_lights_off (light:int→bool) (i j:int) : int
  (** number of ‘l‘, ‘i ≤ l < j‘ such that ‘light l‘ is ‘false‘ *)
```

Notice that the first index i is counted but not the last index j. A canevas of its definition is given in the file `lights.mlw` under the form of a ghost recursive function.

3. *Complete the definition of* `number_of_lights_off`

4. *The lemma* `no_lights_off` *aims at proving that the predicate* `number_of_lights_off` *returns 0 when it is given a function* `light` *that always returns true. That proof must be made by induction, so we propose to use the mechanism of lemma functions. Complete the definition of* `no_lights_off` *so as to be able to prove it with automatic provers.*

The final step for formally specifying the City of Lights problem is to define a predicate

```
predicate city_of_lights_specification (c: lighting_configuration) (answer:int)
```

which expresses that `answer` is the expected answer for the input values given in configuration `c`. We propose to define it in term of an auxiliary predicate

```
predicate city_of_lights_specification_aux (n j:int) (light_on:int→int→bool) (answer:int)
  (** ‘answer‘ is the expected answer for ‘n‘ lights and ‘j‘ uses of the hackers’ program giving
      then sequence of lighting described by ‘light_on‘ *)
```

5. *Complete the definition of* `city_of_light_specification_aux`

6. *Prove the goal* `sample_specification` *which expresses that the expected answer for the sample input is 6.*

3 A Naive Implementation

This section proposes to prove a naive implementation of the problem, which iteratively computes the successive configurations of lights in an array of Boolean values, computes the number of lights off in that array and updates the known maximum stored in a mutable variable.

The first step is to define an auxiliary function `compute_lights_off` which, given an array of Boolean values, computes the number of occurrences of `False` in it. Notice that since the lights are numbered from 1 in the problem, we do not consider the first cell of that array.

7. *Fill the missing parts of the specification of* `compute_lights_off` *to express its intended behavior.*

8. *Add an appropriate loop invariant so as to prove that function correct.*

The second step is the prove the function `run_lights` that solves the City of Lights problem, whose code calls the previous function. It has the profile

```
run_lights (n:int) (k:int) (light_index:int → int) :
                (max_lights_off:int, ghost c:lighting_configuration)
```

where `n` is the number of lights, `k` the number of use of hackers’ program, and `light_index` provides the successives indexes of light. It returns a pair of the regular result `max_lights_off` and a ghost result `c` reporting the lighting configuration that resulted from the computation. The ghost result `c` is used to give an appropriate specification to `run_lights`, as already given in the canvas file. We suggest first to concentrate on the proof of the post-condition expressing that the returned ghost lighting configuration is valid.

9. *Add the necessary loop invariants to prove the post-condition on validity of the lighting configuration* `c`*. In order to clearly separate that part of the proof with the next question,* ***it is highly recommended that you write your report at the same time your prove your code****. Do not hesitate to report also the issues you meet, why it is difficult, and how you decided to solve it.*

Finally let's concentrate on the post-condition expressing that the regular result is correct.

10. *Add the necessary loop invariants for proving the final post-condition expressing that* `max_lights_off` *is the appropriate answer. Hint: you may need to insert an extra assertion before the loop, to make explicit that the lighting configuration is valid on turn 0.*

To play with the code, we now propose to extract that code to OCaml, and run a few tests. The command for extracting the Why3 code to OCaml is as follows:

```
why3 extract -D ocaml64 -o lights.ml lights.mlw
```

There is a file `test_lights.ml` that can be used to test your code. Indeed the extraction and the OCaml compilation can be run using simply

```
make tests
```

This will extract your code, compile the OCaml code to an executable and then run it. The code is run first on the example given above, and then on a large sample with pseudo-random values.

11. *Report the results obtain with your extracted code. Report the time taken on your computer to compute the results, and comment.*

4 A More Efficient Implementation

We propose now a better implementation, given by the pseudo-code below. The idea is that instead of computing the number of lights off at each iteration of the outer loop, we maintain this number in an extra mutable variable. Moreover, for the inner loop, instead of iterating on each cell of the array, we update only the light indexes that are a multiple of the index chosen by the hackers' program.

```
let max be 0
let current be 0
let a be an array of size n+1 with all cells initialized to true
for j from 1 to k do
  let step be the j-th light index
  let l be step
  while l is less or equal n do
    if a[l] is true then
      set a[l] to false
      increment current
    else
      set a[l] to true
      decrement current
    end if
    set l to l + step
  done;
  if max < current then set max to current
done
return max
```

12. *Implement this pseudo-code in Why3, by filling the canvas given for function* `run_lights_fast`. *Test your implementation using* `make tests` *as described in previous section. Report the results and times of computation on your computer, and comment.*

13. *Prove that your code is terminating and free of run-time errors. Explain any annotation that you needed to add in your code.*

The next step is to prove the post-condition expressing that the returned ghost lighting configuration is valid. For that purpose, you need to add ghost code in your program, similarly as in the code for the naive implementation.

14. *Prove the post-condition expressing that the returned ghost lighting configuration is valid. Explain the ghost code and the extra annotations you needed to add. Hint: you may need to help provers by stating (and proving it of course) some general lemmas about modulo, such as*

$$\forall x, y, z.\ x \bmod y = 0 \land x < z < x + y \rightarrow z \bmod y \neq 0$$

The last step is to prove the post-condition expressing that the computed result is correct.

15. *Prove the last post-condition expressing that the computed result is correct. Explain the extra annotations you needed to add. Hint: you may need to state (and prove) an additional lemma about the* framing *of the* `number_of_lights_off`*, that is to express that the result of (*`number_of_lights_off` *f i j) does not change when the function f is changed outside the range $[i, j[$. You may also invent similar lemmas when f is changed inside the range $[i, j[$.*

16. *As a conclusion of your report, comment on the difficulties of proving the two different programs solving the City of Lights problem.*