Homework #4

Deadline: Tuesday, 7 October 2014, 12:00 noon

Note: For this homework, and in general, please make sure that you submit well-organised and well-documented programs. For each predicate that you implement, you should say *what* it does and *how* you have achieved this. In some cases, it may be appropriate to show sample runs of certain queries as well. For some of the exercises on sorting and searching you can and should use code provided by me. You may leave such code uncommented, provided you have not made any changes to it and provided you clearly identify where in your file you are using such code. (In fact, one of the points I want to get across is that these are *general* problem-solving methods, which means that you do not need to change them at all to apply them to your specific problem.)

Question 1 (6 points)

The objective of this exercise is to implement a number of tools to experiment with different sorting algorithms and to empirically assess their complexity. Use the implementation of the sorting algorithms presented in class exactly as given on the slides.

(a) Implement a counter. You should provide the following predicates: `init_counter/0` to initialise the counter (that is, to set it to 0), `step_counter/0` to increment the counter by 1, and `get_counter/1` to return the current value of the counter. Examples:

```
?- init_counter, get_counter(Value).
Value = 0
Yes

?- step_counter, step_counter, step_counter, get_counter(Value).
Value = 3
Yes
```

Hint: You will need to use dynamic predicates.

(b) The number of comparison operations performed by a sorting algorithm (that is, the number of times it has to call the predicate `check/3`) is a good proxy for the time complexity of that algorithm. Write a predicate `experiment/3` that takes as input the name of a sorting algorithm and a list of integers, and that returns the number of primitive comparison operations required to run the given sorting algorithm on the given list. Examples:

```
?- experiment(quicksort, [5,8,1,4,3], Count).
Count = 7
Yes
```

```
?- experiment(bubblesort2, [10,9,8,7,6,5,4,3,2,1], Count).
Count = 90
Yes
```

Do not touch the implementation of the sorting algorithms themselves; all that is required is a minor change in the implementation of `check/3`.

(c) Write a predicate `random_list/3` that takes as input two positive integers `L` and `M` and returns a list of length `L` of random numbers between 1 and `M`. Example:

```
?- random_list(7, 49, List).
List = [3, 47, 5, 46, 33, 45, 36]
Yes
```

(d) Implement the following two predicates:

- `random_experiment(+Algorithm, +Length, +MaxElem, -Count).`
  This predicate should return the number of primitive comparison operations required to run the given sorting `Algorithm` on a randomly generated list of integers of the given `Length` (with random elements between 1 and `MaxElem`). Examples:

  ```
  ?- random_experiment(quicksort, 10, 50, Count).
  Count = 23
  Yes

  ?- random_experiment(quicksort, 10, 50, Count).
  Count = 27
  Yes
  ```

- `random_experiments(+Algorithm, +Length, +MaxElem, +N, -AvgCount).`
  This predicate should run `N` random experiments and return the average number of comparison operations required, rounded to the nearest integer. Example:

  ```
  ?- random_experiments(quicksort, 10, 50, 100, AvgCount).
  AvgCount = 24
  Yes
  ```

How many comparison operations are needed, on average, to sort a list of 100 random numbers between 1 and 500 using (improved) bubblesort and quicksort, respectively? How many milliseconds does it take on average? Show the queries (and Prolog's response) that you used to answer these questions.

(e) Write a predicate `chart/4` to visualise the number of primitive comparison operations required for a series of randomised experiments with list lengths running from 1 up to a specified maximum. The input parameters should be (1) the name of the algorithm, (2) the maximum length of the list, (3) the maximum random integer, and (4) the number of experiments to be run for every given list length. The details of the visualisation are up to you. Examples:

```
?- chart(bubblesort, 8, 50, 100).
1 >
2 > **
3 > ****
4 > *********
5 > ****************
6 > **************************
7 > ****************************************
8 > *********************************************************
Yes

?- chart(quicksort, 15, 50, 100).
 1 >
 2 > *
 3 > ***
 4 > *****
 5 > *******
 6 > ***********
 7 > *************
 8 > *****************
 9 > *********************
10 > *************************
11 > ****************************
12 > ********************************
13 > ************************************
14 > ******************************************
15 > **********************************************
Yes
```

Show and briefly comment on at least two examples in the file that you submit.

Question 2 (4 points)

Consider the following puzzle. A farmer wants to get a fox, a goat, and a cabbage across a river, from the left bank to the right bank. His boat can only take (at most) one of these items at a time (together with the farmer himself). He cannot leave either the fox and the goat, or the goat and the cabbage together on one side of the river without supervision. Is it possible to get everything safely from the lefthand side of the river to the righthand side?

(a) Formulate the problem in Prolog using the state-space representation: Specify how you propose to represent states and give an example. Implement the `move/2` predicate to model moves from one state to the next and implement a suitable `goal/1` predicate.

(b) Give a Prolog query that will solve the puzzle. To do so, use one of the search algorithms introduced in class so far. Briefly comment on your choice of algorithm.

(c) Provide a predicate `show_plan/1` that turns the output of your query into a readable form, so a user can easily understand the solution. Show this output and the query producing it. How many times does the farmer have to cross the river?