Computer Science 433
Programming Languages
The College of Saint Rose
Fall 2014

Program/Problem Set 6: Backtracking

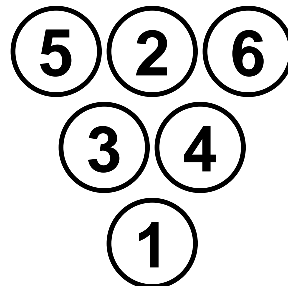
Due: 11:59 PM, Wednesday, October 15, 2014

In this second part of your Scheme programming tasks, you will implement a backtracking algorithm to find solutions to the billiard ball problem.

You may work alone or in a group of 2 or 3 on this assignment.

The Problem

The *billiard ball problem* consists of arranging a triangle of numbered billiard balls such that the resulting layout has the following arithmetic property: every ball below the top level is the absolute value of the difference between two balls immediately above it. For instance, the following is a solution to the problem when the top row consists of three balls (which requires a total of six balls).



Note that the balls are numbered from 1 to 6 in this case.

For the problems with four and five balls in the top row, there are a total of 10 and 15 balls, respectively.

Read more about the problem in Martin Gardner's *Penrose Tiles to Trapdoor Ciphers: And the Return of Dr Matrix* (Cambridge Press 1997) on pages 119-120. (`http://books.google.com/books?id=8-FlYl6-ML8C&lpg=PP1&pg=PA119#v=onepage&q&f=false`)

As far as I can tell, there are no known solutions for problems larger than five balls in the top row.

A Backtracking Approach

A backtracking algorithm, similar to the one we saw that solved the N-Queens problem in class, is one approach to solving this problem.

The idea is that we attempt to build up a candidate solution by adding balls to the top row. Each time a ball is added, we make sure we have not broken any of the rules (in this case, there is just

one: there are no repeated numbers in the triangle generated by that top row). If we have not yet broken a rule, we either have found a solution (if the top row now contains the desired number of balls) or we have a partial candidate solution and we should add another ball. Any time we generate a candidate solution that does violate the rule, we have hit a dead end, so we undo the most recent addition and try the next option. If we ever backtrack all the way to the beginning and have run out of options for our first move, we know no solution exists.

For example, consider a backtracking solution to the problem where there are 2 balls in the top row, and we choose numbers from the largest to the smallest each time we reach a decision point. (Note that this is a good strategy to get a solution more quickly, as larger numbers will tend to be in the top row.)

We start with an empty solution, and we are ready to add the first ball to the top row. Since we are trying numbers from the largest to smallest, we start with 3:

```
(3)
```

This is a legal configuration: there are no repeated digits when we expand this out (in fact, there is no expansion needed for a single ball). So we accept this as a partial solution and move on, trying to add a ball to the second position. The first ball we attempt to place at this position is the highest numbered, 3:

```
(3 3)
```

which expands to

```
(3 3)
 (0)
```

This is not a legal configuration: it includes two 3's. So we backtrack and erase our last move, and instead try the next option, which is to use the 2 ball:

```
(3 2)
```

which expands to

```
(3 2)
 (1)
```

This is a legal solution: no repeats. Plus, we now have filled the top row, so our solution is complete.

? Question 1:
Show the steps in a backtracking solution to the problem with two balls in the top row if we instead chose balls for each position in increasing instead of decreasing numerical order. (4 points)

Now, let's consider the start of the procedure for the much more interesting (and much longer) backtracking computation of a solution to the problem with three balls in the top row. Note that here, we have a total of six balls.

Our first move is to place the largest number into the top row.

```
(6)
```

This is again legal, so we continue by adding a second number.

```
(6 6)
```

This contains a duplicate, so we backtrack and try a 5 in the second position.

```
(6 5)
 (1)
```

This is legal, so we accept the 5 for now, and start working on the third ball. We begin, as before, with the highest numbered ball and work our way down if we encounter illegal moves.

```
(6 5 6)
 (1 1)
  (0)
```

This has duplicates, so it is not legal. In fact, all of our choices for the third ball will result in illegal configurations here:

```
(6 5 6)
 (1 1)
  (0)

(6 5 5)
 (1 0)
  (1)

(6 5 4)
 (1 1)
  (0)
```

```
(6 5 3)
 (1 2)
  (1)

(6 5 2)
 (1 3)
  (2)

(6 5 1)
 (1 4)
  (3)
```

So this means 5 in the second position of the top row was a dead end. And we backtrack, and try a 4 there instead:

```
(6 4)
 (2)
```

So far so good here, so we move on trying each ball in the 3rd position of the top row...

Question 2:
Complete the hand trace to the solution that will result from the above procedure. Note that this will not lead to the sample solution pictured at the top of the page. (6 points)

Implementing in Scheme

Your program should use a similar approach to our N-Queens backtracking example from class. Your "startup" function should take just one parameter: the number of balls to be placed in the top row. It will then call another function with additional parameters needed to manage the backtracking and to compute and return a solution (if one exists for the given input).

If a solution exists, the function should return the list of numbers of the balls in the top row. It can return `#f` otherwise. This list, as it is being built up is a good choice to use as the `state` of the algorithm.

After each time a ball is added to the state, the following possibilities could occur:

- The new state is illegal: the top row and the rows beneath that would be required for that top row to exist contains a duplicate ball. If so, backtrack out of this move.
- The new state is legal and the desired number of balls are now in the top row. In this case, we have a solution and are very happy.
- The new state is legal but more balls are needed in the top row. In this case, trying to add another ball, starting with the highest-numbered ball for the problem size.

You will very likely want a number of additional functions that help your main recursive function to do its work. For example, you might want a function that checks that all elements of a given list are unique (no duplicates).

Tips and Tricks

- When your program doesn't work, it can be difficuly to track down just what's wrong. Always suspect your helper functions and test them thoroughly on their own. For example, if you have function to check if there are no repeated elements in a list, make sure that's working before you rely on it as part of the solution for the larger problem.
- Additional items may be added here as questions come in as everyone works on this assignment.

Trying it Out

> **? Question 3:**
> Run your program for 1, 2, 3, 4, 5, and 6 balls in the top row, and give both the answer you obtain and the amount of time it took to compute. (3 points)

Note that the function `runtime` reports the CPU time used by Scheme since it started. So if you start up Scheme, run your program for a given problem size, then call `runtime`, you will get a reasonable approximation of the time it takes.

Submission

Before 11:59 PM, Wednesday, October 15, 2014, submit your work for grading. Create and submit a single archive file (a `.7z` or `.zip` file containing all required files) using Submission Box at `http://sb.teresco.org` under assignment "PS6".

Grading

This assignment will be graded out of 50 points.

Feature	Value	Score
Hand trace for 2 balls in top row	4	
Hand trace completion for 3 balls in top row	6	
Program correctness	30	
Program documentation	5	
Program style and formatting	2	
Computed solutions and run times	3	
Total	50	