Problem Set 3 Solutions

Jeffrey M. Radcliffe

November 16, 2000

Exercise 2

a.

Sloppy Joe's solution does not work for a number of reasons. The problem lies in the second recursive call, in which the bottom half of discs is moved. The program utilizes the *Using1* peg, which is sadly containing discs already, all of which are smaller than any from the bottom half.

b.

While Fruity Freddie correctly identified the error in Sloppy Joe's algorithm, this correction does not completely address the problem. *Using1* is still occupied, and when the time comes to do recursion, there will be a violation of the rules.

c.

The Scheme code for Sloppy Joe's original algorithm is as follows:

```
(define (faulty-towers n from to using1 using2)
  (if (= n 1)
      (begin (display (list 'move 'disc 'from from 'to to)) (newline))
      (let ((m  (quotient n 2)))
          (faulty-towers (- n m) from using1 to using2)
          (faulty-towers m from to using2 using1) ;; **
          (faulty-towers (- n m) using1 to from using2))))
```

Fruity Freddie's changes change the commented line to

```
(faulty-towers m from to using1 using2)
```

When evaluated at n = 4, the algorithm solved the puzzle correctly. At n = 8, an error was made:

```
(faulty-towers 8 1 2 3 4)

(move disc from 1 to 3)
(move disc from 1 to 2)
(move disc from 3 to 2)
(move disc from 1 to 4)
(move disc from 1 to 3)
(move disc from 4 to 3)
(move disc from 2 to 1)
(move disc from 2 to 3)
(move disc from 1 to 3) ;; 4 disks have been moved to peg 3
(move disc from 1 to 2)
(move disc from 1 to 4)
(move disc from 2 to 4)
(move disc from 1 to 3) ;; oops!!
```

d.

The recurrence equation for Sloppy Joe's algorithm is $T_n = 3T_{\frac{n}{2}}$.

e.

Since we know that $T_1 = 1$, we can solve the equation solves as follows:

$$T_n = 3T_{\frac{n}{2}} = 3^2 T_{\frac{n}{2^2}} = 3^3 T_{\frac{n}{2^3}} \ldots = 3^{\log_2 n} T_{\frac{n}{n^{\log_2 n}}}$$

$$= 3^{\log_2 n} T_1 = 3^{\log_2 n}(1) = 3^{\log_2 n} = n^{\log_2 3}$$

Exercise 3

a.

The problem with Sloppy and Fruity's system lies in using only 3 pegs for the second recursive call. This is easily done by calling the 3-peg version of Towers of Hanoi. While this not might be the fastest way, it is significantly faster.

b.

Here is a working algorithm for four pegs in Scheme:

```
(define (tower-of-power n from to using1 using2)
  (if (= n 1)
      (begin (display (list 'move 'disc 'from from 'to to)) (newline))
      (let ((m (quotient n 2)))
        (tower-of-power (- n m) from using1 to using2)
```

```
        (towers m from to using2) ;; call to 3-peg towers
        (tower-of-power (- n m) using1 to from using2))))
```

The results for $n = 4$:

```
(tower-of-power 4 1 2 3 4)

(move disc from 1 to 2)
(move disc from 1 to 3)
(move disc from 2 to 3) ;; top half moved
(move disc from 1 to 4)
(move disc from 1 to 2)
(move disc from 4 to 2) ;; bottom half in place
(move disc from 3 to 1)
(move disc from 3 to 2)
(move disc from 1 to 2) ;; all done
;Value: #[unspecified-return-value]
```

c.

This algorithm is represented by the recurrence equation $T_n = 2T_{\frac{n}{2}} + (2^{\frac{n}{2}} - 1)$, which can be reduced to

$$T_n = 2^{\log_2 n} T_1 + [2^0 2^{\frac{n}{2}} + 2^1 2^{\frac{n}{4}} + \ldots + 2^{\log_2 n - 2} 2^2]$$

Exercise 4

Diagram not available

Exercise 6

a.

Let the base case be $n = 1$, where one disc moves through all of 3 configurations in the game, from *from* to *to* via *using*. Assuming that our theory holds true for n discs, we look at the $n + 1$ case. First we move the top disc to the *using* peg, then to *to*. With the $n + 1$st disc in place, we move n to *using*. Next, we move the $n + 1$ disc back to *from* (via *using*, which is legal), and the n disc to *to*. We then repeat the base case with the $n + 1$ disc, moving to *to* via *using*. This is clearly expressed in a table:

move	*from*	*to*	*using*
	n , (n + 1)		
1	n		(n + 1)
2	n	(n + 1)	
3		(n + 1)	n
4			n , (n + 1)
5	(n + 1)		n
6	(n + 1)	n	
7		n	(n + 1)
8		n , (n + 1)	

All possible combinations of legal moves are represented.

b.

Our modified Towers of Hanoi rule-set is expressed in the recurrence equation $T_n = 3T_{n-1}+2$.

c.

Taking $T_1 = 2$ as the base case:

$$T_n = 3(3T_{n-1} + 2) + 2 = 3^2(3T_{n-2}) + 2 + 2(3) = 3^r T_{n-r} + 2(1 + 3 + 3^2 + 3^3 + \ldots + 3^{r-1})$$

Substituting (n - 1) for r:

$$T_n = 3^{n-1}(2) + 2(1 + 3 + 3^2 + 3^3 + \ldots + 3^{n-2}) = 2(1 + 3 + 3^2 + 3^3 + \ldots + 3^{n-1})$$

$$= 3^n - 1.$$

Exercise 7

a.

Double-disk Hanoi will take exactly twice as many moves as tradition Hanoi. Since reversal of identically sized disks is allowed, each move of traditional Hanoi will involve moving the topmost disc to the new peg, followed immediately by the disk of identical size below it.

b.

The number of moves needed to solve double-disk Hanoi is $T_n = 2T_{(n-1)} + 2$. Given that $T_0 = 0$:

$$T_n = 2^r T_{n-r} + (2 + 2^2 + 2^3 + \ldots + 2^r) = 2^n(0) + (2 + 2^2 + 2^3 + \ldots + 2^n) = 2^{n+1} - 2.$$

Exercise 8

a.

Here is the standard Gray code sequence using 4 binary digits:

0	0	0	0
0	0	0	1
0	0	1	1
0	0	1	0
0	1	1	0
0	1	1	1
0	1	0	1
0	1	0	0
1	1	0	0
1	1	0	1
1	1	1	1
1	1	1	0
1	0	1	0
1	0	1	1
1	0	0	1
1	0	0	0

b.

diagram not available

Exercise 9

Our base case will be 2 bit Gray code, which forms a Hamiltonian circuit. Assume the (n - 1) case is true. Adding another bit will double the number of nodes in the circuit, the original with the new bit set to 0, the copy with the new bit set to 1. It follows that every node on the duplicated (n - 1) circuit will have a corresponding node on the original circuit differing by only one bit (the new bit). By choosing 2 related pairs of nodes and connecting these together, a new Hamiltonian circuit is formed.

Exercise 10

column	1	2	3
	0	0	0
	0	0	1
	0	1	1
	0	1	0
	1	1	0
	1	1	1
	1	0	1
	1	0	0

a.

A simple theorem is that any column i of a Gray code will have the same number of zeroes and ones.

b.

If we take $n = 1$ (a 1 bit circuit) as a base case, it is clear that there are the same number of zeroes and ones, one of each. The method of adding a new column entails adding zeros to the left side of the previous columns, then taking the reverse of the previous columns and adding ones. Since the previous sets of columns were even size, then the new column has the same number of ones and zeroes. One could further postulate that for any Gray code with n digits, there are an even number of ones and zeroes.

Exercise 11

To simplify computation, let us represent our numbers as lists of 1's and 0's. So, the binary number 1101 can be created with `'(1 1 0 1)`. We will also need a version of XOR which works on 1's and 0's which we write as follows:

```
(define (b-xor a b)
  (if (= a b) 0 1))
```

a.

With that, we can write a procedure which converts binary numbers into Grey codes:

```
(define (bin->grey n)
  (define (helper n l)
    (if (null? n)
    nil
    (cons (b-xor (car n) l)
          (helper (cdr n) (car n)))))
```

```
  (helper n 0))

(bin->grey '(1 1 0 1 1))
;Value: (1 0 1 1 0)
```

b.

And one to convert back:

```
(define (grey->bin n)
  (define (helper n l)
    (if (null? n)
    nil
    (cons (b-xor (car n) l)
          (helper (cdr n) (b-xor (car n) l)))))
  (helper n 0))

(grey->bin '(1 0 1 1 0))
;Value: (1 1 0 1 1)
```

Exercise 12

The function a^n can be written as follows:

$$a^n = \begin{cases} 1 & \text{if } n = 0 \\ (a^{\frac{n}{2}})^2 & \text{if } n \text{ is even} \\ a(a^{n-1}) & \text{if } n \text{ is odd} \end{cases}$$

a.

The function uses $\log n$ multiplications when n is a power of two because each one cuts the problem in half.

b.

When n is one less than a power of two, the number of multiplications is $2\lfloor \log n \rfloor$ because the algorithm alternates between the odd and even cases.

c.

The following table shows the number of multiplications for the first 16 n.

n	n (binary)	$\lfloor \log n \rfloor$	multiplications
1	1	1	0
2	10	1	1
3	11	1	2
4	100	2	2
5	101	2	3
6	110	2	3
7	111	2	4
8	1000	3	3
9	1001	3	4
10	1010	3	4
11	1011	3	5
12	1100	3	4
13	1101	3	5
14	1110	3	5
15	1111	3	6
16	10000	4	4

The table shows that the number of multiplications is equivalent to $\lfloor \log n \rfloor$ plus the number of ones in the binary representation of n minus 1.