Dayou Du
N13825075
dd2645@nyu.edu

Homework1-Self Assessment
Fundamental Algorithms

2017-09-14

1 Ex. 1.1

The step-by-step configurations of $ToH_{standard}(4, A, B, C)$ is drawing below.

Step Num.	A	B	C	Step Num.	A	B	C	Step Num.	A	B	C
0	1 2 3 4			1	2 3 4		1	2	3 4	2	1
3	3 4	1 2		4	4	1 2	3	5	1 4	2	3
6	1 4		2 3	7	4		1 2 3	8		4	1 2 3
9		1 4	2 3	10	2	1 4	3	11	1 2	4	3
12	1 2	3 4		13	2	3 4	1	14		2 3 4	1
15		1 2 3 4									

2 Ex. 1.2

2.1 Question a

As a beginner, we can find a simple, non-recursive solution, which listed below:

Algorithm 1 Simple DTH using a for-loop

```
1:  procedure $DTH_{simple}(n, A, B, C)$
2:      for $i \in [1, n)$ do
          //here we omit the process to find the position of ring $i$ and ring $i+1$ using function $IsItThere()$
3:          if ring $i$ and ring $i+1$ are not on the same pole then
4:              move rings $[1, i]$ on ring $i+1$ using $ToH_{standard}(i, x, y, z)$
5:          end if
6:      end for
7:      if ring $n$ is not on pole $B$ then
8:          move rings $[1, n]$ to pole $B$ using $ToH_{standard}(n, x, B, z)$
9:      end if
10: end procedure
```

The idea is simple : each loop we put the smallest i rings on ring$i + 1$(if they are not on it already), then we "bind"them together and continue the next loop cycle.

Let $Moves_{ToHstandard}(n)$ be the number of ring moves for standard ToH problem, which is equal to $2^n - 1$.

The total moves in the worst case for $DTH_{simple}(n, ...)$ is $Moves_{ToHstandard}(1) + Moves_{ToHstandard}(2) + Moves_{ToHstandard}(3) + ... + Moves_{ToHstandard}(N)$, which is $2^{n+1} - 2 - n$. It will do more moves than the recursive version since sometimes we can move ring $i + 1$ to another pole before we move the smaller i rings on it, which will avoid some useless moves in further loops. Although we can use more ifs to optimize this solution, it will make the algorithm not elegant at all if we do so.
To be honest I cannot guess what other one would think about... Maybe in this question I moved a step further than the beginners.

2.2 Question b and c

Algorithm 2 recursive DTH solution

```
procedure $DTH_{recursive}(n, A, B, C)$
    if $n$ equals 1 then
        if $IsItThere(n, B)$ then
            return
        else
            if $IsItThere(n, A)$ then
                move ring 1 from $A$ to $B$
            else
                move ring 1 from $C$ to $B$
            end if
        end if
    else
        if $IsItThere(n, B)$ then
            $DTH_{recursive}(n - 1, A, B, C)$
        else
            if $IsItThere(n, A)$ then
                $DTH_{recursive}(n - 1, A, C, B)$
                move ring $n$ from $A$ to $B$
                $DTH_{recursive}(n - 1, C, B, A)$
            else
                $DTH_{recursive}(n - 1, B, A, C)$
                move ring $n$ from $C$ to $B$
                $DTH_{recursive}(n - 1, A, B, C)$
            end if
        end if
    end if
end procedure
```

2.3 Question d

Algorithm 3 recursive DTH solution

```
procedure $DTH_{recursive}(n, A, B, C)$
    if $n$ equals 1 then
        if $IsItThere(n, B)$ then
            return
        else
            if $IsItThere(n, A)$ then
                move ring 1 from $A$ to $B$
            else
                move ring 1 from $C$ to $B$
            end if
        end if
    else
        if $IsItThere(n, B)$ then
            $DTH_{recursive}(n-1, A, B, C)$
        else
            if $IsItThere(n, A)$ then
                $DTH_{recursive}(n-1, A, C, B)$
                move ring $n$ from $A$ to $B$
                $ToH_{standard}(n-1, C, B, A)$
            else
                $DTH_{recursive}(n-1, B, A, C)$
                move ring $n$ from $C$ to $B$
                $ToH_{standard}(n-1, A, B, C)$
            end if
        end if
    end if
end procedure
```

2.3.1 Calculation of the total moves for the worst case

In this section we calculate the total moves for the worst case.
Let $Moves_{DTHrecursive}(i)$ be the total moves for $DTH_{recursive}(n, ...)$ **in the worst case**.Then,

$$Moves_{DTHrecursive}(n) = Moves_{ToHstandard}(n-1) + 1 + Moves_{DTHrecursive}(n-1), n > 1 \tag{1}$$

Which means

$$Moves_{DTHrecursive}(n) = 2^{n-1} + Moves_{DTHrecursive}(n-1), n > 1 \tag{2}$$

Since we have

$$Moves_{DTHrecursive}(1) = 1 \tag{3}$$

Then

$$Moves_{DTHrecursive}(n) = 2^{n-1} + 2^{n-2} + ... + 2^1 + 1 = 2^n - 1, n > 1 \tag{4}$$

The formula above stands true for $n = 1$, thus we have

$$Moves_{DTHrecursive}(n) = 2^n - 1 \tag{5}$$

2.3.2 Proof of the efficiency

In this section we prove that this solution achieved best efficiency(i.e. use least moves)
lemma 1. *For standard ToH rules set, the only way to move ring n from pole X to pole Y, is to put all [1..(n-1)] ring sets on the other pole Z*

Then we can prove our solution's efficiency using mathematical induction with the help of Lemma 1.

lemma 2. *$DTH_{recursive}(...)$ will use least moves to gather all rings on the pole required.*

For an easier present,we assume that our goal is to move all rings on pole B, since the poles are symmetrical the proof below also stands true for A and B. Proof of Lemma 2

1. For the situation of $n = 1$, we can easily find that $DTH_{recursive}(1, ..)$ is the most efficient.
2. Given that for situation of $n = k - 1$, $DTH_{recursive}(k - 1)$ is the most efficient way to finish the give task.
3. Then for situation of $n = k$:
   - If ring n is already on pole B, then we should move the other $[1..k - 1]$ rings on the top of it. That is to call $DTH_{recursive}(k - 1, ..)$, which as we assume above.
   - If ring k is not on pole B. Then, according to lemma 1, we can divide our procedure in to 3 steps.(For an easier present we assume ring k is on pole A.)
     (a) Put rings $[1..k - 1]$ to pole C. According to 2, the most efficient way is to call $DTH_{recursive}(k - 1, ...)$
     (b) move ring k from A to B
     (c) move rings $[1..k - 1]$ from C to B. According to our text book, $ToH_{standard}(k - 1, ..)$ is the most efficient way to do it.

Then, according to mathematical induction, we proved Lemma 2.
I'm not sure whether the calculations above are correct or not. They are not given in the answer.

3 Ex 1.3

3.1 Question a

To be honest I don't know what other one would think about... Maybe they put the smallest ring on pole D and then do a standard ToH on the other three poles?

3.2 Question b

Put a "large"ring on pole D to reduce moves when we design our recursion function, and move it out from D before we call the recursive function to avoid violating the rules.
I got the right though when I was working on the problem: there is one more pole to put one more "large"ring, thus we can use a (n-2) recursion.

3.3 Question c

Algorithm 4 ToH with pole D

```
procedure $ToH_{ex1.3}(n, A, B, C, D)$
    if $n$ equals 1 then
        move ring 1 from $A$ to $B$
    else
        if $n$ equals 2 then
            move ring 1 from $A$ to $C$
            move ring 2 from $A$ to $B$
            move ring 1 from $C$ to $B$
        else
            $ToH_{ex1.3}(n-2, A, C, B, D)$
            move ring $n-1$ from $A$ to $D$
            move ring $n$ from $A$ to $B$
            move ring $n-1$ from $D$ to $B$
            $ToH_{ex1.3}(n-2, C, B, A, D)$
        end if
    end if
end procedure
```

3.4 Calculation of the total moves

Let a(n) be the total moves for $ToH_{ex1.3}(n, ...)$. Then we have

$$a(n) = 2a(n-2) + 3 \tag{6}$$

It is a linear recursion equation with constant coefficients. We have characteristic equation as

$$x^2 = 2 \tag{7}$$

Thus we have $x_1 = \sqrt{2}$, $x_2 = -\sqrt{2}$
Then we have general solution as

$$a(n) = A(\sqrt{2}^n) + B(-\sqrt{2})^n, n > 2 \tag{8}$$

where A and B are constants.
We can also find a particular solution as

$$a(n) = -3 \tag{9}$$

Then the solution is

$$a(n) = A(\sqrt{2}^n) + B(-\sqrt{2})^n - 3, n > 2 \tag{10}$$

Use the initial condition

$$a(1) = 1, a(2) = 3 \tag{11}$$

We have

$$\begin{array}{l} a(n) = (\frac{5}{4} + \sqrt{2})(\sqrt{2}^n) + (\frac{5}{4} - \sqrt{2})(-\sqrt{2})^n - 3, n > 2 \\ a(1) = 1 \\ a(2) = 3 \end{array} \tag{12}$$

1. Actually from formula 8 we already know the 'about' complexity. There's no need to figure out the exact number.

2. I didn't figure out how to prove that we already achieved the best efficiency.

4 Ex 1.4

4.1 Question a

Answer: The top sides of ring $1, 2...n-1$ are red, the top side of ring n is white.
Explain: During the procedure of $ToH_{standard}(n, ...)$,it calls $ToH_{standard}(n-1, ...)$ twice, thus all the rings from 1 to n-1 that flipped up in the first call are flipped back in the second call. The ring n only flipped once, thus the top side of it is white.

4.2 Question b

I found two algorithms that can do it, however they actually consumes same number of moves... **Note that solution 2 is a tail recursion algorithm, thus actually we can convert it in to a for-loop to reduce memory cost.**

4.2.1 Solution 1

Algorithm 5 ToH 1.4b, a simple solution

```
1: procedure $RTH_1(n, A, B, C)$
2:     $ToH_{standard}(n, A, C, B)$
3:     $ToH_{standard}(n, C, B, A)$
4: end procedure
```

This algorithm moves the full set of n rings twice, thus all of them are flipped back.
Calculation of the total moves:

$$Move_{RTH1}(n) = Move_{ToHstandard}(n) + Move_{ToHstandard}(n) = 2^n - 1 + 2^n - 1 = 2^{n+1} - 2 \tag{13}$$

I didn't figure out how to prove that we have achieved the best efficiency.

4.2.2 Solution 2

Algorithm 6 ToH 1.4b, a tail recursion solution

```
1: procedure $RTH_2(n, A, B, C)$
2:     if $n$ equals to 1 then
3:         move ring 1 from $A$ to $C$
4:         move ring 1 from $C$ to $B$
5:     else
6:         $ToH_{standard}(n-1, A, B, C)$
7:         move ring $n$ from $A$ to $C$
8:         $ToH_{standard}(n-1, B, A, C)$
9:         move ring $n$ from $C$ to $B$
10:        $RTH2(n-1, A, B, C)$
11:    end if
12: end procedure
```

Line 6 and Line 8 moves full set of $n-1$ rings twice. Line 10 call the recursive function to ensure nothing is flipped up.
Calculation of the total moves:

$$Move_{RTH2}(n) = Move_{ToHstandard}(n-1) * 2 + 2 + Move_{RTH2}(n-1), n > 1 \tag{14}$$

$$Move_{RTH2}(n) = (2^{n-1} - 1) * 2 + 2 + Move_{RTH2}(n-1) = 2^n + Move_{RTH2}(n-1), n > 1 \tag{15}$$

And we have

$$Move_{RTH2}(1) = 2 \tag{16}$$

Then

$$Move_{RTH2}(n) = 2^n + 2^{n-1} + 2^{n-2} + ... + 2 = 2^{n+1} - 2, n > 1 \tag{17}$$

The formula above stands true for $n = 1$,thus

$$Move_{RTH2}(n) = 2^{n+1} - 2 \tag{18}$$

4.3 Question c

Similar with Algorithm 6, we can use a tail recursion algorithm listed below. **Still, we can convert it into a for-loop to reduce memory cost.**

Algorithm 7 ToH 1.4c, tail recursion solution

```
procedure $WTH(n, A, B, C)$
    if $n$ equals to 1 then
        move ring 1 from $A$ to $B$
    else
        $ToH_{standard}(n-1, A, C, B)$
        move ring $n$ from $A$ to $B$
        $ToH_{standard}(n-1, C, A, B)$
        $WTH(n-1, A, B, C)$
    end if
end procedure
```

Line5 and Line7 moves full set of $n-1$ rings twice, and Line8 flip them up. Line6 flips ring n up.
Calculation of the total moves:

$$Move_{WTH}(n) = Move_{ToHstandard}(n-1) * 2 + 1 + Move_{WTH}(n-1), n > 1 \tag{19}$$

$$Move_{WTH}(n) = (2^{n-1} - 1) * 2 + 1 + Move_{WTH}(n-1) = 2^n - 1 + Move_{WTH}(n-1), n > 1 \tag{20}$$

And we have

$$Move_{WTH}(1) = 1 \tag{21}$$

Then

$$Move_{WTH}(n) = (2^n - 1) + (2^{n-1} - 1) + (2^{n-2} - 1) + ... + (3 - 1) + 1 = 2^{n+1} - n - 2, n > 1 \tag{22}$$

The formula above stands true for $n = 1$,thus

$$Move_{WTH}(n) = 2^{n+1} - n - 2 \tag{23}$$

The codes are **correct**, however it doesn't achieve **best efficiency**. The fault I made is that I directly apply the idea of algorithm 6 (Exercise 1.4b solution2) to this problem, and didn't think too much about whether it can be further optimized. I knew enough to find that answer, but I was blind by the intuition I got in 1.4b and didn't check out the (n-2) recursion method.
Actually there's a small bug in the solution sheet: The algorithm didn't check $if(n == 2)$ situation as it use a (n-2) recursion. I think the correct code should be :

Algorithm 8 ToH 1.4c, revised n-2 recursion solution

```
procedure $WTH(n, A, B, C)$
    if $n$ equals to 1 then
        move ring 1 from $A$ to $B$
    else
        if $n$ equals to 2 then
            move ring 1 from $A$ to $C$
            move ring 2 from $A$ to $B$
            move ring 1 from $C$ to $A$
            move ring 1 from $A$ to $B$
        else
            $ToH_{standard}(n-1, A, C, B)$
            move ring $n$ from $A$ to $B$
            $ToH_{standard}(n-2, C, B, A)$
            move ring $n-1$ from $C$ to $A$
            $ToH_{standard}(n-2, B, C, A)$
            move ring $n-1$ from $A$ to $B$
            $WTH(n-2, C, B, A)$
        end if
    end if
end procedure
```

Then we check that algorithm 8 is indeed better than algorithm 7:

$$Move_{WTHrevised}(n) = Move_{ToHstandard}(n-1)+2*Move_{ToHstandard}(n-2)+3+Move_{WTHrevised}(n-2), n > 2 \tag{24}$$

$$Move_{WTHrevised}(n) = 2^n + Move_{WTHrevised}(n-2), n > 2 \tag{25}$$

Then we have:

$$Move_{WTHrevised}(n) \approx \frac{4}{3} * 2^n = \frac{2}{3} * 2^{n+1} < 2^{n+1} \approx Move_{WTH}(n) \tag{26}$$

The formula above stands true for n large enough. Actually we can prove for each n = 1,2,3... if we calculate the exact number,but it's just no need to do it.

5 Ex 1.5

Algorithm 9 ToH 1.5, recursion solution

```
procedure $CTH(n, A, B, C)$
    $ShiftOnePoleToTheRight(n, A, B, C)$
end procedure
                ▷ The procedure below shifts all the n rings one pole to the right, i.e. $A \to B$ or $B \to C$ or $C \to A$
procedure $ShiftOnePoleToTheRight$(n,X,Y,Z)    ▷ X is the source pole , Y is destination, Z is the other
    if $n$ equals to 1 then
        move ring 1 from $A$ to $B$
    else
        $ShiftTwoPolesToTheRight(n-1, A, C, B)$
        move ring $n$ from $A$ to $B$
        $ShiftTwoPolesToTheRight(n-1, C, B, A)$
    end if
end procedure
                ▷ The procedure below shift all the n rings two poles to the right, i.e. $A \to C$ or $B \to A$ or $C \to B$
procedure $ShiftTwoPolesToTheRight$(n,X,Y,Z)    ▷ X is the source pole , Y is destination, Z is the other
    if $n$ equals to 1 then
        move ring 1 from $A$ to $B$
        move ring 1 from $B$ to $C$
    else
        $ShiftTwoPolesToTheRight(n-1, A, C, B)$
        move ring $n$ from $A$ to $B$
        $ShiftOnePoleToTheRight(n-1, C, A, B)$
        move ring $n$ from $B$ to $C$
        $ShiftTwoPolesToTheRight(n-1, A, C, B)$
    end if
end procedure
```

5.1 Calculation of the total moves

Let A(n) be the total moves of $ShiftOnePoleToTheRight(n, ...)$
Let B(n) be the total moves of $ShiftTwoPolesToTheRight(n, ...)$
Then we have:

$$A(n) = B(n-1) + 1 + B(n-1) = 2B(n-1) + 1, n > 1 \tag{27}$$

and

$$B(n) = B(n-1) + 1 + A(n-1) + 1 + B(n-1) = 2B(n-1) + A(n-1) + 2, n > 1 \tag{28}$$

Use (27) in (28),we have

$$B(n) = 2B(n-1) + 2B(n-2) + 3, n > 2 \tag{29}$$

It is a linear recursion equation with constant coefficients. Similar with Section 3.3, we have solution as

$$\begin{aligned} &B(n) = (\tfrac{1}{2} + \tfrac{\sqrt{3}}{3})(1+\sqrt{3})^n + (\tfrac{1}{2} - \tfrac{\sqrt{3}}{3})(1-\sqrt{3})^n - 1, n > 2 \\ &B(1) = 2 \\ &B(2) = 7 \end{aligned} \tag{30}$$

Then we have

$$\begin{aligned} &A(n) = (1 + \tfrac{2\sqrt{3}}{3})(1+\sqrt{3})^{n-1} + (1 - \tfrac{2\sqrt{3}}{3})(1-\sqrt{3})^{n-1} - 1, n > 2 \\ &A(1) = 1 \\ &A(2) = 5 \end{aligned} \tag{31}$$

The total moves of $CTH(n, ..)$ is $A(n)$

6 Ex 1.6

6.1 Question a

Algorithm 10 Color problem part a

```
 1: procedure $Color_{ex1.6a}(n, Adj[1..n]; Color[1..n])$
 2:     Create linked list $UnColored$
 3:     Put vertics $1, 2...n$ into $UnColored$
 4:     Create array MostRecentNeiborColor[1..n] with all entries initialize to Nil;
 5:     $ColorId \leftarrow 0$
 6:     while $UnColored.empty() == False$ do
 7:         for each $vertexv \in UnColored$ do
 8:             if $MostRecentNeiborColor[v] \neq ColorId$ then
 9:                 delete $v$ from $UnColored$
10:                 $Color[v] \leftarrow ColorId$
11:                 for each $vertexj \in Adj[v]$ do
12:                     $MostRecentNeiborColor[j] \leftarrow ColorId$
13:                 end for
14:             end if
15:         end for
16:         $ColorId \leftarrow ColorId + 1$
17:     end while
18: end procedure
```

Note: In algorithm above the index of different colors start at 0. We can set it at line 5 if we would like to change it into 1.

6.2 Question b

Algorithm 11 Color problem part b

```
procedure $Color_{ex1.6b}(n, Adj[1..n]; Color[1..n])$
    create boolean array/bitmap NeiborColorUsed[1..n] with all entries initialized to $False$;
    for each vertex $i$ do
        $NumOfNeighbors \leftarrow Adj[i].size()$
        for each neighbor $j \in Adj[i]$ do
            $ColorJ \leftarrow Color[j]$
            if $ColorJ \neq Nil$ then
                if $ColorJ \leq NumOfNeighbors$ then ▷ Only record small-enough colors, good for clean-up
                    $NeiborColorUsed[ColorJ] \leftarrow True$
                end if
            end if
        end for
        $CurrColored \leftarrow False$          ▷ indicates whether current node has already been colored
        for $k \in [1, NumOfNeighbors + 1]$ do
            if $CurrColored == False$ and $NeiborColorUsed[k] == False$ then
                $ColorOf[i] \leftarrow k$
                $CurrColored \leftarrow True$
            else
                $NeiborColorUsed[k] \leftarrow False$          ▷ Clean up the marker array
            end if
        end for
    end for
end procedure
```

Note: If vertex v have e neighbors, then we can always find a color, of which $id <= e + 1$, that vertex v can use.

6.3 Performance Analysis

The array accesses counts are as follows:

1. Get the start pointer for $Adj[*]$. Once each vertex, then total is $|V|$
2. go through the neighbor list, which consumes $2|E|$
3. Read Color of each vertex i's neighbors in line 6. Since there are $|E|$ edges, then the total reads is $2|E|$
4. Mark neighbors' colors in $NeiborColorUsed[]$ in line 9, only when the neighbor is colored already. Thus each edge only accessed once, then the total is $|E|$
5. Color the vertices in line 16. Total is $|V|$
6. Check and clean marks in line 15 and 19. Not exceed $2 * NumberOfNeighbors$. Thus the total is less than $4|E|$

To sum up, the algorithm can be done in no more than a constant multiple of $|V| + |E|$

7 Ex 1.7

Note: Next time we have to claim pass by reference/pass by value more clearly

7.1 Question i

Algorithm 12 List Processing i, move front record

```
procedure $MoveFrontRecord(Y, M)$          ▷ Y is the source and M is the destination
    $l \leftarrow Y$
    $Y \leftarrow l.next$
    $l.next \leftarrow M$
    $M \leftarrow l$
end procedure
```

Note: Although in the question description it already assumed that Y is not empty, we'd better make a judgment $If(Y \neq Nil)$ at the beginning.

7.2 Question ii

Algorithm 13 List Processing i, move front record

```
procedure $ReverseList(Y)$
    $l \leftarrow Y$
    $m \leftarrow Nil$
    while $l \neq Nil$ do
        $MoveFrontRecord(l, m)$
    end while
    $Y \leftarrow m$
end procedure
```

7.3 Question iiia

1. Reverse the list using the algorithm in Question ii
2. Calculate the prefix sum for the reversed list using the algorithm in Question 0
3. Reverse the list back

Actually it is not efficient as in iiib) since we try to use the algorithms above.

7.4 Question iiib

Algorithm 14 List Processing iiib, ßuffixßum, two-pass solution

```
procedure $SuffixSum_{iiib}(L)$
    $sum \leftarrow 0$
    $l \leftarrow L$
    while $l \neq Nil$ do
        $sum \leftarrow sum + l.dat$
        $l \leftarrow l.next$
    end while
    $l \leftarrow L$
    while $l \neq Nil$ do
        $temp \leftarrow l.dat$
        $l.dat \leftarrow sum$
        $sum \leftarrow sum - temp$
        $l \leftarrow l.next$
    end while
end procedure
```

7.5 Question iiic

Algorithm 15 List Processing iiic, ßuffixßum, gluing solution

```
procedure $SuffixSum_{iiic}(L)$
    $m \leftarrow Nil$
    while $L \neq Nil$ do
        $MoveFrontRecord(L, m)$
    end while
    $Sum \leftarrow 0$
    while $m \neq Nil$ do
        $MoveFrontRecord(m, L)$
        $Sum \leftarrow Sum + L.dat$
        $L.dat \leftarrow Sum$
    end while
end procedure
```

Note: we can change the value of the list-item before or after we move it. Both ways are Ok, just ensure we are accessing the right list to find the item.

7.6 Question iiid

At beginning we call $PrefixSum_{recursive}(L, 0)$

Algorithm 16 Prefix Sum, recursive solution

```
procedure $PrefixSum_{recursive}(L, sum)$
    if $L \neq Nil$ then
        $L.dat \leftarrow L.dat + sum$
        $PrefixSum_{recursive}(L.next, L.dat)$
    end if
end procedure
```

Note: Next time we have to claim pass by reference/pass by value more clearly

7.7 Question iiie

Algorithm 17 Suffix Sum, recursive solution

```
procedure $SuffixSum_{recursive}(L)$        ▷ L is the input list, and this procedure return a integer
    if $L \neq Nil$ then
        $L.dat \leftarrow L.dat + SuffixSum_{recursive}(L.next)$
        return $L.dat$
    else
        return 0
    end if
end procedure
```

Note: As we just learned, we'd better use a *function* instead of *procedure* if we would like to return a value.

7.8 Question iv.a

Algorithm 18 Determine if we have a loop structure

```
procedure $DeterminStructure(L)$ ▷ This procedure return a boolean:$True$ means we have a loop, $False$ means the list ends with a pointer to $Nil$
    $OneStepPointer \leftarrow L$
    $TwoStepPointer \leftarrow L$
    repeat
        $TwoStepPointer \leftarrow TwoStepPointer.next$
        if $TwoStepPointer == Nil$ then
            return False
        end if
        $TwoStepPointer \leftarrow TwoStepPointer.next$
        if $TwoStepPointer == Nil$ then
            return False
        end if
        $OneStepPointer \leftarrow OneStepPointer.next$
    until $OneStepPonter == TwoStepPointer$
    return True
end procedure
```

Explain:

- If there is indeed a loop, then the $TwoStepPointer$ will eventually "catch up"$OneStepPointer$ no later than $OneStepPointer$ reach the end of the list.Thus the total pointer moves is no more than $3|L|$.(Here is a simple proof: After $OneStepPointer$ enter into the cycle, wherever the $TwoStepPointer$ is, it will cost no more than one round for $OneStepPointer$ to be catch up. Thus for $OneStepPointer$, the total moves it take will be no more than a "leading part"+ öne cycle", that is exactly $|L|$)
- If there is no loop, then the total pointer moves is no more than $1.5|L|$

7.9 Question iv.b

Algorithm 19 Determine the cycle length

```
procedure $DeterminCycleLength(L)$                ▷ This procedure returns length of the cycle
    $OneStepPointer \leftarrow L$
    $TwoStepPointer \leftarrow L$
    repeat
        $TwoStepPointer \leftarrow TwoStepPointer.next$
        $TwoStepPointer \leftarrow TwoStepPointer.next$
        $OneStepPointer \leftarrow OneStepPointer.next$
    until $OneStepPonter == TwoStepPointer$
    $Counter \leftarrow 0$
    repeat
        $OneStepPointer \leftarrow OneStepPointer.next$
        $Counter \leftarrow Counter + 1$
    until $OneStepPonter == TwoStepPointer$
    return $Counter$
end procedure
```

Explain:

1. When the $TwoStepPointer$ catch up $OneStepPointer$ first time, they must be already inside the cycle. The moves that $OneStepPointer$ take will be no more than $|L|$, thus the total pointer moves will be no more than $3|L|$
2. Then in the next repeat-until loop, $TwoStepPointer$ will just stand there and wait for $OneStepPointer$ go for a round. That would cost no more than $|L|$ pointer moves.

7.10 Question iv.c

Algorithm 20 Determine the cycle length and the leader length

```
procedure $DeterminBothLength(L; CycleLength, LeaderLength)$ ▷ This procedure gives out the length
of the cycle and the the length of the "leader"that precedes the cycle.
    $CycleLength \leftarrow DeterminCycleLength(L)$
    $OneStepPointer1 \leftarrow L$
    $OneStepPointer2 \leftarrow L$
    $tempCounter \leftarrow CycleLength$
    repeat
        $OneStepPointer \leftarrow OneStepPointer.next$
        $tempCounter = tempCounter - 1$
    until $tempCounter == 0$
    $LeaderLength \leftarrow 0$
    while $OneStepPonter1 \neq OneStepPointer2$ do
        $OneStepPointer1 \leftarrow OneStepPointer1.next$
        $OneStepPointer2 \leftarrow OneStepPointer2.next$
        $LeaderLength \leftarrow LeaderLength + 1$
    end while
    return $CycleLength, LeaderLength$
end procedure
```

Explain:

1. First we use the method in iv.b to get the length of the cycle in $4|L|$ pointer moves. (We do not actually "call"the function here since we do not have enough memory to afford a call stack,we just omit the repeat

parts. The counters in procedure $DeterminCycleLength$ can be reused or deleted thus we can store new things.)

2. We let one pointer moves $CycleLength$ steps first, and then they moves simultaneously.
3. Then they will met exactly at the entrance of the cycle.
4. The total pointer moves will be no more than $4|L| + |L| + |L| = 6|L|$

8 Ex 1.12

Algorithm 21 Slowest Hanoi

```
procedure $SloTh(n, A, B, C)$
    if $n == 1$ then
        move ring 1 from A to C
        move ring 1 from C to B
    else
        $SloTh(n - 1, A, B, C)$
        move ring $n$ from $A$ to $C$
        $SloTh(n - 1, B, A, C)$
        move ring $n$ from $C$ to $B$
        $SloTh(n - 1, A, B, C)$
    end if
end procedure
```