Assignment 1

Xiaoyu Xue

2017-09-11

1 Ex 1.1

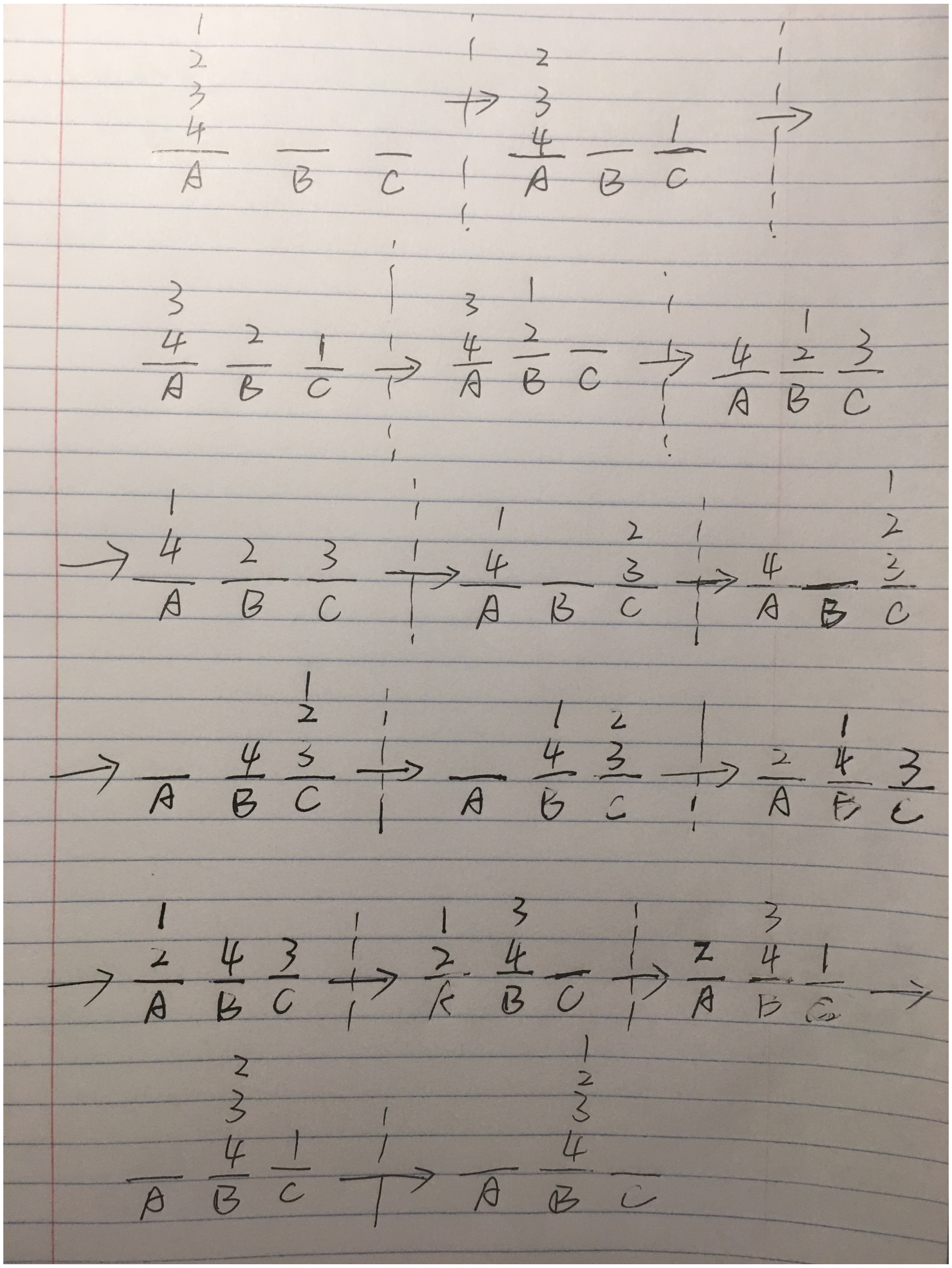

Figure 1: *Exe 1.1 Solution.*

2 Ex 1.2

2.a

Beginners will find the smallest ring first and move it to either A or C. And then find the second small one and figure out which pole should move to.

2.b

Suppose I have a n-1 DTH sovler called $minorDTHSolver$, I will test if pole B contains the ring n first. If it is true, move n-1 rings to pole B and problem solved. If ring n is on pole A, I will move all n-1 rings to pole C with the solver and then move the ring n from pole A to B, then move all n-1 rings to B with the solver. If ring n is on pole C, it's the same.

2.c

```
procedure DTH(n, A, B, C);
  if n equals 1 then
    if IsItThere(1, A) equals True then
      move ring 1 from the top of A to the top of B
    elseif IsItThere(1, C) equals True then
      move ring 1 from the top of C to the top of B
    endif
  else
    if IsItThere(n, A) equals True then
      DTH(n-1, A, C, B)
      move ring n from the top of A to the top of B
      DTH(n-1, A, B, C)
    elseif IsItThere(n, C) equals True then
      DTH(n-1, B, A, C)
      move ring n from the top of C to the top of B
      DTH(n-1, A, B, C)
    elseif IsItThere(n, B) equals True then
      DTH(n-1, A, B, C)
    endif
  endif
end_DTH;
```

2.d

```
procedure DTH(n, A, B, C);
  if n equals 1 then
    if IsItThere(1, A) equals True then
```

```
      move ring 1 from the top of A to the top of B
    elseif IsItThere(1, C) equals True then
      move ring 1 from the top of C to the top of B
    endif
  else
    if IsItThere(n, A) equals True then
      DTH(n-1, A, C, B)
      move ring n from the top of A to the top of B
      TowersOfHanoi(n-1, C, B, A)
    elseif IsItThere(n, C) equals True then
      DTH(n-1, B, A, C)
      move ring n from the top of C to the top of B
      TowersOfHanoi(n-1, A, B, C)
    elseif IsItThere(n, B) equals True then
      DTH(n-1, A, B, C)
    endif
  endif
end_DTH;
```

3 Ex 1.3

3.a

Ring 1 (the smallest)

3.b

How can I use the pole D to make the TowersOfHanoi procedure more efficient in $EACH$ recursive round. Make the size of subproblem as small as possible.

3.c

```
procedure DPoleTH(n, A, B, C, D);
  if n equals 1 then
    move ring 1 from the top of A to the top of B
  elseif n equals 2 then
    move ring 1 from the top of A to the top of D
    move ring 2 from the top of A to the top of B
    move ring 1 from the top of D to the top of B
  else
    DPoleTH(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;
    DPoleTH(n-1, C, B, A, D)
```

```
    endif
end_DPoleTH;
```

4 EX 1.4

4.a

Ring n's top sides is white, the other n-1 rings' top sides color are red cause they are flipped twice.

4.b

```
procedure RTH(n, A, B, C);
    if n equals 1 then
        move ring 1 from the top of A to the top of C
        move ring 1 from the top of C to the top of B
    else
        RTH(n-1, A, C, B);
        move ring n from the top of A to the top of B;
        RTH(n-1, C, B, A)
    endif
end_RTH;
```

4.c

```
procedure WTH(n, A, B, C);
    if n equals 1 then
        move ring 1 from the top of A to the top of B
    else
        WTH(n-1, A, C, B);
        move ring n from the top of A to the top of B;
        WTH(n-1, C, A, B)
        WTH(n-1, A, B, C)
    endif
end_WTH;
```

5 EX 1.5

```
procedure CTHOneJump(n, A, B, C);
    if n equals 1 then
        move ring 1 from the top of A to the top of B
    else
        CTHOneJump(n-1, A, B, C);
        CTHOneJump(n-1, B, C, A);
        move ring n from the top of A to the top of B;
```

```
        CTHOneJump(n-1, C, A, B);
        CTHOneJump(n-1, A, B, C);
    endif
end_CTHOneJump;
```

6 EX 1.6

6.a

```
procedure GColor(n, Adj[1..n], ColorOf[1..n] );
    input: the vertices 1,2,..., n, where for each j, Adj[j] is a list of j's neighbouring vertices;
    output: Color[j] is the color assigned to vertex j
    Create empty sets of Colorable
    Create an array MostRecentNeighborColor[1..n] with initial value is NIL Copy the n vertices into Colorable
    while Colorable not empty do
        Select a new color for this new round and call it c;
        foreach vertex v in the set Colorable do
            if MostRecentNeighborColor[v] not equals c then
                ColorOf[v] ← c;
                Remove v from Colorable;
                foreach vertex w in the list Adj[v] do
                    MostRecentNeighborColor[w] ← c
                endfor
            endif
        endfor
    endwhile
end_GColor;
```

6.b

```
procedure GColor(n, Adj[1..n], ColorOf[1..n] );
    input: the vertices 1,2,..., n, where for each j, Adj[j] is a list of j's neighbouring vertices;
    output: Color[j] is the color assigned to vertex j
    Create an array Color[1..m] with initial value is 0
    while Colorable not empty do
        Select a new color for this new round and call it c;
        foreach vertex i do
            foreach vertex j in Adj[i] do
                Color[ColorOf[j]] ← 1
            endfor
            foreach c in Color[m] do
                if c equals 0 then
                    ColorOf[i] ← c
                endif
                foreach vertex j in Adj[i] do
```

$Color[ColorOf[j]] \leftarrow 0$
endfor

endfor
endwhile
end_$GColor$;

6.c

As we can see there are 3 $Foreach$ loop with each vertex, if they traverse all neighbor of vertex v then there are maximum 3*13 array access. At last each vertex should be colored which needs two array access. Hence, the pseudocode solve all subgoals in no more than 3*13+2 array access.

7 EX 1.7

7.a

$y \leftarrow Y$;
$m \leftarrow M$;
$temp \leftarrow y.next$
$y.next \leftarrow m$
$m \leftarrow y$
$y \leftarrow temp$

7.b

procedure $Reverse$(Y);
$p \leftarrow Y$;
$q \leftarrow Y.next$
p.next = Nil;
while q not equals Nil **do**
$temp \leftarrow q.next$;
$q.next \leftarrow p$;
$p \leftarrow q$;
$q \leftarrow temp$;
endwhile
$Y \leftarrow p$;
end_$Reverse$;

7.c

Reverse the list.
Use the Part 0 solution change each record's dat;

Reverse the list again;

7.d

Iterate all record and get the sum of all record's data, called SumOfAll.
Iterate all record, calculate the sum of all precede records' dat called SumOfPre, modify the each record's dat with SumOfAll – SumOfPre. Which is the sum of succeed records' dat.

7.e

procedure $SumSucWithReverseList$(Y);
  $Reverse(Y)$;
  $SumPre(Y)$;
  $Reverse(Y)$;
end_$SumSucWithReverseList$;

7.f

procedure $SumPreWithRecur$(Y, sum);
  $p \leftarrow Y$;
  $sum \leftarrow sum + p.dat$;
  $p.dat \leftarrow sum$;
  if $p.next$ not equals Nil **then**
    $SumPreWithRecur(p.next, sum)$;
  endif
end_$SumPreWithRecur$;

7.g

function $SumSucWithRecur(Y)$;
  $p \leftarrow Y$;
  if $p.next$equals Nil **then**
    return$(p.dat)$;
  else
    $p.dat \leftarrow SumSucWithRecur(p.next) + p.dat$;
  endif
end_$SumSucWithRecur$;

7.h

procedure $TestCircle$(Y);
  $p \leftarrow Y.next$;
  $q \leftarrow Y.next.next$;
  while q not equals p **do**

```
        if q equals Nil then
            return(False)
        endif
        p ← p.next;
        q ← q.next.next;
    endwhile
    return(True);
end_TestCircle;
```

7.i

```
procedure CircleSize(Y);
    cnt1 ← 1;
    p ← Y.next;
    q ← Y.next.next;
    while q not equals p do
        p ← p.next;
        q ← q.next.next;
        cnt1 ← cnt1 + 1;
    endwhile
    return(cnt1);
end_CircleSize;
```

7.j

```
procedure CircleLeaderSize(Y);
    cnt1 ← 1;
    cnt2 ← 0;
    p ← Y.next;
    q ← Y.next.next;
    r ← Y;
    while q not equals p do
        p ← p.next;
        q ← q.next.next;
        cnt1 ← cnt1 + 1;
    endwhile
    while r not equals p do
        p ← p.next;
        r ← r.next;
        cnt2 ← cnt2 + 1;
    endwhile
    return(cnt1, cnt2);
end_CircleLeaderSize;
```

8 EX 1.12

8.a

```
procedure SloTH(n, A, B, C);
   if n equals 1 then
      move ring 1 from the top of A to the top of C
      move ring 1 from the top of C to the top of B
   else
      SloTH(n-1, A, B, C);
      move ring n from the top of A to the top of C;
      SloTH(n-1, B, A, C);
      move ring n from the top of C to the top of B;
      SloTH(n-1, A, B, C);
   endif
end_SloTH;
```

8.b

```
procedure SloTH(n, A, B, C);
   output: the n rings are moved from A to B in a way where all legal ring configurations are achieved for n rings on poles A, B and C, but just once for each configuration;
   if n equals 1 then
      move ring 1 from the top of A to the top of C
      move ring 1 from the top of C to the top of B
   else
      SloTH(n-1, A, B, C);    { so with ring n sitting on A, rings 1 through n - 1 have achieve all possible legal configurations on poles A, B and C with each configuration achieved just once }
      move ring n from the top of A to the top of C;
      SloTH(n-1, B, A, C);    { so with ring n sitting on C, rings 1 through n - 1 have achieve all possible legal configurations on poles A, B and C with each configuration achieved just once }
      move ring n from the top of C to the top of B;
      SloTH(n-1, A, B, C);    { so with ring n sitting on B, rings 1 through n - 1 have achieve all possible legal configurations on poles A, B and C with each configuration achieved just once }
   endif
end_SloTH;
```