CS/ECE 374 B Fall 2019 Jin Yucheng (yucheng9)
Homework 4 Problem 1 Fan Wenyan (wenyanf2)

Solution:

(a) The exact solution is:

$$A(n) = n^2, \text{ for any non-negative integer } n$$

Assume n is an arbitrary non-negative integer, for every non-negative integer $k < n$, we have $A(k) = k^2$, then:

- for $n = 0$, $A(0) = 0$ satisfies $A(n) = n^2$.
- for $n \geq 1$, by the inductive hypothesis, $A(n-1) = (n-1)^2$, such that:

$$A(n-1) + 2n - 1 = (n-1)^2 + 2n - 1 = n^2 - 2n + 1 + 2n - 1 = n^2 = A(n)$$

Therefore, $A(n) = n^2$ is the exact solution for part (a).

(b) The exact solution is:

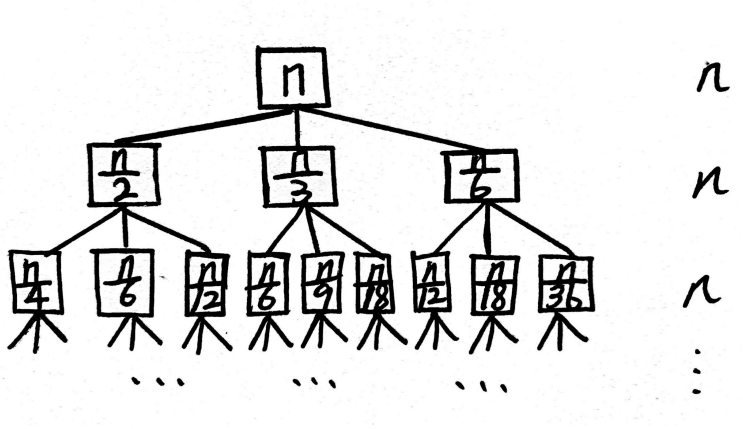
$$B(n) = \tfrac{1}{6}n^3 - \tfrac{1}{6}n, \text{ for any non-negative integer } n$$

Since for any non-negative integer $n > 1$, we have:

$$\begin{aligned}
B(n) &= B(n-1) + \binom{n}{2} \\
&= B(n-2) + \binom{n-1}{2} + \binom{n}{2} \\
&= \binom{2}{2} + \ldots + \binom{n-1}{2} + \binom{n}{2} \\
&\text{Because } \binom{n}{2} = \frac{n!}{2!(n-2)!} = \frac{(n-1)n}{2}, \text{ we have,} \\
&= \frac{1}{2}[1 \cdot 2 + 2 \cdot 3 + \ldots + (n-2)(n-1) + (n-1)n] \\
&= \frac{1}{2} \cdot \frac{n(n-1)(n+1)}{3} \\
&= \frac{1}{6}n^3 - \frac{1}{6}n
\end{aligned}$$

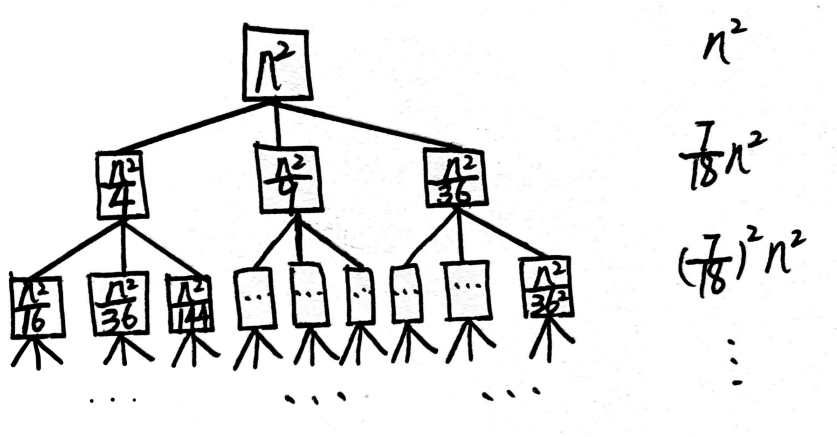
Since $B(0) = 0$, $B(1) = B(0) + \binom{1}{2} = 0$ also satisfy $B(n) = \frac{1}{6}n^3 - \frac{1}{6}n$. Therefore, $B(n) = \frac{1}{6}n^3 - \frac{1}{6}n$ is the exact solution for part (b).

(c) We draw out the first few levels of the recursion tree for $C(n) = C(n/2)+C(n/3)+C(n/6)+n$:



Since $\frac{1}{2}+\frac{1}{3}+\frac{1}{6} = 1$, in all levels we have the equal sum, n, we immediately have $C(n) = O(n \cdot L) = O(nlogn)$ (see Note 1.7).

(d) We draw out the first few levels of the recursion tree for $D(n) = D(n/2)+D(n/3)+D(n/6)+n^2$:



Since $\frac{1}{2^2}+\frac{1}{3^2}+\frac{1}{6^2} = \frac{7}{18}$, in all levels the sum decreases exponentially, which means every term is a constant factor smaller than the previous term, we immediately have $D(n) = O(n^2)$ (see Note 1.7).

■

Solution: The original algorithm is:

```
# move "ndisks" from "src" to "dest" via "tmp" as temporary space
1 : def hanoi(ndisks, src, dest, tmp):
2 :     if ndisks > 0:
3 :         hanoi(ndisks - 1, src, tmp, dest)
4 :         moveone(src, dest)
5 :         hanoi(ndisks - 1, tmp, dest, src)
6 :     else: # do nothing
```

(a) To make sure that moveone obeys the restriction that either the source or destination must be tower 0, after we move the largest disk from source to destination, we should put all the remaining disks back to source, so the modified algorithm is:

```
# move "ndisks" from "src" to "dest" via "tmp" as temporary space
1 : def hanoi(ndisks, src, dest, tmp):
2 :     if ndisks > 0:
3 :         hanoi(ndisks - 1, src, tmp, dest)
4 :         moveone(src, dest)
5 :         hanoi(ndisks - 1, tmp, src, dest)
6 :         hanoi(ndisks - 1, src, dest, tmp)
7 :     else: # do nothing
```

Where line 5 moves the remaining disks from "tmp" to "src" via "dest", which starts over the procedure to move disks from the source to destination, and what line 6 does is just moving the remaining disks from "src" to "dest" via "tmp".
Suppose the number of disks to move is n, the total number of calls to moveone is $F(n)$ if we need to move n disks, therefore, we have:

$$F(n) = F(n-1) + 1 + F(n-1) + F(n-1)$$

Where three $F(n-1)$ represent the total number of calls to moveone in line 3, line 5, line 6, respectively. By solving:

$$F(n) = 3F(n-1) + 1,\ F(0) = 0, F(1) = 1$$

We find that $F(n) = 3F(n-1) + 1 = 3F(n-2) + 3^1 + 3^0 = 3F(n-3) + 3^2 + 3^1 + 3^0 = ...$
So $F(n) = 3^0 + 3^1 + ... + 3^{n-1} = \frac{3^n-1}{2}$, totally $\frac{3^n-1}{2}$ calls to moveone are needed to move n disks from the source to destination.

(b) The algorithm for moving n disks from tower 0 to tower 1 is as follows:

move n disks from tower 0 to tower 1 via tower 2 as temporary space

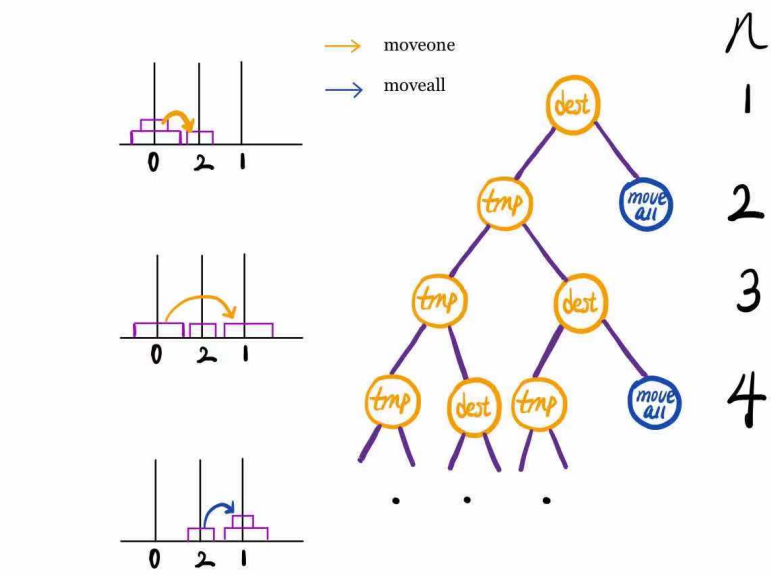
move "ndisks" from "src" to "dest" via "tmp" as temporary space

```
1 : def hanoi(ndisks, src, dest, tmp):
2 :     if ndisks > 0:
3 :         if src = 2:
4 :             moveall(2, tmp)
5 :         else:
6 :             hanoi(ndisks - 1, src, tmp, dest)
7 :         moveone(src, dest)
8 :         if tmp = 2:
9 :             moveall(2, dest)
10 :        else:
11 :            hanoi(ndisks - 1, tmp, dest, src)
12 :    else: # do nothing
```

Compared to the original algorithm, we add two if and else statements to utilize moveall function. These two if and else statements mean to move the entire remaining stack of $n-1$ disks from tower 2 to tower 0 or tower 1.

To calculate how many calls are needed to moveone and moveall, let us analyze two simple cases.

- if we only have 1 disk, then we only need to use moveone once.
- if we have 2 disks, then we need to use moveone tiwce and moveall once, the steps are indicated in the graph below.



Where the yellow arrows and circles show the number of moveone functions we call and blue arrows and circles show the number of moveall functions we call. Since the hanoi function is recursive, denote the number of moveall functions we need to call as $S(n)$, the number of moveone functions we need to call as $F(n)$, we find that:

$$S(n) = \lfloor n/2 \rfloor$$

$$F(n) = F(n-1) + F(n-2) + 1, \text{ for any integer } n > 2, \text{ where } F(1) = 1, F(2) = 2$$

By solving the linear recurrence relation of $F(n)$, we find that: ■

$$F(n) = \frac{1}{2}(3F_n + L_n - 2)$$

$$F_n = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right]$$

$$L_n = \left(\frac{1+\sqrt{5}}{2}\right)^n + \left(\frac{1-\sqrt{5}}{2}\right)^n$$

Jin, Yucheng (yucheng9) Fan, Wenyan (wenyanf2)

3. (a)

```
l := 1 (set l(eft) at 1)
r := n
FindBroken(l, r):
    if l > r
        return None

    mid ← int((l + r)/2)

    if test(mid, mid) = False
        return mid
    if test(l, mid - 1) = False
        return FindBroken(1, mid - 1)
    if test(mid + 1, r) = False
        return FindBroken(mid + 1, r)
```

Its runtime is **O(log(n))**.
FindBroken reduces potential broken lights by half in each step,
i.e., T(n) = T(n/2) + C, C is a constant.
Following is the initial recursion tree for the given recurrence relation.
Level 1: C
Level 2: T(n/2)
If we further break down the expression, we get following recursion tree.
Level 1: C
Level 2: C
Level 3: T(n/4)
Breaking down further gives us following.
Level 1: C
Level 2: C
Level 3: C
Level 4: ...
To know the value of T(n), we need to calculate sum of tree nodes level by level.
If we sum the above tree level by level, we get the following series
T(n) = C + C + C + ⋯
The above series is geometrical progression with ratio 1.
Refer to "1.7 Recursion Trees" in "Notes on recursion" (01-recursion.pdf).
If all terms in the series are equal, we immediately have
T(n) = O(C · L) = O(C · log(n)) = O(log(n)).
Its runtime is O(log(n)).

3. (b)

```
l := 1 (set l(eft) at 1)
r := n
Broken := {}
FindBroken(l, r):
        if l > r
                return None

        mid ← int((l + r)/2)

        if test(mid, mid) = False

                k ← k - 1

                Broken.add(mid)
        if k = 0:
                return Broken
        if test(l, mid - 1) = False
                return FindBroken(1, mid - 1)
        if test(mid + 1, r) = False
                return FindBroken(mid + 1, r)
```

k can be big as 0.25n before algorithm is no longer faster than testing each light.

If k is bigger than **0.25n**, algorithm has to test k lights with each light runtime **O(log(n))**.

3. (c)

1 <= first_left <= first_right < second_left <= second_right <= t

```python
def find_key_with_shared_factor(first_left, first_right, second_left, second_right):
    if first_right - first_left <= 1 and second_right - second_left <= 1:
        gcd = batchgcd(first_left, first_left, second_left, second_left)
        if gcd != 1:
            print ("The pair of key is: ", first_left, " and ", second_left)
        gcd = batchgcd(first_left, first_left, second_right, second_right)
        if gcd != 1:
            print ("The pair of key is: ", first_left, " and ", second_left)
        gcd = batchgcd(first_right, first_right, second_left, second_left)
        if gcd != 1:
            print ("The pair of key is: ", first_left, " and ", second_left)
        gcd = batchgcd(first_right, first_right, second_right, second_right)
        if gcd != 1:
            print ("The pair of key is: ", first_left, " and ", second_left)
        return
    first_middle = int((first_left + first_right) / 2)
    second_middle = int((second_left + second_right) / 2)
    gcd = batchgcd(first_left, first_right, second_left, second_right)
    if gcd != 1:
        gcd = batchgcd(first_left, first_middle, second_left, second_middle)
        if gcd != 1:
            return find_key_with_shared_factor(first_left, first_middle, second_left, second_middle)
        gcd = batchgcd(first_left, first_middle, second_middle, second_right)
        if gcd != 1:
            return find_key_with_shared_factor(first_left, first_middle, second_middle, second_right)
        gcd = batchgcd(first_middle, first_right, second_left, second_middle)
        if gcd != 1:
            return find_key_with_shared_factor(first_middle, first_right, second_left, second_middle)
        gcd = batchgcd(first_middle, first_right, second_middle, second_right)
        if gcd != 1:
            return find_key_with_shared_factor(first_middle, first_right, second_middle, second_right)
    else:
        gcd = batchgcd(first_left, first_middle, first_middle + 1, first_right)
        if gcd != 1:
            return find_key_with_shared_factor(first_left, first_middle, first_middle + 1, first_right)
        else:
            return find_key_with_shared_factor(second_left, second_middle, second_middle + 1, second_right)
```

Its runtime is **O(log(n))**.

find_key_with_shared_factor reduces potential broken lights by half in each step,

i.e., T(n) = T(n/2) + C, C is a constant.

Following is the initial recursion tree for the given recurrence relation.
Level 1: C
Level 2: T(n/2)
If we further break down the expression, we get following recursion tree.
Level 1: C
Level 2: C
Level 3: T(n/4)
Breaking down further gives us following.
Level 1: C
Level 2: C
Level 3: C
Level 4: ...
To know the value of T(n), we need to calculate sum of tree nodes level by level.
If we sum the above tree level by level, we get the following series
$T(n) = C + C + C + \cdots$
The above series is geometrical progression with ratio 1.
Refer to "1.7 Recursion Trees" in "Notes on recursion" (01-recursion.pdf).
If all terms in the series are equal, we immediately have
$T(n) = O(C \cdot L) = O(C \cdot \log(n)) = O(\log(n))$. Its runtime is $O(\log(n))$.