Problem 1

(a) $A(n) = A(n-1) + 2n - 1$, $A(0) = 0$

Idea: suppose that:

$$\begin{aligned}
A(n) &= \sum_{i=1}^{n} 2i - 1 \\
&= (\sum_{i=1}^{n} 2i) - i && \text{-1 repeated n times is n} \\
&= 2(\sum_{i=1}^{n} i) - i && \text{pull out 2 by distributive property} \\
&= 2\frac{n(n+1)}{2} - n && \text{See provided source [1]} \\
&= n(n+1) - n && \text{2's cancel} \\
&= n^2 && \text{distribute and simplify}
\end{aligned}$$

Prove via induction:

- Base Case: n = 1:
  $A(1) = A(0) + 2(1) - 1 = 0 + 2 - 1 = 1$
  $(1)^2 = 1$
- Inductive Hypothesis:
  Whenever $A(n)$ is defined recursively as $A(n) = A(n-1) + 2n - 1$, then $A(n) = n^2$
- Inductive step:

$$\begin{aligned}
A(i) &= A(n-1) + 2n - 1 && \text{definition of A(n)} \\
&= (n-1)^2 + 2n - 1 && \text{Inductive Hypothesis} \\
&= n^2 - 2n + 1 + 2n - 1 && \text{Simplify } (n-1)^2 \\
&= n^2 && \text{simplify expression}
\end{aligned}$$

We have now proven that $A(n) = n^2$.

[1] Sum of n, n^2, or n^3. *Brilliant.org*. Retrieved October 3, 2019, from https://brilliant.org/wiki/sum-of-n-n2-or-n3/

(b) $B(n) = B(n-1) + \binom{n}{2}$, $B(0) = 0$

idea: suppose that:

$$
\begin{aligned}
B(n) &= \sum_{i=1}^{n} \binom{i}{2} \\
&= 0 + \sum_{i=2}^{n} \frac{i!}{2!(i-2)!} && \text{definition of k-choose-n} \\
&= \frac{1}{2} \sum_{i=2} \frac{i!}{(i-2)!} && \text{factor out } 1/2 \\
&= \frac{1}{2} \sum_{i=2} \frac{i(i-1)(i-2)!}{(i-2)!} && \text{definition of factorial} \\
&= \frac{1}{2} \sum_{i=2}^{n} i(i-1) && \text{factor out (i - 2)!} \\
&= \frac{1}{2} \sum_{i=2}^{n} i^2 - i && \text{simplify} \\
&= \frac{1}{2} \sum_{i=2}^{n} i^2 - \frac{1}{2} \sum_{i=2}^{n} i && \text{group similar terms} \\
&= (-1 + \frac{1}{2} \sum_{i=1}^{n} i^2) - (-1 + \frac{1}{2} \sum_{i=1}^{n} i) && \text{change summation limits} \\
&= \frac{1}{2} \sum_{i=1}^{n} i^2 - \frac{1}{2} \sum_{i=1}^{n} i && \text{simplify} \\
&= \frac{1}{2} \frac{n(n+1)(2n+1)}{6} - \frac{1}{2} \frac{n(n+1)}{2} && \text{see source provided below } [2] \\
&= \frac{n^3 - n}{6} && \text{simplify}
\end{aligned}
$$

$B(0) = 0$

$B(1) = 0$

$B(n) = \frac{n^3 - n}{6} \forall n > 1$

Prove via induction

- base case:

  $B(0) = 0$
  $B(1) = B(0) + \binom{1}{2} = 0 + 0 = 0$
  $B(2) = B(1) + \binom{2}{2} = 0 + 1 = 1$
  $B(2) = \frac{2^3 - 2}{6} = 1$

[2] Sum of n, n^2, or n^3. *Brilliant.org*. Retrieved October 3, 2019, from https://brilliant.org/wiki/sum-of-n-n2-or-n3/

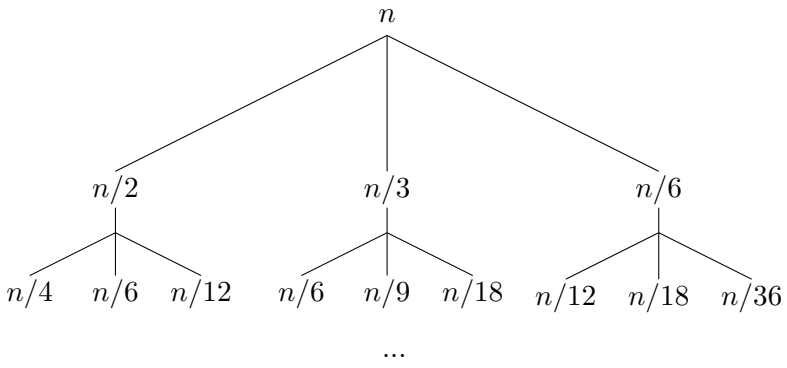
- Inductive Hypothesis:
  Suppose that we have the recurrence $B(n) = B(n-1) + \binom{n}{2}$. Then we can also say that $B(n) = \frac{n^3-2}{6}$.
- Inductive Step:

$$
\begin{aligned}
B(n) &= B(n-1) + \binom{n}{2} \\
&= \frac{(n-1)^3 - (n-1)}{6} + \binom{n}{2} && \text{Induction hypothesis} \\
&= \frac{(n-1)^3 - n + 1}{6} + \frac{n!}{2!(n-2)!} && \text{definition of k-choose-n} \\
&= \frac{(n-1)^3 - n + 1}{6} + \frac{n(n-1)(n-2)!}{2(n-2)!} && \text{definition of factorial} \\
&= \frac{(n-1)^3 - n + 1}{6} + \frac{n(n-1)}{2} && \text{(n-2)!'s cancel out} \\
&= \frac{n^3 - 3n^2 + 2n}{6} + \frac{n(n-1)}{2} && \text{simplify} \\
&= \frac{(n^3 - 3n^2 + 2n) + (3n^2 - 3n)}{6} && \text{combine terms} \\
&= \frac{n^3 - n}{6} && \text{simplify}
\end{aligned}
$$

We have now shown that $B(n) = \frac{n^3-n}{6} \forall n > 1$, with $B(1) = 0$.

(c) $C(n) = C(n/2) + C(n/3) + C(n/6) + n$

Develop closed form expression using the tree method:



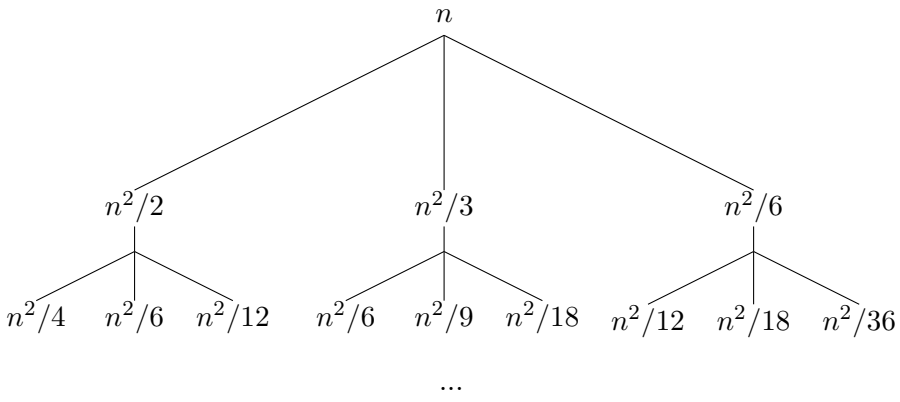
Where the top level of the tree is representative of $C(n)$ and all terms connected to it are representative of the quantity added by the recursive calls of $C(n)$.

Notice how when adding the terms across each level of the tree, the values sum up to n. Also, as the leftmost branch of the tree grows downward, it will be the first to end in a leaf. Therefore the sum across each branch is actually less than or equal to n, and we can now develop a bound. The height of the tree is determined by the rightmost branch, which is proportional to $log_6(n)$. Therefore, we can develop an asymptotic solution to the recurrence as:

$$C(n) < n \log_6 n$$
$$C(n) \in \Theta(n \log_6 n)$$

(d) $D(n) = D(n/2) + D(n/3) + D(n/6) + n^2$

We can develop a closed form expression using, again, the tree method:



The resulting tree follows the same pattern as the previous problem, except this time the levels individually sum up to n^2. Following the same rules, we find that:

$$D(n) < n^2 \log_6 n$$
$$D(n) \in \Theta(n^2 \log_6 n)$$

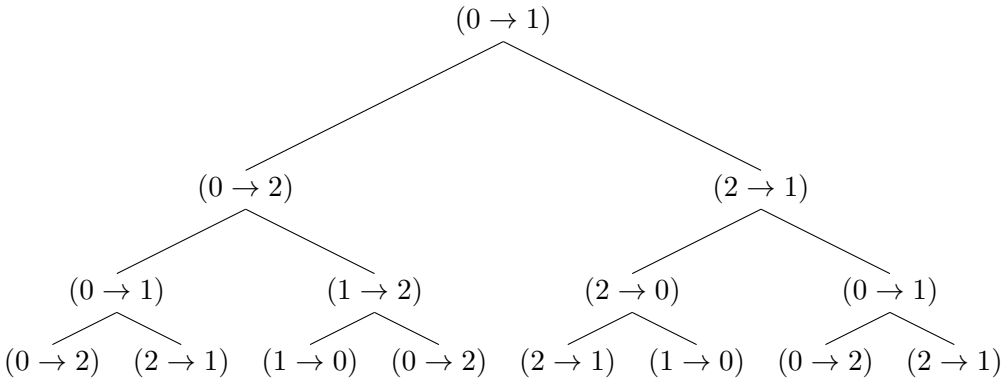
Problem 2

(a) Suppose that moveone had a restriction that either the source or the destination must be tower 0. Modify the recursive algorithm to abide by this restriction. Analyze exactly how many calls to moveone are needed to move n disks in your solution.

There is an important insight to realize when solving this problem: even though we cannot move disks directly between poles 1 and 2 there is a work-around. Instead of going pole 1 $\rightarrow$ pole 2, we can instead do pole 1 $\rightarrow$ pole 0 $\rightarrow$ pole 2. And for the reverse situation of pole 2 $\rightarrow$ pole 1, we can instead do pole2 $\rightarrow$ pole 0 $\rightarrow$ pole 1. The newly modified algorithm is as follows:

```
def hanoi(ndisks, source, dest, tmp):
        """Move 'ndisks' from the 'source' tower to
        the 'dest' tower, using the 'tmp' tower as
        temporary space"""
        if ndisks > 0:
                # recursively move stack of n-1 disks to
                # tmp tower
                hanoi(n - 1, source, tmp, dest)
                # move one disk from source to destination
                if source != 0 and dest != 0:
                        moveone(source, 0)
                        moveone(0, dest)
                else:
                        moveone(source, dest)
                # recursively move stack of n - 1
                # disks to dest tower
                hanoi(n - 1, tmp, dest, source)
        else:
                pass #do nothing
```

The above algorithm means that every time we move something between poles 1 and 2, we have to make 2 calls to moveone, whereas if we move something in an action that uses 0 as a pole, we only have to call moveone once. In order to analyze the runtime of the program, let us first look at the recursion tree generated by the program, where $a \rightarrow b$ represents hanoi(ndisks, a, b, tmp):



...

Denote the hanoi calls that include pole 0 as either a source or a destination as the "good calls" (because the require only one call to moveone). Denote the hanoi calls that do not include pole 0 as a source or a destination as the "bad calls" (because they require 2 calls to moveone). Also denote the top level of the tree as level $l = 1$.

Notice one very important pattern about how good calls and bad calls operate. Bad calls will always recursively call 2 good calls, as it uses 0 as the temporary pole. Good calls will recursively call one good call and one bad call, because it uses either 1 or 2 as a transition pole. Using this information, we can derive a recurrence relationship for the number of bad calls and good calls. *Notation note: $N_{good,l}$ means the number of good calls on level l whereas $N_{bad,l}$ means the number of bad calls on level l.*

$$N_{good,l} = N_{good,l-1} + 2N_{bad,l-1}$$
$$N_{bad,l} = N_{good,l-1}$$

Rearranging the formulas a bit yields the following:

$$N_{good,l} = N_{good,l-1} + 2N_{good,l-2}$$
$$N_{bad,l} = N_{good,l-1}$$

Now we have a way to solve our problem. The game plan is as follows: Develop a closed form expression for the number of good calls and bad calls for a hanoi tower of size n. After we have those closed form expression, the number of calls is equal to $2N_{bad}(n) + N_{good}(n)$ because every call to N_{bad} requires 2 calls to moveone and N_{good} only requires one call.

To develop our closed form expression, put the equation into matrix form:

$$\begin{bmatrix} N_{good,l+1} \\ N_{good,l} \end{bmatrix} = \begin{bmatrix} 1 & 2 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} N_{good,l} \\ N_{good,l-1} \end{bmatrix}$$

Getting closer to a closed form expression we see that:

$$\begin{bmatrix} N_{good}(l) \\ N_{good}(l-1) \end{bmatrix} = \begin{bmatrix} 1 & 2 \\ 1 & 0 \end{bmatrix}^n \begin{bmatrix} N_{good}(1) \\ N_{good}(0) \end{bmatrix} = \begin{bmatrix} 1 & 2 \\ 1 & 0 \end{bmatrix}^l \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

We can diagonalize the matrix using MATLAB, to get the following result:

$$\begin{bmatrix} N_{good}(l) \\ N_{good}(l-1) \end{bmatrix} = \left(\begin{bmatrix} 2 & -1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 2 & 0 \\ 0 & -1 \end{bmatrix} \frac{1}{3} \begin{bmatrix} 1 & 1 \\ -1 & 2 \end{bmatrix} \right)^l \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

by property of diagonalization we find that:

$$\begin{bmatrix} N_{good}(l) \\ N_{good}(l-1) \end{bmatrix} = \begin{bmatrix} 2 & -1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 2^l & 0 \\ 0 & -1^l \end{bmatrix} \frac{1}{3} \begin{bmatrix} 1 & 1 \\ -1 & 2 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

Multiply everything together and now we get:

$$\begin{bmatrix} N_{good}(l) \\ N_{good}(l-1) \end{bmatrix} = \frac{1}{3} \begin{bmatrix} 2^{l+1} - (-1)^{l+1} \\ 2^l = (-1)^l \end{bmatrix} \implies N_{good}(l) = \frac{2^{l+1} - (-1)^{l+1}}{3}$$

Because we already have an expression for N_{bad}, l, we can plug this our newly derived expression and get our desired results of:

$$N_{good,l} = \frac{2^{l+1} - (-1)^{l+1}}{3}$$
$$N_{bad,l} = \frac{2^{l} - (-1)^{l}}{3}$$

But these are simply the number of good and bad terms on each row of the tree. We want the total number of each in the entire tree, across all levels. Well to do this, we simply sum up the value of each expression across the height of the tree, which goes for level $l = 0$ to level $l = n - 1$.

$$\begin{aligned}
T(n) &= 2\sum_{l=0}^{n-1} N_{bad,l} + \sum_{l=0}^{n-1} N_{good,l} \\
&= 2\sum_{l=0}^{n-1} \frac{2^l - (-1)^l}{3} + \sum_{l=0}^{n-1} \frac{2^{l+1} - (-1)^{l+1}}{3} && \text{previously derived expressions} \\
&= \frac{2}{3}\sum_{l=0}^{n-1} 2^l + \frac{1}{3}\sum_{l=0}^{n-1} 2^{l+1} - \frac{1}{3}\sum_{l=0}^{n-1} 2(-1)^l + (-1)^{l+1} && \text{group similar terms} \\
&= \frac{2}{3}\sum_{l=0}^{n-1} 2^l + \frac{2}{3}\sum_{l=0}^{n-1} 2^l - \frac{1}{3}\sum_{l=0}^{n-1} 2(-1)^l - (-1)^l && \text{pull out a -1 and 2 from the exponents} \\
&= \frac{2}{3}\sum_{l=0}^{n-1} 2^l + \frac{2}{3}\sum_{l=0}^{n-1} 2^l - \frac{1}{3}\sum_{l=0}^{n-1} (-1)^l && \text{simplify (-1) series} \\
&= \frac{4}{3}\sum_{l=0}^{n-1} 2^l - \frac{1}{3}\sum_{l=0}^{n-1} (-1)^l && \text{add together } 2^l \text{ series} \\
&= \frac{4}{3}(2^{n-1+1} - 1) - \frac{1}{3}\frac{1 - (-1)^n}{2} && \text{simplify series } [3] \\
&= \frac{4}{3}(2^{n} - 1) - \frac{1}{3}\frac{1 - (-1)^n}{2} && \text{simplify again}
\end{aligned}$$

Thus concludes the proof. We have found that with the given modifications, the number of calls to moveone is equal to $\frac{4}{3}(2^n - 1) - \frac{1}{3}\frac{1-(-1)^n}{2}$ for a hanoi tower of size n.

[3] This solution is a bit of a hack: the right summation term fluctuations between 0, 1/3, 0, 1/3, 0, 1/3 so on and forever. And that's what the little function I replaced it with does. The series of 2 is a well-known summation, i.e. the power series of 2.

(b) Suppose instead that you are give another call, moveall that can move an entire stack of disks from one tower to another, but moveall can only be called to move disks from tower 2.

The moveall function allows us to do a shortcut between poles 2 and any other poles. To use this in our code, we can do the following:

```
def hanoi(ndisks, source, dest, tmp):
        """Move 'ndisks' from the 'source' tower to
        the 'dest' tower, using the 'tmp' tower as
        temporary space"""
        if ndisks > 0:
                if source == 2:
                        # if we're moving a stack from 2 to
                        # another pole directly, use the
                        # movall as a shortcut
                        moveall(2, dest)
                else:
                        # recursively move stack of n-1 disks to
                        # tmp tower
                        hanoi(n - 1, source, tmp, dest)
                        # move one disk from source to destination
                        moveone(source, dest)
                        # recursively move stack of n - 1
                        # disks to dest tower
                        hanoi(n - 1, tmp, dest, source)
        else:
                pass #do nothing
```

To generate an expression that calculates the number of calls to moveone and moveall, we can (ab)use linear algebra. In order to do this, we have to notice some patterns. Again, use the shorthand notation of $a \to b$ being short for hanoi(ndisks, a, b, tmp). For every call to hanoi, there are up to 2 recursive calls. The calls are as follows:

$(0 \to 1)$ calls $(0 \to 2)$ and $(2 \to 1)$
$(0 \to 2)$ calls $(0 \to 1)$ and $(1 \to 1)$
$(1 \to 0)$ calls $(1 \to 2)$ and $(2 \to 0)$
$(2 \to 0)$ performs no recursive subcalls
$(2 \to 1)$ performs no recursive subcalls
$(1 \to 2)$ calls $(1 \to 0)$ and $(0 \to 2)$

Before the next step, one more notation note: Every time we perform a recursive subcall we go up a "level". For example, let the first call of hanoi() be denoted level 0. Hanoi() then calls itself twice, with each of those functions being level 1. In other words, "level" has the same meaning as it does in part A.

Now, let the phrase $N_{ab,l}$ denote the number of hanoi calls on level l that go from source a to destination b (aka $(a \to b)$). We can define a relationship for each $N_{ab,l}$ as follows, when combining it with the previously noticed relation:

$$\begin{bmatrix} N_{01,l+1} \\ N_{02,l+1} \\ N_{10,l+1} \\ N_{12,l+1} \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} N_{01,l} \\ N_{02,l} \\ N_{10,l} \\ N_{12,l} \end{bmatrix}$$

Since we know the starting conditions for our call to hanoi, we can do the following:

$$\begin{bmatrix} N_{01,l} \\ N_{02,l} \\ N_{10,l} \\ N_{12,l} \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}^l \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

The matrix is diagonizable and we can use this to form a closed-form expression, with the help of MATLAB of course (Note: The calculations of the below matrix were extremely messy, to save everyone the effort I did not write down the individual steps but rather the final answer). A is the matrix of eigenvectors, B is the matrix of eigen values.

$$A = \begin{bmatrix} \frac{-\sqrt{5}-1}{2} & \frac{-\sqrt{5}-1}{2} & \frac{\sqrt{5}-1}{2} & \frac{\sqrt{5}-1}{2} \\ -1 & 1 & 1 & -1 \\ \frac{\sqrt{5}+1}{2} & \frac{-\sqrt{5}-1}{2} & \frac{\sqrt{5}-1}{2} & \frac{-\sqrt{5}-1}{2} \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

$$B = \begin{bmatrix} \frac{\sqrt{5}-1}{2} & 0 & 0 & 0 \\ 0 & \frac{-\sqrt{5}+1}{2} & 0 & 0 \\ 0 & 0 & \frac{\sqrt{5}+1}{2} & 0 \\ 0 & 0 & 0 & \frac{-\sqrt{5}-1}{2} \end{bmatrix}$$

$$C = A^{-1}$$

$$\begin{bmatrix} N_{01,l} \\ N_{02,l} \\ N_{10,l} \\ N_{12,l} \end{bmatrix} = A * B^l * C$$

$$\begin{bmatrix} N_{01,l} \\ N_{02,l} \\ N_{10,l} \\ N_{12,l} \end{bmatrix} = \begin{pmatrix} \frac{\sqrt{5}\left(\frac{\sqrt{5}}{2}-\frac{1}{2}\right)\left(-\frac{\sqrt{5}}{2}-\frac{1}{2}\right)^n}{10} + \frac{\sqrt{5}\left(\frac{\sqrt{5}}{2}-\frac{1}{2}\right)\left(\frac{\sqrt{5}}{2}+\frac{1}{2}\right)^n}{10} + \frac{\sqrt{5}\left(\frac{1}{2}-\frac{\sqrt{5}}{2}\right)^n\left(\frac{\sqrt{5}}{2}+\frac{1}{2}\right)}{10} + \frac{\sqrt{5}\left(\frac{\sqrt{5}}{2}-\frac{1}{2}\right)^n\left(\frac{\sqrt{5}}{2}+\frac{1}{2}\right)}{10} \\ -\frac{\sqrt{5}\left(\left(-\frac{\sqrt{5}}{2}-\frac{1}{2}\right)^n+\left(\frac{1}{2}-\frac{\sqrt{5}}{2}\right)^n-\left(\frac{\sqrt{5}}{2}-\frac{1}{2}\right)^n-\left(\frac{\sqrt{5}}{2}+\frac{1}{2}\right)^n\right)}{10} \\ \frac{\sqrt{5}\left(\frac{\sqrt{5}}{2}-\frac{1}{2}\right)\left(\frac{\sqrt{5}}{2}+\frac{1}{2}\right)^n}{10} - \frac{\sqrt{5}\left(\frac{\sqrt{5}}{2}-\frac{1}{2}\right)\left(-\frac{\sqrt{5}}{2}-\frac{1}{2}\right)^n}{10} + \frac{\sqrt{5}\left(\frac{1}{2}-\frac{\sqrt{5}}{2}\right)^n\left(\frac{\sqrt{5}}{2}+\frac{1}{2}\right)}{10} - \frac{\sqrt{5}\left(\frac{\sqrt{5}}{2}-\frac{1}{2}\right)^n\left(\frac{\sqrt{5}}{2}+\frac{1}{2}\right)}{10} \\ \frac{\sqrt{5}\left(\left(-\frac{\sqrt{5}}{2}-\frac{1}{2}\right)^n-\left(\frac{1}{2}-\frac{\sqrt{5}}{2}\right)^n-\left(\frac{\sqrt{5}}{2}-\frac{1}{2}\right)^n+\left(\frac{\sqrt{5}}{2}+\frac{1}{2}\right)^n\right)}{10} \end{pmatrix}$$

So now we have closed-form expressions for the number of each hanoi transition at each level. We can calculate the number of calls to moveone() in the following manner:

1. Notice that we can represent the running of the program as a recursion tree, much like in problem one, with the height of the tree equal to $n-1$. To get the number of calls to moveone(), we simply sum up the number of calls to $N_{01,l}$, $N_{02,l}$, $N_{10,l}$, and $N_{12,l}$. Like this:

$$\text{calls to moveone} = \sum_{ab \in \{01,02,10,12\}} \left(\sum_{l=0}^{n-1} N_{ab,l} \right)$$

2. Now that we have the summation, group like terms of constants that are to the power of l in their own summation term
3. Once we have these summation terms, we can use the property of the geometric series to find their closed-form sum.

$$\sum_{l=0}^{n-1} ar^l = a \left(\frac{1 - r^{n-1}}{1 - r} \right)$$

To develop a closed form expression for the number of calls to moveall(), we need to sum up the values of $N_{20,l}$ and $N_{21,l}$ across all levels from $l = 0$ to $l = n - 1$. Because we know what hanoi calls recursively call $(2 \to 0)$ and $2 \to 1$, we can do the summation as follows:

1. The number of calls to hanoi() with pole 2 as a source per level is equal to the number of $0 \to 1$ and $1 \to 0$ calls in the previous level. Therefore, we can sum them as follows:

$$\sum_{ab \in \{01,10\}} \left(\sum_{l=0}^{n-1} N_{ab,l-1} \right)$$

2. Again, group like terms that are to the power of l in their own summation term.
3. And finally, develop a closed form expression using the power series again.

We have now developed a closed form expression for the number of calls to moveone() and moveall() for a hanoi tower of size n. Or more precisely, we have shown exactly how to calculate these closed form expressions (and hopefully that is worth some partial credit).

Problem 3

(a) Suppose you have a string of n Christmas lights, numbered 1 ,..., n that are wired in series. One of the lights is broken and you want to find out which. You have a multimeter that you can use to test whether any section of the string works. I.e., test(i, j) returns True if lights i through j (inclusive) are all working, and False if one of them is broken. Design a recursive algorithm to identify the broken light (you should assume there is exactly one) and analyze its runtime. For full credit your algorithm should make a sublinear number of calls to test (i.e., o(n))

The solution to this problem is an algorithm that is extremely similar to binary search. To add some generality, I will present my algorithm which is able to return the index of any and all broken lightbulbs, regardless of the number that are broken. *My algorithms use python notation.*

```
def test(bulbs):
    if False in bulbs:
        return False
    else:
        return True

def FindBrokenBulbs(bulbs, l, r):
    # Base cases - check one individual light
    if (r - l) == 1:
        # We have found the singular bulb that is bad
        return [l]
    else:
        bbulbs = []
        m = (l + r) // 2
        #Check if there's bad bulbs in the left subhalf
        if not test(bulbs[l:m]):
            bbulbs = bbulbs + FindBrokenBulbs(bulbs, l, m)
        #Check if there's bad bulbs in the right subhalf
        if not test(bulbs[m:r]):
            bbulbs = bbulbs + FindBrokenBulbs(bulbs, m, r)
        #return the array of indices of bad bulbs recursively
        return bbulbs
```

The best runtime case for the above algorithm is whenever exactly one bulb is broken. In that case, the recurrence that defines the algorithm is given by $T(n) = T(n/2) + 2$ for a string of lights of size n. This is because exactly one half of the string of lights is chosen at one time, and each time 2 calls to test() are performed.

So for every recursive call to FindBrokenBulbs() 2 calls to test are performed. But how many recursive calls are performed? Each time the function calls itself for a value of $n/2$, which means that there can be at most $\log_2 n$ calls. So, overall the runtime is $2\log_2 n$.

$$T(n) = 2\log_2 n$$

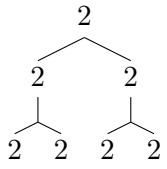
$$T(n) \in \Theta(\log n)$$

And since $\log_2 n < n$, we have now found a good solution.

(b) Suppose now that up to k lights may be broken. Modify your algorithm to find all the broken lights. How big can k be before your algorithm is no longer faster than testing each light?

Because the algorithm given in part A is completely general, no modifications need to be made. If we check each bulb individually to see which ones are bad, our runtime is $\Theta(n)$ because we must check each n bulbs one-by-one.

The worst runtime case for my given algorithm is whenever every single bulb is broken. In that case, the recurrence relation for the algorithm is $T(n) = 2T(n/2) + 2$. We can draw the tree as follows:



...

If we denote the top level of the tree as level $l = 0$, we then find that the number of calls to test() per level is equal to 2^{l+1}. Once again, the height of the tree is $\log_2 n$, except the last level of the tree calls no test(). So the real number of levels we need to sum is $l = 0$ to $l = \log_2 n - 1$ for a string of lights of size n. Therefore, the calls to test() is equal to:

$$\begin{aligned} T(n) &= 2 \sum_{l=0}^{\log_2 n - 1} 2^l \\ &= 2(2^{\log_2 n + 1 - 1} - 1) && \text{geometric series of 2} \\ &= 2(2 * 2^{\log_2 n}) && \text{pull out another 2} \\ &= 2(n-1) && \text{log cancels out} \\ &= 2n - 2 && \text{simplify} \end{aligned}$$

One thing to note about the above is that whenever every single light is broken, you end up checking every single light anyway. Suppose we have an even numbered string of lights with one half being good and the other half being bad. By the above derivation we can see that the number of calls to test() is equal to $2 * (n/2) - 2 + 2 = n$. The "+2" comes from the 2 calls to test on the first level.

Notice what happens when we add a broken light to the the opposite half. Let the left half be the side with one broken light, and the right half be the side with all broken lights (the exact positioning doesn't actually matter, it just makes it easier to explain). Then the number of calls to test is equal to 2 + (left side calls) + (right side calls) $= 2 + \log_2 n + n - 2 = \log_2 n + n$ which is obviously greater than n.

Therefore, the absolute maximum number of broken bulbs where my algorithm is faster is n/2 for a string of length n (and even then under the condition all the broken bulbs are on one side).

Maximum number of bulbs = n / 2 (under specific conditions)

(c) Design a recursive algorithm that finds a pair of keys with a shared factor in your collection of t keys and analyze it's runtime. Your algorithm may assume there is exactly one such pair. For full credit your algorithm should make $o(t^2)$ calls to batchgcd, which you can assume takes constant time.

The following algorithm works in a fairly intuitive way: first we find 2 "blocks" of keys where each one contains a bad key. Once we have 2 blocks, with each one containing a single key we wish to find, we recursively break up the blocks until we find the 2 desired keys (i.e. we end up with blocks of size 1).

Note: the algorithm uses python notation and was also written where batchgcd uses an extra argument of the set of keys, i.e. batchgcd(keys, i, j, k, l)

```
# keys = set of keys to test
# ll = i
# lr = j
# rl = k
# rr = l
def FindKeys(keys, ll, lr, rl, rr):
    # Base case - whenever we're looking at
    # only 2 single elements
    if (ll == lr) and (rl == rr):
        # Check batchgcd to see if the two keys
        # we're looking at are the bad keys
        if batchgcd(keys, ll, lr, rl, rr):
            return [ll, rl]
        else:
            return []

    else:
        #find midpoints
        lm = (ll + lr) // 2
        rm = (rl + rr) // 2

        # Begin by checking if the 2 "blocks" we're
        # looking at contain the bad keys
        if batchgcd(keys, ll, lr, rl, rr):
            # Break up the blocks into smaller pieces
            # recursively until we find the 2
            # keys
            if batchgcd(keys, ll, lm, rl, rm):
                return FindKeys(keys, ll, lm, rl, rm)
            elif batchgcd(keys, ll, lm, rm+1, rr):
                return FindKeys(keys, ll, lm, rm+1, rr)
            elif batchgcd(keys, lm+1, lr, rl, rm):
                return FindKeys(keys, lm+1, lr, rl, rm)
            else:
                return FindKeys(keys, lm, lr, rm+1, rr)

        # if the 2 blocks don't contain the bad keys, then
        # search the left and right halves until we find
        # 2 seperate blocks that contain the bad keys.
```

```
else:
    bad = []
    # check the left half
    bad = bad + FindKeys(keys, ll, lm, lm+1, lr)
    # if it's not in the left half, check
    # the right half
    if bad == []:
        bad = bad + FindKeys(keys, rl, rm, rm+1, rr)
    return bad
```

For every recursive call the number of keys we're looking at is halved. Therefore, the number of recursive calls to FindKeys() will equal $\log_2 n$ for a set of keys of size n. Knowing this, the worst case runtime is where we call batchgcd 3 times every time we recurse (i.e. the case where the the set of keys have the bad keys in the left and right halves initially). We can find the worst-case runtime analysis as follows:

$$\begin{array}{lll} & \text{Number of calls to FindKeys()} & = \log_2 n \\ & \text{Number of calls to Batchgcd() per FindKeys()} & = 3 \\ \implies & \text{Calls to Batchgcd()} & = 3\log_2 n \end{array}$$

This algorithm is $o(t^2)$ (little-o notation) because the minimum number of input keys can be 2 (otherwise using this algorithm makes no sense) and $3\log_2 2 = 3 < 2^2 = 4$. And because $3\log_2 n < n^2 \ \forall \ n >= 2$ (this solution uses the smallest constant multiplier of n^2 so we know we're justified in the little-o notation), we know that we have found a good solution.