CS/ECE 374 B Fall 2019
Homework 4 Problem 1

Siyuan Chen (siyuanc2)
Megan Walsh (meganew2)
Jiashuo Tong (jtong8)

Solution:

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

Unroll 1: $A(n) = A(n-1) + 2n - 1$
Unroll 2: $A(n) = A(n-2) + 4n - 4$
Unroll 3: $A(n) = A(n-3) + 6n - 9$
Unroll 4: $A(n) = A(n-4) + 8n - 16$

So, unrolling k times gives $A(n) = A(n-k) + 2kn - k^2$.

Prove by induction:

Base case $k = 1$:

$A(n) = A(n-1) + 2n - 1$ ✓

Inductive case $k = k-1$:

$A(n) = A(n-(k-1)) + 2(k-1)n - (k-1)^2$
$A(n) = (A(n-k-1+1) + 2(n-k+1)-1) + 2(k-1)n - (k-1)^2$
$A(n) = A(n-k) + 2n - 2k + 2 - 1 + 2kn - 2n - k^2 + 2k - 1$
$A(n) = A(n-k) + 2kn - k^2$ ✓

Unrolling n times and applying the given initial value,

$A(n) = A(0) + 2nn - n^2$
$A(n) = 2n^2 - n^2$
$\boxed{A(n) = n^2}$

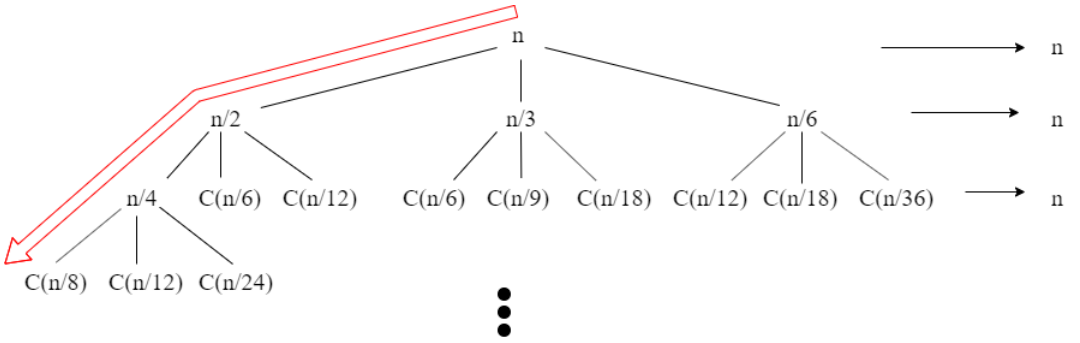
(b) $B(n) = B(n-1) + \binom{n}{2}$; $B(0) = 0$

$B(n) = B(n-1) + \binom{n-1}{2} + \binom{n-1}{1}$
$B(n) = B(n-1) + \binom{n-2}{2} + \binom{n-2}{1} + \binom{n-1}{1}$
$B(n) = B(n-1) + \binom{2}{2} + \binom{2}{1} + \binom{3}{1} + \ldots + \binom{n-1}{1}$
$B(n) = B(n-1) + \sum_{k=1}^{n-1} k$
$B(n) = B(n-1) + \frac{n}{2}(n-1)$
$B(n) = \sum_{k=1}^{n} \frac{k}{2}(k-1)$
$B(n) = \frac{1}{2}(\sum_{k=1}^{n} k^2 - \sum_{k=1}^{n} k)$
$B(n) = \frac{1}{2}(\frac{n(n+1)(2n+1)}{6} - \frac{n(n+1)}{2})$

$\boxed{B(n) = \frac{(n-1)(n)(n+1)}{6}}$

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

The asymptotic solution for this recurrence can be obtained by drawing out the corresponding recursion tree.



In the above recursion tree, the longest path in this recurrence is indicated by the red arrow and the sum of each level is noted on the right side of the tree.
Along that longest path and where i represents the corresponding level in the tree, we have:

$$n \rightarrow n/2 \rightarrow n/4 \rightarrow n/8 \rightarrow \ldots \rightarrow 1$$
$$\frac{n}{2^i} = 1$$
$$n = 2^i$$

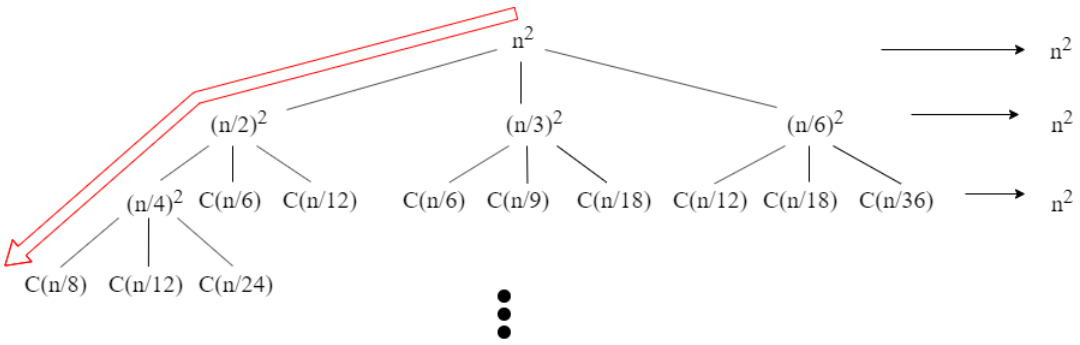
Solving for the cost of the problem at level $i = n$, we get:

$$C(n) = n\frac{lg(n}{lg(2)} = nlgn$$

Therefore, the asymptotic solution for this recurrence is $\boxed{O(nlgn)}$.

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

The asymptotic solution for this recurrence can be obtained by drawing out the corresponding recursion tree.



In the above recursion tree, the longest path in this recurrence is indicated by the red arrow and the sum of each level is noted on the right side of the tree.
Along that longest path and where i represents the corresponding level in the tree, we have:

$n^2 \to (n/2)^2 \to (n/4)^2 \to (n/8)^2 \to \ldots \to 1$
$(\frac{n}{2^i})^2 = 1$
$n = 2^i$

Solving for the cost of the problem at level $i = n$, we get:

$$C(n) = n^2 \frac{lg(n}{lg(2)} = n^2 lgn$$

Therefore, the asymptotic solution for this recurrence is $\boxed{O(n^2 lgn)}$.

■

CS/ECE 374 B Fall 2019
Homework 4 Problem 2

Siyuan Chen (siyuanc2)
Megan Walsh (meganew2)
Jiashuo Tong (jtong8)

Solution:

(a) The algorithm for moving hanoi disks under the new restriction is described below:

```
# hw4p2a.py
def moveone(ndisks, src, dst):
    """ Move the disk numbered 'ndisks' from src to dst, without
        violating any rules of hanoi. """
    global ref, cnt
    print("Disk %i: %i -> %i" % (ndisks, src, dst))
    if len(ref[src]) == 0 or ref[src][-1] != ndisks:
        raise ValueError('Requested disk is not on top of source pole')
    elif len(ref[dst]) != 0 and ref[dst][-1] < ndisks:
        raise ValueError('Smaller disks are already on dest pole')
    else:
        ref[dst].append(ref[src].pop())
        cnt += 1

def canmove(ndisks, src, dst):
    """ Determine whether ndisks can be directly moved from src to dst.
        """
    global ref
    if len(ref[src]) == 0 or ref[src][-1] != ndisks:
        return False
    elif len(ref[dst]) != 0 and ref[dst][-1] < ndisks:
        return False
    else:
        return True

def hanoi(ndisks, src, dst, tmp, master=False):
    global directmove, indirectmove
    """ 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(ndisks-1, src, tmp, dst, True)
        # move one disk from source to destination
        if master: # Keep track of moves without the restriction set in
            p2a
            print("Master Disk %i: %i -> %i" % (ndisks, src, dst))
        if (src != 0) and (dst != 0):
            # If neither src nor dst are 0, calculate new moves that start
                or ends in 0
            if canmove(ndisks, src, 0):
                moveone(ndisks, src, 0)
                moveone(ndisks, 0, dst)
                directmove += 1 # Counter to track number of case 1 moves
            else:
```

```
41              hanoi(ndisks-1, 0, dst, src) # recursively move stack of
                    n-1 disks from 0 to dst tower
42              moveone(ndisks, src, 0)
43              hanoi(ndisks-1, dst, 0, src) # recursively move stack of
                    n-1 disks from dst to 0 tower
44              hanoi(ndisks-1, 0, src, dst) # recursively move stack of
                    n-1 disks from 0 to src tower
45              moveone(ndisks, 0, dst)
46              hanoi(ndisks-1, src, 0, dst) # recursively move stack of
                    n-1 disks from src to 0 tower
47              indirectmove += 1 # Counter to track number of case 2 moves
48          else:
49              # If either src or dst is 0, move as usual
50              moveone(ndisks, src, dst)
51          if master:
52              print(ref)
53          # recursively move stack of n-1 disks to dest tower
54          hanoi(ndisks-1, tmp, dst, src, True)
55      else:
56          pass # do nothing
57
58  if __name__ == "__main__":
59      # Define initial condition of the 3 poles and disks
60      ref = {0: [8, 7, 6, 5, 4, 3, 2, 1], 1: [], 2: []}
61      cnt = 0
62      directmove = 0
63      indirectmove = 0
64      hanoi(6, 0, 1, 2, True)
```

If moveone had a restriction that either the source or the destination needs to be tower 0, then the original Hanoi Tower problem can be divided into 2 subcases:

Subcase 1: Neither the source nor the destination is tower 0, that is, we want to move n disks either from tower 1 to tower 2, or from tower 2 to tower 1.

Subcase 2: Either the source or the destination is tower 0, that is, we want to move n disks from tower 1 to tower 0, from tower 2 to tower 0, from tower 0 to tower 1, or from tower 0 to tower 2.

Suppose the number of calls to function moveone for Subcase 1 is $A(n)$, and the number of calls to function moveone for Subcase 2 is $B(n)$. We obtain the recurrence relations between $A(n)$ and $B(n)$ from the following analysis.

Analysis of Subcase 1:

Suppose we want to move n disks from tower 1 to tower 2. Since moveone must take tower 0 as either the source or the destination, we need to conduct the following subroutines to avoid breaking this rule for using moveone:

1. Move n-1 disks from tower 1 to tower 2 (completed with $A(n-1)$ calls to moveone)

2. Move the nth disk from tower 1 to tower 0 by calling moveone (completed with 1 call to moveone)

3. Move n-1 disks from tower 2 to tower 1 (completed with $A(n-1)$ calls to moveone)

4. Move the nth disk from tower 0 to tower 2 (completed with 1 call to moveone)

5. Move n-1 disk from tower 1 to tower 2 (completed with $A(n-1)$ calls to moveone)
Summarizing the function calls to moveone in the above steps yields the following recurrence relation:

$$A(n) = 3A(n-1) + 2 \tag{1}$$

with the first term being $A(1) = 2$.

Analysis of Subcase 2:

Suppose we want to move n disks from tower 0 to tower 1. We can apply the regular subroutines as listed below to complete the task:

1. Move n-1 disks from tower 0 to tower 2 (completed with A(n-1) calls to moveone)

2. Move the nth disk from tower 0 to tower 1 (completed with 1 call to moveone)

3. Move n-1 disks from tower 2 to tower 1 (completedd with B(n-1) calls to moveone)
Summarizing the function calls to moveone in the above steps yields the following recurrence relation:

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

with the first term being $B(1) = 1$.

In order to know how many calls to moveone are needed to move n disks from tower 0 to tower 1, we should solve recurrence relation (2) for $B(n)$. Since there is an unkonwn term $A(n-1)$ in (2), we need to first solve recurrence relation (1) for $A(n)$.

Solve for A(n):

Notice that Eqn. (1) can be rewritten as

$$A(n) + 1 = 3\big(A(n-1) + 1\big)$$

which implies that $A(n) + 1$ is a geometric sequence with a common ratio of 3 and the first term of $A(1) + 1 = 3$. Thus, the general formula for this geometric sequence is given by

$$\begin{aligned} A(n) + 1 &= \big(A(1) + 1\big) \times 3^{n-1} \\ &= 3 \times 3^{n-1} \\ &= 3^n \end{aligned}$$

Therefore, we have the following general formula for $A(n)$

$$A(n) = 3^n - 1$$

Solve for B(n):

Substituting $A(n-1)$ with $3^{n-1} - 1$ in Eqn. (2) gives

$$\begin{aligned} B(n) &= (3^{n-1} - 1) + 1 + B(n-1) \\ &= 3^{n-1} + B(n-1) \end{aligned} \tag{3}$$

We unroll Eqn. (3) and obtain the general formula for $B(n)$ as shown below

$$\begin{aligned} B(n) &= 3^{n-1} + B(n-1) \\ &= 3^{n-1} + 3^{n-2} + B(n-2) \\ &= 3^{n-1} + 3^{n-2} + \ldots + 3^1 + B(1) \\ &= 3^{n-1} + 3^{n-2} + \ldots + 3^1 + 3^0 \\ &= 3^0 \times \frac{1-3^n}{1-3} \\ &= \frac{3^n - 1}{2} \end{aligned}$$

Conclusion

For Problem 2(a), We conlude that for the standard hanoi problem (move n disks from tower 0 to tower 1), we need a total of $\boxed{\frac{3^n-1}{2}}$ function calls to moveone.

(b) The algorithm for moving hanoi disks utilizing *moveall* where possible is described below:

```
# hw4p2b.py
# Import packages
import random

'''
2(b) Hanoi Tower Modified Problem:
Suppose that we can take benefit of a function moveall
that moves all disks from one tower to another
'''
class Tower:
   def __init__(self, index, num_disks = 0):
      self.index = index
      self.num_disks = num_disks

def hanoi(ndisks,source,dest,temp):
   if (source.index!=2 and dest.index!=2):
      if (ndisks >= 0):
         hanoi(ndisks-1,source,temp,dest)
         moveone(source,dest)
         moveall(ndisks,temp,dest)
   if (source.index!=2 and dest.index==2):
      if (ndisks >= 0):
         hanoi(ndisks-1,source,temp,dest)
         moveone(source,dest)
         hanoi(ndisks-1,temp,dest,source)
   if (source.index==2 and dest.index!=2):
      if (ndisks >= 0):
         moveall(ndisks,source,dest)

def moveone(source,dest):
   global count_moveone
   source.num_disks -= 1
   dest.num_disks += 1
```

```
34      count_moveone += 1
35
36  def moveall(ndisks,source,dest):
37      global count_moveall
38      source.num_disks -= ndisks
39      dest.num_disks += ndisks
40      count_moveall += 1
41
42  def print_count():
43      global count_moveone
44      global count_moveall
45      print("Total number of calls to moveone is ", count_moveone)
46      print("Total number of calls to moveall is ", count_moveall)
47
48  if __name__ == "__main__":
49      '''
50      Test Code for 2(b)
51      '''
52      n = 9
53      src = Tower(0,n)
54      dst = Tower(1)
55      tmp = Tower(2)
56
57      count_moveone = 0
58      count_moveall = 0
59
60      hanoi(n,src,dst,tmp)
61      print_count()
62
63      def fabonacci(n):
64          if (n==0 or n==1):
65              return 1
66          return fabonacci(n-1)+fabonacci(n-2)
67
68      print("The correct answer should be", fabonacci(n+2)-1, "and",
            fabonacci(n))
```

If we want to take advantage of moveall which can move an entire stack of disks from tower 2, then the original Hanoi Tower problem can be divided into 3 subcases:

Subcase 1: Tower 2 is neither the source nor the destination, that is, we want to move n disks from 0 to 1 or from 1 to 0. **(This includes the standard hanoi problem, where all disks are moved from pole 0 to pole 1)**

Subcase 2: Tower 2 is the destination, that is, we want to move n disks from 0 to 2 or from 1 to 2.

Subcase 3: Tower 2 is the source, that is, we want to move n disks from 2 to 0 or from 2 to 1.

Notice that **Subcase 3** can be completed using one function call to moveall and thus is a trivial case. We focus on the analysis for **Subcase 1** and **Subcase 2** in the following discussion.

Suppose the number of calls to function moveone and to function moveall for **Subcase 1** are $A(n)$ and $B(n)$, respectively, and the number of calls to function moveone and to function moveall for **Subcase 2** are $C(n)$ and $D(n)$, respectively. We obtain the recurrence

relations between $A(n)$ and $C(n)$ and the recurrence relations between $B(n)$ and $D(n)$ from the following analysis.

Analysis of Subcase 1:

Suppose we want to move n disks from tower 0 to tower 1. We need to conduct the following subroutines which incorporate the use of moveall:

1. Move n-1 disks from tower 0 to tower 2 (completed with $C(n-1)$ calls to moveone, and $D(n-1)$ calls to moveall)

2. Move the nth disk from tower 0 to tower 1 by calling moveone (completed with 1 call to moveone)

3. Move n-1 disks from tower 2 to tower 1 (completed with 1 call to moveall) Summarizing the function calls to moveone and moveall in the above steps yields the following recurrence relation:

$$A(n) = C(n-1) + 1 \tag{4}$$

with the first term being $A(1) = 1$.

$$B(n) = D(n-1) + 1 \tag{5}$$

with the first term being $B(1) = 1$. **Here, we assume that the call with input n=0 is a valid call to function moveall**.

Analysis of Subcase 2:

Suppose we want to move n disks from tower 0 to tower 2. We can apply the regular subroutines as listed below to complete the task:

1. Move n-1 disks from tower 0 to tower 1 (completed with $A(n-1)$ calls to moveone, and $B(n-1)$ calls to moveall)

2. Move the nth disk from tower 0 to tower 2 (completed with 1 call to moveone)

3. Move n-1 disks from tower 1 to tower 2 (completed with $C(n-1)$ calls to moveone, and $D(n-1)$ calls to moveall) Summarizing the function calls to moveone and moveall in the above steps yields the following recurrence relations:

$$C(n) = A(n-1) + 1 + C(n-1) \tag{6}$$

with the first two terms being $C(0) = 0$, and $C(1) = 1$.

$$D(n) = B(n-1) + D(n-1) \tag{7}$$

with the first two terms being $D(0) = 0$, and $D(1) = 0$.

In order to know how many calls to moveone are needed to move n disks from tower 0 to tower 1, we should solve recurrence relations (4) and (6) for $A(n)$. In order to know how many calls to moveall are needed to move n disks from tower 0 to tower 1, we should solve recurrence relations (5) and (7) for $B(n)$.

Solve for A(n):

Substituting Eqn. (4) in Eqn.(6), we get

$$\begin{aligned} C(n) &= \big(C(n-2) + 1\big) + 1 + C(n-1) \\ &= C(n-1) + C(n-2) + 2 \end{aligned}$$

Rearranging the above equation, we get

$$C(n)+2=\big(C(n-1)+2\big)+\big(C(n-2)+2\big)$$

with $C(0)+2=2$ and $C(1)+2=3$. Recognizing that $\big(C(n)+2\big)$ is a Fibonacci sequence, we apply the general formula for Fibonacci sequences and get

$$C(n)+2=\frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^{n+3}-\left(\frac{1-\sqrt{5}}{2}\right)^{n+3}\right]$$

Thus, the general formula for $C(n)$ is

$$C(n)=\frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^{n+3}-\left(\frac{1-\sqrt{5}}{2}\right)^{n+3}\right]-2$$

Applying this formula to Eqn. (4), we obtain the general formula for $A(n)$ as follows

$$A(n)=\frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^{n+2}-\left(\frac{1-\sqrt{5}}{2}\right)^{n+2}\right]-1 \quad (n\geq 0)$$

which starts by $A(0)=0, A(1)=1, A(2)=2$.

Solve for B(n):

Substituting Eqn. (5) in Eqn.(7), we get

$$\begin{aligned} D(n) &= \big(D(n-2)+1\big)+D(n-1) \\ &= D(n-1)+D(n-2)+1 \end{aligned}$$

Rearranging the above equation, we get

$$D(n)+1=\big(D(n-1)+1\big)+\big(D(n-2)+1\big)$$

with $D(0)+1=1$ and $D(1)+1=1$. Recognizing that $\big(D(n)+1\big)$ is a Fibonacci sequence, we apply the general formula for Fibonacci sequences and get

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

Thus, the general formula for $D(n)$ is

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

Applying this formula to Eqn. (5), we obtain the general formula for $B(n)$ as follows

$$B(n)=\frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^{n}-\left(\frac{1-\sqrt{5}}{2}\right)^{n}\right] \quad (n\geq 0)$$

which starts by $B(0)=0, B(1)=1, B(2)=1$.

Conclusion

For Problem 2(b), we conclude that we need

$$A(n)=\frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^{n+2}-\left(\frac{1-\sqrt{5}}{2}\right)^{n+2}\right]-1 \quad (n\geq 0)$$

calls to function moveone, and

$$B(n) = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right] \quad (n \geq 0)$$

calls to function moveall, in order to move n disks from tower 0 to tower 1. ■

CS/ECE 374 B Fall 2019
Homework 4 Problem 3

Siyuan Chen (siyuanc2)
Megan Walsh (meganew2)
Jiashuo Tong (jtong8)

Solution:

(a) The following algorithm recursively finds the location of broken lights x in the range $[i, j]$ by exploiting the $test()$ function.
The algorithm, which is depicted in $testlight(i, j)$ function above, performs binary search to locate the broken light. At each recursion, the algorithm checks if $x \in [i, k]$, where $k = floor(\frac{i+j}{2})$. If x is indeed in $[i, k]$, then the algorithm recurses to find the baroken light from the halved interval of $[i, k]$. Otherwise, the algorithm recurses to search in $[k+1, j]$. The base case is encountered when $i = j$. Because, $x \in [i, j]$, then $x = i$ must be the broken light. The algorithm then simply reports the broken has been found.

```
# hw4p3a.py
import numpy as np

def test(i, j):
    """ Implementing test function as specified in homework problem.
        test(i, j) returns true if all lights i through j (inclusive)
        are all working, and false otherwise. """
    global lights
    # Checks if any of the lights in [i, j] is broken. If so, return
        False. The array is 0-index but problem is 1-index, hence the
        slightly unclear conversion in lights[i-1:j]
    if np.sum(lights[i-1:j]) >= 1:
        return False
    else:
        return True

def testlight(i, j):
    """ Recursively identify the broken light. The function assumes
        broken light is within [i, j] """
    if i==j:
        # Base case
        print('Light %i is broken' % i)
    else:
        k = (j - i)//2 + i # Middle indice between i and j
        if test(i, k):
            # test(i, k) returns True means broken light is in [k+1, j]
            testlight(k+1, j)
        else:
            # test(i, k) returns False means broken light is in [i, k]
            testlight(i, k)

if __name__ == "__main__":
    # Preparing a long array to simulate a bunch of lights with
        exactly one not working (value=1)
    lights = np.zeros((10000,), dtype=int)
    randidx = np.random.randint(0, 10000)
    lights[randidx] = 1
```

```
32     print('Light %s is broken' % (randidx+1))
33     # Actually running the algorithm with 10000 lights
34     testlight(1, 10000)
```

Runtime analysis: This algorithm is standard-fare binary search. Although each recursion of the algorithm $testlight(i, j)$ has two branches, only one branch will be explored, thanks to insight provided by $test(i, j)$. Given that $n = j + 1 - i$, the algorithm obeys the recurrence $T(n) = T(n/2)+1$ and therefore $\boxed{O(logn)}$ calls will be made to $test(i, j)$, which is sublinear.

(b) In order to find all broken lights, the algorithm above was modified such that the two recursive branches can be explored at the same time:

```
1  # hw4p3b.py
2  import numpy as np
3
4  def test(i, j):
5      """ Implementing test function as specified in homework problem.
           test(i, j) returns true if all lights i through j (inclusive)
           are all working, and false otherwise. Also keeps track of how
           many calls to test() have been made. """
6      global lights, cnt
7      cnt += 1
8      if np.sum(lights[i-1:j]) >= 1:
9          return False
10     else:
11         return True
12
13 def testlight(i, j):
14     """ Recursively identify all broken lights in range of [i, j] """
15     if i==j:
16         # Base case
17         print('Light %i is broken' % i)
18     else:
19         k = (j - i)//2 + i # Middle indice between i and j
20         if not test(i, k):
21             # test(i, k) returns False means there are at least one
                   broken light in [i, k]
22             testlight(i, k)
23         if not test(k+1, j):
24             # test(k+1, j) returns False means there are at least one
                   broken light in [k+1, j]
25             testlight(k+1, j)
26
27 if __name__ == "__main__":
28     # Preparing a long array to simulate a bunch of lights with some
           broken ones (value=1)
29     lights = np.zeros((3750,), dtype=int)
30     for i in range(469):
31         lights[8*i] = 1
32     print('%s lights are broken' % (np.sum(lights)))
33     cnt = 0
34     # Actually running the algorithm with 3750 lights
35     testlight(1, 3750)
```

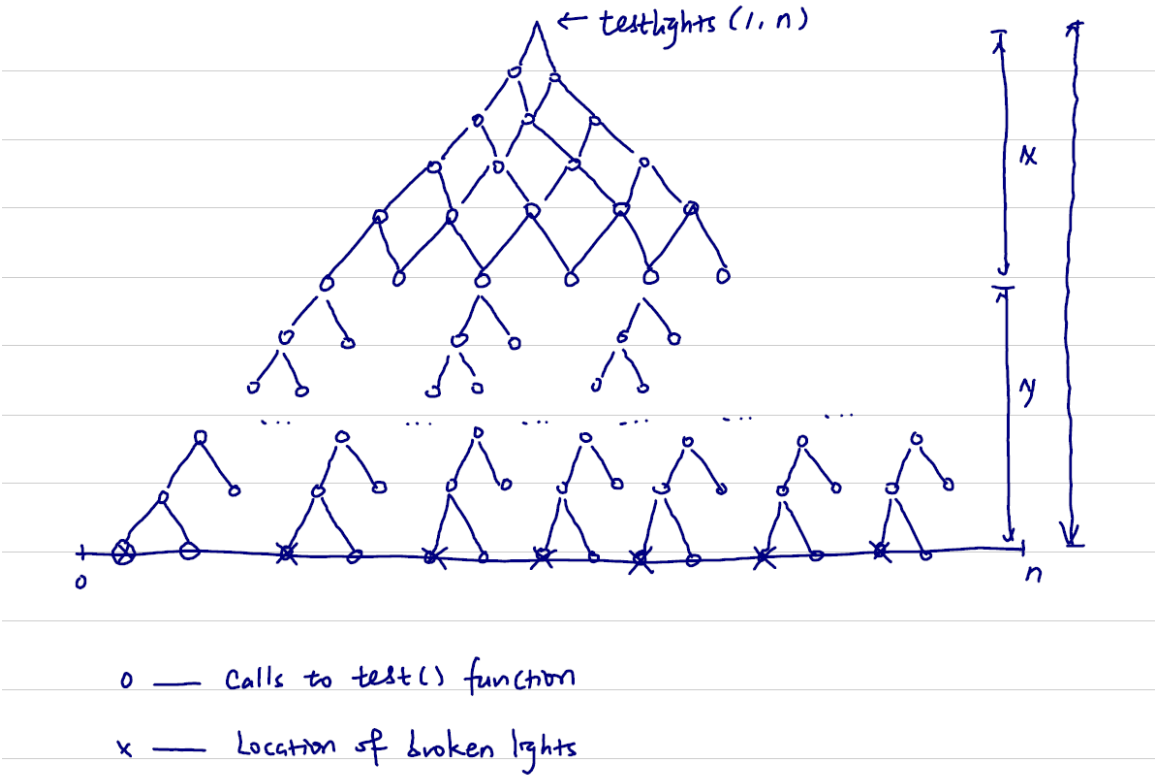
```
36    print('%s calls to test() were made' % cnt)
```

The algorithm now checks if there are broken lights in both left and right halves of the input range. Therefore, in the worst case that every single light is a broken light, the algorithm follows the recurrence $T(n) = 2T(n/2)+2$, with base case $T(2) = 2$ and $T(1) = 0$. We claim that unrolling this recurrence gives $T(n) = 2^{(log_2 n)+1} - 2 = 2n - 2$, which can be proved by induction:

$$\begin{aligned}
T(1) &= 2*1-2 = 0 && \text{[Base case]}\\
T(2) &= 2*2-2 = 2 && \text{[Base case]}\\
T(n) &= 2T(n/2)+2 \\
&= 2(2(n/2)-2)+2 && \text{[Induction hypothesis]}\\
&= 2n-2
\end{aligned}$$

In order to make this algorithm slower than linear search, which takes precisely n calls to the $test()$ function (assuming we don't know how many lights are broken), we can strategically place k broken lights among other $n-k$ lights such that each broken lights are separated as much as possible. A way to estimate the value of k given n is provided below (Note that this is an estimation as some assumptions used in calculation below is not always correct):

The $testlight()$ algorithm described above always run $test()$ on both left and right halves of the given range to decide whether to further explore each path. This can be envisioned as a tree depicted below, with circles indicating calls to $test()$. The total height of the tree is $log_2 n$.



For some initial height x, the tree will be a perfect tree without the root (as no $test()$ was

called at that point). The bottom of this tree should have $2k$ nodes and therefore $2^x = 2k$ and $x = (log_2 k) + 1$. From recurrence discussed above, the total number of calls at this stage is $T_x = T(2k) = 2(2k) - 2 = 4k - 2$.
In the remaining height of the tree y, as all broken lights are separated as much as possible, the algorithm explores the path to the broken lights and no path overlap with each other. During exploration of each path, $2y$ calls to $test()$ were made. Then total number of calls made here is approximately $T_y = 2ky = 2k(log_2 n - x) = 2k(log_2 n - log_2 k - 1)$.
Therefore, the total number of calls made is: $T = T_x + T_y = 4k - 2 + 2k(log_2 n - log_2 k - 1) = 2k(log_2 n - log_2 k + 1) - 2$. Solving for $T \geq n$ for $n = 100, 500, 1000, 5000, 10000$ generates the following result and are then verified by code provided above:
From the tabulated results, it seems that when $\boxed{k \geq \frac{n}{8}}$ broken lights are strategically placed

n	min value of k	k used for test	actual calls made
100	12.9	12	96
500	62.9	62	496
1000	125.4	125	998
5000	625.4	625	4998
10000	1250.4	1250	9998

in a line of n lights, the algorithm may not be faster than brutal force.

(c) The algorithm presented below recursively finds 0-based index of pair of keys, noted below as x and y, that shares factors from the range of $[0, t)$. The algorithm is roughly divided into 3 stages:

Stage 1: `findbp_(a, bp, b)` Given a range $[a, b)$ that contains both keys, the algorithm first recursively find a breakpoint bp such that $x \in [a, bp)$ and $y \in [bp, b)$ in a binary search fashion: $bp = floor((a + b)/2)$ and $batchgcd(a, bp, bp, b)$ is called exactly once in each recursion. If $batchgcd(a, bp, bp, b)$ returns false, then both x and y are in the same side of bp; the algorithm then explores $[a, bp)$ and will return to explore $[bp, b)$ if it fails to find x and y in the left branch. If $batchgcd(a, bp, bp, b)$ returns true, then the algorithm enters the second stage to find exact value of x in range $[a, bp)$ again using binary search.

Stage 2: `findx_(a, b, c, d)` Once it is known that only one key (here conveniently noted as x) is in $[a, b)$, the algorithm searches for exact location of x using standard-fare binary search algorithm. Upon finding x, the algorithm then proceeds to stage 3 and finds exact location of y from range $[c, d)$.

Stage 3: `findy_(a, b, c, d)` Once it is known that only one key (here conveniently noted as y) is in $[c, d)$, and the other key x is located at a, the algorithm searches for exact location of y using standard-fare binary search algorithm. The algorithm exits once y has been found.

```
1  # hw4p3c.py
2  import numpy as np
3
4  def batchgcd(i, j, k, l):
5      """ Implementing the batchgcd function as described in homework
           problem. Also tracks number of calls to batchgcd() made. """
6      global cnt
```

```
7     cnt += 1
8     # batchgcd() returns true if one of the keys, pi, is in [i, j)
          and the other, pj, is in [k, l)
9     if (i <= pi and pi < j) and (k <= pj and pj < l):
10       return True
11    else:
12       return False
13
14 def findmatch(t):
15    """ Main function of the algorithm. Returns true if the said
          pair of keys were found, false otherwise. """
16    if findbp_(0, t//2, t):
17       print('%s calls to batchgcd were made' % cnt)
18       return True
19    else:
20       print('None of the keys in [1, t] shares factor with another
             key')
21       return False
22
23 def findbp_(a, bp, b):
24    """ Given a range of numbers [a, b), find breakpoint bp such that
          x in [a, bp) and y in [bp, b), where x, y are the two keys"""
25    if a < bp and bp < b:
26       if batchgcd(a, bp, bp, b):
27          # x in [a, bp) and y in [bp, b)
28          print('x in [%s, %s) and y in [%s, %s)' % (a, bp, bp, b))
29          # Start to find x from the range [a, bp)
30          return findx_(a, bp, bp, b)
31       else:
32          # both x and y are in either [a, bp) or [bp, b)
33          if findbp_(a, (bp-a)//2+a, bp):
34             return True
35          else:
36             return findbp_(bp, (b-bp)//2+bp, b)
37    else:
38       return False
39
40 def findx_(a, b, c, d):
41    """ Given a, b, c, d such that x in [a, b) and y in [c, d),
42       find exact location of x"""
43    if a+1 < b:
44       m = (b-a)//2+a
45       if batchgcd(a, m, c, d): # x is in [a, m)
46          print('x in [%s, %s)' % (a, m))
47          return findx_(a, m, c, d)
48       else: # x is in [m, b)
49          print('x in [%s, %s)' % (m, b))
50          return findx_(m, b, c, d)
51    else: # Base case
52       print('x = %s' % a)
53   # With exact value of x known, start to find y from range [c, d)
54       return findy_(a, b, c, d)
55
56 def findy_(a, b, c, d):
```

```
57      """ Given a, b, c, d such that x = a = b-1 and y in [c, d),
58          find exact location of y"""
59      if c+1 < d:
60          m = (d-c)//2+c
61          if batchgcd(a, b, c, m): # y is in [c, m)
62              print('y in [%s, %s)' % (c, m))
63              return findy_(a, b, c, m)
64          else: # y is in [m, d)
65              print('y in [%s, %s)' % (m, d))
66              return findy_(a, b, m, d)
67      else: # Base case
68          print('y = %s' % c)
69      # Both x and y have been found, exit
70          return True
71
72  if __name__ == "__main__":
73      # Generate two random index to use as location of the two keys
            that share factors
74      randidx = sorted(np.random.randint(0, 10000, (2,)))
75      print('# %s and # %s share factors' % (randidx[0], randidx[1]))
76      pi = randidx[0]
77      pj = randidx[1]
78      cnt = 0
79      # Run algorithm on 10000 keys
80      findmatch(10000)
```

Runtime analysis: Since this algorithm has 3 different stages, we use T_1, T_2, and T_3 to represent runtime for each of the stage, and total runtime on t keys $T(t) = T_1(t) + T_2(t) + T_3(t)$. There are two extreme cases to consider:

In stage 1, the algorithm first completely explore the left branch before entering the right recursive branch, much like inorder traversal done on a perfect binary tree. A worst case that maximizes runtime for stage 1 is when x and y are the very last two keys (i.e. $t-1$ and $t-2$ in 0-based indices) in the series of t keys, in which case the algorithm must traverse the entire perfect tree before finding bp, which happens to be equal to y. T_1 follows the recurrence $T_1(t) = 2T(t/2) + 1$ with base case $T_1(2) = 1$. This is conceptually equivalent to the number of nodes in a perfect binary tree with t nodes on the bottom layer, which we know from CS 225 is $T(t) = 2t - 1$. However, the algorithm need not to explore the very bottom layer because of the base case, and because there is no point finding a breakpoint in just one number. Then $T_1(t) = 2t - 1 - t = t - 1$ in the **worst** case. However, this happens to make T_2 and T_3 trivial, as here $a = x, b = x + 1 = y, c = y, d = y + 1$, which hits the base case of `findx_(a, b, c, d)` and `findy_(a, b, c, d)` without a single call to `batchgcd()`. Therefore we know that maximum value of $T_1(t)$ is $t - 1$ and minimum value of $T_2(t)$ and $T_3(t)$ is 0.

Another extreme case is that $x \in [0, floor(t/2))$ and $y \in [floor(t/2), t)$, such that stage 1 is completed after exactly 1 call to `batchgcd()`. Here T_2 and T_3 encounters the worst case, and each of which follows the recursion $T_2(t) = T_2(t) + 1$, $T_3(t) = T_3(t) + 1$. From lab 6.5, we know that $T_2(t) = T_3(t) = log_2 floor(t/2)$. Therefore we know that minimum value of $T_1(t)$ is 1 and maximum value of $T_2(t)$ and $T_3(t)$ is $log_2 floor(t/2)$.

From above two cases, we know that $T_1(t) \in [1, n-1]$ and $T_2(t)$, $T_3(t) \in [0, log_2 floor(t/2)]$. Then $T(t)$ cannot be larger than $\max(T_1(t))+\max(T_2(t))+\max(T_3(t)) = n-1+2log_2 floor(t/2)$, which is $O(t)$ and strictly $o(t^2)$.

■