Question 1

a)

Time complexity =n-1

b)

let length of arr = n and assuming n is even

Since comparison is 1 once per call, No. of arrays is number of comparisons

arr	2^0/n * n array x 1 comparison
½ arr + ½ arr	2^1/n * n arrays x 1 comparison
¼ arr + ¼ arr + ¼ arr + ¼ arr	2^2/n * n arrays x 1 comparison
……..	¼ n arrays x 1 comparison
……..	½ n arrays x 1 comparison
Unit 1+1+1+1+1	1/1 n arrays x 0 comparison #return 0 no comparison

Since this recursion is breaking up arr into half all the until having len(arr) number off arr

Recursion of the function will happen len(arr) times

Summing up all comparison $\rightarrow$ 2^0/n * n + 2^1/n * n + 2^2/n * n + ... ½ n

=(1/n) n + (2/n) n + (4/n) n + ... ½ n

= 1+2+4+...... + ½ n $\rightarrow$ (n-1/n*n)

= O(n)

Question 2

a)

For example, let q1 and q2 be the queues used in all cases

Since Stack is Last in First out in nature, and Queue is first in first out in nature. To implement stack using Queues we just need to make sure the last push value is at the Head of the queue.

<u>Function: Push </u>#O (1)

```
q2 = Queue () #empty q2
If q1 is empty
        q1. enqueue(x) #if no element in q, directly enqueue
```

```
else
    q2. enqueue(x)
    for item in q1:
        q2. enqueue (q1. dequeue) #q2 will be in order of Last in First out (Stack nature)
    q1 = q2 #
```

<u>Function: Pop #O (1)</u>

```
Return q1. dequeue ()
```

<u>Function: Peek # O (1)</u>

```
Return q1. peek ()
```

<u>Function: count #O (1)</u>

```
Return q1. count ()
```

b)

Yes, It can be implemented with one queue

```
def pop ():
count = 1
while q.count != count:  # e.g. q = head(1,2) tail
    temp = q.dequeue()  #q = (2)
    q.enqueue(temp) # q= (2,1)
    count += 1
return q.dequeue()  #  return 2
```

Question 3

a)

```
def print_kth_level_leaves(root,k)
declare an empty queue e.g. (q = Queue ())
enqueue root to q
enqueue a None as variable to q #indicate end of level that currently checking
```

```
level = 1 #start off with level 1/height 1

declare an empty list leaf = []

The algorithm starts off by finding all the leaves inside this binary tree

------------------------------

#Start of while loop O(n)

While q is not empty

    Node = q.dequeue ()

    if Node is None

        if q1 is not empty

            enqueue a None as variable to q1 #indicate new level

        level += 1

    else

        If Node. Left is not empty

            enqueue Node. Left to q

        If Node. Right is not empty

            enqueue Node. Right to q

        If node. isLeaf

            Append (node.value,level) to leaf

#End of while loop

------------------------------

#Start of For loop O(n)

For l in leaf #(Look something like [[D,3], [G,3], [F,4]])

    If l [1] == k:

        Print (l [0])

#End print_kth_level_leaves
```

b)

For the while loop I also use Queue to help me as queue having a First in First out nature, it help me to keep track which level/height that I am checking by providing the data in order (FIFO) so that the code will generate the correct node level.

The item inside the queue look like: head (A, None, B, C, None, D, G, E, None, F) Tail

c)

In my algorithm I used a while loop that iterate through all nodes in a binary tree and a for loop to print result. Assuming enqueue, dequeue, count and print etc. is O (1). The time complexity for the while loop is the number of nodes in the binary tree. In worst time complexity

2^0 + 2^1 + 2^2 + ….2^h-1 = n

If height of binary tree is 4 worst case n = 1+2+ 4+2^(4-1) = (2^h) - 1 = 15

Since the for loop is looping through all the leaves, the time complexity for the for loop should be

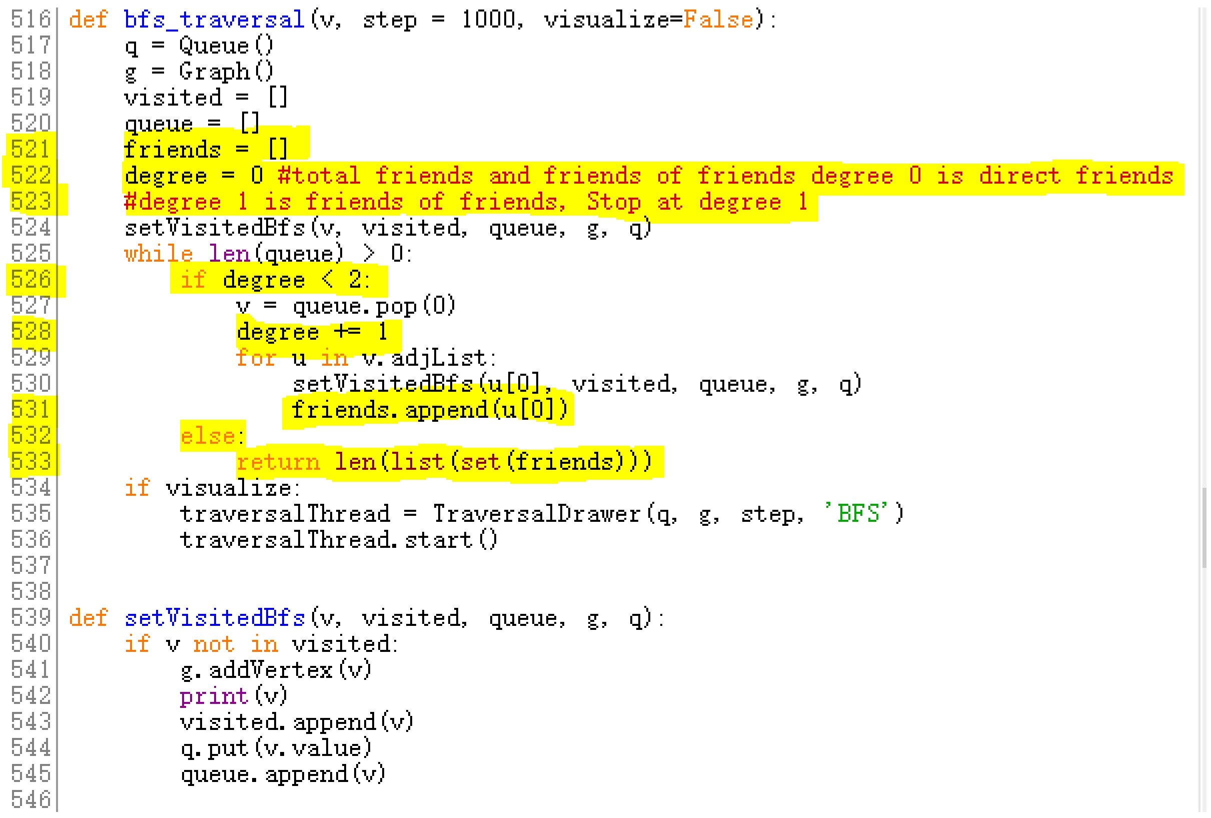
If height of binary tree is 4.

2^(h-1)= 2^(3-1)=2^2 = 4

In general time complexity is (2^h)-1 + 2^(h-1)

Question 4

a)

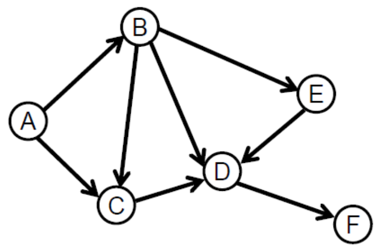


```
def bfs_traversal(v, step = 1000, visualize=False):
    q = Queue()
    g = Graph()
    visited = []
    queue = []
    friends = []
    degree = 0 #total friends and friends of friends degree 0 is direct friends
    #degree 1 is friends of friends, Stop at degree 1
    setVisitedBfs(v, visited, queue, g, q)
    while len(queue) > 0:
        if degree < 2:
            v = queue.pop(0)
            degree += 1
            for u in v.adjList:
                setVisitedBfs(u[0], visited, queue, g, q)
                friends.append(u[0])
        else:
            return len(list(set(friends)))
    if visualize:
        traversalThread = TraversalDrawer(q, g, step, 'BFS')
        traversalThread.start()


def setVisitedBfs(v, visited, queue, g, q):
    if v not in visited:
        g.addVertex(v)
        print(v)
        visited.append(v)
        q.put(v.value)
        queue.append(v)

```

b)

I. The out-degree of the right most node in a topological sort is 0

Example.



II. With the example above the topological order is [A, B, E, C, D, F]. If I add a back edge from C to E it will not create a circle and the topological sort will become [A, B, C, E, D, F].

III. i) n(n-1)/2 ii) n(n-1)