**Question 1:**

**Pseudo code:**

if array[0] in second\_bag: return 0

i = 1

while True:

if array[i] in first\_bag:

bag1\_pointer = i

if array[i+1] in second\_bag:

return i+1

else:

i \*= 2

else: # outcomes[i] in bags[2]

if array[i-1] in first\_bag:

return i

i -= (i - bag1\_pointer)/2

**Complexity Analysis:**

Array =[ ‘Y’, ‘G’, ‘G’, ‘B’, ‘Y’, ‘R’, ‘O’, ‘W’, ‘W’, ‘R’, ‘R’, ‘O’, ‘W’, ‘R’]

**i** will go like this: 1, 2, 4 (switch found (4+1) ) and that is the best case, i.e when the switch happens at N where N is a (power of 2) >>> O(log n)

İf we have one more ball from the first bag (i.e N is bigger by 1)

Array =[ ‘Y’ , ‘Y’, ‘G’, ‘G’, ‘B’, ‘Y’, ‘R’, ‘O’, ‘W’, ‘W’, ‘R’, ‘R’, ‘O’, ‘W’, ‘R’, ‘R’]

**i** will go like this: 1, 2, 4, 8, 6 (switch found 6) and that is the worst case, i.e when the switch happens at N where N is a (power of 2) + 2 >>> O(2 log n)

**worst case: ~ O(2 \* log n)**

There are 23 iterations when N\*2 is 4100

There are 25 iterations when N\*2 is 8196

There are 27 iterations when N\*2 is 16388

**Best case: O(log n)**

There are 12 iterations when N\*2 is 4096

There are 13 iterations when N\*2 is 8192

There are 14 iterations when N\*2 is 16384

**Question 2:**

#Generate the end of the left subsequence O(n)

Left\_end = loop through elements in the array until you find i where a[i+1] < a[i]

#Generate the start of the right subsequence O(n)

Start\_right = loop through elements (from the end to the start) in the array until you find i where a[i-1] > a[i]

# shrink the left subsequence until you find something **smaller** than the **minimum** item in the middle subsequence whose index will be the **starting** index of the middle subsequence.

# shrink the **right** subsequence until you find something **bigger** than the **maximum** item in the middle subsequence whose index will be the **ending** index of the middle subsequence.

The worst case complexity would be then O(2\*n) which is if the array is already sorted so the middle subsequence will have to keep stretching from both sides covering the whole array. However, we can easily eliminate this scenario by checking if the left\_end (the end of the left subsequence) is already at the end of the array. Thus, the worst case complexity is approximately O(n)

**Question 3:**

Processing the input to convert it to a dictionary as follows:

Graph\_table = {Node: [child1, child2], …}

Input graph\_table, start, end

if start == end:

return 1

return crawl(graph\_table, start, end)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Crawl function:**

paths = 0

if start == end:

return 1

for child in children of start:

paths += crawl(graph\_table, start=child, end=end)

The algorithm goes over all the nodes and all the edges just like depth first search but it doesn’t stop once the goal is reached but rather once all edges have been visited, which makes its growing order O(|V|+|E|)