# 1.3.3

a)

this sequence can occur because   
push -> 0 1 2 3 4 [0 1 2 3 4]  
pop -> 4 3 2 1 0 []

push -> 5 6 7 8 9 [5 6 7 8 9]  
pop -> 9 8 7 6 5 []

b)

this sequence can occur because

push -> 0 1 2 3 4 [0 1 2 3 4]  
pop -> 4 [0 1 2 3 ]

push -> 5 6 [0 1 2 3 4 5 6]  
pop -> 6 [0 1 2 3 4 5]

push -> 7 8 [0 1 2 3 4 5 7 8]  
pop -> 8 7 5 3 2 [0 1]

push -> 9 [0 1 9]  
pop -> 9 0 1 []

c)

This sequence is invalid because after popping 2, it attempts to push in 5 before pushing 3 and 4

push -> 0 1 2 [0 1 2]  
pop -> 2 1 [0]

push -> 5 [0 1 2 ?]

d)

This sequence is valid because of the following :

push -> 0 1 2 3 4 [0 1 2 3 4]  
pop -> 4 3 2 1 0 []

push -> 5 [5]  
pop -> 5 []

push -> 6 [6]  
pop -> 6 []

push -> 7 [7]  
pop -> 7 []

push -> 8 [8]  
pop -> 8 []

push -> 9 [9]  
pop -> 9 []

e)

this pattern works observe :

push -> 0 [0]

push -> 1 [0 1]  
pop -> 1 [0]

push -> 2 [0 2]  
pop -> 2 [0]

push -> 3 [0 3]  
pop -> 3 [0]

push -> 4 [0 4]  
pop -> 4 [0]

push -> 5 [0 5]  
pop -> 5 [0]

push -> 6 [0 6]  
pop -> 6 [0]

push -> 7 8 9 [0 7 8 9]  
pop -> 9 8 7 0 []

f)

This sequence is invalid because it attempts to pop 1 before 7 or 2

push -> 0 [0]  
pop -> 0 []

push -> 1 2 3 4 [1 2 3 4]  
pop -> 4 [1 2 3]

push -> 5 6 [1 2 3 5 6]  
pop -> 6 5 3 [1 2]

push -> 7 8 [1 2 7 8]  
pop -> 8 1 [1 2 7 ?]

push -> 7 8 [0 1 2 3 4 5 7 8]  
pop -> 8 7 5 3 2 [0 1]

push -> 9 [0 1 9]  
pop -> 9 0 1 []

g)

This pattern doesn’t work since it attempts to pop out 0 before popping out 2, observe

push -> 0 1 [0 1]  
pop -> 1 [0]

push -> 3 4 [0 2 3 4]  
pop -> 4 [0 2 3]

push -> 5 6 7 [0 2 3 5 6 7]  
pop -> 7 [0 2 3 5 6]

push -> 8 9 [0 2 3 5 6 8 9]  
pop -> 9 8 6 5 3 0 [0 2 ?]

h)

This pattern works, observe

push -> 0 1 2 [0 1 2]  
pop -> 2 1 [0]

push -> 3 4 [0 3 4]  
pop -> 4 3 [0]

push -> 5 6 [0 5 6]  
pop -> 6 5 [0]

push -> 7 8 [0 7 8]  
pop -> 8 7 [0]

push -> 9 [0 9]  
pop -> 9 0 []

# 1.4.5

a)

, because at large values of N, 1 becomes irrelevant

b)

Because using limits,

c)

For the same reason as b)

d)

, because increases significantly faster than at high values, thus at large values of N, the tilde expression removes the smaller powers

e)

Log (2N)/log (N) Log (2) through the basic rules of logarithmic division

f)

Through the same rule of b), limit of 1/N goes to zero, thus N is the only value that stays

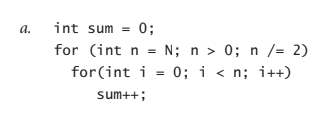
g)

Through L’ Hôpital’s rule, if applied 100 times, it is shown that the limit of the function goes to zero. Thus the tilde expression is

N^100/2^N 0

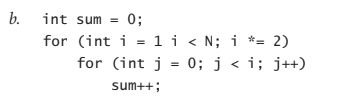
For large values of N

# 1.4.6



Notice how the first for loop starts at n= N and divides by 2 every single time it returns to the loop, the first time n = N, then n= N/2, then n = N/4 and the cycle continues until it is equal to 0. This in turn seems like the first loop is approaching the iteration through a log (N) approach since it’s a continuous binary division.   
  
The second loop begins at I = 0 and loops towards n. The first time it’s N iterations, but on the second portion it becomes N/2 and then N/4 and so on until it is 0. This means that if we did try to add up all the values of the inner loop it becomes (N +N/2 + N/4 +…+1)

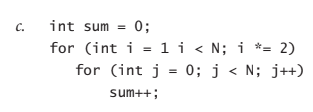
This in turn is equivalent to **O (N)** (because constants are omitted).



This may look a little intimidating but if one looks closely, i starts at one and doubles at each turn until it is greater than N, which means the loop is done a total of log (N) times.

For the second loop, the value starts at zero and loops until it reaches i. Just like the first problem, this can be written as (1 + 2 + 4 + 8 + … + N/2 + N)

This in turn is equivalent to **O (N)** (because constants are omitted).



Using the same logic as the b), the first loop operates log (N) times.

The second inner loop is different in that it starts with 0 and loops until it reaches the actual value of N. This means the loop in proportion to the value of N. so let’s take this value here as N.

This means that the total cost is log (N) times N which is equal to **O (N log (N))**.

# 1.4.20

0 -> count

For integer i starting at zero and i < size of array

For integer j equal to i + 1 and j < size of array

For integer k equal to j+ I and k < size of array

For integer m equal to k + 1 and m < size of array

If a[i] + a[j] + a[k] + a[m] = 0

Then increase count

Else do nothing

\*\*Look down to see drawing equivalent

