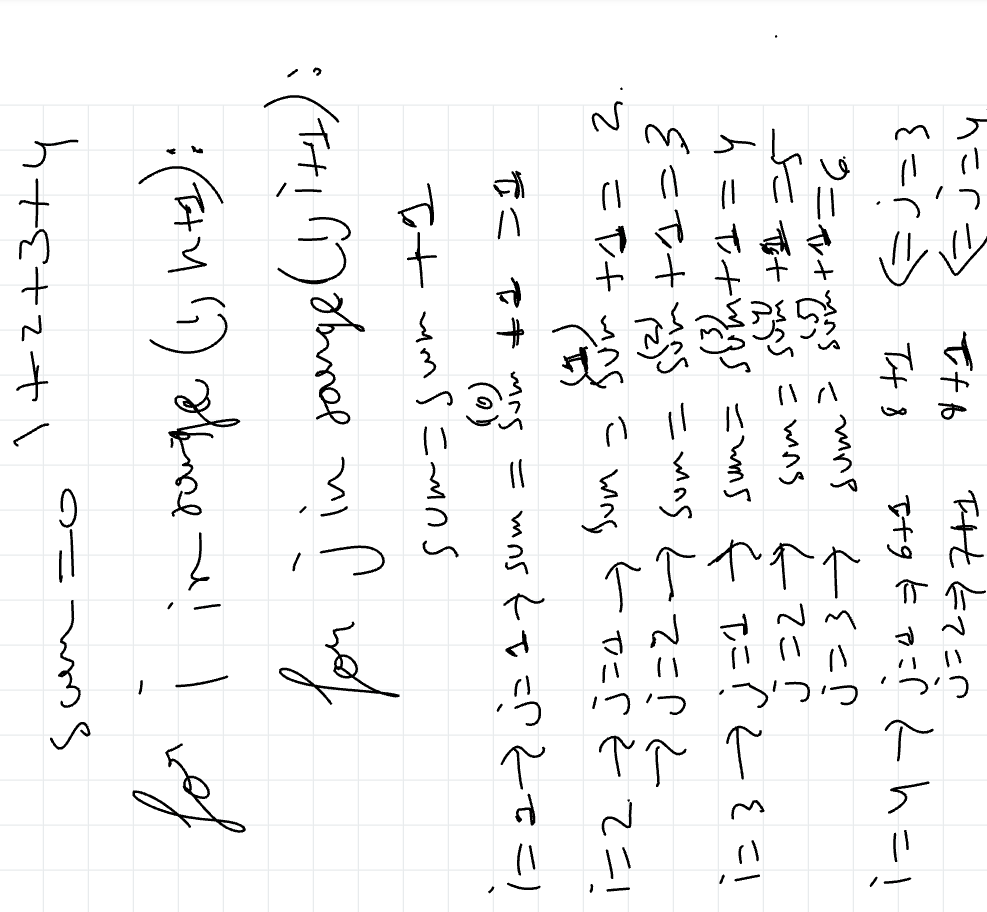
**DSA Practice Problems**

**Analysis of Algorithms:**

**return total** in an interactive environment (like a Python shell, Jupyter notebook, or a script), it will not display the result unless you explicitly print it. So, use print when **calling the FUNCTION**

**** 🡪 Here c4n2+c5n+c6 🡪 1+(1+1)+(1+1+1) 🡪 inner running Constant + is running N\*N times and then N times and then returning sum Constants

def natural\_sum(num):

total=0

for i in range(1,n+1):

sum=sum+i

return total

print(natural\_sum(10))

**Primitive and Non-Primitive understandings..**

**Order of Growth:**

Def funct():

Return n\*(n+1)/2 Here \*,+,/ all are constants.

For every N the program will run 3 constants

Whether you run 100\*100 or 1000\*1000, both having the same constant (\*), so it will take same time

In this 🡪 Programs is running one time, and the functionality only depends on constants(\*,+,/) 🡪 funct() 🡪 c1

Def funct2():

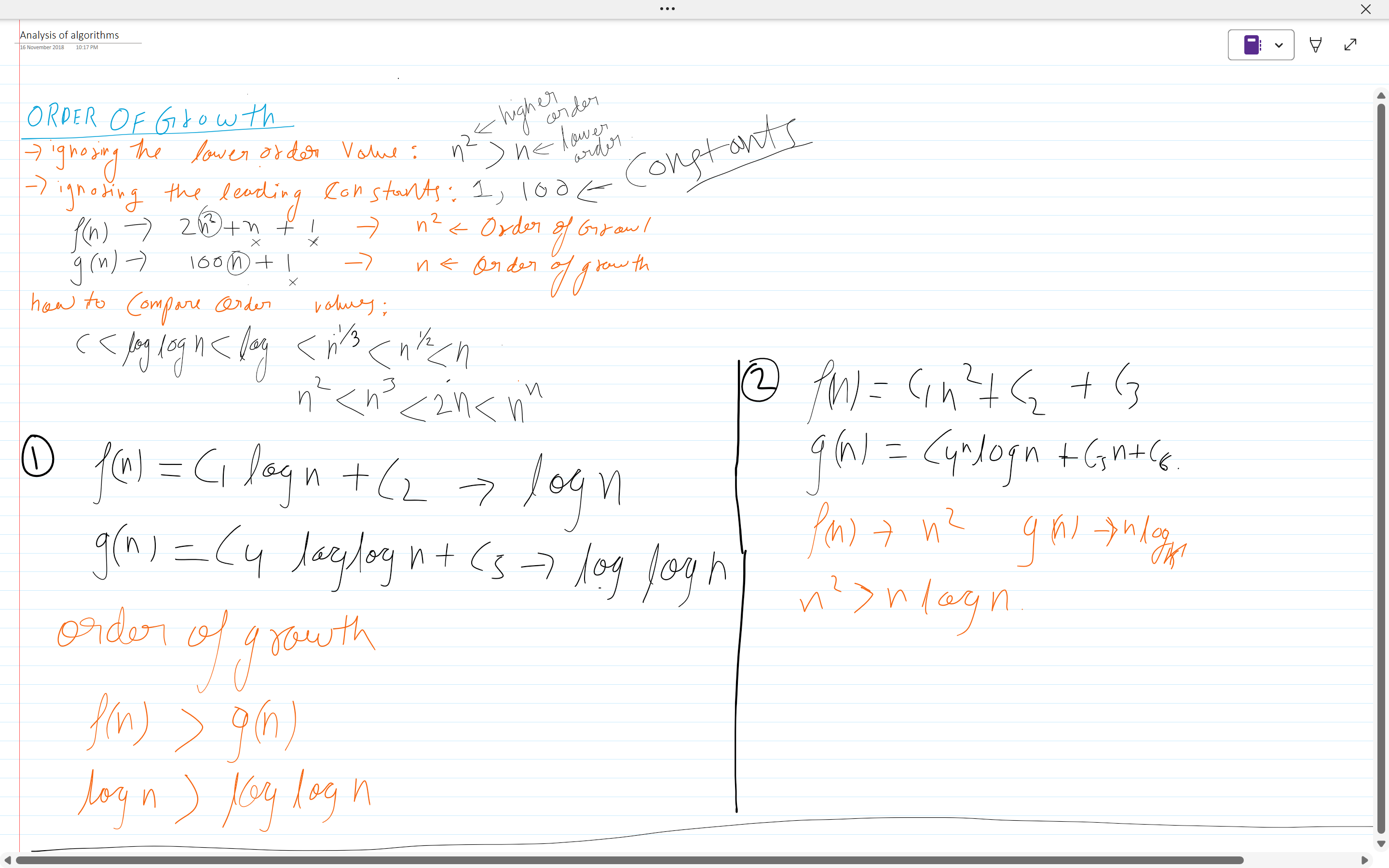
Sum=0 Here funct2 () the loop is running N times,

for i in range(1,n+1): and + Constant is runnign N times 🡪 c2n

Sum=sum+i Also there are constants like: sum=0, returning sum

return sum 🡪 c3 == c2n+c3

**Direct way to find and Compare Order of Growth:** The most time is taken by the last nn

****

**Big O Notation:**

O(n) means Big O of n denotes the Upper bound on an Order of Growth.

**Definition:** We say f(n)=O(g(n)), iff there exists Constant C and nosuch that f(n) <= Cg(n) for all n >= n0

**Example:**

f(n) = 2n+3 written as O(n)

f(n) <= Cg(n) g(n)🡪 O(n)🡪 n

2n+3 <= Cn

**To find C and n0:**

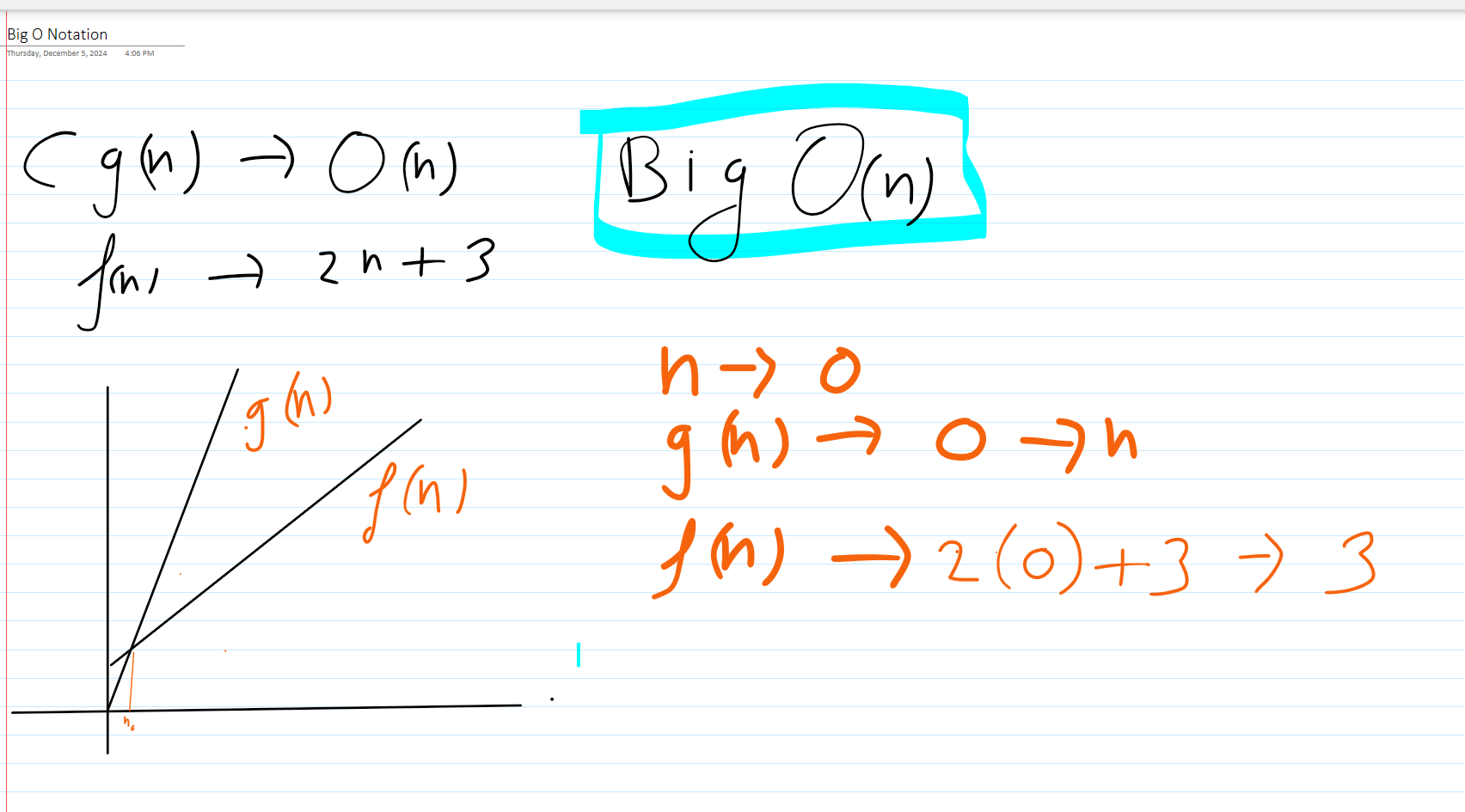
the constant of highest order is 2, the C of Cn should be +1🡪 Cn🡪 3n

2n+3 <= 3n

3 <= n 🡪 n0=3 till n=3 f(n)=g(n)

But for every n > 3, g(n) > f(n)

Pictorial Representation:



O(n) Big O represents order of growth for equal and lesser order of growth:

{ n/4, 2n+3, n/100+log n, n+1000, n/1000, 100, log n+100…. } – O(n)

Where, this includes order of growth that is equal to O(n)🡪 n/4, n/100+log n and O(n) also includes lesser order growth🡪 100, log n+ 100

{n2+n, 2n2, n2+100, …} – O(n2)

{100, 2, 3, …} – O(1)

O(n2) includes🡪 O(n) (as lesser order growth also includes in it.)

Also, O(n) includes🡪 O(1)

**Omega Notation:**

Ω 🡪 Omega this denotes the lower bound on an order of growth

**Definition:** f(n)= Ω(g(n)), iff there exists positive Constants C and n0 such that 0 <= Cg(n) <= f(n) for all n >= n0

**Example:**

Ω(2n+3)🡪 Ω (n)🡪 Cg(n)

f(n)= 2n+3

Cg(n) <= f(n)

Cn <= 2n+3

**To find C and n0:**

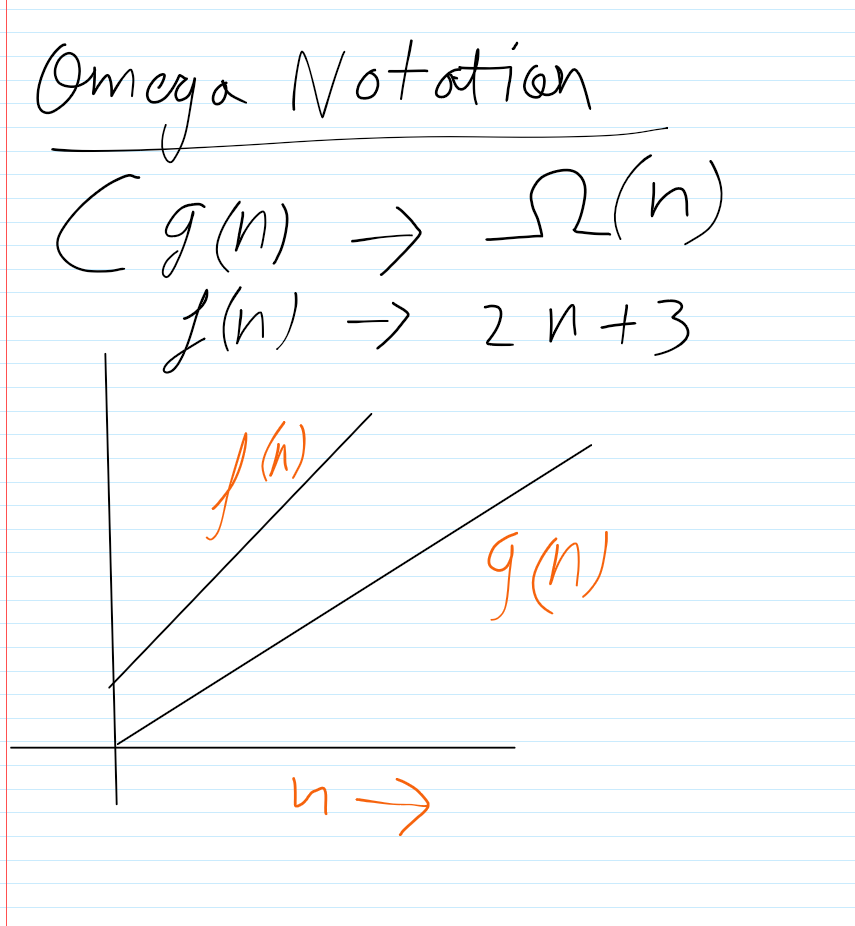
Here C, as per lower bound, should be -1 to the Constant of the highest order, (2n+3), n is the highest order and 2 is it’s Constant, so 🡪 Cn🡪 (2-1)n

n <= 2n+3

n <= -3 but the value should be in positive, so the next positive value after -3 = 0

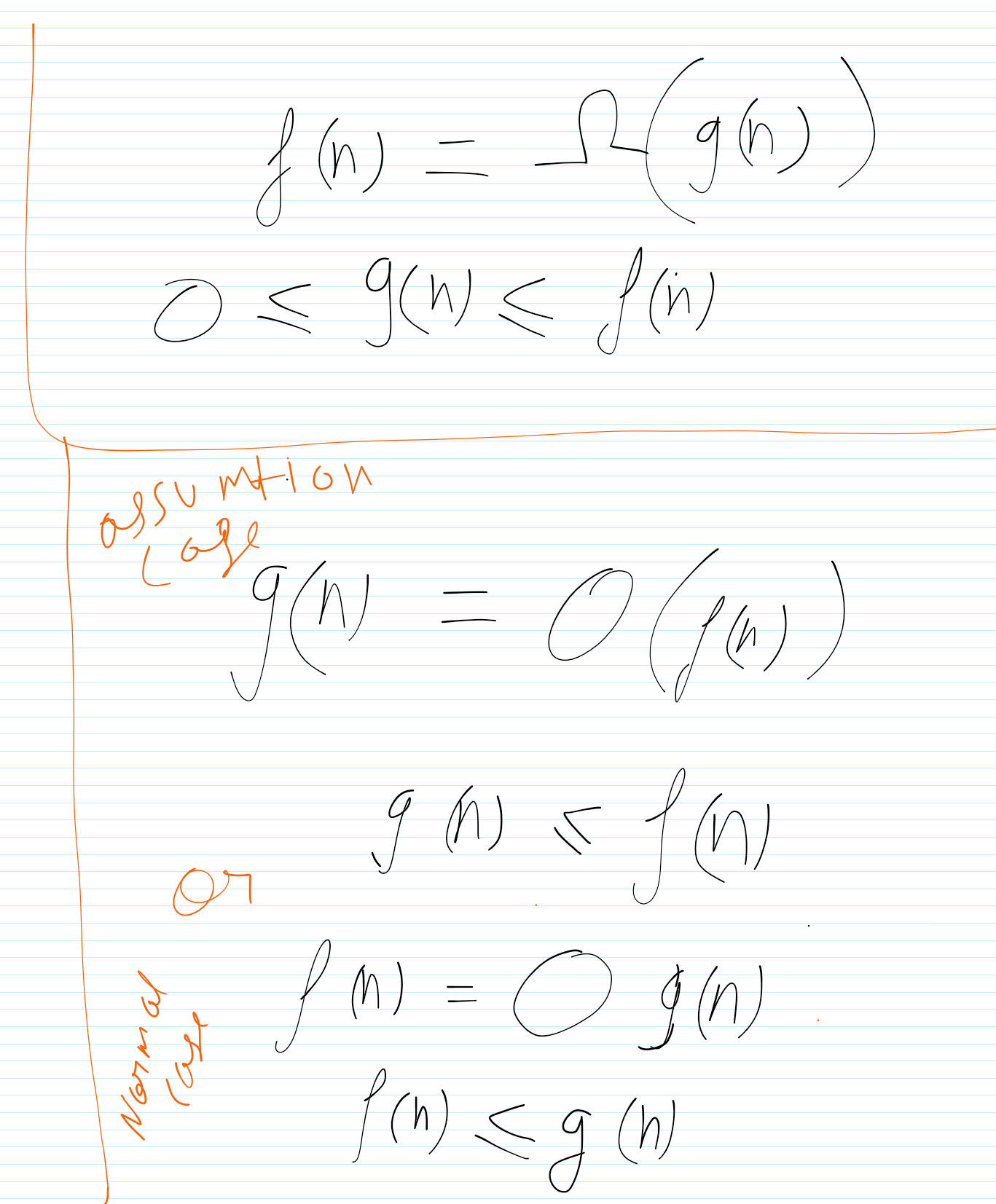
n0 = 0 for every n >= 0, f(n) >= Cg(n)

Pictorial Representation:



Ω represents all order of growth equal to linear and higher order of growth, so the lower bound is linear order means 🡪 n is linear and n2 is higher, which is also included in Omega Notation.

{n/4, 2n, n, 2n+3, n2, 2n2, ….., nn} -- Ω(n)



**Theta Notation:**

(n) represents the exact bound



We say f(n) = (g(n)), iff there exists Constants C1 and C2 where C1>0 and C2>0 and n0 where n0 >= 0 such that C1g(n) <= f(n) <= C2g(n)

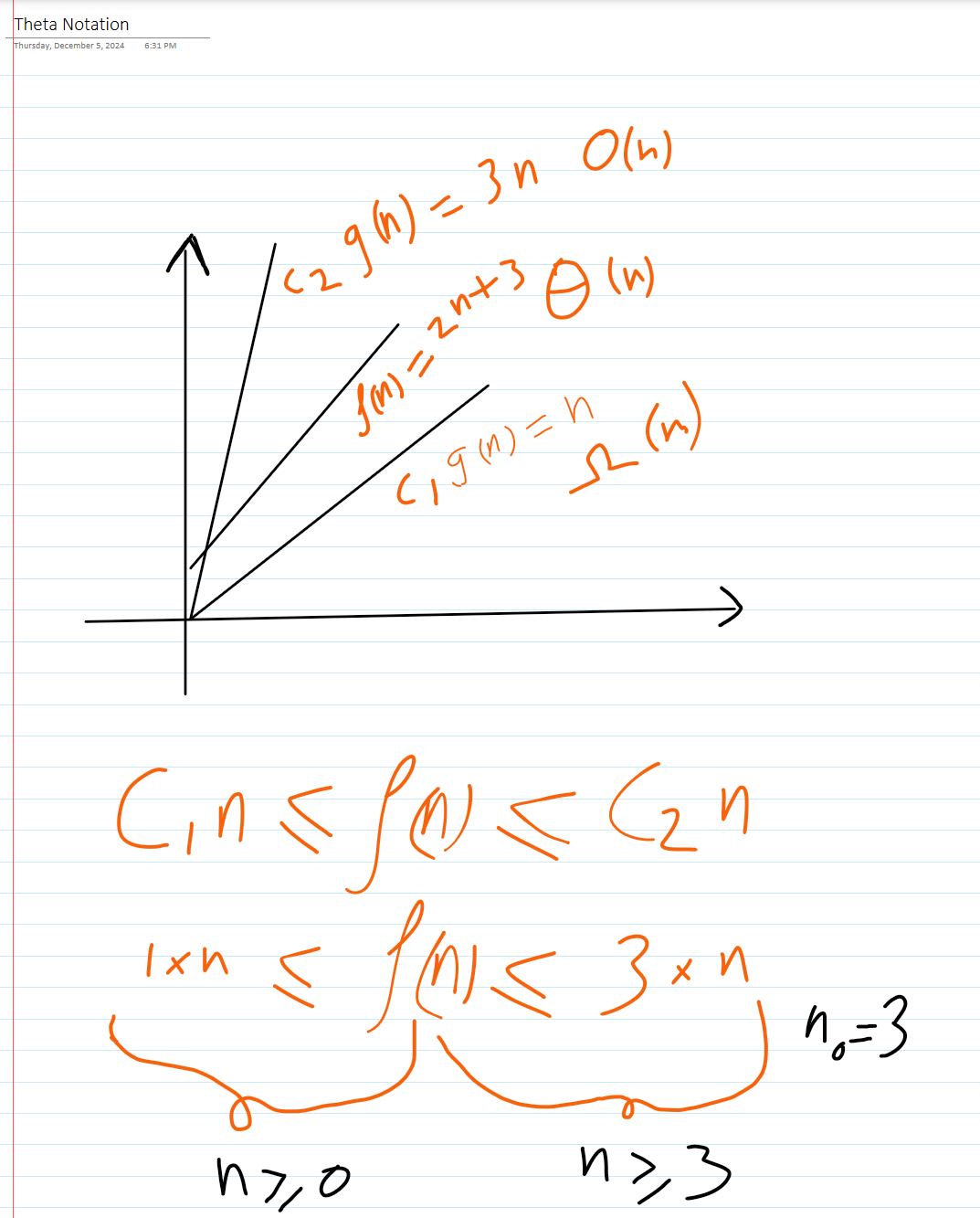
**Direct:** 1000n2+100 nlogn+2n🡪 (n2)

**If** f(n) = (g(n))

**then** f(n) = O(g(n)) and f(n) = Ω(g(n)) as they all have the exact same order of growth.

**and** g(n) = O(f(n)) and g(n) = Ω(f(n)) as (n) is the exact bound.

Pictorial Representation:



{100, 105, log 2000, ….} -- (1)

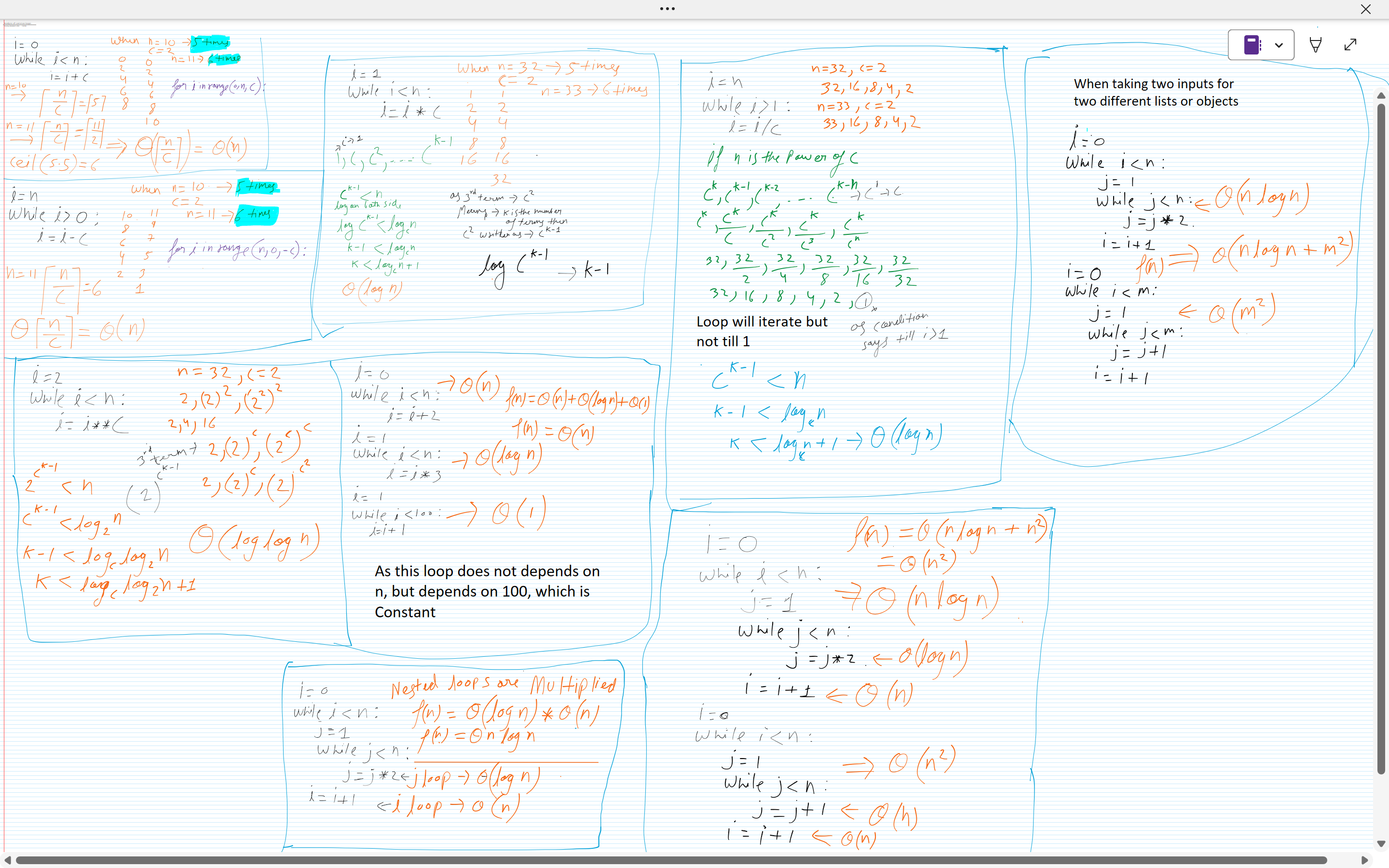
{100n, 2n+log n, 5n+3, …} -- (n)

{2n2, n2/4+5 n log n, …} -- (n2)

**When you have an algorithm, knowing it’s exact bound use🡪** theta Notation

* e.g. maximum in an array, it will have to search for all the terms(n), which is exact bound of (n).
* Using O(n) will not tell exact, it could be n or less than n, as it only tells the upper bound.
* In insertion sort, we can use O(n2) for sorted array, as it covers the best and worst case. The insertion could be at n2 or at n, and Big O notation covers the upper bound and the one lesser order terms than that.
* For reverse sorted, insertion sort takes n times

**Analysis of Common loops:**

****