ASSIGNMENT 2

Jaspreet Kaur

11529746

Qs1 Why Algorithm Analysis is important both in terms of running time and Space Complexity?

An Algorithm is determined by 2 factors – Space and Time.
Time Complexity determines how much time an algorithm takes with respect to input size.
Space Complexity determines how much space has been taken by an algorithm.

If for small values, an algorithm **"A1" takes less time** and **time increases very large** (exponential maybe) as the **input size increases** – algorithm "A1" cannot be said as GOOD ALGORITHM.
On the other hand, if an **algorithm "A2" takes more time when compared to "A1" for small value of input but takes less time for large input size** – Algorithm "A2" can be stated as better algorithm than "A1".

Therefore, Time complexity helps us to better understand which algorithm performs better.
Space complexity is helpful as computer system has limited memory and it becomes crucial to use that memory wisely.

For example – Suppose we want to find if an element exist in particular sorted array arr[].

Two Approaches –

1st Approcah – Linear Search

```
Linear Search(int arr[]){
for(int i=0;i<n;i++)
    {
        If(arr[i] == element)
                Return true;
    }
}
```

2nd Approcah – Binary Search

```
Binary(int arr[], int element){
	While(low <= high){
		int mid = low+(high-low)/2;
		if(arr[mid] == elment){
			return true;
		}
		If(arr[mid] , element){
			low = mid+1;
		}
		Else{
			Low = mid -1;
		}
	}
	Return -1;
}
```

Both Approach1 and Approach2 will give the desired answer, but binary search will take less time to execute than linear search. Therefore, Binary Search Algorithm is better than Linear Search in this case.

Qs2 The Order of growth of an Algorithm is how long the time of execution depends on the length of the input array. Mathematically, show worst-case (upper-bound), average-case (tight-bound), best-case (lower-bound) of an algorithm. Explain clearly. What is asymptotic in order of growth?

Asymptotic Notation

By dropping the less significant terms and constant coefficient, we can focus on important part of an algorithm running time – it's rate of Growth. When we drop the constant coefficients and less significant terms – we use Asymptotic Notation.

For ex → Suppose an algorithm running on input size n takes $6n^2 + 100n + 300$ machine instruction. we can ignore less smaller term $(100n+300)$ and say running time of algorithm is n^2.

Asymptotic Notation → Ω-Notation

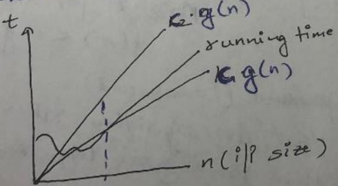
O-Notation

Θ-Notation

(Tight Bound)

Θ-Notation → When we say that a particular running time is $\Theta(n)$, we are saying that once n gets large enough, the running time is atleast $k_1 \cdot n$ and atmost $k_2 n$ for some constants k_1 and k_2.

AVERAGE CASE



Let f is a non-negative function such that –

$$c_1 g(n) \le f(n) \le c_2 g(n)$$

where $c_1, c_2 > 0$ & $n \ge n_0$.

for eg. → $f(n) = 10n^3 + 5n^2 + 17 \simeq \Theta(n^3)$

$$10n^3 \le f(n) < (10+5+17)n^3$$

$$\boxed{10n^3 \le f(n) \le 32n^3} \Rightarrow f(n) \sim \Theta(n^3)$$

where $c_1 = 10$; $c_2 = 32$ and $n \ge 1$.

When we use big-Θ Notation - we say that we have Asymptotically tight bound on the running time.

(upperbound)

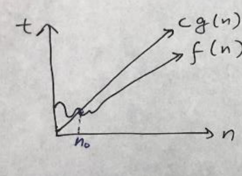
Big-O Notation - It bounds the growth of the running time ~~from~~ atmost ($c \cdot g(n)$) ~~$g(n)$, above~~ for large enough input size.

WORST CASE

if $f(n)$ is a non-negative function, ~~funct~~
there exist constant c_2 & n_0 for which

$f(n) \leq c_2 \, g(n)$

where $c_2 > 0$ & $n \geq n_0$.



If $f(n) \leq c g(n)$
$n \geq n_0$
$c > 0$ & $n_0 \geq 1$
then, $f(n) \sim O(g(n))$

eg → $f(n) = 3n+2$ $\quad g(n) = n$

$f(n) = O(g(n))$

$f(n) \leq c\, g(n)$

$3n+2 \leq c\, n$

let $c = 4$ ⇒ $3n+2 \leq 4n$

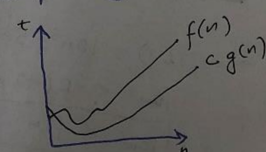
$\boxed{n \geq 2}$ ⇒ we have found $c = 4$ and $n_0 \geq 2$ for which $f(n) <= g(n)$

Big-Omega Notation (Big-Ω) ~~(Upperbound)~~ (LowerBound)

Sometimes, we want to say that an algorithm takes at least a certain amount of time, without providing an upper bound.

BEST CASE

if f is a non-negative function
then there exist constant c_1 & n_0
for which $f(n) \geq c_1 \, g(n)$
for $n \geq n_0$.



ALGORITHM EXAMPLE TO DISTINGUISH

Average Case, Best Case And Worst Case →

Suppose there is array [2, 5, 4, 6, 10, 12]

if we want to find number 2 → BEST Case which $\boxed{O(1)}$ will take constant time $O(1)$.

if we want to find number 13 → WORST case $\boxed{O(n)}$ because it doesnot exist in the array.

if we want to find number 6 → Average Case because it is in the middle of the array.

3. Consider the following code:

A) a) Why the total count of this algorithm is:

; $0+1+2+\ldots+(N-1)=\frac{N*(N-1)}{2}$

and b) why time-complexity is O(N^2)?

```
int count = 0;
for (int i = 0; i < N; i++)
   for (int j = 0; j < i; j++)
```

```
count++;
```

B) Why time-complexity of the following algorithm is O(N) and not O(N * LogN)?

```
int count = 0;
for (int i = N; i > 0; i /= 2)
   for (int j = 0; j < i; j++)
      count++;
```

C) What is the time-complexity of this algorithm?

```
int count = 0;
for (int i = 0; i < N; i++)
   for (int j = 0; j < i; j++)
      count++;
```

D) What is the time-complexity of this algorithm?

```
int count = 0;
for (int i = N; i > 0; i /= 2)
   for (int j = 0; j < i; j++)
      count++;
```

(3) (a)

```
int count = 0;
for (int i = 0; i < N; i++)
{
    for (int j = 0; j < i; j++)
        count++;
}
```

The number of operations performed by the algorithm is $0 + 1 + 2 + 3 + \ldots (N-1) = \frac{N \times (N-1)}{2}$

Let us assume $N = 5$

The outer loop will run N times and inner loop will run 'i' times.

i	j	count
0	X	0
1	0	1
2	0, 1	2
3	0, 1, 2	3
4	0,1,2,3	4
5	~~0,1,2,4~~	~~5~~ X

loop terminates

From the table we can count how many times i & j executed

$(0 \times 0) + (1 \times 1) + (1 \times 2) + (1 \times 3) + (1 \times 4)$

$= 0 + 1 + 2 + 3 + 4$

$\Rightarrow 10 = \frac{N \times (N-1)}{2} = \frac{5 \times \cancel{4}^{2}}{\cancel{2}} = 10$

4

(b) Time-Complexity for Algorithm is $O(n^2)$ because inner loop runs 'n' times and Outer loop also execute 'n' times $\longrightarrow n \times n = n^2$

B)

```
int count=0
for (int i=N; i>0; i/=2)
{ for (int j=0; j<i; j++)
    count++;
}
```

At first, it seems like Time complexity is $N * \log N$
But, if we analyse algorithm we find that -
Outer loop takes value $\rightarrow N, \frac{N}{2}, \frac{N}{4} \dots \frac{N}{2^{\log N}}$

$$N\left[1 + \frac{1}{2} + \frac{1}{4} + \dots \frac{1}{2^{\log N}}\right] \simeq O(N)$$

c)

```
int count=0
for (i=0; i<N; i++)
{
  for (j=0; j<i; j++)
    count++;
}
```

The time-complexity for the algorithm is $O(N^2)$ because inner loop runs N times & Outer loop runs N times in worst-case.

d)

```
count=0
for (i=N; i>0; i/=2)
  for (j=0; j<i; j++)
    count++;
```

The time complexity is $N + \frac{N}{2} + \frac{N}{4} + \dots \frac{1}{2^{\log N}} \simeq O(N)$

4. The worst-case running time of an Algorithm can be: (constant 1, logN, N, NlogN, N^2, N^3, 2^N). Mathematically, each instance follows the following model, describe each case:

$$\binom{N}{3} = \frac{N(N-1)(N-2)}{3!} \sim \frac{1}{6}N^3$$

(4) To develop a mathematical expression, we depend on 2 factors :

(i) Number of machine instructions

(ii) time for each instruction.

```
int N = a.length;
int count = 0
for (int i=0; i<n; i++)
{
  for (int j=i+1; j<N; j++)
  {
    for (int k=j+1; k<N; k++)
    {
      if (a[i]+a[j]+a[k]==0)
      {
        count++;        Block E
      }
    }
  }
}
```

Block A, Block B, Block C, Block D

Block	time	frequency	Total time	
E	t_0	x (depends on input)	$t_0\,x$	$\sim O(x)$
D	t_1	$\frac{N^3}{6}-\frac{N^2}{2}+\frac{N}{3}$	$t_1\left(\frac{N^3}{6}-\frac{N^2}{2}+\frac{N}{3}\right)$	$\sim O(N^3)$
C	t_2	$\frac{N^2}{2}-\frac{N}{2}$	$t_2\left(\frac{N^2}{2}-\frac{N}{2}\right)$	$\sim O(N^2)$
B	t_3	N	$t_3 N$	$\sim O(N)$
A	t_4	1	t_4	$\sim O(1)$

logN Example –

```
Binary(int arr[], int element){
        While(low <= high){
                int mid = low+(high-low)/2;
                if(arr[mid] == elment){
                        return true;
                }
                If(arr[mid] , element){
                        low = mid+1;
                }
```

```
                    Else{
                              Low = mid -1;
                    }
          }
          Return -1;
}
```

NlogN Example –

```
For(int i=0;i<N;i++){

          For(int j=1;j<N;j++){

                    I=i*j;

          }

}
```

Total Time = nlogn

2^n Example –

```
 int Fibonacci(int number)
{
 if (number <= 1) return number;

 return Fibonacci(number - 2) + Fibonacci(number - 1);
}
```

Total work done will sum of work done at each level, hence it will be 2^0+2^1+2^2+2^3...+2^(n-1) since i=n-1. By geometric series this sum is 2^n, Hence total time complexity here is **O(2^n)**

5. Estimate the running time (or memory) as a function of input size *N*. Explain as to why the results are the same for the following three examples.

$$\tfrac{1}{6} N_3 + 20 N + 16 \quad \sim \quad \tfrac{1}{6} N_3$$
$$\tfrac{1}{6} N_3 + 100 N_{4/3} + 56 \quad \sim \quad \tfrac{1}{6} N_3$$
$$\tfrac{1}{6} N_3 - \tfrac{1}{2} N_2 + \tfrac{1}{3} N \quad \sim \quad \tfrac{1}{6} N_3$$

smaller values

(5) $\frac{1}{6}N^3 + 20N + 16 \sim \frac{1}{6}N^3$

$\frac{1}{6}N^3 + 100N^{4/3} + 56 \sim \frac{1}{6}N^3$

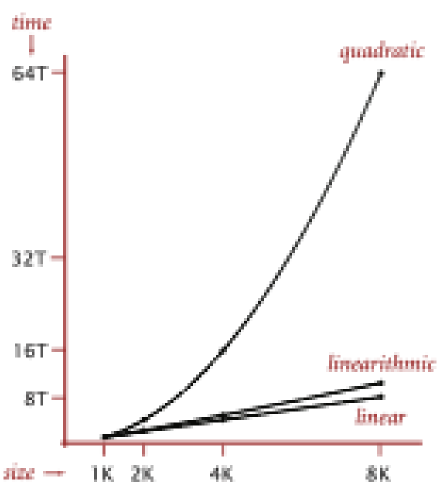
$\frac{1}{6}N^3 + \frac{1}{2}N^2 + \frac{4}{3}N \sim \frac{1}{6}N^3$

All three function will have running of $O(N^3)$ because we can neglect smaller values while considering the running time

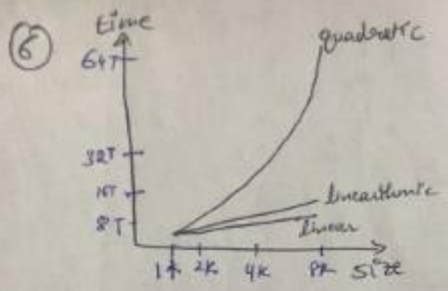
for some machine dependent Constant a,

$T(N) \sim aN^3$

6. Explain this graph



6



7

→ Linear suggest that as the input size increases, running time increases correspondingly.

→ Linearithmic suggest that running time for a size N has order of growth NlogN.

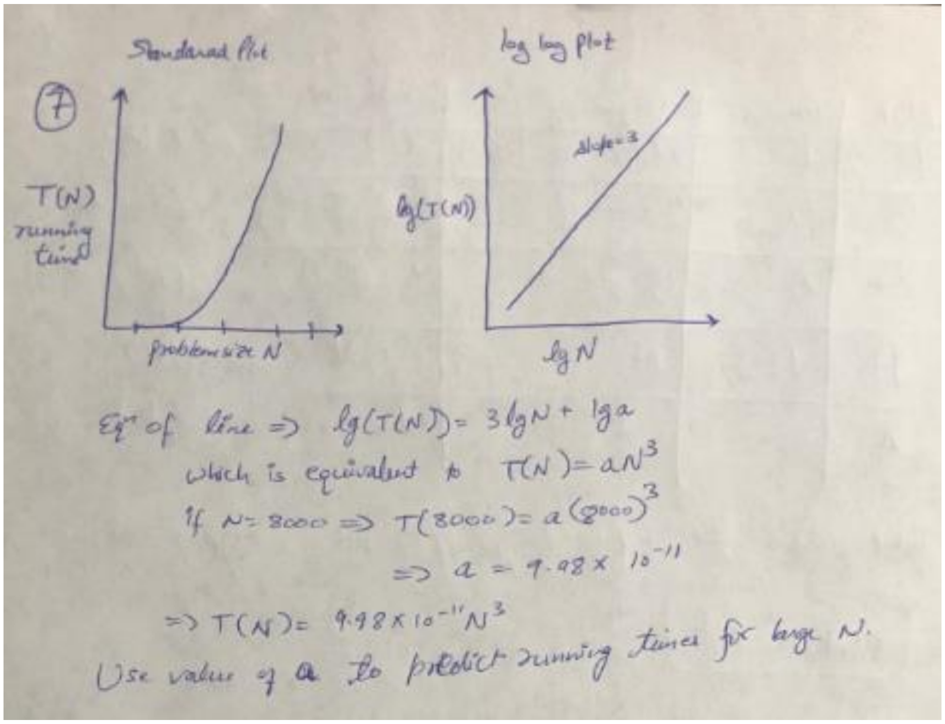
→ Quadratic suggest that, as the input size grows, the running time grows Quadratically.

If N = 8K => time ~64T.

7. Explain this data with various input sizes and measure running time, What is the graph looks like?

N	time (seconds) †
250	0

N	time (seconds) †
500	0
1,000	0.1
2,000	0.8
4,000	6.4
8,000	51.1
16,000	?



8. Explain as to why this is Brute-Force Algorithm;

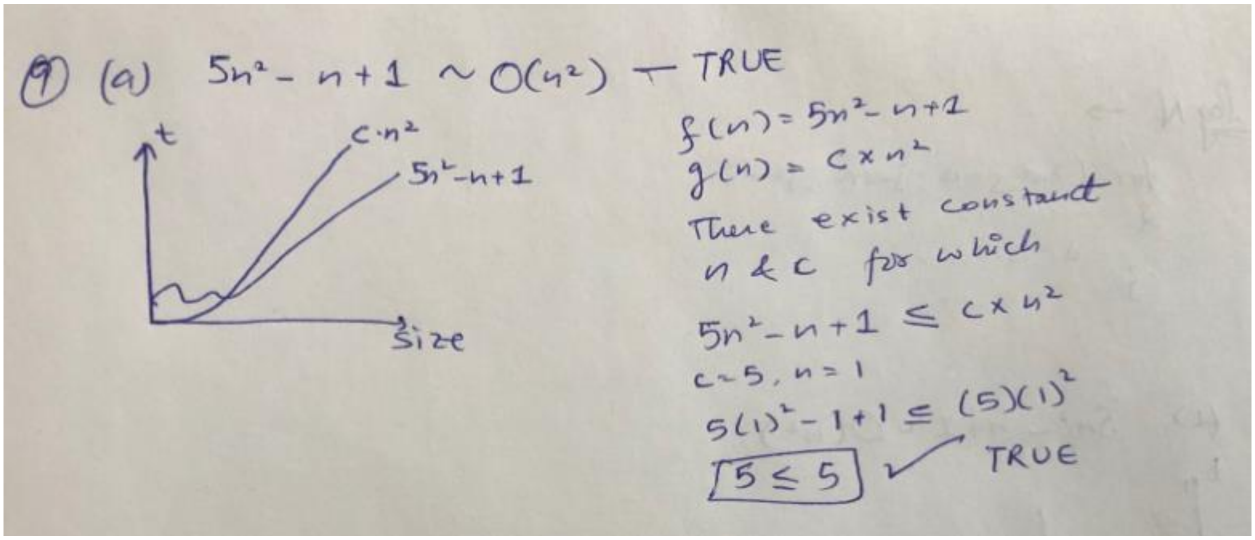
```
for (int i = 0; i < N; i++)
      for (int j = i+1; j < N; j++)
        for (int k = j+1; k < N; k++)
          if (a[i] + a[j] + a[k] == 0)
            count++;
```

Answer8 - The above algorithm has three loops and each loop runs N times –
It calculate every possible combination – that's why its Brute Force.
N * N * N ~ O(N^3)

9. Consider the following functions asymptotically:

A) true or false
B) draw the graph
C) explain Why true or false

a) **5n2-n+1 is Big O(n2)**



(9) (a) $5n^2 - n + 1 \sim O(n^2)$ — TRUE

$f(n) = 5n^2 - n + 1$

$g(n) = c \times n^2$

There exist constant n & c for which

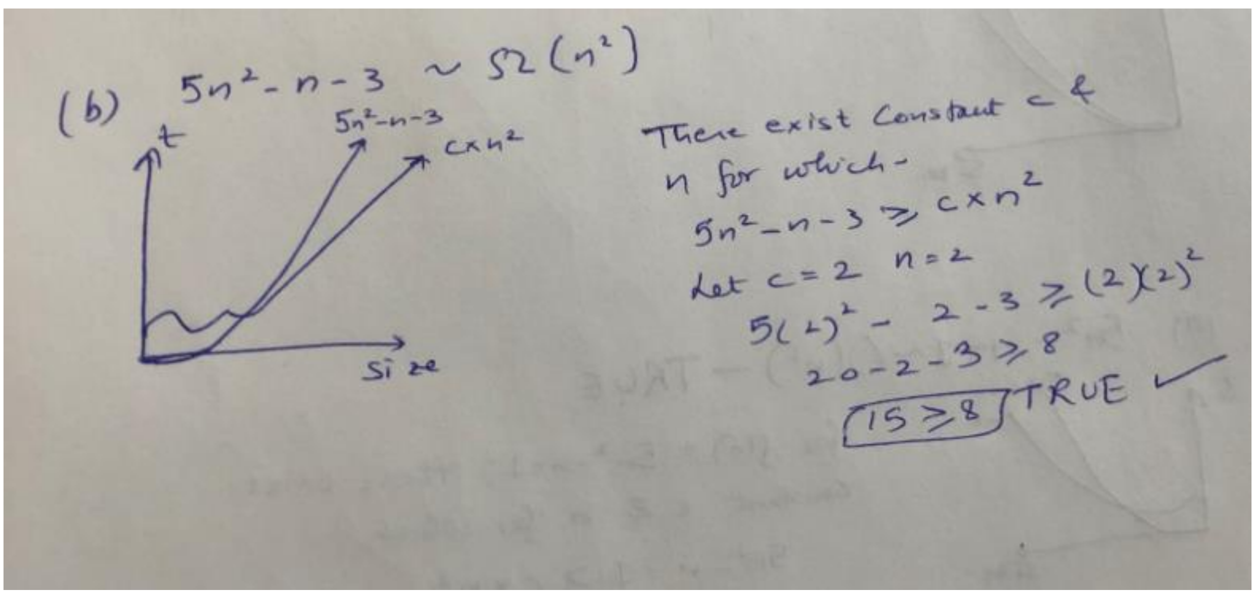
$5n^2 - n + 1 \leq c \times n^2$

$c = 5,\ n = 1$

$5(1)^2 - 1 + 1 \leq (5)(1)^2$

$\boxed{5 \leq 5}$ ✓ TRUE

b) 5n2-n-3 is Ω(n2)



(b) $5n^2 - n - 3 \sim \Omega(n^2)$

There exist Constant c & n for which -

$5n^2 - n - 3 \geq c \times n^2$

Let $c = 2$ $n = 2$

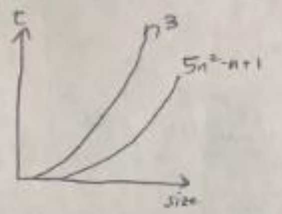
$5(2)^2 - 2 - 3 \geq (2)(2)^2$

$20 - 2 - 3 \geq 8$

$\boxed{15 \geq 8}$ TRUE ✓

c) 5n2 -n+1 is Big O(n3)

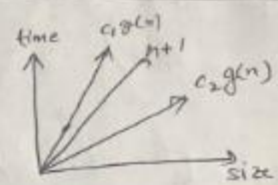
(9) (c) $5n^2 - n + 1 \sim O(n^3)$ - False



for $5n^2-n+1$ to be $O(n^3)$, there should have been some constant c & n_0 for which $5n^2 - n + 1 \le n^3$; but we can analyse that, it's not possible.

d) 4n+1 is Θ(n)

(d) $4n+1 \sim \Theta(n)$ - True



$c_1 n \le 4n+1 \le c_2 n$

Let $c_1 = 5$; $c_2 = 6$; $n = 1$

$(5)(1) \le 4(1)+1 \le (6)(1)$

$5 \le 5 \le 6$ ✓

for $c_1 = 5$; $c_2 = 6$ & $n = 1$ $\Rightarrow$ $c_1 n \le 4n+1 \le c_2 n$

Therefore $4n+1 \sim \Theta(n)$

e) 5n2-n+1 is Big O(n)

(e) $5n^2 - n + 1 \sim O(n)$ – False

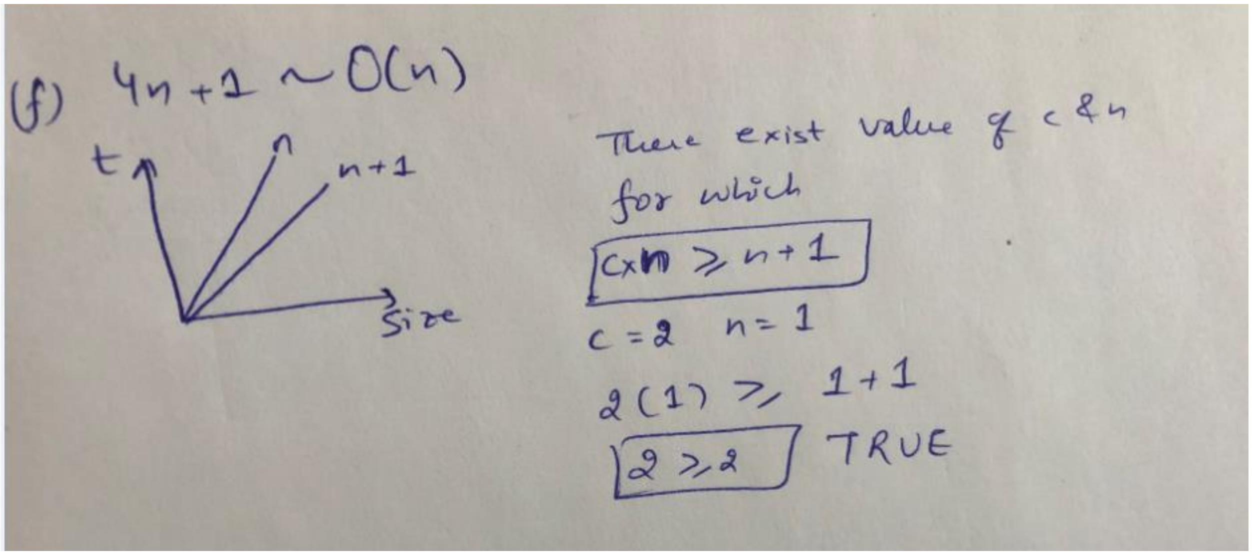
Let us check if there exist c_1; c_2 & n for which →

$c_1 n \le 5n^2 - n + 1 \le c_2 n$

Since n^2 will always be a larger term so we cannot find any constant c_2 for which $5n^2 - n + 1 \le c_2 n$

Therefore $5n^2 - n + 1$ is not of the Order $O(n)$

f) 4n+1 is Big O(n)



(f) $4n+1 \sim O(n)$

There exist value of c & n for which

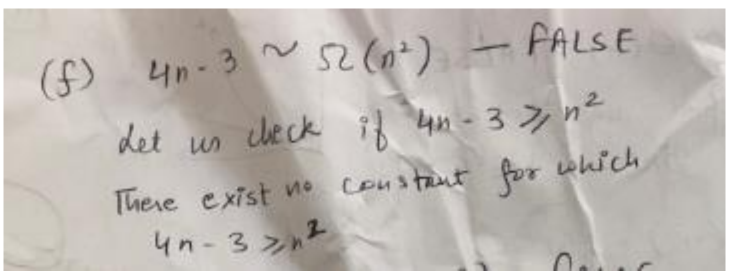
$\boxed{c \times n \geq n+1}$

$c = 2 \quad n = 1$

$2(1) \geq 1+1$

$\boxed{2 \geq 2}$ TRUE

g) 4n-3 is Ω(n2)



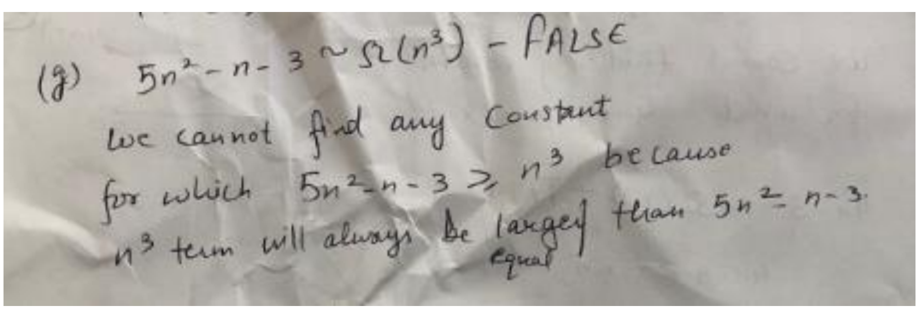
(f) $4n-3 \sim \Omega(n^2)$ — FALSE

Let us check if $4n-3 \geq n^2$

There exist no constant for which

$4n-3 \geq n^2$

h) 5n2-n-3 is Ω(n3)



(g) $5n^2-n-3 \sim \Omega(n^3)$ - FALSE

We cannot find any Constant for which $5n^2-n-3 \geq n^3$ because n^3 term will always be larger/equal than $5n^2-n-3$.

i) 4n+1 is Big O(n2)

(h) $4n+1 \sim O(n^2)$ — TRUE

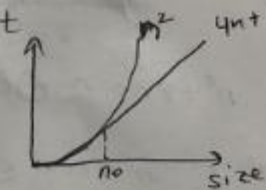
We can find upperbound for $4n+1$ for which -

$c_1 \times n^2 \le 4n+1 \le c_2 n^2$

$c_1 = 4$; $c_2 = 5$; $n = 1$

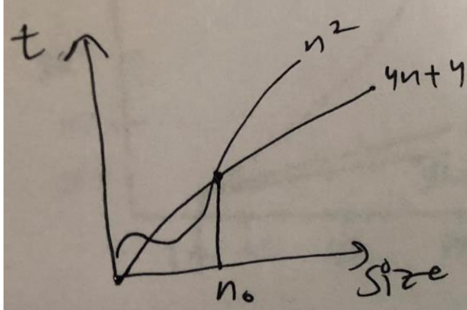
$(4) \times (1)^2 \le 4(1)+1 \le (5)(1)^2$

$\boxed{4 \le 4 \le 5}$ ✓



j) 4n+4 is Θ(n2)

$4n+4 \sim \Theta(n^2)$ TRUE



$c_1 n^2 \le 4n+4 \le c_2 n^2$

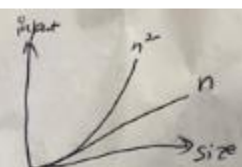
for $c_1 = 4$; $c_2 = 8$; $n = 1$

$4 \le 8 \le 8$

k) 5n2-n+1 is Θ(n)

(8) $5n^2 - n + 1 \sim \Theta(n)$ - FALSE

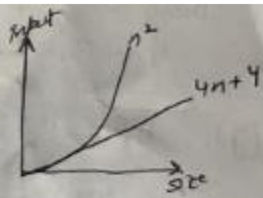
We can not find any constant c for which $5n^2 - n + 1 \le c \times n$



l) 4n+4 is Θ(n2)

(k.) $4n+4 \sim \Theta(n^2)$ - FALSE

we cannot find any constant c for which $4n+4 \geq c \times n^2$



m) 4n-3 is Ω(n)

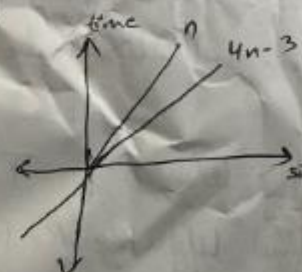
(l) $4n-3 \sim \Omega(n)$ — TRUE

$4n-3 \geq c \times n$

for $c=1$ & $n=1$

$4(1)-3 \geq (1)\times(1)$

$\boxed{1 \geq 1}$ TRUE



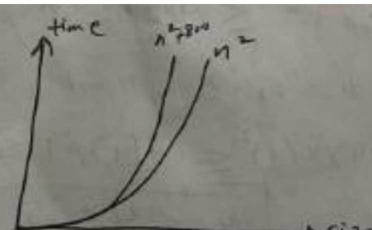
n) n2+800 is Θ(n2)

(M) $n^2+800 \sim \Theta(n^2)$ - TRUE

$c_1 \times n^2 \leq n^2+800 \leq c_2 \times n^2$

$c_1 = 800$; $c_2 = 801$, $n=1$

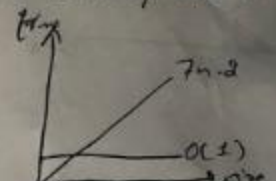
$\boxed{800 \leq 801 \leq 801}$



o) 7n-2 = Θ(1)

(n) $7n-2 \sim \Theta(1)$ - FALSE

$7n-2$ corresponds to size of n.



10. Fill in the asymptotic relationship in table below: T(n) = 5n^2 – n + 1

	BigO	Omega	Theta
n!	yes	No	No
2^n	Yes	No	No
n^2	Yes	Yes	Yes
nlogn	No	Yes	No
n	No	Yes	No
logn	No	Yes	No
1	NO	Yes	No