Vaishnavi Vishwas Pawar
vpawar@iu.edu

ASSIGNMENT NO. 1

Q1. Prove or disprove the following questions on asymptotic notation.

a. Prove or disprove that $f(n) \in \Theta(g(n))$ where $f(n) = 64n^2$ & $g(n) = n^4$

→ We have a Big Theta $\Theta()$ Notation

$$\Theta(g(n)) = \{ f(n) \mid 0 \le c_2 \cdot g(n) \le f(n) \le c_1 \cdot g(n)), \forall n \ge n_0, \exists (c_1 > 0, c_2 > 0, n_0 > 0) \}$$

To prove $f(n) \in \Theta(g(n))$ we need to find explicit values of c_1, c_2 & n_0

For upper bound, $f(n) = 64n^2$ & $g(n) = n^4$

$\therefore f(n) \le c_1 \cdot g(n)$

$64n^2 \le c_1 \cdot n^4$

Divide both the sides by n^2

$64 \le c_1 \cdot n^2$

Let's consider $n = 1$

$64 \le n^2$

$\therefore \underline{\underline{8 \le n}}$

If we consider $n_0 = 10$, then the upper bound condition is satisfied. as we get $8 \le 10$, which is true.

For lower bound, $f(n) = 64n^2$ & $g(n) = n^4$

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

$c_2 \cdot n^4 \le 64n^2$

Divide both the sides by n^2

$c_2 \cdot n^2 \le 64$

Let's consider ~~n~~ $c_2 = 1$

$n^2 \le 64$

$\therefore n \le 8$

If we consider $n_0 = 10$, then the lower bound condition does not ~~get~~ satisfy, as we get $10 \le 8$, which is false.

As both the conditions are not satisfied, we can say that
$f(n) \notin \Theta(g(n))$.

Hence, we can say $f(n) \notin \Theta(g(n))$, we disprove it

b. Prove or disprove that $f(n) \in \Omega(g(n))$ where $f(n) = \frac{n^2}{3} + 10n - 2$ and $g(n) = n^2$

-> We have a Big Omega Notation $\Omega(\,)$

$$\Omega(g(n)) = \{ f(n) \mid 0 \leq c \cdot g(n) \leq f(n), \forall n \geq n_0, \exists c > 0, \exists n_0 > 0 \}$$

To prove $f(n) \in \Omega(g(n))$ we need to find explicit values of c and n_0, where $f(n)$ is always larger than or equal to $c \cdot g(n)$ for all $n > n_0$

$\therefore \quad c \cdot g(n) \leq f(n)$

$f(n) = \frac{n^2}{3} + 10n - 2$ & $g(n) = n^2$

$c \cdot n^2 \leq \frac{n^2}{3} + 10n - 2$

Let's consider $c = 1$ and $n = 10^2$

$(1)(100)^2 \leq \frac{(10^2)^2}{3} + 10(10^2) - 2$

$10000 \leq \frac{10000}{3} + 1000 - 2$

$10000 \leq 4331.33$

$\therefore \quad 10^4 \not\leq 4331.33$

Here, if we consider $c = 3$ & $n = 2$

$3(2)^2 \leq \frac{(2)^2}{3} + 10(2) - 2$

$12 \leq \frac{4}{3} + 20 - 2$

$12 \leq 19.33$

Here the condition gets satisfied, but if we take n_0 as a huge number it doesn't get satisfied.
But $f(n)$ should always be greater than $c \cdot g(n)$

Hence, for $\forall n \geq n_0$ the condition is not proved

$\therefore \quad \frac{n^2}{3} + 10n - 2 \notin \Omega(n^2)$

Hence, we can say that $f(n) \notin \Omega(g(n))$, so it is disproved.

c. Prove or disprove that $f(n) \in O(g(n))$ where $f(n) = 300000n^3+1$ & $g(n) = n^4$.

→ We have a Big-oh Notation $O()$

$O(g(n)) = \{f(n) \mid 0 \le f(n) \le c\,g(n),\ \forall n \ge n_0,\ \exists c > 0, \exists n_0 > 0\}$

To prove $f(n) \in O(g(n))$, we need to find explicit values of c and n_0, where $f(n)$ is always less than or equal to $c.g(n)$, for all $n > n_0$

$\therefore f(n) \le c.g(n)$

Lets consider $c = 3$ & $n = 10^{18}$

$\therefore 300000n^3 + 1 \le c.(n^4)$

$\therefore$ We can drop +1, as it is a small value from the function.

$\therefore 300000n^3 \le 3n^4$

Divide by n^3

$\therefore 300{,}000 \le 3n$ for $n = 10^{18}$

$3 \times 10^5 \le 3(10^{18})$

$\therefore$ Hence the condition is satisfied.

$\therefore f(n) \in O(g(n))$

$\therefore f(n)$ is bounded by $O(g(n))$, so it is proved.

d. Prove or disprove that $f(n) \in o(g(n))$ where $f(n) = 15n^{15}$ and $g(n) = n^{15}$

→ Here we have a Little Oh Notation $o()$

$o(g(n)) = \{f(n) \mid 0 \le f(n) < c.g(n),\ \forall n \ge n_0,\ \forall c > 0,\ \exists n_0 > 0\}$

Other Alternative definition is

$$\{f(n) \mid \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0\}$$

$f(n) < c.g(n) \quad \forall n > n_0$

$15.n^{15} < c.n^{15}$

Here for $o(g(n))$ the condition for $\forall c$ is that $\forall c > 0$.

$\therefore c = 2$ and $n = n_0 = 3$

$\therefore 15(3)^{15} < 2(3)^{15}$

Here the $f(n)$ function is greater than $c.g(n)$.

Hence $f(n) \notin o(g(n))$

As the condition of $f(n)$ should be less than $c.g(n)$ is not satisfied, $f(n) \notin o(g(n))$, so it is disproved

e. Prove or disprove that $f(n) \in \omega(g(n))$ where $f(n) = n^{20} - 17$ and $g(n) = n^{16}$

→ We have a Little omega Notation $\omega(g(n))$

$\omega(g(n)) = \{f(n) \mid 0 \leq c.g(n) < f(n), \forall n \geq n_0, \forall c > 0, \exists n_0 > 0\}$

$\therefore c.g(n) < f(n)$

Here we have to prove $f(n)$ is always greater than $c.g(n)$

$c.n^{16} < n^{20} - 17$

$\therefore$ Lets consider $c = 1$.

$n^{16} < n^{20} - 17$

$17 < n^{20} - n^{16}$.

$17 < n^4$

If we consider $n = 5$

$17 < (5)^4$

Other alternative is,

$c = 3, n = 7$.

$3(7)^{16} < (7)^{20} - 17$

Either way, the condition is satisfied

Hence, condition is satisfied, $\therefore f(n) \in \omega(g(n))$

As the condition is satisfied $f(n) \in \omega(g(n))$, it is proved.

Q2. Find the time complexities.

a.
```
def function1(n):
    for i in range (0, n):
        for j in range (0, i+1):
            print ("*")
            break
    return.
```

Soln: For the above code,

The outer for loop gives a time complexity of $O(n)$

The inner for loop also gives $O(n)$ time complexity, but as there is a break statement inside the loop, the inner loop will terminate after printing just one "*" on each iteration of the outer for loop.

∴
```
def function1(n)
    for i in range (0,n)          ....  runs n times
        for j in range (0,i+1)     ...  will also run
            print("*")                  n times but as there
            break.                      is break statement
    return.                             so it is O(1)
```

∴ Time complexity of the above code is $O(n)$

b.
```
def function2(n):
    i=1
    while i**2 <= n:
        i = i+1
    return
```

The above code uses the while loop, it iterate till $i^2 <= n$. In each iteration i is incremented by 1.

$i = 1$		def function2(n)	
i	$i^2 <= n$	i=1	... i=1
1	✓	while i**2 <= n:	
2	✓	i=i+1	... i is incremented
3	✓	return.	the loop works, floor (sqrt n) i.e($\sqrt{n}$)
⋮			

x iteration where x. iteration of i will satisfy the while loop.

∴ The loop runs ~~iter~~ 'x' times, where x is the integer part of the square root of 'n'
∴ The time complexity = O(sqrt(n))

∴ Time complexity = $O(\sqrt{n})$.

c. def ffunction3(m, n):

```
def function3(m, n):
    while (m != n):
        if (m > n):
            m = m - n
        else:
            n = n - m
    return
```

Lets consider the values here.
~~As its~~ If m = 2 and n=2, the while loop won't work. giving the best case as O(1).
But ~~if~~ m, n and value are different, it gives the worst case time complexity.

	m	n	m = m - n	n = n - m.
if cond 1:	3	2	✓ ∴ m = 1.	×
condition 2:	2	3	×	✓ ∴ n = 1.

Either way, and for any other number, there is a constant difference in each case.
∴ The time complexity depends on m and n values.
∴ Time complexity = max(m, n)

d.
```
def function4(n):
    i = 1                  . . . . .   i = 1.
    while i<n              . . . . .   iterated and depend
                                       on i ∴ log_2(n)
        j = n
        while (j>0):       . . .       log_2 n
            j = j//2       . . .       till j reaches zero
        i = 2*i            . . .       i is doubled.
    return n.
```

Sol^n: Here the outer while loop runs till $i<n$.
The i value gets doubled in each iteration
∴ The outer while loop iterates and depends upon 'i', as it is doubled in each iteration, it is expressed in log(n)

The inner while loop run as long as j is greater than 0, j is divided by 2 in each iteration
It will run as long as the j value reaches zero.
∴ It can also be expressed in $\log_2 n$

∴ The Total Time complexity = $O(\log_2 n)$

e.
```
def functions(n):
    for i in range(0, n//2):
        for j in range(1, n-(n//2)+1):
            m=1
            while m<=n:
                m *= 2
    return n
```

Sol^n: The outer for loop iterates from range 0 to n//2
∴ It completely on the value of n
Hence outer for loop complexity is O(n)

The inner for loop iterates from range 1 to n-(n//2)+1
The loop runs n/2 + 1 times, and is completely depended on the n value. Hence the inner for loop complexity is also O(n)

The while loop has a condition where $m \leq n$. It will run as long as it satisfies the above condition.
The iteration depends on how many times m is doubled until it exceeds n. Which give the time complexity of the while loop as $\log_2 n$

∴ Total Time Complexity = Outer for Loop * Inner for loop * While loop

$$= n \times n \times \log_2 n$$

$$= n^2 \log_2 n$$

∴ Time Complexity $= O(n^2 . \log_2 n)$