Problem set 2.1

Q.1

Solution: Consider the pseudocode for x^y.

```
Power_bitwise(x, y)
1. result = 1
2. While 'y' greater than 0.
3.     if AND operation (y, 1) is 1
           result = result * x
4.
       x = x * x
5.
       y = y >> 1 (Right shift operation).
6.
7. return result.
```

We loop through each bit of 'y' (in Binary). At line 3; we do 'AND' operation to check whether it is odd (or) even. if it set to 1. we are making result multiplied x. Now we have one multiplication done and remaining with (y-1) operations.

but when we are shifting bit until we traveled to last bit in line '6' (Right shift operation).

Time Complexity: As we are looping through each bit of 'y' which has 'n' bits. so the time complexity is $O(n)$

Question 2:-

Solution

Given $g(n) = 1 + c + c^2 + \cdots + c^n$

This is in the form of sum of increasing powers.

so $g(n) = \frac{1 - c^{n+1}}{1 - c}$

(a) if $c < 1$;

then $\frac{1 - c^{n+1}}{1 - c} < 1$

$(1 - c) < (1 - c^{n+1}) < 1$

Dividing with $(1 - c)$

$1 < \frac{1 - c^{n+1}}{1 - c} < \frac{1}{1 - c}$

$1 < g(n) < \frac{1}{(1 - c)}$

if we keep any value below 1 at 'c'; bounded to constant. so

$g(n) = \Theta(1)$.

(b) if $c = 1$: $g(n) = 1 + 1 + (1)^2 + \cdots (1)^n \Rightarrow \underline{1 * n} \Rightarrow \underline{n}$

so $g(n)$ is same as 'n' value at any point of 'n'.

so $g(n) = \Theta(n)$.

(c) if $c > 1$; $g(n) = \frac{c^{n+1} - 1}{c - 1} \Rightarrow c^{n+1} - 1 > c - 1$

so, $\Rightarrow (c^{n+1}) > (c^{n+1} - 1) > (c - 1)$

$\Rightarrow c^{n+1} > c^{n+1} - 1 > c^n$

dividing with $(c - 1)$

$$\frac{c^n \cdot c}{c-1} > \frac{c^{n+1}-1}{c-1} > \frac{c^n}{c-1}$$

$$\frac{c^n}{c-1} < \frac{c^{n+1}-1}{c-1} < c \cdot \frac{c^n}{c-1}$$

$$\frac{c^n}{c-1} < g(n) < c \cdot \frac{c^n}{c-1}$$

So, $g(n)$ grows exponentially when ~~A~~ values increases

So, $g(n) = \theta(c^n)$

Question: 3

(a) - upper bound $\Rightarrow O(f(n))$

As it has double "for" loop. the outer loop iterate over 'i' from 'i' to 'n'. The inner loop iterate over 'j' from 'i+1' to n.

↳ Total number of operations

$$T(n) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} (j-i+1)$$

The summation is roughly quadratic, as the number of terms being summed grows as $(n-i)$

Total running time

$$T(n) = O(n^3)$$

Thus $f(n) = n^3$, and the running time is $O(n^3)$.

(b) Lower bound $\Omega(n^3) \Rightarrow \Omega(f(n))$

To show algorithm run-time is $\Omega(n^3)$, we need to show that it takes at least n^3 time in the worst case.

Outer loop → 'n' times

Inner for loop → "n−i" times.

Summation $A[i] + A[i+1] \cdots + A[j] \Rightarrow (j-i+1)$ summations.

- There are $O(n^2)$ pairs (i,j) in total
- For each pair, summing elements between 'i' and 'j' takes $O(j-i)$ growth cubically.

Thus, the overall time complexity $\Omega(n^3)$

we have both $O(n^3)$ and $\Omega(n^3)$, so, it's running time is $\Theta(n^3)$.

(c) we need an algorithm with running time $O(g(n))$

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

we have $f(n) = n^3$

we need $g(n)$ to be atleast n^2

so, we need to find algorithm that runs in $O(n^2)$ time.

So, the algorithm can be written like below:-

```
Compute_sum_Array(A):
(1). Initialize pref_sum array with size of length of A.
(2). for K in (1, length of (A)+1):                          } O(n)
(3)      prefix_sum[K] = prefix_sum[K-1] + A[K-1]            }

(4) Initialize 2D_array Result.
(5) For i in (0, length(A)):                                 } O(n^2)
(6)      For j in (i+1, n):                                  }
(7)          Result[i][j] = prefix_sum[j+1] - P[i]           }
(8) return B
```

(2) — O(n) time
(5) — n times
(6) — 'n-i' times
(7) — O(1) time

Therefore, the total time complexity is $O(n^2)$ which is significantly better than original $O(n^3)$

Complexity Analysis:-

step: 1	O(1) time
step: 2	O(n) time
step: 3	O(1) time
step: 4	O(n) time
step: 5	$O(n^2)$ times
step: 6	$O(n^2)$ times

The dominant step is $O(n^2)$;

So, the complexity is $O(n^2)$