Q.4 Ans

```
def Count Maximal (set of points in list L):    // i write def count Maximal.
    M = compute the Median of x coordinates of L
a)  if list L is NULL
        return 0.
    LL = points in L with x-coordinates < M
    LR = points in L with x-coordinates >= M
b)  y_max = maximum y-cordinates of any point in LR.
    remove points in LL whose y-coordinates is less-than y_max.

    x1 = CountMaximal (LL)
    x2 = Count Maximal (LR)

    return x1 + x2.
```

c) Use linear time median + linear scan to implement function.

$T(n) <= 2T(n/2) + O(n)$.

by using master theorem,

$T(n) = O(n \log n)$.

Q.2

Ans:
```
def FindHC (graph H):
a) // base case.
   if given graph has only one node
      return NULL
   if given graph has no any edge
      return NULL      // isolated vertex in graph).
   if given graph has only one edge
      return NULL.
b) to store the all the vertices that belongs to Hamiltonian
   cycle made a list here. H[]// H is a list.

   made a exact duplicate or copy of graph H.
for edge e in H:
   // there is a Hamil. cycle in H after removing the edge e.
   if Has HC (H-{e} returns true:
 c)
   • we remove that edge & store the number of edges
   z= need to made a Hamiltonian cycle

   • we ensure that and check that if we consider that
   y= edge then further how many edges required.

   • if z need less edge then we ~~stored~~ stored in list
     H. otherwise, store the y in the list H[]

else:

     no-op.

d) return H[]  // H is the list.
```

=) so, that's how it says ~~the~~ & ~~check~~ store k vertices to identify & determines the hamiltonian cycle.

Q.3

Ans:-

APPROX = solution for approximate algorithm,

OPT = optimal solution.

→ for r-absolute algorithm, 0-1 knapsack is the maximization problem.

→ Either we choose item fully or leave it as it is, we can't do fraction of in this particular problem.

→ let's suppose knapsack is NP, then proof is the set S of items that are choosen and the verification process is compute $\sum_{i \in S} s_i$ & $\sum_{i \in S} v_i$ ($S \subseteq \{1, 2, \dots n\}$. n item with size s_i.

→ so, Is there a subset $S \subseteq \{1, 2, \dots, n\}$ such that $\sum_{i \in S} s_i < B$

& $\sum_{i \in S} v_i \geq V$ (B = capacity. V = value. $(v_1, v_2, v_n \dots\}$.

→ Construct the poly. time r-abso. algo. for given problem shows the contradiction.

→ we will show that proof by contradiction. Assume there is any algorithm X with P positive integer.

K = instance K of knapsack, K' = we will construct the instance K'. such that $w_i'' = w_i$, value $v_i = (r+1)v_i'$.

→ values are space up in K'. such that every solution in K is feasible in K'

→ we can compute

$r \geq |X(K') - OPT(K'|$

$\therefore r \geq |(r+1)f(\beta) - (r+1)\,OPT(r)|$

$\therefore \frac{r}{r+1} \geq |f(\beta) - OPT(K)|$

$\therefore 0 \geq |f(\beta) - OPT(K)|$ ⇒ we assume that algo X with solution in P time.

(∵ β = solution for K.

→ so, it is contradiction that our observation & assumption wrong. (contradict the solution).

→ for any integer r, it's not possible to design poly-time r-absolute appro. algorithm for 0-1 knapsack.

Q.6

Ans:

a) LP (T, PS, n, S) :-

T = task, PS = current slot, n = total slot, S = Penalty.

=> So, it calculate the every possibility by recursion to fullfill the criteria & we use sub-problem for the overlapping & used DP-based approach to store the intermediate result. that's how we can improve time complexity.

b) use 2D-array arr[][].

```
arr[2][n] = -1
    LP (T, PS, n, S)
{
    if (PS == n) then ~~arr[[T][PS~~ arr[T][PS] = x[T][PS];
        return x[T][PS]
}
if (LP[T][PS+1]! = -1) than x = arr[T][PS+1];
else   z = ~~LP[1-T][PS+1]~~; LP(T, PS+1, n, S);
if (arr[1-T][PS+1]! = -1) then y = dp[1-T][PS+1];
else.
      z = arr[1-T, PS+1, n, S);
      return arr[T][S];
```

"I don't know"

c)

"I don't know".

d). Time-complexity = O(n)

→ for correctness, we can say that we check every possibility ~~for~~ optimal solution. & it gives the correct result.

Q. 1

" I don't know"

Q. 5

" I don't know".