Problem 1:

Here we mantain a array and update The arrays element by the following rule.

Let Arr be Input array.

We Create Dar array. by initializing

$Dar[0] = Arr[0]$

Updation rule

① if $Arr[i] \geq Dar[-1]$

$Dar[-1] = Arr[i]$

[Dar[-1] = Last element of Dar]

② If $Arr[i] < Dar[-1]$

Binary search on Dar an find

if: I such that

$Dar[I] \leq Arr[i] \leq Dar[I+1]$

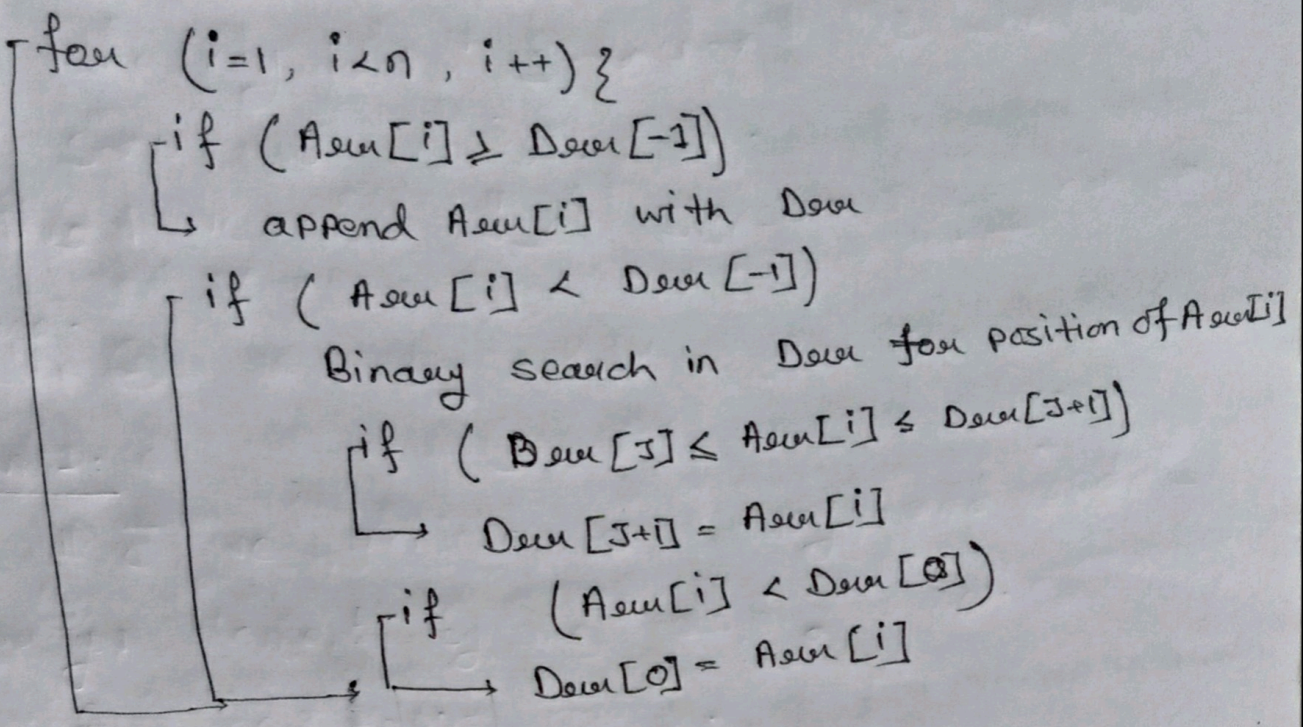
then $Dar[I+1] = Arr[i]$

③ If : $Arr[i] < Dar[0]$

then $Dar[0] = Arr[i]$

Psudo Code:

Input Arr:
Output: length of longest increasing subsequence.
Initialize: Dar (array) with Dar[0] = Arr[0]



S= len(Dar)
Return S;

Time Complexcity:

First for loop takes $O(n)$ time and binary search takes $O(\log n)$ time. So in total that will take $O(n \log n)$ time.

Correctness:

in first if Condition (Arr[i] ≥ Dar[-1]) we take new input to the array. [So Dar is in increasing order]

Also we use binary search to find position of the ~~arr~~ in Dar if that Arr[i] is less than highest element. In this step we update Dar[J+1]. by this we did not change the length but increases the possible count.

Demonistration:

Arr = | 1 | 2 | 4 | 3 | 7 | 8 | 6 | 0 |

Dem = | x | 2 | 4 | x | 8 | | |

0 3 6

So Dem = | 0 | 2 | 3 | 6 | 8 | |

The final Dem does not give correct order but gives correct length.

Problem 2:-

(a) Psudo code Initialize:- array G, V, W

Input: Intervals and weights.

output: minimized sum of weight of Non overlap intervals

① Sort $(f_1, \dots f_n)$ in increasing order.

• $G = (g_1, \dots g_n)$ be the sort of $(f_1, \dots f_n)$

② for (i=1 to n){
  for(J=1 to n , J≠i):
    P(i,J) = 0 if the intervals overlap
           = 1 if the intervals not overlap.
}

③ $V = (v_1, \dots, v_n)$ sorted weight according to G or weight of $g_i = v_i$

W = V (Initial)

for (i = 1, to n){
  for (J=0, J<i, J++){
    if (P(i,J) ≠ 0 & • W[i] < W[J] + V[i])
      W[i] = W[J] + V[i]
  }
}

④ find maximum element of W array.

Time Complexcity:

① step ① takes $O(n \log n)$ time
② Step ② " $O(n^2)$ "
③ In step ③ two for loop takes $O(n^2)$ time.
④ in step ④ takes $O(n)$ time

So in total it takes $O(n^2)$ time.

Correctness:

Claim: At ith step of iteration the maximum of $W[:i-1]$ is maximum weighted sum that can be done with out over laping interval using i interval (sorted)

→ Here we use induction on i

for $i=1$, $j=0$

so $W[1] = W[1] + V[0]$ if interval i and j does not overlap. otherwise no change.

So this is correct for $i=1$

Now we assume true for i.

in $i+1$ step we over away W is updated. accordingly

Through ith step.

meang $W[0], W[1] \cdots W[i]$ are correct weight

if we done upto ith step.

Now we update $W[i]$ if

$W[i+1] < W[j] + V[i+1]$ if $p(i,j) \neq 0$.

That is this is true for $i+1$ also.

Thus by induction this is true for all n.

Problem 3:

In this case we use longest Common subseaquence by taking input seavence and reverse of input seaquence.

Psudo code:

Input: Seavence H

output: Palindeom

G = Reverse of H

Print-LCS (G, H)

where LCS is longest Common subseavence. finding length algorithm. and Print-LCS is finding the seavence.

Psudo code of LCS

m = len (H)

n = len (H)

for Initialize C[i,0] = 0 for i = 1, n

and C[0,J] = 0 for J = 1, to n,

```
for (i = 1 to n) {
    for (J = 1 to n)
        if xi = Yi
            C[i,J] = C[i-1, J-1] + 1
            B[i,J] = "K"
        else if C[i-1, J] ≥ C[i, J-1]
            C[i,J] = C[i+1, J] + 1
        else
            C[i,J] = C[i, J-1]
            B[i,J] = "L"
```

```
Print-LCS(B, X, i, J)
    if
        B[i, J] = 'K'
            Print-LCS(B, X, i-1, J-1)
            Print (Xi)
    else if
        B[i, J] = 'K'
            Print-LCS(B, X, i-1, J)
    else
            Print-LCS(B, X, i, J-1)
```

Time Complexcity:

In total this LCS and Print LCS will take $O(n^2)$ time.

So and calculation of reverse of a seqvence takes $O(n)$ time.

So in total this takes $O(n^2)$ time.

Correctness:

In palindrom seqvence, the seqvence is same if we reverse the seqvence. That is by taking the reverse of a seqvence then finding Common seqvence is same as finding palindrom.

(4) Yes Dijkstra's algorithm can be use in this case.

⇒ In case of Diskstra's algorithm we first Initialize the neighbourhood weigh in a array. Then update the distance accordingly.

Main ide of diskstra algorithm is that at any step, if we know the shortest distance of u. and v is in the neighbourhood of u.

Then we update $d[v] = \min_{u \in Nghb\ v} (d(u) + w(u,v))$

Here negative weight are only in the outgoing edges of s, where s is the source node.

So weights are initialize at the first step.

Then the algorithm goes as it is. and it does not harm the key fact $s \rightsquigarrow u \rightsquigarrow t$ is shortest path than $s \rightsquigarrow u$ also shortest path.

Thus we can use Dijkstra algorithm in this case.

Problem 5:

Psudo code:

Input-S. H and G;

Output distance from

Problem 5:

Psudo code:

Input:- H, G, S and t.

Output:- Shortest distance.

① Run Dijkstra on H, source s.

② Reverse edges of H.

③ Run Dijkstra on ~~H~~ Reverse H from t (source)

④ Initialize K=0 Inf

```
for (u,v) in E{ [edges in G]
    K = d(s,u) + w(u,v) + d1(t,v)
    if S ≤ K{
        S = K
    }
}
```

where $d(s,u)$ is the return from step ①
$d_1(s,u)$ is " " " " ③

Time Complexcity:

① step ①,③ takes $O(|E| \log |V|$

② Step ② take $O(|E|)$ time.

④ Step ④ take $O(|E|)$ time.

In total these will take $O(|E| \log |V|)$ time.

Correctness:

Suppose our shortest path is joining (u, v) which is not in E but in G.

Then our path become $s \rightsquigarrow u \rightarrow v \rightsquigarrow t$.

Here u and v must be in H otherwise The system not possible.

Since this $s \rightsquigarrow u \rightarrow v \rightsquigarrow t$ shortest path.

$s \rightsquigarrow u$ and $v \rightsquigarrow t$ also shortest path.

in step (1) we use dijkstra algo theorem to find Single source all node (In H) shortest path.

Also finding shortest path from $v \rightsquigarrow t$ is as similar as reversing edges then finding distance from t to v.