ASSIGNMENT 4 (RITIKA GOYAL) (301401516)

Ques1:

<u>Structure of optimal solution:</u>

For an ordered digraph G,

V(G)={v1, v2, v3, v4, …….., v_n}

Let E(G) be the set of all edges.

To calculate the longest path from v_i to v_n, firstly all the other nodes are found that v_i is connected and then the problem is subdivided by removing one edge that connected v_i to other edges and find the longest path from each of those edges to the v_n.

<u>Bellman equation:</u>

Longest_path(v_n) = { 0 when i = n;

max (Longest_path(v_i) + 1) for every i such that (v_i, v_n) belongs to E }

<u>Pseudo code:</u>

```
Longest_path(E, i, n)
    Let S be the array of size n which stores the longest path from v1 to vn initialised to 0
    if i = n  then return 0;
        for j = i+1 to n
            if (vi, vj) belongs to E then
                S[i] = max ( Longest_path(E,j,n) +1, S[i]);
    return S[i];
```

Ques2:

<u>Structure of optimal solution:</u>

To compute maximum number of steps, this problem can be subdivided into smaller problem and can be solved recursively. Firstly, the total possibilities of choosing next step by each node that is each a_i and each b_i is calculated through recursion following the total value of the maximum number of steps of task that can be executed on both machines.

- Firstly, the next possible steps for each a_i and b_i is computed including the possibility of switching between two machines.
- Then the maximum value is calculating for each a_i following the maximum value of a_{i+1} and so on.

Bellman equation:

If a job is executing on a machine, it has two possibilities of either remain on the same machine and execute next steps, or skip the maximum steps and switch the machine to get the more steps from other machine. Let the plan to store the solution is stored in an array temp of length of total time interval and t[i] is equal to a, b and s if job is executed on machine a, b and skipped respectively for i^{th} time interval.

Let S be an array to which stores the maximum number of steps at i^{th} interval.

Max_number_of_steps(i) = { a_i (if job is executing on machine a) or b_i (if job is executing on machine b) for i = n;

S[i] + max(Max_number_of_step(x)) for all x that can be the next option for current step to choose;

}

Pseudo code:

Input: I is current task, n= total number of tasks, temp array to store solution and job_location for tracking current machine.

```
Max_number_of_steps(i,n, arr, job_location)
    if i = n then return m[i]
    max_steps1 = Max_number_of_steps(i+1, n, arr, job_location)
    if i not equal to n-2 then
        S[i] = max_steps1
    else
        job_location = switch(job_loation);
        max_steps2 = Max_number_of_steps (i+2, n , arr, job_location)
        S[i] = S[i] + max(max_steps1, max_steps2);
    If(max(max_steps1,max_steps2) = nax_steps1) then
        job_location = a;
        temp[i] = a;
    else
        job_location = b;
        temp = b;
    return S[i];
```

Ques3:

Matrix e[i,j]:

i-> j ↓	0	1	2	3	4	5	6	7
1	0.06	0.28	0.62	1.02	1.34	1.83	2.44	3.12
2		0.06	0.30	0.68	0.93	1.41	1.96	2.61
3			0.06	0.32	0.57	1.04	1.48	2.13
4				0.06	0.24	0.57	1.01	1.55
5					0.05	0.30	0.72	1.20
6						0.05	0.32	0.78
7							0.05	0.34
8								0.05

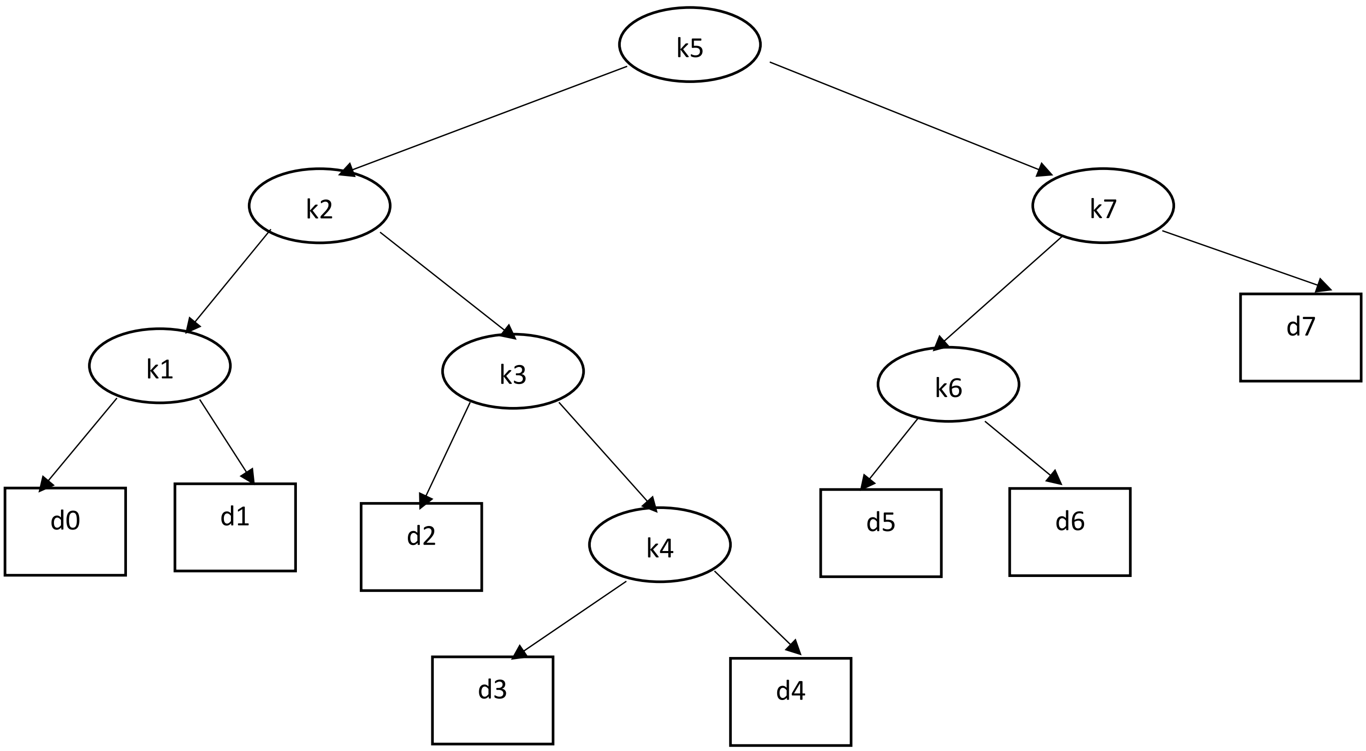
Weight w[i,j]

i-> j ↓	0	1	2	3	4	5	6	7
1	0.06	0.16	0.28	0.42	0.49	0.64	0.81	1.00
2		0.06	0.18	0.32	0.39	0.54	0.71	0.90
3			0.06	0.20	0.27	0.42	0.59	0.78
4				0.06	0.13	0.28	0.45	0.64
5					0.05	0.20	0.37	0.56
6						0.05	0.22	0.41
7							0.05	0.24
8								0.05

Root[i,j]:

i-> j ↓	1	2	3	4	5	6	7
1	1	2	2	2	3	3	5
2		2	3	3	3	3	5
3			3	3	4	4	5
4				4	5	4	6
5					5	6	6
6						6	7
7							7

Tree structure: root at k5



Cost table:

Node	Depth	Probability	Contribution
k1	2	0.04	0.12
k2	1	0.06	0.12
k3	2	0.08	0.24
k4	3	0.02	0.08
k5	0	0.10	0.10
k6	2	0.12	0.36
k7	1	0.14	0.28
d0	3	0.06	0.24
d1	3	0.06	0.24
d2	3	0.06	0.24
d3	4	0.06	0.30
d4	4	0.05	0.25
d5	3	0.05	0.20
d6	3	0.05	0.20
d7	2	0.05	0.15
Total			3.12

Ques4:

Let free_resources be an array which stores all the resources that are not in use and busy_resources be an array which stores all the resources that are being used. When any resource is occupies at start time s_i, then that resource is removed from free_resources and added in busy_resources. When an activity is finished at time f_i, the resource is removed form busy_resource and added to free_resource. When there is an activity that requires a resource and free_resources are empty, then an additional new resource is added by appending the free_resource to make room for new activity.

<u>Pseudocode:</u>

```
Greedy_scheduling_algo (s,f)
        Sort(s);    /* Sort the resources by increasing start time*/
        Busy_resource = 0;
        For i=1 to n
                Busy_resources = empty
                For j = 1 to i-1
                        Busy_resources.append(0)
                        If [si , fi) ∩ [sj , fj ) is not equal to ϕ        then
                                Busy_resources[i]++;
        Return Busy_resources.size();
```

Optimal: It is seen that algorithm never assigns the same resource to two conflicting activities because the inner loop always checks the conflicting conditions and eliminates the resource if conflicting. When an activity conflicts with already running activity, then it is given a resource of its own it means there are I activities running together. At the end, when all the activities are checked and returned the number of subset of mutually compatible activities.

Running time: since there are two for loops in this algorithm which can go maximum to n, thus the running time is: $O(n^2) + O(1) = O(n^2)$.

Ques5:

<u>Pseudocode:</u>

Input: Let S be a array of size n to store ratios of v_i and w_i that is v_i/w_i

```
Greedy-Fractional-knapsack(n,W,S)
        Let Solution be a new array which is returned with the solution.
        Median = S[ceil(n/2)];
        Let S1, S2, S3 be two new arrays and W1, W2 and W3 be variables to store weight
        For i=1 to n
                if S[i] > median then
```

```
                S1.append(S[i]) and W1 = W1 + wi ;
        Else if S[i] = median then
                S2.append(S[i]) and W2 = W2 + wi;
        Else
                S3.append(S[i]) and W3 = W3 + wi;

If W1>W then
        Return Greedy-Fractional-knapsack(S1.size(), W, S1)
Else
        While (S2 is not empty and knapsack is not full)
                Solution = Solution + S2

        if (knapsack is full)
                Solution = S1 + S2
                Returns Solution;
        Else
                W = W1 + W2;
                Return Greedy-Fractional-knapsack(S3.size(), W, S3)
```

Correctness:
Loop invariant: Before every loop iteration, the total weight of knapsack does not increase W.
Initialisation: Initially the knapsack is empty so total weight of knapsack is 0 which is less that W thus loop invariant is satisfied.
Maintenance: Every time while adding an item in knapsack, it is checked that combined value of weights does not exceed W and if it does, the item is not added in knapsack.
Termination: When the total weight of knapsack is equal to W then the algorithm returns the set of al elements.

Running time: Since there is one for loop from 1to n and one while loop which iterates on less than n elements and the rest is recursive calls, the running time is: O(n) + T(n/2) = O(n)

Ques6:

a) Aggregate analysis:
   Let cost of i^{th} operation be c_i.
   if(i is exact power of 2)

   $c_i = i$

   else

   $c_i = 1$

   Cost of n operations:

$\sum_{i=1}^{n} ci = \sum_{j=1}^{\lg n} 2^j + \sum_{i \leq n \text{ not power of } 2} 1$

$\leq 2^{1 + \text{ceiling}(\log(n))} - 1 + n$

$\leq 4n - 1 + n$

$\leq 5n$

Average cost of operation = Total cost / number of operations

= 5n / n

By aggregate analysis, the amortised cost per operation is O(1).

b) Accounting method of analysis:

Let the charge for each operation is \$3.

Then if i is not a power of 2, pay \$1 and store \$2 as credit .

If i is a power of 2, pay \$i with credit

Operation	Cost	Actual Cost	Credit
1	3	1	2
2	3	2	3
3	3	1	5
4	3	4	4
5	3	1	6

The amortized cost of one operation = \$3, the sum of all amortized c_i = 3n.

Every 2^{i-1} operation has cost 1 and has credit 2 which gives total of 2^{i+1} -2 credit.

2^{i+1} has actual cost of 2^{i+1} and gives credit of 2^{i+1} +1.

When combined, it leaves 1 credit => amount of credit is never negative.

Since the amortized cost of each operation is O(1), and amount of credit never goes negative, the upper bound on total cost of n operations is O(n).

c) Potential method:

Let $\phi(D_i) = k+3$ if $i = 2^k$

Potential function is defined as follows:

$\phi(D_0) = 0$ and, $\phi(D_i) = \phi(D_{2^k}) + 2(i - 2^k)$

The potential is always nonnegative, so $\phi(D_i) \geq 0$

Potential difference:

(if i is not power of 2)

$\phi(D_i) - \phi(D_{i-1}) = 2$

else

$$\phi(D_i) - \phi(D_{i-1}) = -2^k + 3$$

Total amortized cost of n operation $= \sum_{i=1}^{n} ci$

$= \sum_{i=1}^{n} 3$

$=3n$

$= O(n)$

Ques 7:

INSERT(S,x)

The insert(S,x) add the element x to the end of array S. If before insertion, S is full then a new temporary array is made whose size is twice the size of S. Then all elements are copied from S to the temporary array, then make S equal to new array.

The running time of Insertion would be O(1) for inserting one element in array as it take constant time. To insert m elements, its running time will be O(m).

DELETE-LARGER-HALF(S)

DELETE-LARGER-HALF(S) deletes the largest half array. Firstly all the elements are found whose order is ceiling(|S|/2) by calling RANDOMIZES-SELECT function. The elements who are smaller than max is copied in a new temporary array whose size is half the size of S and then make S equal to temporary array.

The running time of Delete-Larger-Half is affected by RANDOMIZES-SELECTION and copying of the elements. The RANDOMIZES-SELECTION takes O(m) time and copying takes O(m/2) time so running time of delete-larger-half is O(m).

Output of S: To output the element of array S, The S can be iterated using any loop and then output each element. Since the looping condition is not greater than size of S which gives the running complexity in O(|S|) time.