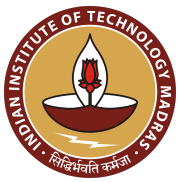


Indian Institute of Technology, Madras

Department of Computer Science & Engineering

CS2800 Course Assignment

Assignment 4
Greedy Approach

Akshat Singh - CS18B001
B Abhijit - CS18B004
B V S Sudheendra - CS18B006
Archana Kutumbaka - CS18B023
Naman Mohnot - CS18B030
Venu Madhava Reddy - CS18B051
Meesa Shivaram Prasad - CS18B056
Ritvik Rishi - CS18B057
Vishav Rakesh Vig - CS18B062
Yuvraj Singh - CS18B064
Nischith Shadagopan M N - CS18B102
Shreesha G Bhat - CS18B103
Prateek Mishra - BS15B024

April 1, 2020

Problem 1

Explanation

Assuming we have an empty tank initially and the distance between point A and point B is D in kilometers, the minimum fuel required F_{min} in liters to reach B from A is

$$F_{min} = F \times D$$

where F is the rate of fuel consumption in litres/kilometer. Now, given the rate of refuelling is 'r' in litres/minute, the minimum time spent for refuelling T_{min} in minutes is

$$T_{min} = \frac{F_{min}}{r}$$

Therefore, any optimal solution should give the same result as above. Now, considering the second algorithm:
Now consider a gas station x_i where we stop for refuelling. The destination B can be reached using less fuel than the capacity 'c' of the tank. But according to the second algorithm, we refuel to the full capacity and therefore wasting time. Hence, the second algorithm cannot be the most optimal one.

Now, consider the first algorithm. In this greedy approach, we try to minimize the fuel filled by filling just enough to reach the next station. Using this approach, we fill just enough fuel to cover the total distance.

Let f_i be the minimum fuel required to travel between the i^{th} and the $(i+1)^{th}$ station for $1 \leq i \leq (n-1)$.

Then $sum(\frac{f_1}{r}, \frac{f_2}{r}, ..., \frac{f_{n-1}}{r}) = \frac{F_{min}}{r} = T_{min}$.

Hence the greedy algorithm (a) gives the optimal result.

Problem 2

Q2)

a) It is easy to see that this solution is wrong. I have constructed a counter example for the same.

Let $A = \{0, 0.8, 0.9, 1, 1.1, 1.2, 2\}$

So, by the above algorithm we would choose a set containing all of $\{0.8, 0.9, 1, 1.1, 1.2\}$ which cannot contain $\{0\}$ or $\{2\}$. So, once this is the first interval chosen, we need two more intervals to cover '0' and '2'. So, going by this algorithm, we would need 3 intervals. But consider the following two intervals: $[0, 1]$ and $[1, 2]$. They are able to cover the points.

Hence, the above mentioned algorithm is incorrect.

2 b)

Proof of correctness:

To see how our solution satisfies the "greedy" choice property", let us see how there always exists an optimal set of intervals that contain the interval $[x_1, x_1+1]$ where x_1 is the left most element among those that need to be covered.

Assume on the contrary that there is no such optimal set. And now, let the left most interval be $[a, a+1]$. Now, a can't be greater than x_1, since x_1 needs to be covered. Also, a can't be x_1, by our assumption. So, $a < x_1$.

Now, consider the interval $[a, x_1)$. We might as well remove this interval since there are no points here (by assumption, x_1 is the left most point). So, even if we replace $[a, a+1]$ with $[x_1, x_1+1]$ all points are still covered and number of intervals is the same.

Hence, proved that there exists an optimal set of intervals containing $[x_1, x_1+1]$ where x_1 is the leftmost element.

This way an optimal solution can be built greedily by solving the subproblem removing all points already covered by $[x_i, x_i+1]$. Let's call the optimal solution for the subproblem S'. The optimal solution for the whole problem would be $[x_i, x_i+1] \cup S'$. So, by this recursively defined greedy algorithm, we can solve the problem in $O(n \log n)$ time.

Note: Here, sorting is the dominating factor, hence the complexity of $O(n \log n)$.

Problem 3

(a)

No matter which line the i^{th} word is present in, it contributes $w_i \times L$ to the total penalty. When we count total penalty this way we get

total penalty $= \sum_{i=1}^{i=n} w_i \times L$

total penalty $= L \times \sum_{i=1}^{i=n} w_i$

Therefore total penalty is independent of the way in which way we layout them. Therefore vacuously, the given greedy algorithm also solves this problem correctly.

(b)

We disprove that the given greedy algorithm correctly solves the problem with a counter example. Consider

$words = \{aa, aa, aaa\}$

$L = 4$

Layout obtained by applying the given algorithm is:

aaaa
aaa

This layout has a total penalty of 4.

Now consider the layout

aa
aa
aaa

This has a total penalty of 3 which is lesser than the total penalty previously obtained. Therefore this disproves that the given greedy algorithm correctly solves the problem.

Problem 4

(a)

FASLE.
Consider the case when we have 3 skiers, with heights as 5, 9, 1 and the ski heights as 2, 3, 6, respectively. According to this algorithm, we assign the skier with height 5, a ski of height 6, since for that skier, the minimum height difference comes from the ski of height 6. Now for the next skier, we will assign the ski of height 3, as 9 is closest to 3 in the remaining ski's. Therefore shier with height 1 gets the ski of height 2.
Calculating the summation, we have SUM-1 $= |5-6| + |9-3| + |1-2| \Rightarrow$ SUM-1 $= 8$.
However, consider the case when we assign the skier with height 1 to ski of height 2, the skier of height 5 a ski of height 3, and the skier of height 9 a ski of height 6. In this case, we have the sum to be :-
SUM-2 $= |1-2| + |5-3| + |9-6| = 6$. Clearly this sum is less that the sum produced by the algorithm given and therefore this algorithm is false
Hence Proved.

(b)

Without loss of generality, let us assume that the input $p_i^{'s}$ are sorted in increasing order, and so are $s_i^{'s}$. Consider i_1^{th} skier to be allotted the j_1^{th} ski, and the i_2^{th} skier is allotted the j_2^{th} ski. If for every i_1 and i_2, such that $i_1 \leq i_2$ (Which means that $p_{j_1} \leq p_{j_2}$ since they are in sorted order), we have $j_1 \leq j_2$(Which means that $s_{j_1} \leq s_{j_2}$ since they are in sorted order) then this case is what is produces by the algorithm given in the question, since the shortest of all the skier's will get the shortest of all the ski's and the second shortest skier will get the second shortest sk and so on .. .
However let us assume that there exists another correct solution to the above problem. We will now show that we can convert the aforementioned solution into a solution generated by our greedy algorithm by undergoing exchanges that at no step worsen the quality of the solution.
In that solution let us consider $p_{i_1} \leq p_{i_2}$, which are allotted s_{j_1}, s_{j_2}. If $j_1 \leq j_2$, then continue to the next pair ,else if $j_1 > j_2$:-
In this case we have the sum $|p_{i_1} - s_{\alpha_{i_2}}| + |p_{i_2} - s_{\alpha_{i_2}}| = |p_{i_1} - s_{j_1}| + |p_{i_2} - s_{j_2}|$.
Let $p_{j_1} = a, p_{j_2} = d, s_{j_1} = c, s_{j_2} = b$, hence $a < d, b < c$. Therefore, we have this sum to be equal to $|a-c| + |b-d|$.
Let $b = a + X$, and $c = d - Y$.

$$b = a + X, \quad c = d - Y$$
$$a - c + X = b - d + Y$$
$$c - a = d - a - Y$$
$$d - b = d - a - X$$
$$|a - c| + |b - d| = |d - a - Y| + |d - a - X|$$
$$|a - b| + |c - d| = |X| + |Y|$$
$$b - c = a - d + X + Y$$
$$X + Y = (d - a) + (b - c)$$

Case-1: X<0 and Y<0

$$|a - c| + |b - d| = |d - a + |Y|| + |d - a + |X||$$
$$|a - c| + |b - d| >= |Y| + |X| \quad (\text{Since } d > a).$$
$$|a - c| + |b - d| >= |a - b| + |c - d| \qquad (1).$$

Case-2: X<0 and Y>0

$$|b - d| = |a - b| + |d - c| + |a - c|.$$
$$|a - c| + |b - d| >= |a - b| + |c - d| \qquad (2).$$

Case-3: X>0 and Y<0

$$|a - c| = |a - b| + |c - d| + |d - b|.$$
$$|a - c| + |b - d| >= |a - b| + |c - d| \qquad (3).$$

Case-4: X>0 and Y>0

$$|a - c| >= |a - b|.$$
$$|b - d| >= |c - d|.$$
$$|a - c| + |b - d| >= |a - b| + |c - d| \qquad (4).$$

From $(1), (2), (3), (4)$, we have that $|a - c| + |b - d| \geq |a - b| + |c - d|$. This means that if we assign s_{j_2} to p_{i_1} and s_{j_1} to p_{i_2}, we get a sum which is lesser. Thus for any $p_{i_1} \leq p_{i_2}$, if they are given s_{j_1}, s_{j_2} respectively such that $s_{j_1} > s_{j_2}$, then swap them such that p_{i_1}(The smaller height skier) gets s_{j_2} i.e, lesser of the two and p_{i_2}(The taller among the two) gets s_{j_1} i.e, the longer of the ski's. Now compare any two other pairs of skiers and apply the same swapping procedure. Clearly we are moving towards the solution given by the greedy algorithm, nowhere worsening the quality of the solution but only making it either better or keeping it the same. Thus following these swaps we move towards the solution produced by the greedy algorithm given in the question, and therefore our greedy algorithm is a correct algorithm by itself.

Problem 5

INPUT: A collection of jobs $J_1, \cdots, J_n$. The size of job J_i is x_i, which is a non-negative integer. An integer m.
OUTPUT: A nonpreemptive feasible schedule for these jobs on m processor that minimizes the total n completion time $\sum_{i=1}^{n} C_i$
A schedule specifies for each unit time interval and for each processor, the unique job that that is run during that time interval on that processor. In a feasible schedule, every job J_i has to be run for exactly x_i time units after time 0. In a nonpreemptive schedule, once a job starts running on a particular processor, it has to be run to completion on that particular processor. The completion time C_i for job J_i is the earliest time when J_i has been run for x_i time units.
Variables Used :

```
1  // X is vector of pairs where X[i].first=x_i from input and X[i].second=i
2  // When we perform the step of reducing the time remaining for a job in the algorithm we do X[i].first = X[i].first -
3  // When we say print in the algorithm print X[i].second
4  // n is the number of jobs.
5  // m is the number of processors.
```

Algorithm:

JOB-SCHEDULE(X, n, m)

```
1  Sort(X)
2  for each unit time
3      if Number of Uncompleted jobs >= m
4          Print first m uncompleted jobs from X
5          Reduce the time remaining for the above jobs by 1
6          if Time remaining for any job becomes 0
7              Mark that job as completed
8      elseif Number of Uncompleted jobs < m
9          Print all uncompleted Jobs
10         Reduce the time remaining for these jobs by 1
11         if Time remaining for any job becomes 0
12             Mark that job as completed
13     if All jobs are completed
14         break
```

Correctness:
Let A_l(l = 1,...,L $\leq$ n) be the set of jobs which are scheduled in position l from the end on their processor.
This means that A_1 is set of jobs scheduled last on the m processors and A_2 is the set of jobs scheduled second last on the m processors and so on.
L is the maximum number of jobs any processor will run.
If a job J_i is scheduled as k-last job on a machine i.e. $J_i \in A_k$ then this job contributes k*x_i to $\sum_{i=1}^{n} c_i$.
Consider the set A_1
Here the time of the jobs in this set is only added once i.e. we have m last positions where the processing time is weighted by 1.
Consider the set A_2
Here the time of the jobs in this set is added twice: once when these processes are being executed and once again when the processes in set A_1 will be executed i.e we have m second last positions where the processing time is weighted by 2
(Because this is how we defined C_i of each process)
and so on.
Now, we define

$$\sum D_j = \sum_{l=1}^{L} \sum_{J \in A_l} l x_j$$

$$\sum D_j = \sum_{i=1}^{n} C_i$$

We need to minimise $\sum D_j$
So it is optimal to place the m longest jobs last on their machines,the next longest m jobs one before last and so on.
In other words we assign the shortest jobs to the processors first and then as soon as a job gets completed we give the processor the next shortest uncompleted job that is not being executed.
Time-Complexity:
The time required to sort X is **O(n log n)**.
The for loop runs till all the jobs are completed. The last job is completed say at time T = max(C_i) where i = 1,2,...,n
We are printing m things inside this loop. The remaining time of each job can be decremented as soon as we print and it can be marked as completed in the same step. So, the time complexity of this for loop is **O(T * m)**.
Hence, the overall complexity of this greedy algorithm is
O(n log n + T * m) or **O(n log n + max(C_i) * m)** where i = 1,2,...,n.

Problem 6

Let us assume that there exists another correct solution to the above problem. We will now show that we can convert the aforementioned solution into a solution generated by our algorithm by undergoing exchanges that at no step worsen the quality of the solution.

Since we assume that the aforementioned solution is different than the one generated by our algorithm, it implies that there exist some cells x_{ij} in the matrix such that in our solution the values of these cells are different than the one by the aforementioned one.
Let us assume the first such occurrence happens in the i^{th} row. Therefore we have that in the i-1 rows preceding it the algorithm is followed.
Let the cell be x_{ij} such that the values assigned to it by the algorithm and the solution are different. Since the row sum r_i is constant it means that there also occurs another cell x_{ik} in the row such that its values are also opposite as given by the 2 solutions but they differ in respective values given by the algorithms i.e Our greedy algorithm gives 1 in one cell and 0 in another cell and the solution under consideration gives opposite values in both cell, such that the sum is still conserved.
Without loss of generality we can assume x_{ij} is assigned 0 in the solution and x_{ik} is assigned 1 by our greedy algorithm.

Exchange Argument:
Since our greedy algorithm allocates 1 to x_{ik} as opposed to x_{ij} we know that c_k-$a_k \geq c_j$-a_j .
This implies that no of 1's still to be filled in the k^{th} column after the i^{th} row is greater than equal to that in the j^{th} column.
But since the solution assigns a 1 to the j^{th} we now have that c_k-$a_k > c_j$-a_j. This means that there are more 1's yet to be filled in the k^{th} row as compared to the j^{th} row.
Now lets consider all the 1's in the k^{th} column after the i^{th} row. In all these rows containing a 1 in the k^{th} column there will exist at least some cells in the j^{th} column which have a 0 in the same row i.e for some $l > i$ the cell x_{lk} has a 1 and the cell x_{lj} has a 0 (Since c_k-$a_k > c_j$-a_j) .
Now we have i, l, j, k such that :
$x_{ij} = 1$;
$x_{ik} = 0$;
$x_{lj} = 0$;
$x_{lk} = 1$;
Now if we swap the values between x_{ij} and x_{lj} and similarly between x_{ik} and x_{lk} .We will still maintain the column sums as we are swapping only between the column cells, also since we are swapping a 1 and a 0 respectively with a 0 and a 1, the row sums will also remain constant i.e even after the swaps the sums are constant and satisfy the sum conditions as we assumed the solution is correct.
Now after the swaps:
$x_{ij} = 0$;
$x_{ik} = 1$;
$x_{lj} = 1$;
$x_{lk} = 0$;
Thus following these swaps we move towards the solution produced by the greedy algorithm.
Now following the same steps for all the swaps going row by row we modify the solution without worsening the quality of the solution that is it still satisfies all the sum invariants of the solution, hence we prove that our greedy algorithm is as good as an correct solution and hence is a correct algorithm in itself

Problem 7

1) given $s_1, s_2, s_3, \ldots, s_n$ are speeds

let $P_1, P_2, P_3, \ldots, P_n$ be the permutation of those speeds in increasing order

let $t_1, t_2, t_3, \ldots, t_n$ be the times taken by corrosponding P_i's .

algorithm:

$\exists\ j \in \{3, 4, \ldots, n\} \Rightarrow 2t_2 < t_1 + t_j$

and $\forall\ i < j \neq 1,2 \quad 2t_2 > t_1 + t_i$

For $t_3, t_4, \ldots, t_{j-1}$ we follow one process

$\xleftarrow[P_1, P_2]{t_2}$ P_1 and P_2 cross the bridge in t_2 time

$\xrightarrow{t_1}$ P_1 comes back

$\xleftarrow[P_1, P_3]{t_3}$ P_1 and P_3 cross the bridge

$\xrightarrow{t_1}$ P_1 comes back

⋮

$\xleftarrow[P_1, P_{j-1}]{t_{j-1}}$ P_1 and P_{j-1} cross the bridge

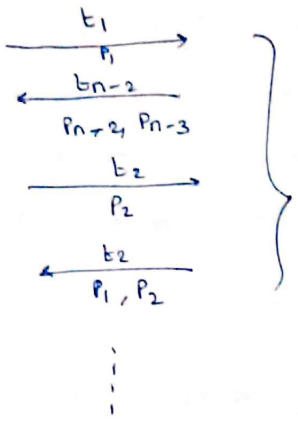
For $t_j, t_{j+1}, \ldots, t_n$ we follow another process

$\xrightarrow{t_1}$ P_1 comes back

$\xleftarrow[P_n, P_{n-1}]{t_n}$ P_n and P_{n-1} crosses bridge

$\xrightarrow{t_2}$ P_2 comes back

$\xleftarrow[P_1, P_2]{t_2}$ P_1 and P_2 crosses bridge



repeats with change at 2nd step [$t_n, t_{n-2}, t_{n-4}, \ldots$]

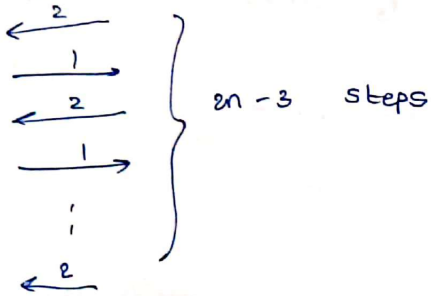
if n is even ends with $\xleftarrow{t_2}{P_1, P_2}$

n is odd ends with $\xleftarrow{t_j}{P_1, P_j}$

Proof of optimality:

Here for n we have to take $2n-3$ steps

if ~~n~~ is ~~even~~



Note a person carrying torch back has to cross the bridge again at some point of time. so to optimize that P_1 can be carrying torches back. But inorder to carry back P_1 must already should have crossed the bridge. so two cases are possible

a) $\xleftarrow{\text{2 people cross}}$ and there is no P_1

b) $\xleftarrow{\text{2 people cross}}$ and one of them is P_1

For case b, the next step would be carrying back because nobody can take less time than P_1 in going back and crossing again.

For case a, in the next step P_2 would be carrying back. because of the people that crossed bridge P_2 can go back and cross the bridge fastest. of avialable people.

Now, P_1 and P_2 should cross the bridge to optimize because they have to cross at some point of time but if they cross now they can carry torch back in less time.

so, we established optimality in case-a and case-b situations. The question is to how to choose b/w case-a and case-b

consider t_K, with $3 \leq K < j$ say t_K travelled with t_P in case-a

For t_K, if we use case - a

time taken = $t_1 + (\max(t_K, t_P)) + t_2 + t_2$

For t_K, if we use case - b

time taken = $t_K + t_P + t_1 + t_1$

Now we know $2t_2 > t_1 + t_K$

$\Rightarrow 2t_2 + t_1 + \max\{t_K, t_P\} > t_1 + t_P + t_1 + t_K$

so case-b is better.

so for $t_3, t_4, \ldots, t_{j-1}$ we use case-b travelling with t_1

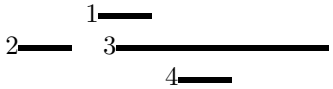
for others we use case-a travelling without t_1.

Hence proved.

Problem 8

Let us assume the line ▬ represents a class and its length is proportional to the duration of the class and the start and end of the class are depicted by the end points of the line segment

(a) Choosing the class that ends last and remove all conflicting classes and recurse
Let us examine the arrangement :



Run of this algorithm :
1.Chooses 3 . Discards 2,4
2.Chooses 1.
Thereby the set will be S = { 1,3}
But let us consider the set S' = { 1,2,4}. It is non conflicting and $|S'| > |S|$.
Hence the algorithm is wrong

(b) Choosing the class which starts first and remove all conflicting classes and then recurse.
Let us examine the arrangement :

Run of this algorithm :
1.Chooses 1 . Discards 2,3
2.Chooses 4.
Thereby the set will be S = { 1,4}
But let us consider the set S' = { 2,3,4}. It is non conflicting and $|S'| > |S|$.
Hence the algorithm is wrong

(c) Choosing the class that starts last and removing all conflicting classes and then recurse.
Proof of Correctness:

PART I: Correctness of structure
In our algorithm, we are choosing the class with largest start time in our given set and remove all the classes which conflict with it from the set. And then choose the one with largest start time the remaining elements of the set. Thereby we donot have any two classes which conflict in our greedy solution

PART II: Optimality
Let us assume an arbitrary optimal solution O = $(x_1, x_2, x_3, x_4, ..x_{j-1}, x_j...x_m)$ where x_i represents a class.
Similarly, let our greedy solution be G = $(g_1, g_2, g_3, g_4, ..g_{j-1}, g_j...g_m)$
Let us assume O and G are same upto the $(j-1)^{th}$ class from the end
Therefore

$$x_m = g_m$$

$$x_{m-1} = g_{m-1}$$

.

.

.

$$x_j = g_j$$

Now let us replace x_j with g_j in O. The obtained sequence is O' O' $= x_1, x_2, x_3, x_4, ..x_{j-1}, g_j, x_{j+1}...)$
Since after x_j , g_{j-1} is the class that starts last in accordance with our Greedy Property.
If s_i and r_i are the start and end time of a class i Then

$$s_{g_j} > r_{s_j}$$

Since O is a valid and optimal solution

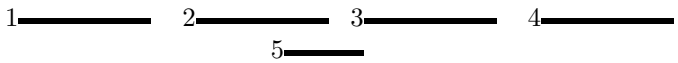
$$s_{x_j} > r_{x_{j-1}}$$

From the above two inequalities, we get Then

$$s_{g_j} > s_{x_{j-1}}$$

From this we can infer that g_j does not conflict with x_{j-1}.
Since O' accommodates as many classes as the optimal solution O, it also a valid and optimal solution. Therefore O' is as good as O
Now we can convert O to G by repeating this steps from $j = n - 1$ to $j = 0$ where n is the number of classes accommodated in the optimal solution.
Therefore G is also an optimal solution.
Hence proved

(d) Choosing The class with shortest duration and then remove all conflicting classes and recurse.
Let us examine the arrangement :

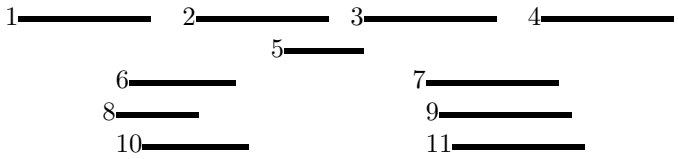


Run of the algorithm

1.Chooses 5
2.Discards 2,3
3.Chooses 1
4.Chooses 4

Therefore we get three classes 1,5,4 . Let us consider this pick
1,2,3,4. There are 4 classes and all of them are non-conflicting.
Hence the algorithm is wrong

(e) Choosing a course that conflicts with least other courses and removing all conflicting classes.
Let us examine the arrangement :



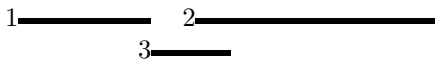
Run of the algorithm

1.Chooses 5
2.Discards 2,3
3.Chooses 8
4.Discards 1,6,10
5.Chooses 9
6.Discards 4,7,9

Thereby we get 3 classes

Let us consider this pick
1,2,3,4. There are 4 classes and all of them are non-conflicting.
Hence the algorithm is wrong

(f) Discarding classes with longest duration in case of conflict between the given set of classes
Let us examine the arrangement :



Run of this algorithm :

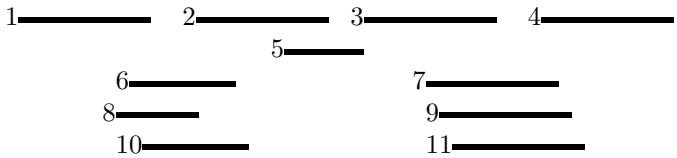
1.There is conflict. Discards 2
2.There is conflict. Discards 1.

We are left with a set S ={3}

Let us consider this pick S' = {1,2}
It is non-conflicting and $|S'| > |S|$
Hence the algorithm is wrong

(g) Discarding classes with most conflicts in case of conflict between the given set of classes

Let us examine the same arrangement in part (e)



Run of the algorithm :
1.Discards 2
2.Discards 3
3.Discards 6
4.Discards 7
5.Discards 8
6.Discards 9
7.Discards 1
8.Discards 4

We have the set S = {5,10,11}
Let us consider this pick S' = {1,2,3,4}
It is non-conflicting and $|S'| > |S|$
Hence the algorithm is wrong

(h) If x is the class with earliest start time and y be the one wills second earliest start time.
Now if :
i) y starts after x ends . Choose x
ii) y starts and ends during the time interval of x. Discard x
iii) y starts during x and ends after x ends. Discard y

If we closely study these conditions. It is equal to picking the one with earliest finish time and removing all conflicting classes
In case i) ,
y starts after x ends. So clearly x ends before y

In case ii),
y starts and ends before x. So it finishes before x. But another class z can exist which finishes before y. If z finishes before y, it conflicts with x. So regardless of z=y or not, it conflicts with x. So x is discarded.

In case iii),
If y starts during x and ends after x completes. Then x finishes before y but there can be another class z that finishes before x and starts after y. But in both cases y conflicts with x or z. Therefore we discard y

Essentially, we are choosing the activity with earliest finish time and removing all conflicting classes.

Proof of Correctness:

PART I: Correctness of structure
In our algorithm, we are choosing the class with earliest finish time in our given set and remove all the classes which conflict with it from the set. And then choose the earliest finish from the remaining elements of the set. Thereby we donot have any two classes which conflict in our greedy solution

PART II: Optimality
Let us assume an arbitrary optimal solution O = $(x_1, x_2, x_3, x_4, ..x_{j-1}, x_j...)$ where x_i represents a class.
Similarly, let our greedy solution be G = $(g_1, g_2, g_3, g_4, ..g_{j-1}, g_j...)$
Let us assume O and G are same upto the $(j-1)^{th}$ class
Therefore

$$x_1 = g_1$$
$$x_2 = g_2$$
$$.$$
$$.$$
$$.$$
$$x_{j-1} = g_{j-1}$$

Now let us replace x_j with g_j in O. The obtained sequence is O' O' = $x_1, x_2, x_3, x_4, ..x_{j-1}, g_j, x_{j+1}...)$
Since after x_{j-1} , g_j is the class that finishes earliest in accordance with our Greedy Property.
If s_i and r_i are the start and end time of a class i Then

$$r_{g_j} < r_{x_j}$$

Since O is a valid and optimal solution

$$r_{x_j} < s_{x_{j+1}}$$

From the above two inequalities, we get Then

$$r_{g_j} < s_{x_{j+1}}$$

From this we can infer that g_j does not conflict with x_{j+1}.
Since O' accommodates as many classes as the optimal solution O, it also a valid and optimal

solution. Therefore O' is as good as O
Now we can convert O to G by repeating this steps from $j = 0$ to $j = n - 1$ where n is the number of classes accommodated in the optimal solution.
Therefore G is also an optimal solution.
Hence proved

(i) If any class x completely contains another class, discard x and recurse. Otherwise choose y that ends last and discard all classes that conflict with it and recurse.

This is equivalent to choosing the class that starts last and removing all classes that conflict and then recurse with the remaining set.

If the last class, let us call y has class z which starts and finishes during class y. Then

$$s_y < s_z$$

$$r_y > r_z$$

Therefore y is not the last class to begin but it will definitely clash with the class that begins last because it ends last. Therefore we discard y. And recurse in the remaining set of classes.

But if y doesnot entirely contain another class then definitely for any arbitrary class z,

$$s_z < s_y$$

This proves our argument that y is indeed the last class to start.

The proving of this particular choice of greedy algorithm is in part(c).

Problem 9

Three possible conditions for two given intervals:

- interval lag behind: if L[i]<L[j] and R[j]<R[i], then we say j lag behind i.
- interval overlap: if L[i]<L[j]≤R[i] and R[i]<R[j], then i and j overlaps.
- interval ahead: if L[i]<L[j] and L[j]>R[i], then j is ahead of i.

Algorithm

```
1  sort intervals based on the left value
2  Step 1: select the interval with least left value. If multiple such exists,
   select the one with the largest right value. Add it in answer.
3  For remaining intervals
4      if interval lag behind the last added interval
5          continue
6      elseif interval overlaps with last added interval
7          while interval overlaps
8              find the overlapping interval with maximum right value.
9              Add it in answer
10             // both this loop and outer loop has the same counter.
11             // if an interval is seen here, it will not be seen in the outer loop
12     elseif interval is ahead of the last added interval
13         repeat from Step 1
```

Correctness and Optimality:
The algorithm uses a greedy approach. Firstly, we sort the intervals based on their left values. Let the solution obtained by this greedy algorithm is G: $f_1, f_2, ..., f_g$. Starting from G = ϕ, we build the solution as follows:
If f_i is not overlapping with the last element of G, we insert the one with the greatest right value. Else, we greedily choose, among all the intervals which overlap with f_i, the interval with the greatest right value.
Proof by Greedy stays ahead:Let,
Greedy solution be,

$$G : f_{i_1}, f_{i_2}, .., f_{i_g}$$

and optimal solution be,

$$O : f_{j_1}, f_{j_2}, .., f_{j_o}$$

By definition, o≤g. Let l_i and r_i be the left and right values of f_i respectively. We will prove that $r_{i_k} \geq r_{j_k}$ for all k≤ o. Thus, an immediate consequence is o=g, as otherwise $r_{i_o+1} > r_{i_o} \geq r_{j_o}$, which means that the optimal solution leaves out the region $(r_{i_o}, r_{i_o+1}]$ covered by f_{i_o+1} by the greedy solution.
Base case: By the greedy choice $r_{i_1} \geq r_{j_1}$ as the $l_{i_1} = l_{j_1}$ to include the left most vertex.
Induction Step: Suppose that the claim is true till k. Let f_{j_k+1} does not overlap with the previous interval. Then it must be included in both G and O. Then the claim is trivially true for k+1.
Otherwise, by induction hypothesis, $l_{j_k+1} \leq r_{j_k} \leq r_{i_k}$. This means that the f_{j_k+1} overlaps or lag behinds f_{i_k}. By greedy choice we know that we choose an overlapping interval in such a way that it has maximum right value. So, $r_{i_k+1} \geq r_{j_k+1}$. Hence, by the claim, our greedy algorithm is also optimal.
Time Complexity:
Sorting takes O(nlogn) time. The loops iterate over all interval once and thus the computation there is of O(1). Thus, time complexity is **O(nlogn)**.

Problem 10

Algorithm

We are given two arrays L[1,2,...n] and R[1,2,...,n] which contain the start and end points of the intervals correspondingly.

1. Sort R[] and L[] in increasing order separately. We will maintain a map of corresponding indices from L to R
2. Pick the first entry in R[], call it x
3. Mark all points where L[i] ¡ x as stabbed and remove corresponding points in R[] using the map
4. Add x to set of stabbing points
5. Repeat till all intervals are marked stabbed

Analysis

The sorting at the start can be performed in $\mathcal{O}(nlogn)$ time using any standard sorting technique. The creation of an index mapping for L[] to R[] would take $\mathcal{O}(nlogn)$ time as the corresponding element in R[] from L[] can be found in $\mathcal{O}(logn)$ time using binary search and has to be repeated for n elements. Once the map has been created, searching and marking the points stabbed will require $\mathcal{O}(n)$ as every element in the array L[] has to be referenced only once and the corresponding element in R[] can be marked out in $\mathcal{O}(1)$ time using the index map. Therefore, the overall time complexity of the algorithm $T(n) = \mathcal{O}(nlogn)$.

Correctness

The algorithm works by greedily selecting the end points sorted in increasing order whenever they are encountered. Once a point is selected, the stabbed intervals are marked and we then pick the next point greedily from the remaining unmarked points. We can prove the correctness of this algorithm using an exchange argument by showing that for any optimal solution set that does not contain the end points but rather the points inside the interval, there exists a solution set that replaces these points by the nearest end point without affecting the cardinality or stabbing. This does require the assumption that the end points are distinct.

Consider an optimal solution set of points $X = x_1, ..., x_k$. Let x_i be the smallest element that is not the end point of some segment. Let x_j be the smallest right end point that exceeds x_i. We then replace x_i with a_j, but because we have increased x_i to a value that is not greater than any of the right end points stabbed by x_i, every interval stabbed by x_i is still stabbed by a_j. The cardinality of the solution set after this step stays the same. After repeating this step at most k times, we are left with a solution set which stabs all the intervals as before but consists of right end points of intervals, which means that the optimality of the solution still holds true. Therefore, this solution is one of the many optimal solutions available for this problem.

Problem 11

11) Given L[1..n] and R[1..n] representing the left and right endpoints of interval x

We need to compute the minimum number of colours needed to compute colour x such that overlapping intervals are assigned different colours

High level Algorithm

A greedy method is as follows:

for each interval I in order of increasing start time do:

assign to I the smallest colour that has not been assigned to any previously assigned intervals that overlap I.

Low-level-Algorithm

1. n=0 // n contains the largest colour ever used
2. A=ϕ // Empty queue of available halls that are ever used

/* priority queue Q contains all the endpoints (with reference to their intervals)

/* It will be increasing order of times (min heap) and between two start times we break tie in favour of finish time between two finish times we break tie in favour of start time between finish & start break tie in favour of finish */

```
3.  Q ← {L[i], R[i] : 1≤i≤n}
4.  while Q ≠ ϕ do {
5.      x ← Extract min (Q)
6.      if (x is a start time) then {
7.          if (A ≠ ϕ) then      // some colour is available
8.              c ← deque(A)     // reuse it
9.          else {
10.             d ← d+1          // new colour
11.             c ← d }
12.         {assign colour c to interval of x}.
13.     else {      // x is a finish time
14.         c ← colour of interval x
15.         enqueue (c, A) // put colour c in queue
16.     }
17. };
18. return d    // d contains total no of colours used.
```

Running time:-

$T(n) = O(n \log n)$ because we do extract min operation at each endpoint & start point, and also we constructed heap

Correctness:-

Let k be the maximum number of intervals that can overlap at any point in time, so we need to have minimum k colours that is $k \leq |S|$ where S is any solution

If G_1 is a current greedy solution and if we can show $|G_1| = k$ then we can conclude G_1 is optimal.

So we need to prove $|G_1| = k$.

Proof by contradiction:

Suppose the algorithm (Greedy) used more than k colours consider the first time (say on some interval x) the greedy algorithm would have used k+1 of colours

which means there are k intervals which are overlapping with intervals, so overall there are k+1 overlapping intervals which is the contradiction to the fact that k is maximum number of intervals that can overlap at any point in time.

Hence $|G_1| = k$, thus G_1 is optimal.

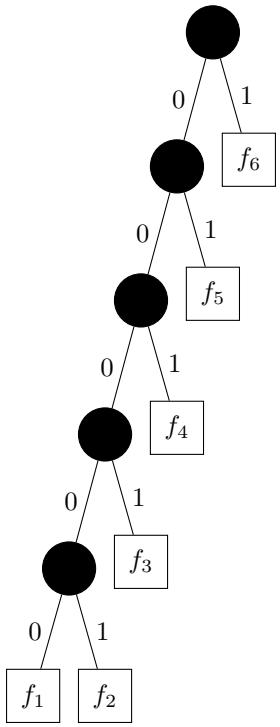
References:-

* Inspired from GIAC- MCS-375 Algorithm's course lecture handout.

Problem 12

(a)

We first note that there is only 1 possible Huffman tree (non-isomorphic) of depth $n-1$. The following is such a tree for $n = 6$.



Where $f_1, f_2, ..., f_6$ are the frequencies of the characters such that $f_1 \leq f_2 \leq ... \leq f_6$. Any such tree will have a similar structure as above with $f_1 \leq f_2 \leq ... \leq f_n$ when there are n characters.
For such a tree to be built using the Huffman encoding algorithm, the following properties must hold,

$$
\begin{aligned}
f_3 &> \textsc{Max}(f_1, f_2) \\
f_4 &> \textsc{Max}(f_1 + f_2, f_3) \\
f_5 &> \textsc{Max}(f_1 + f_2 + f_3, f_4) \\
f_6 &> \textsc{Max}(f_1 + f_2 + f_3 + f_4, f_5) \\
&\vdots \\
f_k &> \textsc{Max}(\sum_{i=1}^{k-2} f_i, f_{k-1})
\end{aligned}
$$

Since the problem requires us to find a set of frequencies $f[1\,..\,n]$ such that the largest frequency is as small as possible, we can take the values to be

$$
\begin{aligned}
f_3 &= \textsc{Max}(f_1, f_2) + 1 \\
f_4 &= \textsc{Max}(f_1 + f_2, f_3) + 1 \\
f_5 &= \textsc{Max}(f_1 + f_2 + f_3, f_4) + 1 \\
f_6 &= \textsc{Max}(f_1 + f_2 + f_3 + f_4, f_5) + 1 \\
&\vdots \\
f_k &= \textsc{Max}(\sum_{i=1}^{k-2} f_i, f_{k-1}) + 1
\end{aligned}
$$

This also gives us a freedom of choosing f_1 and f_2, since the aim is to make the largest frequency as small as possible, we shall choose $f_1 = f_2 = 1$, the smallest positive frequencies available. We can now see that $f_k > f_{k-1}\ \forall k > 2$ as follows

$$\begin{aligned} f_k &= \textsc{Max}(\sum_{i=1}^{k-2} f_i, f_{k-1}) + 1 \\ &\geq f_{k-1} + 1 > f_{k-1} \end{aligned}$$

Induction Proof :
Now, we prove that $\sum_{i=1}^{n-2} f_i \geq f_{n-1}\ \forall n > 2$ inductively.
Base Case : $k = 3$, we know that $f_1 = 1 \geq f_2 = 1$.
Inductive Case : Assume the result holds for all $3 \leq i \leq k$, Now this gives us that,

$$f_i = \sum_{i=1}^{i-2} f_i + 1$$

for $i \in [3, k]$.
This give us that,

$$\begin{aligned} f_i &= \sum_{i=1}^{i-2} f_i + 1 \\ f_{i-1} &= \sum_{i=1}^{i-3} f_i + 1 \\ \implies f_i &= f_{i-1} + f_{i-2} \end{aligned}$$

which holds for $i \in [4, k]$. Now, from the inductive hypothesis,

$$\begin{aligned} \sum_{i=1}^{k-2} f_i &\geq f_{k-1} > f_{k-2} \\ \implies \sum_{i=1}^{k-1} f_i &\geq f_{k-2} + f_{k-1} = f_k \end{aligned}$$

Thus proven for all $n > 2$.

This information can now be used as follows.

$$\begin{aligned} f_k &= \sum_{i=1}^{k-2} f_i + 1 \\ f_{k-1} &= \sum_{i=1}^{k-3} f_i + 1 \\ \implies f_k &= f_{k-1} + f_{k-2} \end{aligned}$$

Therefore f follows the Fibonacci sequence with the values $f_1 = 1, f_2 = 1, ..., f_n = F_n$
where

$$F_n = \frac{1}{\sqrt{5}} \left(\left(\frac{1+\sqrt{5}}{2} \right)^n - \left(\frac{1-\sqrt{5}}{2} \right)^n \right)$$

(b)

First we introduce a change in notation, given a frequency set $f_1, f_2, ..., f_n$ with $f_i \leq f_{i+1}\ \forall i \in [1, n-1]$, we define

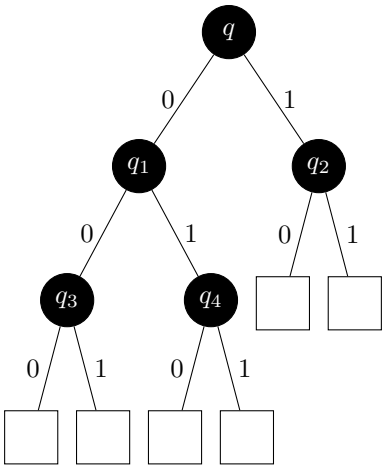
$$p_i = \frac{f_i}{\sum_{k=1}^{n} f_k}$$

as the probability of occurrence of the i^{th} character with frequency f_i. It follows that $p_i \leq p_{i+1}$ $\forall i \in [1, n-1]$.
In the Huffman tree, we denote the probability of each node as the sum of the probabilities of it's 2 children with the base case being for a leaf in which case the probability of the leaf node is just the probability of the associated character. We first show the following result.

Lemma : If q is the probability of the root of a sub-tree of the Huffman tree of depth t (depth of the sub-tree), then

$$q \geq p_1 F_{t+1} + p_2 F_t$$

where F_i denotes the Fibonacci numbers with $F_0 = 0$ and $F_1 = 1$. We prove this by Strong Mathematical Induction on t,
Base Case : For depth 0 and depth 1, we have $q \geq p_1$ and $q \geq p_1 + p_2$ respectively. These are trivially true, for a sub-tree of depth 0 i.e. a leaf, the probability will be greater than or equal to the smallest probability among all leaves which is p_1 and for a sub-tree of depth 1, we know that the two lowest probability characters are grouped together in a sub-tree of depth 1 in the first step itself, therefore any sub tree of depth 1 has probability $\geq p_1 + p_2$.
Inductive Step : Assume the result to be true for all sub-trees with depth $\leq t$



Consider a sub tree T of depth $t+1$ with root of probability q and let it's children sub-trees be T_1 and T_2 with roots of probability q_1 and q_2 respectively. At least one of it's children sub-trees has a depth t, WLOG let this sub-tree be T_1. Let T_1's children sub-trees be T_3 and T_4 with roots of probability q_3 and q_4. Similarly at least one of T_3 and T_4 has a depth $t-1$, WLOG let this sub-tree be T_3.
Now, from inductive hypothesis,

$$q_1 \geq p_1 F_{t+1} + p_2 F_t \tag{1}$$

and

$$q_3 \geq p_1 F_t + p_2 F_{t-1} \tag{2}$$

Now we can notice that $q_2 \geq q_3$, if this were not so, then from the exchange argument, T_2 could be interchanged with T_3 yielding a better set of prefix codes, but this is not possible as the Huffman code generates the optimal sub-tree. This implies that

$$q_2 \geq q_3 \geq p_1 F_t + p_2 F_{t-1} \tag{3}$$

Adding (1) and (3), we get

$$q = q_1 + q_2 \geq p_1(F_{t+1} + F_t) + p_2(F_t + F_{t-1}) = p_1 F_{t+2} + p_2 F_{t+1}$$

Hence we have shown that for any sub-tree of depth t with root of probability q, $q \geq p_1 F_{t+1} + p_2 F_t$.
Now, applying the result to the root of the Huffman tree which has a probability 1 and depth t, we have

$$1 \geq p_1 F_{t+1} + p_2 F_t \tag{4}$$

Now we can apply the formula for the Fibonacci sequence i.e.

$$F_n = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right)$$

Applying this in (4) after a few transformations, we get

$$t \leq \log_\phi\left(\frac{\phi+1}{p_1\phi+p_2}\right) = \log_\phi\left(\frac{(\phi+1)(\sum_{k=1}^{n} f_k)}{f_1\phi+f_2}\right)$$

But the problem gives us that the total length of the un-encoded message i.e. $N = \sum_{k=1}^{n} f_i \in \mathcal{O}(n^k)$. This gives us

$$t \leq \log_\phi\left(\frac{(\phi+1)(cn^k)}{f_1\phi+f_2}\right) \in \mathcal{O}(\log_\phi n)$$

where $\phi = \frac{1+\sqrt{5}}{2}$ is the golden ratio. Therefore the depth of the Huffman tree, $t \in \mathcal{O}(\log(n))$.

Problem 13

For the symbol 1 to be represented by one bit, it is necessary that it is present as an only element in the last step of the Huffman tree building algorithm. To recall, the algorithm takes the pair of elements/trees with the least frequency and creates a node with frequency as the sums of the two frequencies in one stage. The node created at the last stage is the root, and so the element 1 needs to be present as a node in the second last stage in order to be represented by one bit.

First, we prove that $1/3 \leq \alpha \leq 1/2$. For this we consider the second to last step in the Huffman tree building algorithm. We require the node with element 1 in the last step, so it must be present in this step too. Let the nodes present in this step be n_1, u and v, and their frequencies be f_1, f_u, and f_v. N represents the total number of symbols or $\sum_{i=1}^{n} f[i]$.

Now,

$$f_1 + f_u + f_v = N$$

And since $f[1] \geq \alpha N$, to find the largest real number α such that the condition holds for every α-heavy array, we take $f[1] = \alpha N$. Since 1 needs to be present as node in the last step, we have

$$f_u \leq \alpha N$$
$$f_v \leq \alpha N$$

Adding them,

$$f_u + f_v \leq 2\alpha N$$

Also, to get a stricter bound, we have that at the last step $f[1]$ is not the greater one. We chose u and v, and get a node with frequency $f_u + f_v$. Thus,

$$f_u + f_v \geq \alpha N$$

So,

$$\begin{aligned} \alpha N \leq f_u + f_v \leq 2\alpha N \\ 2\alpha N \leq f_1 + f_u + f_v \leq 3\alpha N \\ 2\alpha N \leq N \leq 3\alpha N \\ 2\alpha \leq 1 \leq 3\alpha \end{aligned}$$

Thus,

$$1/3 \leq \alpha \leq 1/2$$

Claim: $\alpha = 0.4$, that is , in every Huffman code for every 0.4-heavy frequency array, symbol 1 is represented by a single bit.

First, we show that there exists a Huffman code for an α-heavy frequency array where $\alpha < 0.4$ and 1 is not represented by a single bit.
Take a stage of the Huffman coding with three nodes : $1, u, v$ and their frequencies $f_1 = 0.4 - x, f_u = 0.2 + x, f_v = 0.4$ where $0 < x < 0.1$(v can be obtained by combining two symbols with frequency 0.2). In this case, f_1 and f_u will be chosen in the second last step and so 1 will not be represented by a single bit.

Next, we show that for every Huffman code for an α-heavy frequency array where $\alpha 0.4$, 1 is the represented by a single bit. We give a proof by contradiction for this.
Take the stage with three nodes : $1, u, v$ and their frequencies $f_1 = 0.4, f_u = u, f_v = v$. We assume that 1 is not represented by a single bit. Thus, WLOG, f_u is the greatest. Now, let

$f_1 = 0.4, f_u = 0.4 + x, f_v = 0.2 - x$ and $x > 0$. u cannot be a symbol as then it would become the most frequent symbol. Thus, it is made up of symbols with frequencies less than or equal to f_v, because if that was not the case, then f_v would have been chosen in an earlier step. So the frequencies of the two nodes in the earlier step would be both less than or equal to $0.2 - x$. Let those two nodes be p and q.

$$\begin{aligned} f_p &\leq 0.2 - x \\ f_q &\leq 0.2 - x \\ f_p + f_q &\leq 0.4 - 2x \end{aligned}$$

But $f_p + f_q = f_u = 0.4 + x$, and $0.4 + x \leq 0.4 - 2x$ is a contradiction as $x > 0$.
Therefore, 1 is always represented by a single bit in every Huffman code for every 0.4-heavy frequency array.

Thus, $\alpha = 0.4$.

Problem 14

a) The greedy algorithm for this problem is simple. For every shelf, we try to fit in as many books as we can by leaving no space between two books. However, if at the end of each shelf, if a book does not fit, we move onto the next shelf.

The algorithm works because of the following reason:
Consider an ordering which does not follow the above approach, which means that at least one shelf has the capacity to hold more books. This can affect the total number of shelves used. Consider the last shelf. Leaving a middle shelf partially empty may force us to use an extra shelf to store the books at the last that are pushed forward because of the wastage of space in one of the middle shelf.

b) Consider the following setup:

There are 'n' books available. Let for $1 \leq i \leq (n-2)$ the height of the books be $H[i] = h_0$ and for $(n-1) \leq i \leq n$, $H[i] = h_1$ where $h_1 > h_0$.

Suppose the first shelf can fit a maximum of $n-1$ books as per the algorithm in part (a). Then the last book has to be placed in the second shelf. Therefore the height of both the shelves has to be h_1 and the total height given by this algorithm is $2 \times h_1$.

However, a better solution exists. The last two books can be placed on the second shelf and rest of the first $n-2$ stay on the first shelf. This way, the height of the second shelf remains h_1 but the height of the first shelf reduces to h_0 which in turn decreases the sum of the heights of the shelves.

Hence, the greedy algorithm used in part (a) does not always give the best solution to this problem.

c) We solve this problem using dynamic programming. Assuming we place k books on the last shelf. Then the sum of the heights of the shelves is $max(H[n-k+1], H[n-k+2], ..., H[n])$+ cost of placing the rest $(n-k)$ books.

Let $cost[x]$ be the minimum height of the shelves after placing the initial x books.

Then, we have the following relation: $cost(x) = min(cost(y) + max(H[y+1], H[y+2], ..., H[x]))$ for $1 \leq y < x$

Also, $sum(T[y+1], T[y+2], ..., T[x]) \leq L$

Pseudocode:
$cost(0) = 0$
for $i = 1$ to n
    $maxheight = 0$
    for $j = i-1$ to 0 step -1
        $currThickness = currThickness + T[j+1]$
        $maxheight = max(maxheight, H[j+1])$
        $if currThickness \leq L$
            $cost(x) = min(cost(x), maxheight + cost(j))$

Problem 15

Q15) a)

The problem can be solved by exploiting the stack data structure.

Since we only have two types of input characters, '(' and ')', our algorithm does the following:

(i) if its a '(', then push it onto the stack.

(ii) if its a ')', then pop the topmost element from the stack.

Only those strings will be rejected, which reach the following conditions:

a) All characters have been read, but stack is non empty.

b) When there is no element in the stack, but we try to pop.

Runtime analysis:

Since for every input character being read, we are either pushing or popping, it takes O(1) time for a character. For an input of size n, the time taken is O(n).

b) Let a balanced string be described by those accepted as in part (a) of the problem. So, we need a count of extra open braces and extra closed braces, which on removal will make the string balanced.

Algorithm:

(i) compute extra open braces count and extra closed braces count.

(ii) If at any point, number of closed braces exceed open braces, then this closed brace is extra and needs to be removed.

(iii) This way, keep track of extra closed braces.

(iv) No of open braces extra are precisely those remaining on the stack from the (a) part's solution.

(v) Final answer is just input size - (extra closed braces + extra open braces)

Pseudo Code explaining the same:

(i) initialise ~~int~~ extra_open = 0.

(ii) initialise extra_closed = 0.

(iii) if input char is '('

extra_open++;

(iv) if input char is ')'

(a) if (extra_open) is 0

extra_closed++;

(b) else extra_open--;

(v) answer = n - (extra_open + extra_closed)

Time complexity:

Similar to the runtime analysis for part (a), for every character it consumes O(1) time. So for an input of size n, it takes O(n) time.

Problem 16(a)

Explanation

For each node in the tree we define an attribute *length* which denotes the number of nodes it is connected to, which have depth greater than its depth, in the final solution. Note that all the leaf nodes will have $length = 0$ as a leaf node cannot be connected to any node which has depth greater than it. The maximum number of climbers allowed is the maximum number of nodes with $length = k - 1$. To optimise the number of climbers,for each node its *length* is given by (the maximum *length* of its children not equal to $k - 1$) + 1. The intuition behind doing this is that the node can share an edge with only one of its children due to the disjoint constraint and the child with maximum *length* has the highest probability of achieving a *length* of $k - 1$. The problem of finding the *length* for each node is equivalent to finding the number of climbers.

Pseudocode

Let $MC[1\,..\,n]$ be an array initialised to -1 where n is the number of nodes. This array is for book keeping the values of MAXIMUM-CLIMBERS$(T, node, k)$ for different values of *node*.

```
MAXIMUM-CLIMBERS(T, node, k)
1  if MC[node] ≠ −1
       return MC[node]
2  if node.children.size == 0
       node.length = 0
       return 0
3  maxChildLength = 0
4  Climbers = 0
5  for child in node.children
       Climbers = Climbers + MAXIMUM-CLIMBERS(T, child, k)
       if child.length > maxChildLength and child.length ≠ k − 1
            maxChildLength = child.length
6  node.length = maxChildLength + 1
7  if node.length == k − 1
       Climbers = Climbers + 1
8  MC[node] = Climbers
9  return Climbers

MAXIMUM-CLIMBERS-MAIN(T, node, k)
1  return MAXIMUM-CLIMBERS(T, T.root, k)
```

Analysis

The function MAXIMUM-CLIMBERS$(T, node, k)$ is computed for each *node* exactly once and stored in MC. Also MAXIMUM-CLIMBERS$(T, node, k)$ has a time complexity of $\mathcal{O}(degree(node))$. Therefore MAXIMUM-CLIMBERS-MAIN$(T, node, k)$ has a time complexity of $\mathcal{O}(V + E)$ where V is the number of vertices in the tree and E is the number of edges in the tree.

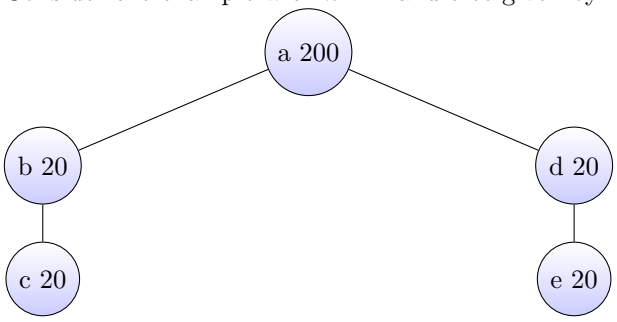
Correctness

Suppose in the optimal solution some node x has *length* different from that given by our algorithm. Now suppose we change x's *length* by adding an edge from it to its child with largest *length* not equal to $k - 1$. By doing this the length of the path involving x does not decrease. If this length becomes greater than $k-1$, we can remove some edges along the path to make the path length $k-1$. By increasing the path length, there is a possibility that we can accommodate another climber else the number of climbers. Therefore by exchange argument the solution given by our algorithm is not worse than the optimal one. Therefore it has to be optimal.

Problem 16(b)

Explanation

Consider the example with $k = 1$ and tree given by



The algorithm from part (a) will choose two paths, one containing (b, c) and the other containing (d, e). This gives us a total reward of 80. But this is not optimal as just choosing the path (a, b) gives me a reward of 220.

Problem 16(c)

Explanation

For each node in the tree we define an attribute *length* which denotes the number of nodes it is connected to, which have depth greater than its depth, in the final solution. For each node in the tree we define an attribute *reward* which denotes the sum of rewards of nodes it is connected to, which have depth greater than its depth, in the final solution. Note that all the leaf nodes will have $length = 0$ ans $reward = 0$ as a leaf node cannot be connected to any node which has depth greater than it. The total reward is the sum of the rewards of each path containing a node with $length = k - 1$. We first find the reward for all the different values of *length* of root. Then we choose the length which gives maximum reward. We recursively do this for all the nodes. The intuition behind doing this is for a given *length* we just need to worry about maximising reward by choosing the child which gives maximum reward.

Pseudocode

Let $MRL[1\,..\,k][1\,..\,n]$ be an array initialised to -1 where n is the number of nodes. This array is for book keeping the values of MAXIMUM-REWARD-LENGTH$(T, node, length, k)$ for different values of *node* and *length*. Also similarly let $MR[1..k]$ be an array initialised to -1.

```
MAXIMUM-REWARD-LENGTH(T, node, length, k)
1  if MRL[length + 1][node] ≠ −1
       return MRL[length + 1][node]
2  if length == 0
       MRL[1][node] = 0
       return 0
3  reward = 0
4  maxChildReward = −1
5  maxChild = −1
6  for child in node.children
       reward = reward + MAXIMUM-REWARD(T, child, k)
       if length > 0
           childReward = MAXIMUM-REWARD-LENGTH(T, child, length − 1, k)
           if childReward ≠ −1
               childReward = childReward + child.reward
               if length == k − 1
                   childReward = childReward + node.reward
           if childReward > maxChildReward
               if length ≠ k − 1
                   maxChildReward = childReward − child.reward
               else
                   maxChildReward = childReward
               maxChild = child
7  if maxChildReward ≠ −1
       node.reward = node.reward + maxChild.reward
       MRL[length + 1][node] = reward + maxChildReward − MAXIMUM-REWARD(T, maxChild, k)
       return MRL[length + 1][node]
8  return -1

MAXIMUM-REWARD(T, node, k)
1  if MR[node] ≠ −1
       return MR[node]
2  maxReward = 0
3  for length in [0 . . k − 1]
       if MAXIMUM-REWARD-LENGTH(T, node, length, k) > maxReward
           maxReward = MAXIMUM-REWARD-LENGTH(T, node, length, k)
4  MR[node] = maxReward
5  return maxReward

MAXIMUM-REWARD-MAIN(T, node, k)
1  return MAXIMUM-REWARD(T, T.root, k)
```

Analysis

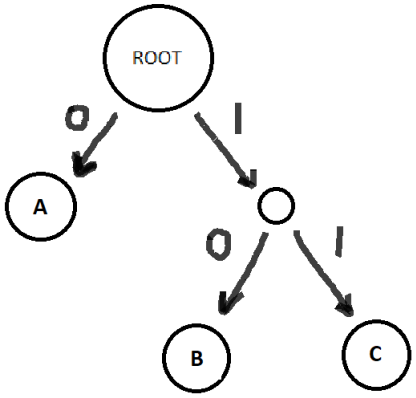
The function MAXIMUM-REWARD-LENGTH$(T, node, length, k)$ is computed for each *node* and each *length* exactly once and stored in MC. Also MAXIMUM-REWARD-LENGTH$(T, node, length, k)$ has a time complexity of $\mathcal{O}(degree(node))$. Therefore MAXIMUM-REWARD-MAIN$(T, node, k)$ has a time complexity of $\mathcal{O}(k(V + E))$ where V is the number of vertices in the tree and E is the number of edges in the tree.

Correctness

For a given $length = l$ of the node, to maximise the reward we connect it to the child which has maximum reward for $length = l - 1$. We then do an exhaustive search on all the possible lengths to find the maximum possible reward of the tree with the node as root. We do this recursively to find the maximum possible reward of the whole tree.

Problem - 17(a)

Yes, this code can be obtained from Huffman's algorithm
For the case where $f_a = 60$, $f_b = 30$, $f_a = 10$.
Consider a tree with only 1 node i.e, the root. First build a min heap, the minimum would be 10, and the next would be 30. We now make a node with frequency 40 with the left child being b, the right child being c. After removing two minimum from the min heap, insert the node into the tree. Now the heap contains 60. Therefore assign root-¿left to 'a', and root-¿right the node we created.This root-¿right in turn also contains 'b' as its left child and 'c' as its right child. Therefore, the root has 'a' as its left child, 'b' as the left child of its right child, 'c' as the right child of its right child.
So, we have the code for 'a': 0 'b' : 10 'c' : 11.
Hence Proved.



This encoding will work for any set of frequencies such that $f_a \geq (f_b + f_c)$, so that 'a' will still get '0', which it should as it has the highest frequency, b ,c get the next two positions, and hence we get the same code for the alphabet.

Problem - 17(b)

No, this code cannot be obtained from Huffman's algorithm for any set of frequencies as this code is wrong.
Proof by contradiction:-
Let us assume that the following code is correct and the code to be 'a': 0 'b' :1 'c' : 00.
Let us consider an encoding '001'. Now decoding this can be done in two ways:-
Case - 1 :-
We look at the first character, and we infer that it must be 'a' since 'a' has the code : '0'. The next character of the encoding is also '0' therefore the string must have 'a' as its second element. Now the last character is '1' implying that the encoding is decoded into 'aab'.
Case - 2 :-
Look at the first two characters. We have '00' which is the code for 'c'. Therefore the first two characters correspond to 'c' and the last '1' corresponds to 'b' => the code '001' is decoded into 'cb'.
Clearly the following encoding can be interpreted as two different string, which is wrong , hence the following code can never be obtained from the Huffman's algorithm since we know that the Huffman's algorithm generates a code which gives us a unique string after decoding. A simple way to put it is that 'a' is the prefix code for 'c' which is not allowed. Hence Proved.

Problem - 17(c)

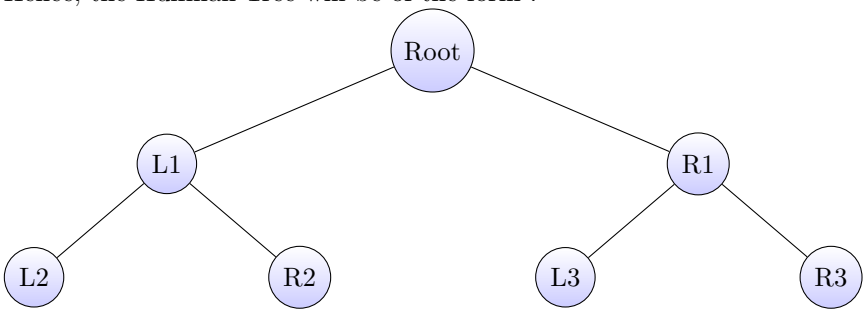
No, this code cannot be obtained from Huffman's algorithm for any set of frequencies
According to the Huffman's algorithm, the last step of tree building will give rise to the character of the alphabet having the maximum frequency to get a single bit code. However, we see that none of the codes is a single bit meaning that the code given in the question can be optimized, and so cannot be obtained from Huffman's algorithm. However, this code can be used since it is unique, but it just isn't obtained from Huffman's algorithm for any set of frequencies.

Problem - 18

(a) To Prove: If some character occurs with frequency more than 2/5, then there is guaranteed to be a codeword of length 1.
Proof :
The depth of the character in Huffman tree represents the length of the codeword.
Let $d_t(c)$ represent the depth of character c's leaf in the tree.
Let C denote the set of alphabets from which the characters are drawn
Assumption : There is no codeword of length 1 i.e.

$$\forall c \in C \quad d_t(c) \geq 2$$

Hence, the Huffman Tree will be of the form :



Note that both L1 and R1 cannot be leaves and will have children.
It is given that there exists some character with frequency more than 2/5.
Assume without loss of generality that this character lies in the subtree L2.
Let F(x) denote the cumulative frequency of node x

Since L2 and R2 were merged and their merged tree was merged with R1 hence, it can be said that F(R1) $\geq$ F(L2). This is because if F(R1) $<$ F(L2) then R1 and R2 or R1 and L2 should have been merged (This comes from the way we construct the Huffman Tree) but this has not happened and , therefore, we can say that F(R1) $\geq$ F(L2).
Now,

$$F(L2) > 2/5 \text{ (}\textit{Since the given character lies in L2 }\text{)}$$

So,

$$F(R1) > 2/5$$

But the sum of cumulative frequency must be 1 .Hence,

$$F(R2) < 1/5 \text{ (} \textit{since } F(Root) \ = \ 1 \ = \ F(R2) \ + \ F(L2) \ + \ F(R1) \text{)}$$

$$F(L1) > F(L2) \text{ (} \textit{Since } F(L1) \ = \ F(L2) \ + \ F(R2) \text{)}$$

$$F(L1) > 2/5$$

But since,

$$F(Root) = 1$$

Hence,

$$F(R1) < 3/5$$

One of the children of R1 will have cumulative frequency less than 3/10. Without loss of generality say that child is R3.
So now we have,

$$F(L2) > 2/5$$

$$F(R2) < 1/5$$

$$F(R3) < 3/10$$

So, from the way we construct Huffman Tree R2 and R3 must have combined. Therefore, this is a contradiction. Hence, our assumption is false. Hence, there is a codeword of length 1.

(b)

To Prove : If all characters occur with frequency less than 1/3, then there is guaranteed to be no codeword of length 1

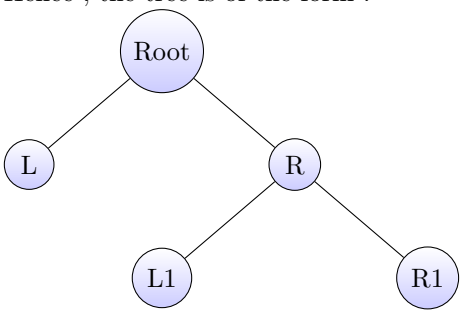
Proof :

The depth of the character in Huffman tree represents the length of the codeword.
Let $d_t(c)$ represent the depth of character c's leaf in the tree.
Let C denote the set of alphabets from which the characters are drawn

Assumption : Suppose that there exists a codeword of length 1 i.e

$$\exists\ c\ \in\ C\ such\ that\ d_t(c)\ =\ 1$$

Depth of character c is 1 in the Huffman Tree.
Hence , the tree is of the form :



Note that L is a leaf here and represents the character c.
F(x) represents the cumulative frequency of node x.
It is given all characters occur with frequency less than 1/3.

$$F(L) < 1/3 \tag{5}$$

Now, L1 and R1 combined first and then R(L1+R1) combined with L. Hence, we can say that F(L) is greater than both F(L1) and F(L2).

$$F(L1) < 1/3 \tag{6}$$

$$F(R1) < 1/3 \tag{7}$$

Now,

$$F(L) + F(L1) + F(R1) = F(Root) \tag{8}$$

$$F(L) + F(L1) + F(R1) < 1 \tag{9}$$

$$F(Root) < 1 \tag{10}$$

This is a contradiction. Hence, our assumption is false. Hence, there is guaranteed to be no codeword of length 1.

Problem 19

We know that the length of the longest code-word is the depth of the Huffman encoding tree.
The Huffman encoding tree is a complete tree and the leaves represent the symbol frequencies and the inner nodes represent the sum of their direct children .The depth of the tree will be maximum if it is a skewed complete tree.
In a skewed complete tree each complete node will have a leaf and another complete node as its children and the last complete node will have 2 leaves as its children. This will lead to the depth of the tree being equal to the no of symbols for $n > 2$.
Claim:
The i^{th} inner complete node from the bottom holds the value which is the sum of the first $i + 1$ smallest frequencies.
Proof:
By induction:
Let the inner nodes be addressed by $A_1, A_2, ...A_{n-1}$. Numbered from bottom up.
Let the frequencies of the symbols be $F_0, F_1, F_2, ...F_{n-1}$. In sorted order .
Base Case:
for $i = 1$ i.e the bottom-most inner node it only has 2 leaves as children and we know that inner nodes represent the sum of the frequencies of its direct children.
Also since the leaves are at the lowest level it means that they are the 2 smallest frequency symbols.
hence $A_1 = F_0 + F_1$
Base case is true.
Induction Hypothesis :
Let us assume that $A_k = \sum_{i=0}^{k} F_i$.
Now we know that the $k + 1^{th}$ inner node has the k^{th} inner node and the $k + 1_{th}$ symbol leaf node as children.
Therefore $A_{k+1} = A_k + F_{k+1}$
We have $A_k = \sum_{i=0}^{k} F_i$.
Substituting we have :
$A_{k+1} = \sum_{i=0}^{k+1} F_i$. Hence proved by induction.
From the fact that the complete encoding tree is skewed we can say that after $k - 1$ merges when we are left with $A_{k-1}, F_k, F_{k+1}, F_{k+2}, ...F_{n-1}$. The smallest no's are still A_{k-1} and F_k since they are the next ones to be merged .
Since we just need an example frequency distribution we can also add the assumption that $A_{k-1} < F_k$ for all $k < n$.
Now we have that the prefix sum of the first k smallest frequencies is less than the $k+1^{th}$ frequency value for all $k < n$.
The geometric Progression series satisfies this condition.
Let us assign $F_i = 2^i \ \forall i \in 0, ..n - 1$.
Now prefix sum of the first k frequency value $Sum_k = 1 + 2 + 4 + 8 + ...2^{k-1}$
On solving the GP we have, $Sum_k = 2^k - 1$.
Now the $k + 1^{th}$ frequency $F_k = 2^k$
$F_k > Sum_k$, hence it satisfies our condition.
Therefore if we have a frequency distribution such that $F_i = 2^i \ \forall i \in 0, 1, 2..., n - 1$, we will have the longest code-word length for the smallest frequency symbol which will be of length n for $n > 2$

Problem 20

20) idea:

A feedback edge set $E' \subseteq E$ is such that removing E' renders G acyclic.

$\Rightarrow$ $E - E'$ becomes a tree

If E' is to have minimum wt the spanning tree must have maximum weight.

So the complement of maximum spanning tree is the feedback edge set the answer required.

algorithm: we use the same algorithm as discussed in class but instead of taking increasing order we use decreasing order.

so arrange edges in decreasing order of weights. if an edge is internal ignore it else include the edge.

The complement of the edges in maximum spanning tree is feedback edge set.

(or) we can store all the edges we are ignoring and that set will be feedback edge set.

Pseudo code:

E be the edge set in decreasing order of weights

```
e = First(E)
T = ∅
E' = ∅
while not end of E
    if ( e is not in component generated by T )
        T = T U {e}
        e = .
    else  E' = E' U {e}
    e = Next(e, E)
```

Problem 21

Changing the MST after the weight of a particular edge e is modified
Given MST = (V , E') and edge weight becomes $\hat{w}$(e) from w(e)

(a) e $\notin$ E' and $\hat{w}$(e) > w(e)
Since e is not in E' with w(e), it cannot be in E' with weight $\hat{w}$(e) which is greater than w(e). Thereby MST will remain the same.

(b) e $\notin$ E' and $\hat{w}$(e) < w(e)
Using cycle property of an MST , we get that when any new edge e $\notin$ E' is added to the MST. We get a cycle. Now we just have to remove the heaviest edge in the cycle and the tree obtained will the new modified MST.
VARIABLES USED :

vector <edge> edgeList // Keeps track of edges in MST
vector <vector <Vertices> > AdjList // AdjList of the MST
edge newEdge // Stores the modified edge and it's weight

ALGORITHM :

```
1  // Writing our own DFS to keep track of the only cycle
2  vector <int> DFS(vector <vector <Vertices> > AdjList,int current ,
   int u ,vector <int> cycleNow) {
3  if (current = u)
4        return cycleTillNow
5  cycleTillNow.pushback(current)
6  for auto it = AdjList[current].begin() to it = AdjList[current].end()
7        DFS(<vector <Vertices> > AdjList, int (*it) ,int u , vector <int> cycleTillNow)
8  // Adding the edge
9  AdjList[newEdge.vertex1].pushback(vertex2)
10 AdjList[newEdge.vertex2].pushback(vertex1)
11 // Finding the cycle
12 vector ¡int¿ Cycle
13 int u = newEdge.vertex1
14 Cycle.pushback(u)
15 Cycle = DFS( AdjList, AdjList[u][0] ,int u ,Cycle) // Find the maximum edge in the cycle and removing it
16 auto it = Max(Cycle.begin(),Cycle.end())
17 EdgeList.erase(it)
18 }
```

Complexity :
The DFS is of O(n) because there is only one cycle in the MST after an edge is added and m = n.
Adding the edge is of O(1)
Finding the maximum edge is done using STL max : O(N)
Erase is also of O(N)
Thereby the overall complexity of the algorithm is O(N)

(c) e $\in$ E' and $\hat{w}$(e) < w(e)
Since it already belongs to the MST, a decrease in the weight of the edge will not affect the MST. MST remains the same

(d) e $\in$ E' and $\hat{w}$(e) > w(e)
We will use a property of a cut. Let us remove the edge e which has modified weight. We get two disconnected trees. Now we color the vertices of each tree with different colors. We will analyse the remaining edges and find the one with least weight which connects the two trees.

Let e connect vertices v1 and v2
VARIABLES USED :
vector <edge> EdgeSet
vector <vertices> VertexSet
vector < *int* > visited(n,0)
vector < vector <vertices> > MST // Stores Adj List of MST
edge MinEdge

ALGORITHM :

```
1  // removing the edge e
2  for auto it = MST[v1].begin() to it = MST[v1].end()
3       if (*it) = v2
4            MST[v1].erase(it)
5  for auto it = MST[v2].begin() to it = MST[v2].end()
6       if (*it) = v1
7            MST[v2].erase(it)
8  // Coloring the vertices
9  for auto it = VertexSet.begin() to it = VertexSet.end()
10      int color = 1
11      if visited(*it) = 0
12           DFS(*it, color)
13      color = -1
14 // Finding edge with least weight which connects the two trees
15 for auto it = EdgeSet.begin() to it = EdgeSet.end()
16      if Visited[(*it).Vertex1] ≠ Visited[(*it).Vertex2]  &  (*it).Weight < MinEdge.Weight
17           MinEdge = (*it)
18 // Add the MinEdge to the MST
19 MST[MinEdge.Vertex1].pushback(MinEdge.Vertex2)
20 MST[MinEdge.Vertex2].pushback(MinEdge.Vertex1)
21 return MST
```

Complexity :
There are five steps in the algorithm
Step 1 : Removing the edge
It is of O(n)
Step 2 : Colouring the vertices
We are using DFS which allots the color to the vertices in visited vector
It is of O(m+n)
Step 3 : Finding smallest edge
It is of O(m)
Step 4 : Adding the smallest edge to the MST
It is of O(1)
Therefore the overall complexity of the algorithm is **O(m+n)**.

Problem 22

(a) Let $(d_1, d_2, d_3, d_4) = (3, 3, 1, 1)$. Here, $\sum_{i=1}^{4} d_i = 8$ and $\forall d_i$, $d_i \leq 3$ but clearly this degree sequence can't form a graph.

(b) Given a graph G=(V,E) and with degree sequence $(d_1, d_2, ..., d_n)$ such that $d_1 \geq d_2 \geq ... \geq d_n$.

(a) It is given that v_1 has d_1 neighbors but not $v_2, v_2, v_3, ..., v_{d_1+1}$. So, $\exists$ j such that $(v_1, v_j) \in E$ and $d_1+1 < j \leq n$ and also, $\exists$ i such that $(v_1, v_i) \notin E$ and $2 \leq i \leq d_1+1$. This implies that $i < j \leq n$. By definition, v_j has d_j neighbors, and v_i has d_i neighbors. Also, as $i < j$, this implies that $d_i \geq d_j$. Now, as v_1 is adjacent to v_j but not to v_i and $d_i \geq d_j$, both these statements imply that $\exists\ u(\neq v_j)$ such that it is a neighbor of v_i but not of v_j. Hence, we can say, $(v_1, v_i), (u, v_j) \notin$ E and $(v_1, v_j), (u, v_i) \in$ E.

(b) As from the previous part, find a v_i, v_j and u satisfying the condition from above part. It is guaranteed to find such vertices as proven above. Now, with same vertex set, construct a new edge set E' where

$$E' = E \cup \{(v_1, v_i), (v_j, u)\}\ /\{(v_1, v_j), (v_i, u)\}$$

Note that for vertices other than these 4, the edges are not changed. For these 4 vertices, v_1, v_i, v_j and u, the degrees remain unchanged. Hence, the degree sequence for graph G'=(V,E') has the same degree sequence as G and $(v_1, v_i) \in$ E'.

(c) To obtain such a graph, we can simply iterate the above procedure until the neighbors of v_1 are v_2, v_3,..., v_{d_1+1}. Note that this process will always terminate because d_1 is finite. Hence, there exists a graph with the given degree sequence where v_1 has v_2, v_3,..., v_{d_1+1} as neighbors.

(c) **Variables Used**
n = number of vertices
D = vector containing degree sequences

bool Check(int n, vector<int> & D)

```
1  if n==1
2       if D[0] == 0
3            // D[0] is d1
4            return true
5       else
6            return false
7  else
8       sort(D)
9       // dn ≤ ... ≤ d2 ≤ d1
10      // here, last element in D i.e D[n-1] = d1
11      if D[n-1]≥n or D[n-1]<0
12           return false
13      else
14           d1 = D[n-1]
15           erase last element of vector
16           for i = n-2 to n-2-d1+1
17                D[i] = D[i] - 1
18           return Check(D, n-1)
```

Correctness:
Proof by induction:
Base Case: n = 1, when there is 1 vertex, degree has to be zero. So, base case is true.
Induction Hypothesis: Let the algorithm be true for n=k.
Induction Step: We will prove for n=k+1
Let $d_1, ..., d_{k+1}$ be the desired degree sequence and $d'_1 \geq d'_2 \geq ... \geq d'_{k+1}$ be the sequence

sorted in non-decreasing order. The previous part showed that if a graph exists with this sequence, then there is a graph with this sequence that also has $\{v_2, ..., v_{d'_1+1}\}$ as the neighbors of v_1. Such a graph, if v_1 is removed, has k vertices and degree sequence given by $d'_2 - 1, d'_3 - 1, ..., d'_{d'_1+1} - 1, d'_{d'_1+2}, ..., d'_{k+1}$. The induction hypothesis implies that the existence of such a graph is correctly determined by the algorithm. If such a graph exists, then the k + 1 vertex graph exists as well by attaching the new vertex v_1 to v_2, . . . , $v_{d'_1+1}$.

Time Complexity:
There is at most 1 recursive call corresponding to each d_i in the original sequence. For every d_i, there is one sorting of O(nlogn), one deletion of O(1) time as element is present at the end of the vector and d_i decrements. So, fir the complete algorithm, time complexity is $O(n^2 logn + \sum_{i=1}^{n} d_i)$. This is polynomial time over n.

Problem 23

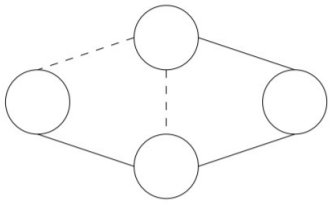
(a) **FALSE**. Any unique heaviest edge that is not part of a cycle must be in the minimum spanning tree. A counter example of this is a graph with only one edge.

(b) **TRUE**. A MST has no cycle, so at least one of the edges of the cycle can be excluded. If e is part of the MST, we can replace e with a different edge from the cycle to get a lighter MST. Therefore, e cannot be part of the MST.

(c) **TRUE**. An edge of minimum weight is trivially minimum weight edge of some cut.

(d) **TRUE**. If the lightest edge is unique, then it must be lightest edge of any cut that separates its end points.

(e) **TRUE**. If e were not the lightest edge across some cut of G, then we could always replace it to get a lighter MST.



(f) **FALSE**. The dashed edge in the picture above is not part of the MST even though it is the lightest edge in the left cycle.

(g) **FALSE**. The Dijkstra's algorithm will use the heaviest edge of a cycle if it is on shortest path from the start node a to a node b.

(h) **FALSE**. From (g), the shortest path from a to b was not part of the MST.

(i) **TRUE**. Prim's algorithm always adds the lightest edge between the visited vertices and the unvisited vertices, which is the lightest edge of this cut. Negative weights does not affect this property.

(j) **TRUE**. Suppose that a graph G contains an r-path from a node s to t but that there is an MST T of G that does not contain an r-path from s to t. Then T contains a path from s to t with an edge e of weight w_e r. Consider the partition ($S|V$-S) of vertices made by removing e from T. One vertex $e^{'}$ of the r-path must be along this cut, as the r-path connects s and t. Since $w_{e'} < r$, we can swap $e^{'}$ for e to get a spanning tree that is lighter than T, a contradiction.

Problem 24

24) Given directed graph with positive edge lengths, we need to return the length of shortest cycle in graph

Approach:-

Find the length of shortest cycle containing a vertex s, $s \in V$, and do this for all $s \in V$ and take minimum of it, this will be the length of shortest cycle in graph

To find length of shortest cycle containing vertex s we can do the following:-

1) Apply dijkstra's algorithm to find shortest distances $d(s,v)$ from s to each vertex v

2) then shortest cycle's length containing s found by $\min_{v \in V \& (v,s) \in E} \{d(s,v) + \ell(v,s)\}$.

Correctness:-

A shortest cycle containing s must be composed of some path from s to a vertex v, followed by an edge (v,s) so shortest cycle must be $\min_{(v,s) \in E \; v \in V} \{d(s,v) + \ell(v,s)\}$.

It's trivial that length of shortest cycle containing in Graph is minimum of {length of shortest cycle containing a vertex s, $s \in V$}.

Algorithm:-

Shortest cycle (G, w)

```
1.  min = Inf
2.  for each vertex v ∈ V
3.      Dijkstra(G, w, s)
4.      for each vertex v ∈ V and v ∉ s
5.          if(s in G.adj[v])
6.              min = Minimum(min, (v.d + w(v, s)))
7.  return min.          // if min is INF then no cycle else
                            it is length of smallest cycle.
```

Dijkstra (G, w, s)

```
1.  Initialize-single-source(G, s)
2.  S = ϕ
3.  Q = G.V          // Q is a priority queue (min heap) where
                        // keys are vertices and values are v.d.
4.
4.  while Q ≠ ϕ
5.      u = Extract-Min(Q)
6.      S = S ∪ {u}
7.      for each vertex v ∈ G.Adj[u]
8.          Relax(u, v, w)
```

```
Initialize-single-source(G,s)
1. for each vertex v ∈ G.V
2.         v.d = ∞
3.         v.π = NIL          // this statement can be ignored as
                              // the problem doesn't require reconstruction
4.    s.d = 0
```

```
Relax(u,v,w)
1. if v.d > u.d + w(u,v)
2.         v.d = u.d + w(u,v)
3.         v.π = u.
```

Time complexity:-

Time taken to run - Dijkstra's Algorithm is $O(|E|+|V|\log|V|)$
and time for calculating minimum is $O(|E|+|V|)$
so for one iteration of for loop at (2) in (shortest-cycle function)
it takes $O(|E|+|V|\log|V|)$ time
there are $|V|$ such iterations
so total Time complexity is $O(|E||V|+|V|^2\log|V|)$
$= O(|V|^3)$.

References:-

1. Taken Dijkstra's Algorithm implementation from clrs book.

Problem 25

Explanation

The idea behind the solution is pretty simple. Let the graph be $G = (V, E)$ and the given edge be $e = (x, y) \in E$, the shortest cycle containing the edge e can be obtained by finding the shortest path between the end points x and y in the graph $G \setminus \{e\}$ and then adding the edge e to get the shortest cycle containing the edge e. Finding the shortest path between the end points x and y in $G \setminus \{e\}$ can be done using Dijsktra's shortest path algorithm.

Pseudocode

Let the graph be $G = (V, E)$, where $|V| = n$ and $V = 1, 2, 3, ..., n$. We assume that the adjacency matrix of G is available as an attribute $G.Adj$ and $G.Adj[u, v] = l_{(u,v)}$ or 0 depending on whether an edge (u, v) exists or not. The distances of vertices from u are stored in an array $dist[1\,..\,n]$. Then calling FIND-SMALLEST-CYCLE-CONTAINING-EDGE$(G, e = (x, y), n)$ returns the correct answer, where $e = (x, y)$ is the given edge.

```
MINDISTANCE(dist[1..n], sptSet[1..n])
    min = ∞
1   for i = 1 to n
        if sptSet[i] == 0 and dist[i] ≤ min
            min = dist[i]
            min-index = i
    return min-index
```

```
DIJSKTRA(G, u, v, n)
    Let dist[1..n] be a new array where each element is initialized to ∞
    Let sptSet[1..n] be a new array initialized to 0.
    dist[u] = 0
1   for t = 1 to n − 2
        i = MINDISTANCE(dist, sptSet)
        sptSet[i] = 1
2       for j = 1 to n
            if (sptSet[j] == 0 and G.Adj[i][j] ≠ 0 and dist[i] ≠ ∞
    and dist[i] + G.Adj[i][j] < dist[j])
                dist[j] = dist[i] + G.Adj[i][j]
    return dist[u][v]
```

```
FIND-SMALLEST-CYCLE-CONTAINING-EDGE(G, e = (x, y), n)
    return DIJSKTRA(G \ {e}, x, y, n)
```

Analysis

The time complexity of the above algorithm arises purely due to the DIJSKTRA() and the auxiliary MINDISTANCE() procedures. The analysis is as follows, each call to the MINDISTANCE() procedure takes $\Theta(n)$ time because of the for loop in line (1) in the procedure. The DIJSKTRA() procedure has a for loop in line (1) inside which is a single call to the MINDISTANCE() procedure and another nested for loop in line (2) which takes $\Theta(n)$ time. Therefore each iteration of the for loop in line 1 takes $\Theta(n)$ time. With the for loop itself running $n - 2 \in \Theta(n)$ time, this makes the total time complexity of the DIJSKTRA() procedure $\Theta(n^2)$. Therefore the time complexity of the above algorithm is also $\Theta(n^2) \equiv \Theta(|V|^2)$

Correctness

The correctness of the above algorithm relies on the correctness of the Dijsktra's shortest path algorithm which is well established. Besides which me must also show the correctness of the result

that the smallest cycle containing the given edge can be obtained by finding the shortest path between the end points of the edge in the graph obtained by removing the edge and then adding the edge to form the cycle. This can be shown as follows, since we are given that the cycle necessarily contains the edge e, this cost cannot be minimized. The rest of the cost which arises purely due to a path between the end points of the edge which does not contain the edge. This is the only cost that can be minimized which is exactly what we are doing using the shortest path algorithm.

Problem 26

(a)

The car's fuel tank capacity is limited, so it can't traverse the edges with length l_eL. So, to check if there is a feasible route from s to t, we can do the following: make a new graph with all the edges present in G that have length $l_e \leq L$ and then do a DFS starting at s to see if t can be reached. The roads that the car cannot traverse were removed and the correctness of DFS is already established, hence this algorithm is also correct with complexity same as that of DFS, $O(V + E)$.

(b)

Explanation

For this, we have to find the path such that the the maximum length of a road (or the maximum weight of an edge) in that path is minimum. Thus we modify Dijkstra's algorithm to give us such a path. For this, we only need to change the edge relaxation condition to be : update $dist[v]$ as $Max(dist[u], w(u, v))$ if $dist[v] > Max(dist[u], w(u, v))$ where $w(u, v)$ is the length of the road or the weight of the edge. So, at the end of the algorithm, $d[i]$ will have the value of the minimum fuel tank capacity needed to reach i from s.

Pseudocode

The code is identical to Dijkstra's algorithm, with the only change being in lines 9 and 10, which is the edge-relaxation conditon.

```
ModifiedDijsktra(G, s, t, n)
 1  Let dist[1..n] be a new array where each element is initialized to ∞
 2  dist[s] = 0
 3  H = makequeue(V) (using dist-values as keys)
 4  While H is not empty
 5      u = deleteMin(H)
 6      sptSet[i] = 1
 7      for j = 1 to n
 8          if (sptSet[j] == 0 and G.Adj[i][j] ≠ 0 and dist[i] ≠ ∞ and Max(dist[i], G.Adj[i][j]) < dist[j])
 9              dist[j] = Max(dist[i], G.Adj[i][j])
10              prev[i] = j
11              decreaseKey(H, j)
12  return dist[s][t]
```

Correctness

Like the original Dijkstra's algorithm stored the length of the shortest path to a vertex i in $d[i]$, this modified algorithm stored the value of the minimum fuel capacity required to reach i in $d[i]$. The proof of correctness is trivial as it would be the same as that of original Dijkstra's, but with the updated relaxation condition.

Analysis

The subroutines used are all the same as that of the original Dijkstra's algorithm, so the time complexity would also be the same, $O((|V| + |E|) \log V)$.

Problem 27

We solve this by using a modified version of Dijkstra's Algorithm implemented using a min-heap.

Let us assume that there exist two different paths connecting s and u. Now we have two cases:

i) The two paths share the last edge leading to u.
ii) The last edge is distinct.

If case (i) is true, then we can check the same by changing Dijkstra's Algorithm to check if a node has been added to the known region earlier that has the same distance from s.

Pseudocode:
We implement the Dijkstra's Algorithm using a min-heap H. All the values in the array usp have been initialized to $true$.

while H is not empty
  $u = deleteMin(H)$
  for all $(u, v) \in E$ do
    if $dist(v) > dist(u) + l_{(u,v)}$ then
      $dist(v) = dist(u) + l_{(u,v)}$
      $decreaseKey(H, v)$
      $usp[v] = usp[u]$
    $elseif dist(v) = dist(u) + l_{(u,v)} then$
      $usp[v] = false$

Proof of correctness:
The algorithm finds the shortest path by Dijkstra's algorithm. Let p_0 be a shortest path given by the Dijkstra's algorithm. Now, if this path is not unique, let p_1 be another path such that these two are distinct. Now, these two paths will either have a common final edge (w, u) (for some vertex w), or not. If the y do have a common edge, then there must be two unique paths connecting s and w. Therefore with an inductive argument, which assumes that we have already set the array usp for all the vertices less that are closer than u, which is checked in the first if statement when it reaches w and searches for the shortest path to u and finds that there exist multiple shortest paths from s to w. However, if the last edge is not common, the algorithm detects the existence of multiple statements with the second if statement. The base case is also valid as there exists a unique shortest path from each vertex to itself. The algorithm takes $O((|V| + |E|) \log |V|)$ time because it uses the binary heap implementation of Dijkstra's algorithm which has the same time complexity..

Problem 28

28)

Approach:

The idea is to modify Dijkstra's algorithm to accomodate the updation of best[v] for every vertex u. The following psedo code explains how it is done:

Pseudo code:

```
while X is not empty   // X is set of
                       // remaining vertices

// best[start] = 0  initially
// best[v] for other vertices = infinite initially
u* ← vertex in X with smallest dist[]
for all edges (u,v) ∈ E
    if dist(v) > dist(u) + l(u,v)
        dist(v) ← dist(u) + l(u,v)
        best(v) ← best(u) + 1
    if dist(v) = dist(u) + l(u,v)
        if best(v) > best(u) + 1
            best(v) ← best(u) + 1
```

} Let's call this (1)

Proof of correctness:

From ①, we can see that if there are multiple possible shortest paths, it picks the one path which minimises best [v].

There is no question of choice, if there is just a single shortest path.

Runtime analysis:

Since we are adding only constant operations and modifying the dijkstra's algorithm, the time complexity is same as dijkstra's algorithm which is $O(V+E \log V)$.

Problem 29

Explanation

We can transfer the weight of a vertex to all of its incoming edges so that there are no weights on the vertices. For all vertices we need to add its weight to all its incoming edges. This way any path which contains this vertex will also contain the weight of this vertex. Now we can use Dijkstra's algorithm to find the shortest path. Finally we need to add the weight of the source vertex to all the paths.

Pseudocode

SHORTEST-PATH(G, s)

```
1  for vertex in G.V
       for edge in G.E[vertex]
           edge.weight = edge.weight + vertex.weight
2  cost[1..n] be an array initialised to −∞
3  vis[1..n] be a boolean array initialised to 0
4  let Q be a min priority queue to store the costs and corresponding vertices pairs of G
5  cost[s] = s.weight
6  insert (s.weight, s) into Q
7  while Q is not empty
       (u, d) = pop from Q
       remove (u, d) from Q
       if vis[u] == 1
           continue
       vis[u] = 1
       for edge in G.E[vertex]
           alt = cost[u] + edge.weight
           if alt < cost[edge.v]
               cost[edge.v] = alt
               insert (alt, edge.v) into Q
8  return cost
```

Analysis

Line 2 to line 6 is Dijkstra's algorithm and has a time complexity of $\mathcal{O}(V + E \log V)$ and line 1 has time complexity of $\mathcal{O}(V + E)$. Therefore overall time complexity is $\mathcal{O}(V + E \log V)$.

Correctness

For correctness, we need to prove that the problem we are solving after weight transformation is equivalent to the original problem. For this consider any path in the original graph, P weight cost c, in the transformed graph the same path will have cost $c - c_s + c_s = c$. Similarly for a path in the transformed graph P' with cost c', the same path in the original graph will also have cost c'. Therefore the problem we are solving is equivalent to the original problem. The correctness of the algorithm now relies on the correctness of Dijkstra's algorithm.

Problem 30(a)

FALSE :-
Consider a graph with nodes A,B,C, cost of edge (A,B) is 3, edge (A,C) is 1 and (B,C) is 2. There are 2 shortest paths from A to B both of length 3.
Hence proved.

Problem 30(b)

FALSE :-
Consider a graph with nodes A,B,C, arcs (B,A), (A,C) and (C,B), each having cost 1. In the directed graph, shortest distance from A to B is 1 + 1 = 2 if it becomes un-directed, the shortest distance from A to B now becomes 1. Clearly the shortest distance is changed.
Hence Proved.

Problem 30(c)

FALSE :-
Consider an un-directed graph on nodes A,B,C. Edges (A,B) and(B,C) have cost 1 and edge (A,C) has cost 3. Length of shortest A to C path is 2. If each edge cost is increased by k= 10 the shortest path length becomes 13. But 13 - 2 is not a multiple of 10.
Hence Proved.

Problem 30(d)

FALSE :-
Consider an un-directed graph on nodes A,B,C. Edges (A,B) and(B,C) have cost 11 and edge (A,C) has cost 15. Length of shortest A to C path is 15. If each edge cost is decreased by k= 10 the shortest path length becomes 2. But 15 - 2 is not a multiple of 10.
Hence Proved.

Problem 30(e)

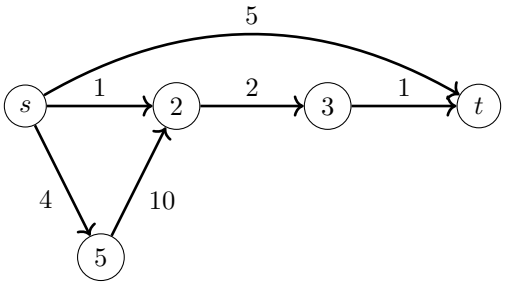
FALSE :-
Consider a graph with nodes A,B,C. Let the edges be (A,B) with weight 3, (B,C) with weight 5, and (A,C) with weight 8. Now consider the shortest path from A to C. Both the paths A - B - C, and A - C have equal weights. However, since Dijkstra's algorithm works on a greedy approach, it will choose the path with each edge having the least possible value and hence in this example, it will choose the path A - B - C. But this is not the one with the least number of arcs.
Hence proved.

Problem 31

31
A vital arc of a network is an arc whose removal from the network causes the shortest distance between two specified nodes, say node s and node t, to increase. A most vital arc is a vital arc whose removal yields the greatest increase in the shortest distance from node s to node t. Assume that the network is directed, arc lengths are positive, and some arc is vital.

(a) A most vital arc is an arc with the maximum value of C_{ij} .

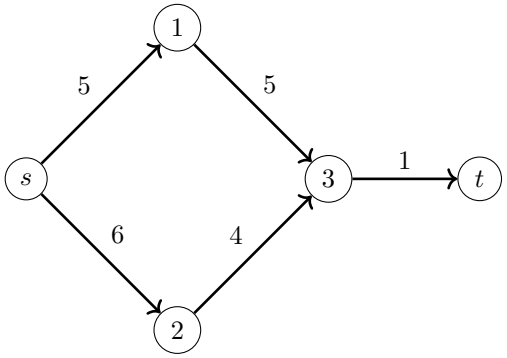
This statement is false .
This is shown by the counterexample to this statement shown below :



The shortest distance between nodes $'s'$ and $'t'$ is 4. Shortest distance comes along the path s-2-3-t.
The maximum value of C_{ij} is $C_{5,2} = 10$. Now, suppose the arc (5,2) is removed.
The shortest distance between nodes $'s'$ and $'t'$ after removal of arc(5,2) is 4.
Hence, shortest distance between s and t did not increase on removal of arc with maximum C_{ij}.
The most vital arc is the arc (2,3) or (3,t) or (s,2). Since the shortest distance from s to t on removal of any one of the above becomes 5.
Hence, it is possible that the most vital arc does not have the maximum value of C_{ij}.

(b)A most vital arc is an arc with the maximum value of C_{ij} on some shortest path from node s to node t.

This statement is false.
This is shown by the counterexample to this statement shown below:



In this example there are 2 shortest paths from s to t i.e.
s - 1 - 3 - t and s - 2 - 3 - t
The shortest distance from s to t is 11.
Consider the path s - 2 - 3 - t.
The arc with maximum value of C_{ij} is (s,2). $C_{s,2} = 6$.
On removing arc (s,2) shortest distance between the nodes s and t is 11.
Hence, the shortest distance between nodes s and t did not increase on removing an arc with max-

imum value of c_{ij} on some shortest path from node s to node t.
The most vital arc here is (3,t) because on removing this arc the distance from s to t becomes infinite and clearly $C_{3,t} < C_{s,2}$.
Hence, it can be the case that most vital arc is not an arc with the maximum value of C_{ij} on some shortest path from node s to node t.

(c) An arc that does not belong to any shortest path from node s to node t cannot be a most vital arc

This statement is true.
Proof:
Suppose an arc (i,j) that does not belong to any shortest path from s to t and is the most vital arc.
Let the path that gives the minimum distance from s to t be called P.

$$(i, j) \notin P$$

Let the minimum distance from s to t be called d_{min}.
On removing the arc (i,j) the minimum distance between s and t increases.(Since, (i,j) is most vital arc)
Say this new minimum distance from s to t is d_{min2}.
Since,

$$(i, j) \notin P$$

s and t are still connected via path P.
So, minimum distance from s to t comes from path P = d_{min}.
Hence,

$$d_{min} = d_{min2}$$

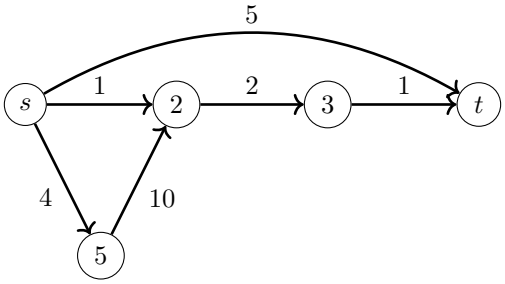
But

$$d_{min} < d_{min2} \text{ (} Since,\ (i, j)\ is\ most\ vital \text{)}$$

This contradicts our assumption that arc (i,j) is most vital.
Hence, our assumption is incorrect.
Therefore, an arc that does not belong to any shortest path from node s to node t cannot be a most vital arc.

(d)A network might contain several most vital arcs.

This statement is true.
Consider the network :



The Minimum distance from node s to node t is 4.
There are 3 most vital arcs here:
(s,2) and (2,3) and (3,t)
Because on removal of any 3 arcs the minimum distance from node s to node t increases to 5 which is the maximum possible increase in the minimum distance from node s to node t.
Hence, a network might contain several most vital arcs.

Problem 32

Following from the solution given for Q-31, we gather certain outcomes:
1) A most Vital arc if exists will lie on the shortest path from node s to node t.
2) Removing any edge from the shortest path will always lead to increasing or keeping constant the shortest path.
3) There can be more than 1 most Vital arc.

We realise that since the most Vital arc will lie on the shortest path, if we remove that edge and perform a shortest path algorithm like Dijkstra's the shortest path between s and t will increase the most in value. Therefore our objective left is to first compute the shortest path in the original graph and then iterate over all its edges removing them in that particular iteration and finding the max increase over all these iterations to find a Most Vital Arc. We also add back the edges for the next iterations. We assume that in our implementation of shortest path algorithm (Dijkstra's) it returns the edge list corresponding to the shortest paths in the form of a parent array and the Shortest Distance also as an array.

Pseudocode:

```
1  PArray,DArray = G.Dijkstra(s)
2  v = t
3  while v ≠ s
4        ShortestPath.push(edge(PArray(v),v)))
5        v = PArray[v]
6  OriginalD = DArray[t]
7  MVArc = edge(-1,-1)
8  MaxDiff = 0
9  for edge in ShortestPath
10       G.remove(edge)
11       PArray,DArray = G.Dijkstra(s)
12       if MaxDiff DArray[t] - OriginalD
13            MVArc = edge
14            MaxDiff = DArray[t] - OriginalD
15       G.add(edge)
16 if MVArc = edge(-1,-1)
17       return any edge from ShortestPath
18 return MVArc
```

1) First we call Dijkstra's in the starting graph. Now we find the shortest path's edges between s,t.
2) Now we iterate over all these edges, deleting them only in their respective iteration and applying Dijkstra's to get the length of the shortest path s-t in this modified graph.
3) Then we compare the difference from the original value and take the maximum of all these differences.
4) If we cannot find any edge such that the Shortest Distance increases then we can say that each of the edges in the shortest path of the original graph is a Most Vital Arc.
5) Otherwise we return the edge that caused the biggest difference.

Time Complexity:
Time Complexity of Dijkstra's: O(V + Elog(V))
Now since we also call Dijkstra's size of the shortest path s-t in original graph, it is called O(V) times.
Getting the edges of the shortest path s-t in the original graph takes O(V) time.
Therefore the overall Time Complexity:
O(V*(V + Elog(V))) + O(V) = O(V*(V + Elog(V)))

Problem 33

33) like Dijkstra's algorithm we generate MCP set (Max capacity path tree) with given source as root. we maintain two sets, one set contains vertices included in the MCP and other not in MCP tree.

at every step we find one vertex which is not in MCP and include it.

algorithm:

1) initially MCP has only source vertex and each adjacent vertices of source vertex has values equal to capacity of edge b/w source and vertex all other vertices stores 0.

2) while MCP doesn't include all vertices

a) pick a vertex u which is not there in MCP and has maximum capacity value.

b) include u to MCP

c) update maximum capacity values of all adjacent vertices of u.

To update the capacity values iterate through all adjacent vertices

For every vertex v adjacent to u,

value of v = Max (value of v , Min(value of u, u-v edge capacity))

justification:

The capacity of a path p is the edge with minimum capacity.

so of all possible paths between source and a vertex we need to find maximum capacity path.

Now,

For vertices included in MCP the values the vertices have are the maximum capacity values from source

when we include new vertex u with maximum capacity value. ∃ a path from source to u with that value and it is maximum for u so far.

so when all the vertices are included u has maximum value by same reason as Dijkstra's has minimum path distance value.

so the algorithm is correct.

Problem 34

Detecting a negative cycle in Floyd-Warshall Algorithm
Developing a solution :

i) The Floyd-Warshall algorithm iteratively develops shortest path between as possible pairs of verticesv (i,j) and eventually the pair (i,i) which is initialised to zero in the beginning
ii) Any path [i,...k ... ,i] can only improve on the initial value zero, if it is negative i.e., if there's a negative cycle involving kth vertex.
iii) Thus in the solution matrix Mat[i][i] will be negative.

From the above conclusion, we have to check whether any of the diagonal elements of the solution matrix are negative before checking any possible shortest paths.

VARIABLES USED:
int Solution[N][N] // Stores the solution developed before the algorithm stops or completes

ALGORITHM:

```
 1  bool check = false
 2  for int ii = 0 to N-1
 3       for int jj = 0 to N-1
 4            for int kk = 0 to N-1
 5                 if Solution[kk][kk] < 0
 6                      check = false
 7                      break;
 8                 if Solution[ii][jj] > Solution[ii][kk] + Solution[kk][jj]
 9                      Solution[ii][jj] = Solution[ii][kk] + Solution[kk][jj]
10            if check = true
11                 break
12       if check = true
13            break
14  return check
```

Problem 35

In the all-pairs shortest path problem, we modify the Floyd-Warshall algorithm as follows to identify a smallest path for a given pair among all possible shortest paths for the same pair.
Variables Used:

1. n : number of vertices
2. distance[n][n]: It is a n×n matrix storing the shortest possible path where each entry (i, j) stores shortest path distance from i to j.
3. path[n][n]: It is a n×n matrix. path[i][j] = k means that the shortest path from i to j is, shortest path(i,k) and then k to j.

Also, distance is initialised to graph's weight values and non-edges are stored as infinity. Also, self loops are considered as zero(diagonal entries are initialised to zero).

Algorithm

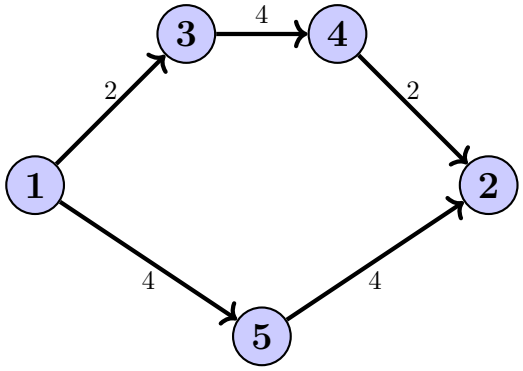
```
1  for k = 0 to n-1
2      for i = 0 to n-1
3          for j = 0 to n-1
4              // Note: here no equality is present in the if condition
5              if distance[i][j] > distance[i][k] + distance[k][j]
6                  distance[i][j] = distance[i][k] + distance[k][j]
7                  path[i][j] = path[k][j]
```

Print_Shortest_Path(u, v)

```
1  // u is source and v is destination
2  s = Stack()
3  s.push(v)
4  while path[u][v] ≠ u
5      s.push(path[u][v])
6      v = path[u][v]
7  print → u
8  while s is not empty
9      print → s.top s.pop()
```

CLAIM: Path produced by this algorithm can be of two types:

- If there is direct edge between vertex (i, j) and it is the shortest path, then path returned will be this edge.
  Proof: As, the matrices are initialised before the run of algorithm, if the edge i-j is the minimum path between (i,j), then as path[i][j] is updated only when distance[i][j] > distance[i][k] + distance[k][j], this condition will never satisfy and this edge will be returned as the shortest path.
- Otherwise, let there be multiple shortest paths $p_1, p_2, ...p_m$. Remove the starting and ending element. Considering only the internal vertices now, sort each path in the decreasing order of vertices. Now the answer, the path chosen by the Floyd-Warshall algorithm, is the sorted path(with only internal vertices) coming first in lexicographical order.



For example, in this figure, for vertex 1-2, two shortest paths are possible, 1-3-4-2 and 1-5-2. Sorting the internal vertices of each path in decreasing order, we get, (4,3) and (5). As (4,3) comes first in lexicographical order, answer is path 1-3-4-2 or path 1.

Proof: Let there be multiple shortest paths $p_1, p_2, ..., p_m$. Let after removing starting and ending element and sorting the remaining vertices(internal) in decreasing order, we get paths as $p'_1, p'_2, ..., p'_m$. As all the vertices in this path p'_1 comes first(lexicographical sorting of paths), they will be updated in the algorithm and thus this path will be updated first by the algorithm. After updating this path, no other path will be updated as we have already found a shortest path. Hence, it will **NOT** be the path with minimum number of arc but the path shown above.

Problem 36

Floyd-Warshall algorithm is a strictly decreasing algorithm. This means that the distance matrix will get updated if and only if the new distance is strictly lower than the previous distance. Now $d[i, i]$ in a matrix is the distance to itself which initially is set to 0, so the only way it becomes negative if there is a negative cycle that contains the node i. If there are multiple negative cycles, we will pick the one with the least number of vertices and try to show this.

Assuming that the cycle has at least two vertices, let k be the highest numbered vertex in the cycle (since we chose the smallest cycle, this will be the least indexed node satisfying the property that a negative cycle exists in the network), and let i be some other vertex in the cycle. ${d_i}_k^{(k-1)}$ and ${d_k}_i^{(k-1)}$ have correct shortest-path weights, because they are not based on the negative weight cycle. (Neither ${d_i}_k^{(k-1)}$ nor ${d_k}_i^{(k-1)}$ can include k as an intermediate vertex, and i and k are on the negative weight cycle with the fewest vertices.) Since i to k to i is a negative weight cycle, the sum of those two weights is negative, d_i^k will be set to a negative value. This value will remain negative at the termination of the algorithm as it's never increased. Therefore, we can say that if there exists a negative weight cycle in a network, $d_i^k < 0$ at some stage will occur with Floyd-Warshall Algorithm.

Problem 37

37)

d[i,i] in FLOYD-WARSHALL algorithm means shortest distance from i to i among all paths from i to i with "0" intermediate distance vertices.

so if we make d[i,i] = INF then at some iteration K+1 such that there exist a path from i to i with only (1 to "K") intermediate vertices and there will be some smallest among them and let it be $d^{K+1}[i,i]$

as $d_{ii}^{(K+1)} = \min(d_{ii}^{(K)}, d_{ik}^{K} + d_{ki}^{K})$

where $d_{ij}^{K+1} = d^{K+1}[i,j]$

so if d[i,i] = 0 then all the time d_{ii}^{K} ; $k \geq 1$ & 0

where as if we make d[i,i] = Inf then we get some length of shortest cycle if it is present.

And if we take minimum among all d_{ii}^{n+1} $i \in \{1,\dots n\}$ then we get length of shortest cycle in graph.

Note that d_{ii}^{K+1} is length of shortest path from i to i which atleast one intermediate vertex (in $v \in \{1,2,3,\dots K\}$ if d_{ii}^{K+1} if Inf., so basically it is a cycle containing i

$\therefore$ $\min\{d_{n+1}[i,i] : 1 \leq i \leq n\}$ is the minimum length of a directed cycle in G. and it is INF if there is no cycle.

Problem 38

Explanation

Let the directed graph be $G = (V, E)$. We are given that an existing arc (p, q) with weight c_{pq} is changed to c'_{pq} such that $c'_{pq} < c_{pq}$.
(1) : If $d_{qp} + c'_{pq} < 0$, then the new graph has a negative cycle.
We first assume that the original graph did not contain a negative cycle, because if it did, checking for a negative cycle again is redundant. We know that d_{qp} is the length of the shortest path from vertex q to p. Let this path be $q \rightsquigarrow p$. On adding the new arc (p, q) to this path, we get a cycle $q \rightsquigarrow p \rightarrow q$ of weight $d_{qp} + c'_{pq}$. If this weight is less then 0, then this cycle has a negative weight, and therefore we have proved the existence of a negative cycle.
(2) : If $d_{qp} + c'_{pq} \geq 0$, then new the new graph has no negative cycles.
If a negative cycle exists, it must contain the new arc (p, q) because the rest of the cycles are also present in the original graph which did not have a negative cycle. We can also observe that the cycle obtained above is the smallest weight cycle containing the arc (p, q). Therefore, if $d_{qp}+c'_{pq} \geq 0$ then, any cycle containing the edge (p, q) has a positive weight and therefore no negative cycles exist in the new graph.
(3) : In the new graph, $d'_{ij} = min\{d_{ij}, d_{ip} + c'_{pq} + d_{qj}\}$.
Case 1 : The shortest path between vertices i, j in the original graph does not contain the edge (p, q). In this case, the shortest path between the vertices i, j in the new graph has 2 possible candidates,
1 : The shortest path in the original graph of weight d_{ij} or
2 : The shortest path containing the new edge (p, q). This path can be obtained by appending the shortest path in the original graph from i to p of weight d_{ip}, the edge (p, q) of weight c'_{pq} and the shortest path in the original graph from q to j of weight d_{qj}. This path has a total weight of $d_{ip} + c'_{pq} + d_{qj}$.
Therefore in this case $d'_{ij} = min\{d_{ij}, d_{ip} + c'_{pq} + d_{qj}\}$.
Case 2 : The smallest weight path from i, j in the original graph contains the edge (p, q). This path can be expanded as the concatenation of the shortest path between the vertices i and p, the old edge (p, q) and the shortest path from vertex q to j. This path has a length of $d_{ij} = d_{ip}+c_{pq}+d_{qj} > d_{ip} + c'_{pq} + d_{qj}$, therefore the shortest path in the new graph is same as before but with the new edge (p, q). This path has a weight $d'_{ij} = d_{ip} + c'_{pq} + d_{qj} = min\{d_{ij}, d_{ip} + c'_{pq} + d_{qj}\}$.
Therefore, in both the cases, $d'_{ij} = min\{d_{ij}, d_{ip} + c'_{pq} + d_{qj}\}$.

Problem 39

Explanation

Let the directed graph be $G = (v, E)$. We are given an existing arc (p, q) with lenght c_{pq}, and the length is increased to c'_{pq} such that $c'_{pq} > c_{pq}$.
We first note that increasing the length of an arc cannot introduce a negative weight cycle.This is quite clear as increasing the weight can only increase the weight of the cycle, and so cannot introduce a negative weight cycle.

Now, for any two vertices i and j, we have the following cases:
Case (1) : The shortest path between vertices i, j in the original graph does not contain the edge (p, q). In this case, $d_{ij} < d_{ip} + c_{pq} + d_{qj}$. This is easy to show as it follows Floyd-Warshall algorithm. In this case, the shortest path would remain the same as $c'_{pq} > c_{pq}$ and so $d_{ij} < d_{ip}+c_{pq}+d_{qj}$.

Case (2) : The shortest path contains the edge (p, q) but there are other paths that do not contain (p, q) of the same length. In this case, $d_{ij} = d_{ip} + c_{pq} + d_{qj}$ but there's another path that does not contain (p, q). In this case, the length of shortest path does not change.

Case (3) : The shortest path contains the edge (p, q), and all the shortest paths contain the edge (p, q). In this case, $d_{ij} = d_{ip} + c_{pq} + d_{qj}$ and the length of (p, q) is increased. So now, the shortest path may change and we have to find a new shortest path by running the algorithm again.

Thus, it follows that $d_{ij} \leq d_{ip} + c_{pq} + d_{qj}$ and equality holds when a shortest path passes through (p, q).
So, we cannot modify the method described in question Q38 because there is no way to distinguish cases (2) and (3). So, every time $d_{ij} = d_{ip} + c_{pq} + d_{qj}$, we will have to find a new shortest path. This is the difficulty encountered in trying to re-optimise shortest path distances in $O(n^2)$ time as done in Q38. We have to run the Floyd-Warshall algorithm again and it would take $O(n^3)$ time.

Problem 40

Explanation

In any all-pairs-shortest-path problem on a graph $G = (V, E)$, the weights of the edges are represented as follows,

$$w_{ij} = \begin{cases} 0, & \text{if } i == j \\ \text{Weight of the edge } e, & \text{if } e \in E \\ \infty, & \text{otherwise} \end{cases}$$

Using this convention, the problem of adding new arcs can be converted to the problem of modifying an existing edge weight. Let the new edge be added between (u, v) of edge weight c'_{uv}. Since this edge did not exist before, $c'_{uv} < c_{uv} = \infty$. So, we can apply the same updating procedure as in Problem 38 to obtain the new set of all-pair-shortest-paths in $\mathcal{O}(n^2)$ time. If we have to add 5 new arcs, successively add them one-by-one taking $\mathcal{O}(5n^2) \equiv \mathcal{O}(n^2)$ time.

Pseudocode

Let the original matrix result of the Floyd-Warshall or some other all-pairs-shortest-path algorithm be kept in $D[1\,..\,n][1\,..\,n]$, where $D[i][j]$ gives the shortest path between the vertices i and j. Let the edge updated/added be between (p, q) with new edge weight w.
To add 5 new edges, successively call the below procedure with the 5 edges and keep updating the matrix orignal matrix D with the one returned from the previous call to UPDATE-EDGE-WEIGHT

```
UPDATE-EDGE-WEIGHT(D[1..n][1..n], p, q, w)
    Let M[1..n][1..n] be a new array
    for i = 1 to n
        for j = 1 to n
            M[i][j] = MIN(D[i][j], D[i][p] + w + D[q][j])
    return M[1..n][1..n]
```