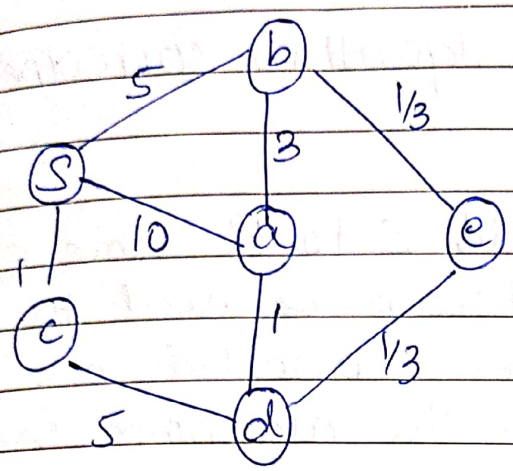
CS 660 - HOMEWORK I

Q1.



Applying dijkstra's algo.

Selected source vertex	destination vertex				
	a	b	c	d	e
→ s	10	5	(1)	∞	∞
s, c	10	(5)	1	6	∞
s, b	8	5	1	6	(5+1/3)
s, b, e	8	5	1	(5+2/3)	(5+1/3)
s, b, e, d	(6+2/3)	5	1	(5+2/3)	(5+1/3)

∴ dist(a) during execution of dijkstras from the following source vertices:

s → 10

s, c → 10

s, b → 8

s, b, e → 8

s, b, e, d → 6+2/3 ≈ 6.66.

∴ dist(d) during execution of dijkstras from the following source vertices:

s → ∞

s, c → 6

s, b → 6

s, b, e → 5+2/3 ≈ 5.66

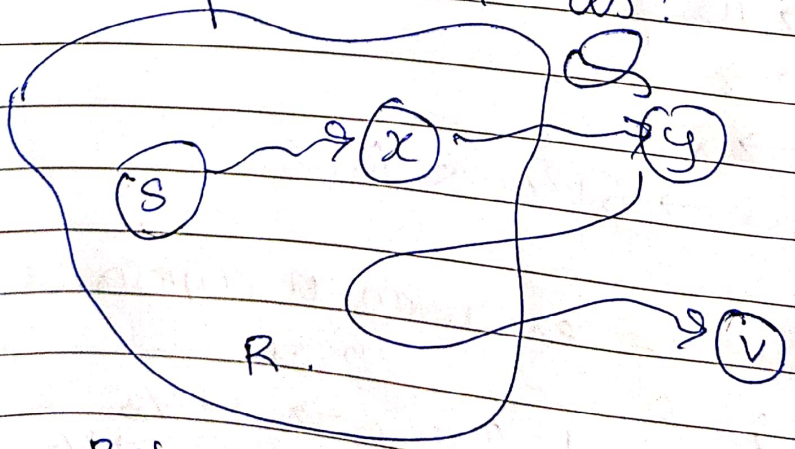
s, b, e, d → 5+2/3 ≈ 5.66.

Part b) Proving Dijkstra's algorithm correctness.

- Base Case:
The algorithm first add S to R since s is the starting vertex. There is no need to visit s again as the edges connected to s can not change later on. The algorithm checks in the list R if the node is visited or not, if the node is visited, it will skip it Or else add it.
dist(s) = 0 as it is the starting vertex and thus does not cost anything as we are already there.

- Using the Induction hypothesis:
To show dist(u) = δ(u)
It is true for current set R for all v ∈ R, we have
dist(v) = δ(v).

The diagram for the proof can be represented as:



Before

- Since Q is the shortest path from s to u. Suppose $dist(u) > l(Q)$. ... {1}
  ∵ Q leaves R at some point to reach the unvisited vertex u, there must be an edge that goes from within R to outside R which is considered as xy in the given proof.
  ∴ there exists xy where $x \in R$ & $y \notin R$ since the path needs to reach u.
- Qx be part of Q that is from s to u.
  ∵ Qx is a subpart from s to x.
  Here, $s \to x \to y \to u$ is supposed to be the path where y and u are outside R.
  ∴ $l(Qx) = s \to x$.
  But $l(Q)$ is shortest path from $s \to u$.
  ∵ $s \to x + x \to y \leq s \to u$.
  $l(Qx) + l(xy) \leq l(Q)$ ... {2}
- Since dist(x) is the length of shortest path from $s \to x$ which is equal to Qx due to the Induction hyp and the fact that $x \in R$.
  ∵ The above eqn can be written as
  $dist(x) + l(xy) \leq l(Q)$ ... {3}
- Similarly, $dist(y) \leq dist(x) + l(xy)$ because y is next to x so the dist(y) is just updated by the algorithm in the similar way as x. {4}
- But since u was selected as the next vertex, u is supposed to have the smallest /shortest distance of the next path.
  ∵ $dist(u) \leq dist(y)$ ... {5}

- Now, we can simplify the eqns as:

$dist(u) \leq dist(y)$ ... {from 5}

$\Rightarrow dist(u) \leq dist(x) + l(xy)$ ... {from 4}

$\Rightarrow dist(u) \leq l(Q_x)$ ... {from 3}

$\Rightarrow dist(u) \leq dist(x) + l(xy) \leq l(Q_x)$

... {combining above eqns 2}

$\Rightarrow dist(u) < dist(u)$ ... {from 1}

which is a contradiction

∴ no such shorter path Q exists.

and $dist(u) = \delta(u)$.

Q2. $A[1 \ldots n][1 \ldots n]$.

claim: $\sum_{i=1}^{n} \sum_{j=1}^{n} (i+j) A[i][j]$

	1	2	3	4
1	1	2	5	10
2	3	4	7	12
3	6	8	9	14
4	11	13	15	16

or

	1	2	3	4
1	1	3	6	11
2	2	4	8	13
3	5	7	9	15
4	10	12	14	16

Algorithm:

Base Cases: The bottom right corner should be the max number since it is the only number with both i & j as max i.e. n. Similarly, top left would be the min number with i & j as 1.

→ We keep on filling the respective numbers from the i^{th} ~~column~~ row & j^{th} column. from outside to inside.

i.e. first move from (4,4)
to (4,3) & (3,4)
to (4,2) & (2,4)
to (4,1) & (1,4).
and then back to 3,3 ...

Here, we check for if [i-k] == [n-1]
{ both indices i & j are equal }
if so, we square the number & place it in array.

Pseudo Code:

```
import numpy as np

def maxArrange(a,n):
    ctr = n * n;
    for i in range(n-1, -1, -1):
        for k in range(0, n):
            if([i-k] != [n-1]):
                if(a[i-k][n-1] == 0):
                    a[i-k][n-1] = ctr
                    ctr = ctr - 1
                if(a[n-1][i-k] == 0):
                    a[n-1][i-k] = ctr
                    ctr = ctr - 1.
            else:
                a[i][i] = n * n
                ctr = ctr - 1.
    n = n - 1

a = np.zeros((n,n))
b = maxArrange(a,n)
print(b).
```

{working code}

Output :

1,	3,	6,	11
2	4	8	13
5	7	9	15
10	12	14	16

Time Complexity : $O(n^2)$.

The formula $\sum_{i=1}^{n} \sum_{j=1}^{n} (i+j)\, A[i][j]$

would get max value with i+j having the highest indices & array at those position being highest during solving.

$\therefore$ 4, 4 $\Rightarrow$ $(4+4) \cdot A[4][4]$

$\Rightarrow$ $16[8] = 128$

The sum can be found by initializing sum = 0 and then :

```
for i in range[0,n]:
    for j in range[0,n]:
        sum += (i+j) * A[i][j]
```

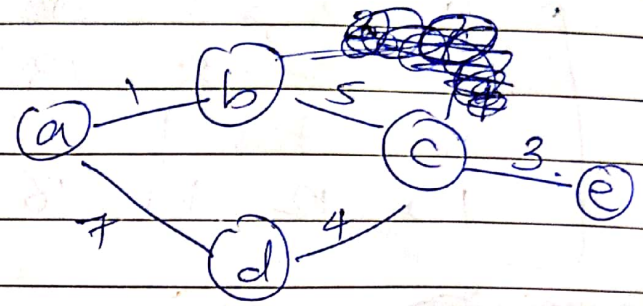
1, 1 $\Rightarrow$ $a[1][1] * (1+1)$

$2 \cdot 1 = 2$.

Pg 268 (Ex 9)

Q3 $G = (V, E)$ Graph with weighted edges.

Let mx be edge with max weight.



and let X be an MST of G.

∴ Let us consider that mx is in G.

Here mx = ~~7~~ 5 in the example.

in cycle b, d, c, a.

~~(Let there be~~ Suppose ~~we add one more~~ edge in ~~cycle~~ y

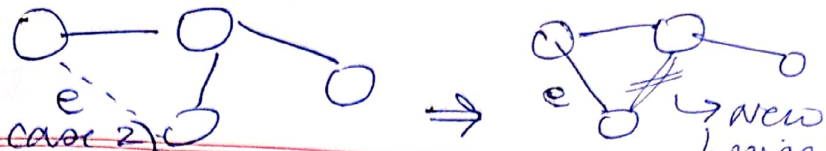
Suppose we add one more edge in X, it will form a cycle.

Let the edge be y.

If we choose an edge other than mx, we can remove the cycle and form a new Minimum spanning tree.

But this implies that mx must not be in MST which is a contradiction.

In forming a minimum spanning tree, the algorithm verifies if a cycle is formed and to get the least weights, the maximum edge is discarded.



Q5. Undirected graph G with weighted edges Min Spanning Tree T of G.

→ There are 2 cases for the problem.
Let us consider the edge with reduced wgt is e.

→ case 1: The edge e is in MST
case 2: The edge e is not in MST.

→ case 1:

If the edge e is there in the original MST, edge e is not the heaviest edge from the original graph or else it would have been eliminated nor forms a cycle. The original MST would remain the same and only the total weight would be affected with the new weight being less by the change in weight of the edge.

→ case 2:

If edge e is not there in MST, it would make the original edge e as the heaviest edge in a cycle.
Let us consider graph G' by adding the decreased e to T.
∴ G' will form a cycle.
To make G' into a new spanning tree, we need to remove the heaviest edge in that cycle which contains e.
This takes O(|V|) time.

Pg 253 (19-a).

Q4 G be a directed arbitrary graph. with V vertices having weight W(V).

To find longest increasing sequence in G:

Base case:

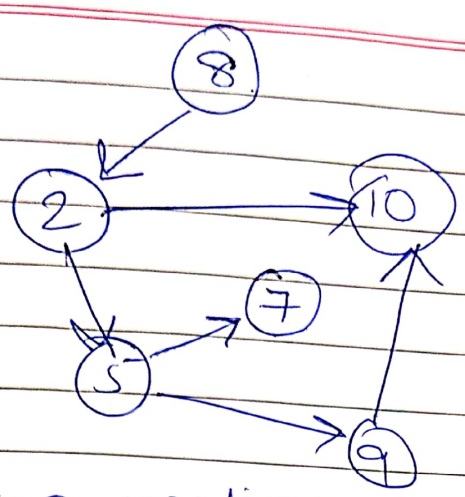
→ We make sure that the graph is a directed graph.

→ We form adjacency list of each vertex

→ We make sure that weight of the destination node is greater than weight of the source node. else discard.

1. We sort all the adjacency lists of all the vertices.
2. We then sort all the vertices & start with the smallest vertex.
3. We compare all the destination vertices of the vertex.. If wgt(destination) > wgt(source) we proceed to the adjacency list of the destination
4. If the vertex is already visited in the same list, next vertex is visited.
5. This is done until all nodes are visited.
6. Once the destination vertex is used or ∴ For all u → v i visited, a counter is incremented and vertex path is stored if counter is greater than original counter (initial or previous path)
7. Counter is reset if new vertex is used as starting node.

e.g.



2 → [5, 10]
5 → [7, 9]
8 → [2]
9 → [10]
10 → [ ]
7 → [ ].

Sorting vertices:

2 → [5, 10]
5 → [7, 9]
7 → [ ]
8 → [2]
9 → [10]
10 → [ ]

2 is used as starting.
2 → 5 [ctr = 1]
2 → 5 → 7 [ctr = 2]
2 → 5 → 7 → [ ]

2 → 5 → 9 [ctr = 2]
2 → 5 → 9 → 10 [ctr = 3]
2 → 5 → 9 → 10 → [ ]

2 → 10 [ctr = 1]
2 → 10 → [ ]

∴ The maximum counter is stored and path is
2 → 5 → 9 → 10.

for all u → v :
  if w(v) > w(u)
    go to v.next

The time complexity for sorting all the adjacency list is O(E log E) since all the edges with destination node > source node are used to sort in final list.
The (V log V) is used to sort all vertices for traversing. ∴ O(E log E + V log V).
The path is found & written in linear time.