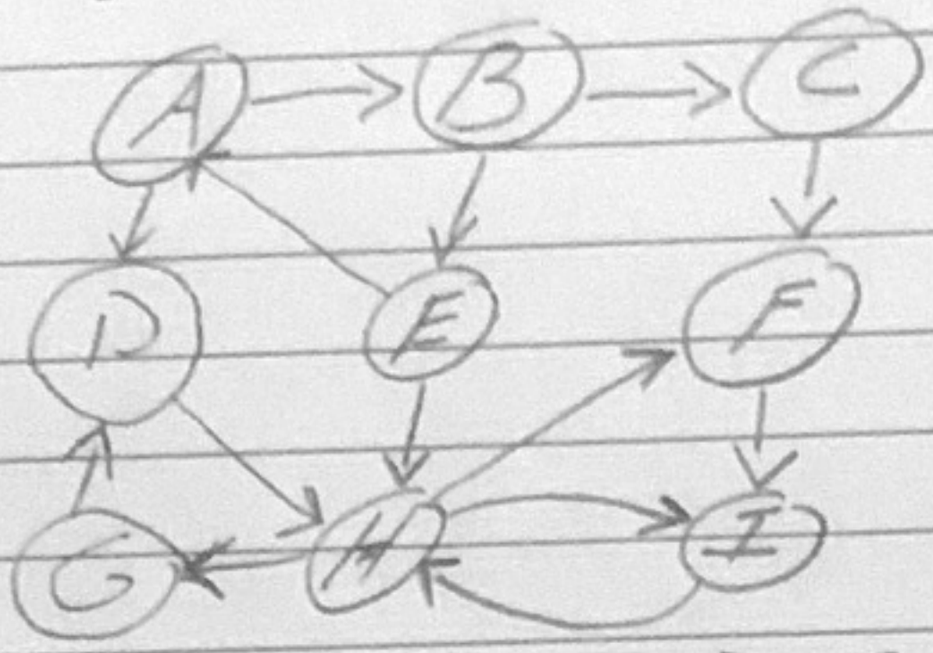
(Q1)

1- T
2- F
3 T
4- T
5- F

(Q2)

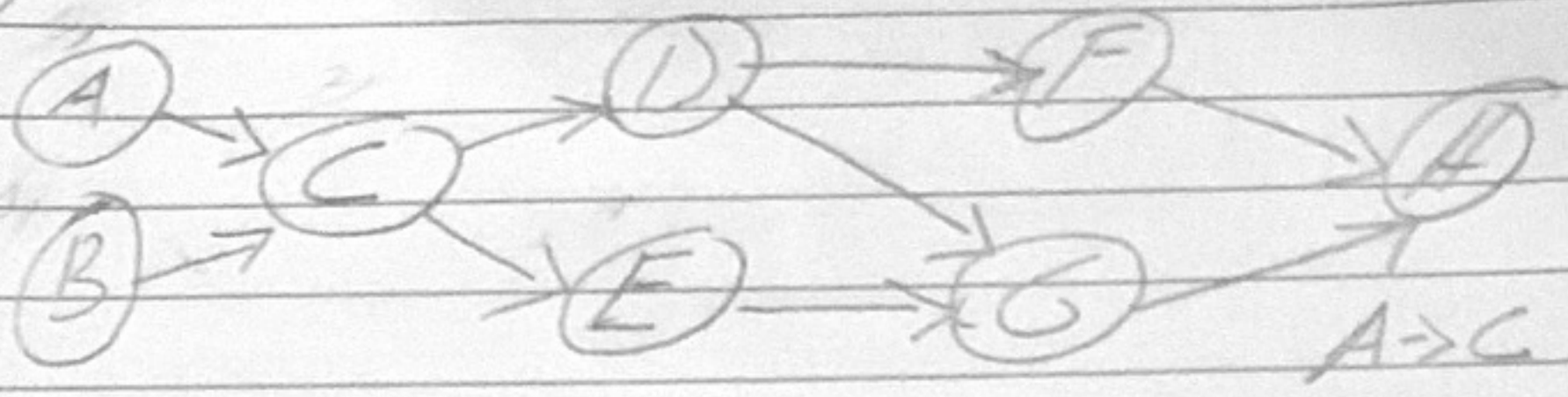


1-

	A	B	C	D	E	F	G	H	I
A	0	1	0	1	0	0	0	0	0
B	0	0	1	0	1	0	0	0	0
C	0	0	0	0	0	1	0	0	0
D	0	0	0	0	0	0	0	1	0
E	1	0	0	0	0	0	0	1	0
F	0	0	0	0	0	0	0	0	1
G	0	0	0	1	0	0	0	0	0
H	0	0	0	0	0	1	1	0	1
I	0	0	0	0	0	0	0	1	0

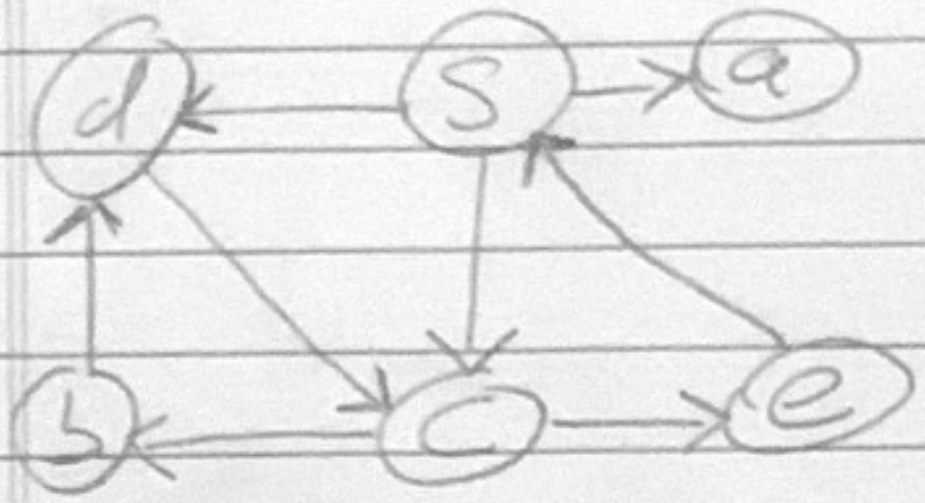
A	B, D
B	C, E
C	F
D	H
E	A, H
F	I
G	D
H	G, I, F
I	H

(Q2)

2) 

B, A, C, E, D, G, F, H

A -> C
B -> C
C -> D, E
D -> F, G
E -> G
F -> H
G -> H
H :

3) 

We will start with vertex 'S'

S a c b e

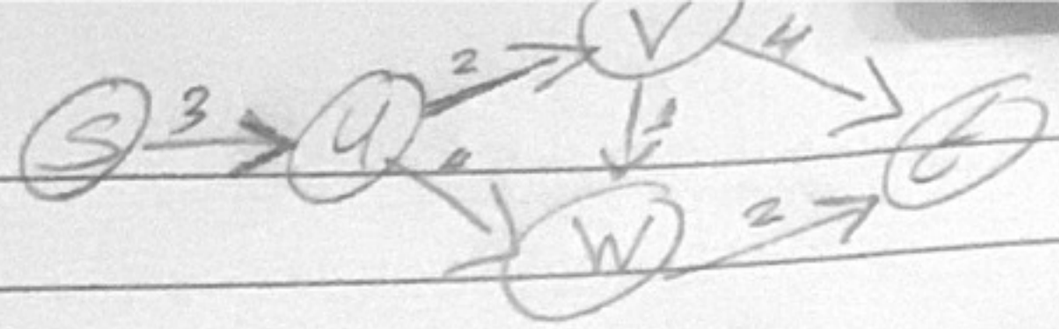
The choices for a is ~~limited~~ none, So we will go to C, we will choose b because we are following a alphabetical order then we will add e.

The shortest path from S to b is

~~S c e b~~ S C b

S
↓
B ← C

(Q3)



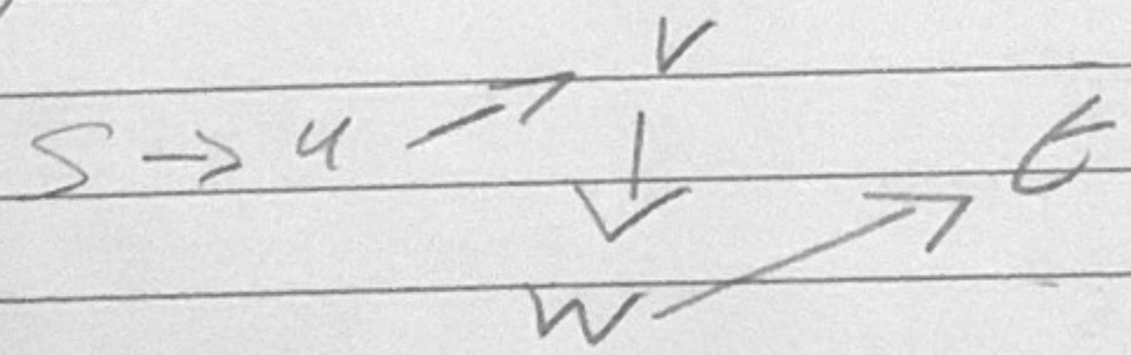
1)

	D[S]	D[u]	D[v]	D[w]	D[6]
Initialization	0	∞	∞	∞	∞
Immediately after iteration 1	0	3	∞	∞	∞
Immediately after iteration 2	0	3	5	7	∞
After iteration 3	0	3	5	6	9
After iteration 4	0	3	5	6	8

	P[S]	P[u]	P[v]	P[w]	P[6]
Initialization	None	None	None	None	None
1	None	S	None	None	None
2	None	S	u	u	None
3	None	S	u	v	v
4	None	S	u	v	w

Final	0	3	5	6	8	None	S	U	V	W

2) The path



d[6] = 8

4)

1- $\frac{(n-j+1)K}{2}$ total edges

probability that Kruger chooses K edges excluding S at j step is at most $\frac{K}{\frac{(n-j+1)K}{2}} = \frac{2}{n-j+1}$

The probability Karger does not choose one of the K edges is $1 - \frac{2}{n-j+1} = \frac{n-j-1}{n-j+1}$

$\left(\frac{6}{8}\right)\left(\frac{5}{7}\right)\left(\frac{4}{6}\right)\left(\frac{3}{5}\right)\left(\frac{2}{4}\right)\left(\frac{1}{3}\right) = \frac{2}{n(n-1)} = \frac{1}{\binom{n}{2}} = \boxed{\frac{1}{28}}$

2- $P(\text{find the min cut after } 1+\varepsilon n) \geq \frac{1}{nC_2}$ where n = total # of nodes

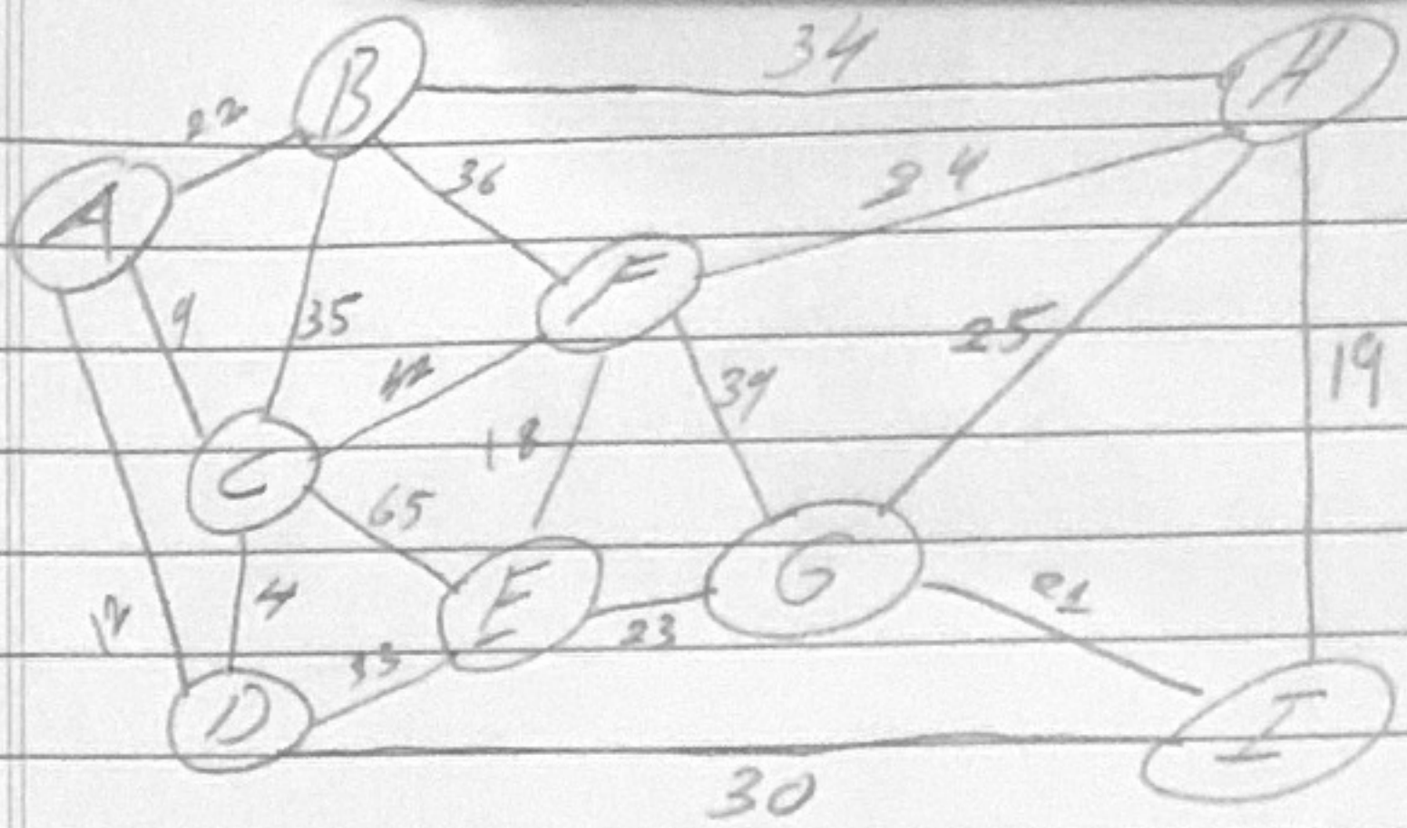
$P(\text{don't find min cut after } 1+\varepsilon n) \leq 1 - \frac{1}{nC_2}$

$T(nC_2)\ln\frac{1}{0.05}$

$= \frac{8!}{(8-2)!\,2!}\left(\ln\left(\frac{1}{0.05}\right)\right)$

$= \frac{8 \cdot 7}{2}\left(\ln\left(\frac{1}{0.05}\right)\right) = 84$ times

5)

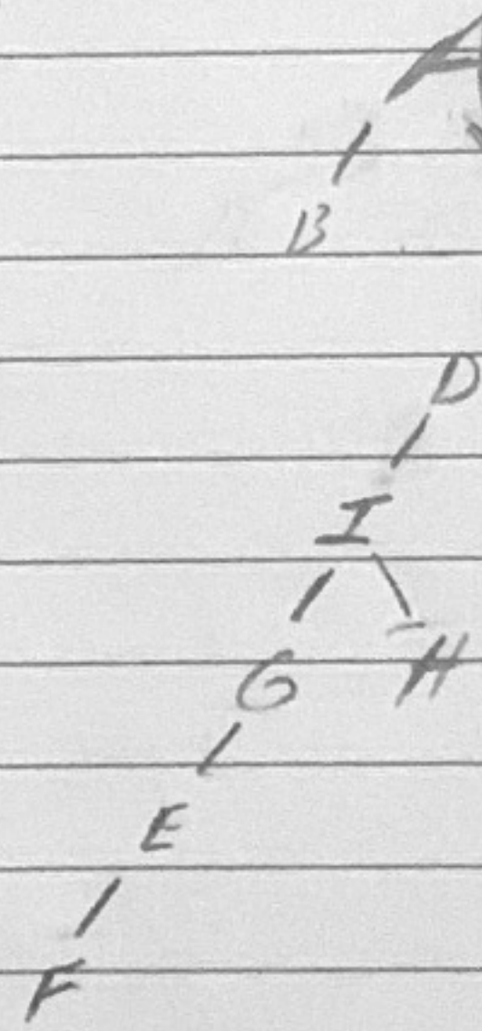


1- A -> C -> D -> I -> H
↓ G -> E -> F

(A, C), (C, D), (A, B), (D, I), (I, H), (I, G), (G, E), (E, F)

2-

(C, D)	22	30
(C, A)	9	34
(E, F)	12	18
(I, H)	4	34
(I, G)	35	23
(A, B)	42	25
(E, G)	65	19
(D, E)	33	21



6) Strongly Connected Components

Rules: Players must finish all the guests
N guest, player can arbitrarily pick one N guests.
Finish one → invited another.

1) a- Proof by contradiction.
The definition of strongly connected says that there is a path from vertex 1 to 2. If we say that that guest A is lucky then is connected to every other in G. If we say that guest B is lucky then it is also connected to every other component in G. If A and B are lucky then there must exist a path between A and B then there must be a path from B to A. This contradicts the assumption that A and B are not strongly connected. Therefore all lucky guests are in the same connected component.

b- Every other guest that guest A and guest B is in this strongly connected component, there is a path from one guest to another. $g \in G\{1,\dots,n\} \rightarrow g \in G\{1,\dots,n\}$

Hence every guest is a strongly connected component.

6)
2. Also findAllVertex(G):
#Step1: Run SCC finding algorithm to obtain all strongly connected components of G.

#Step2: Run topological sort
topological_order = TopologicalSort(G')

#Step3: C be SCC vertex in G in the list topological sort
C = topological_order[0]

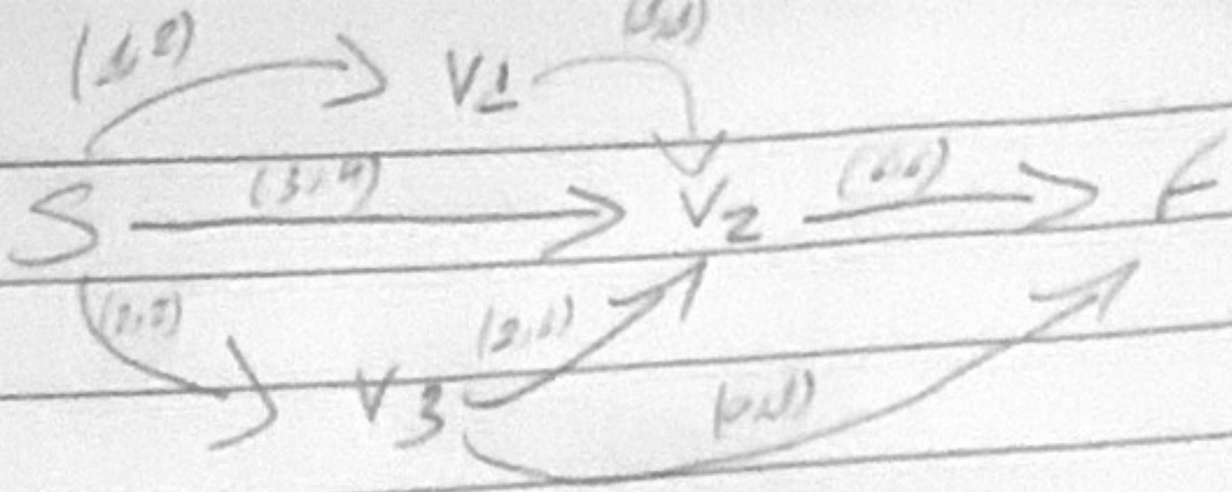
#Step4: Return any vertex v in C
return any vertex in C

7) Also fallowTurkey(n, [][] sleepIntervals){

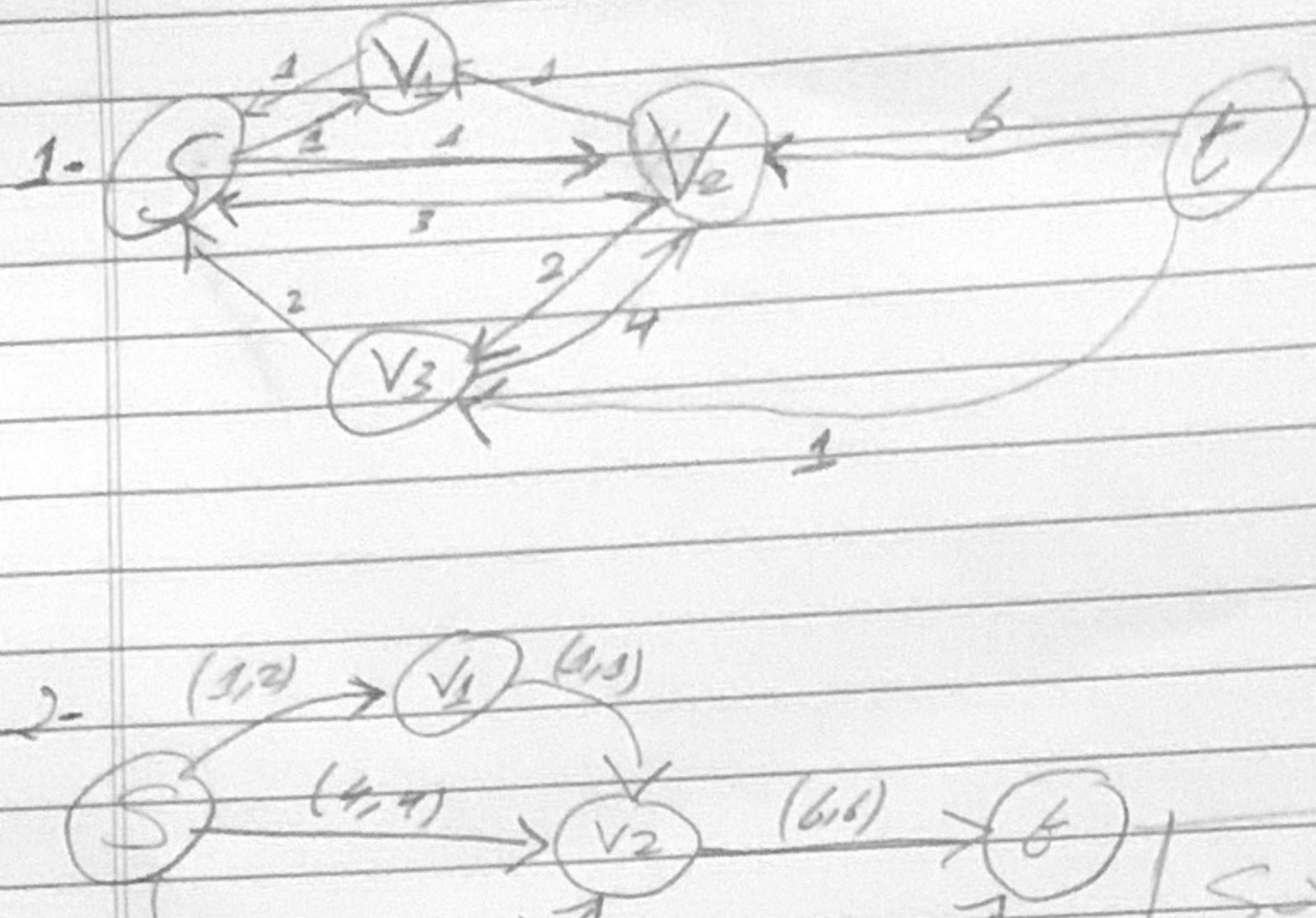
amount_of_turkey = 0
currentFinishTime = 0
Sort_intervals = SleepIntervals.sort()
for(ted time at n){
if(currentFinishTime < Sort_intervals[time][0]){
amount_of_turkey += 1
currentFinishTime = Sort_intervals[time][1]
}
}

return amount_of_turkey

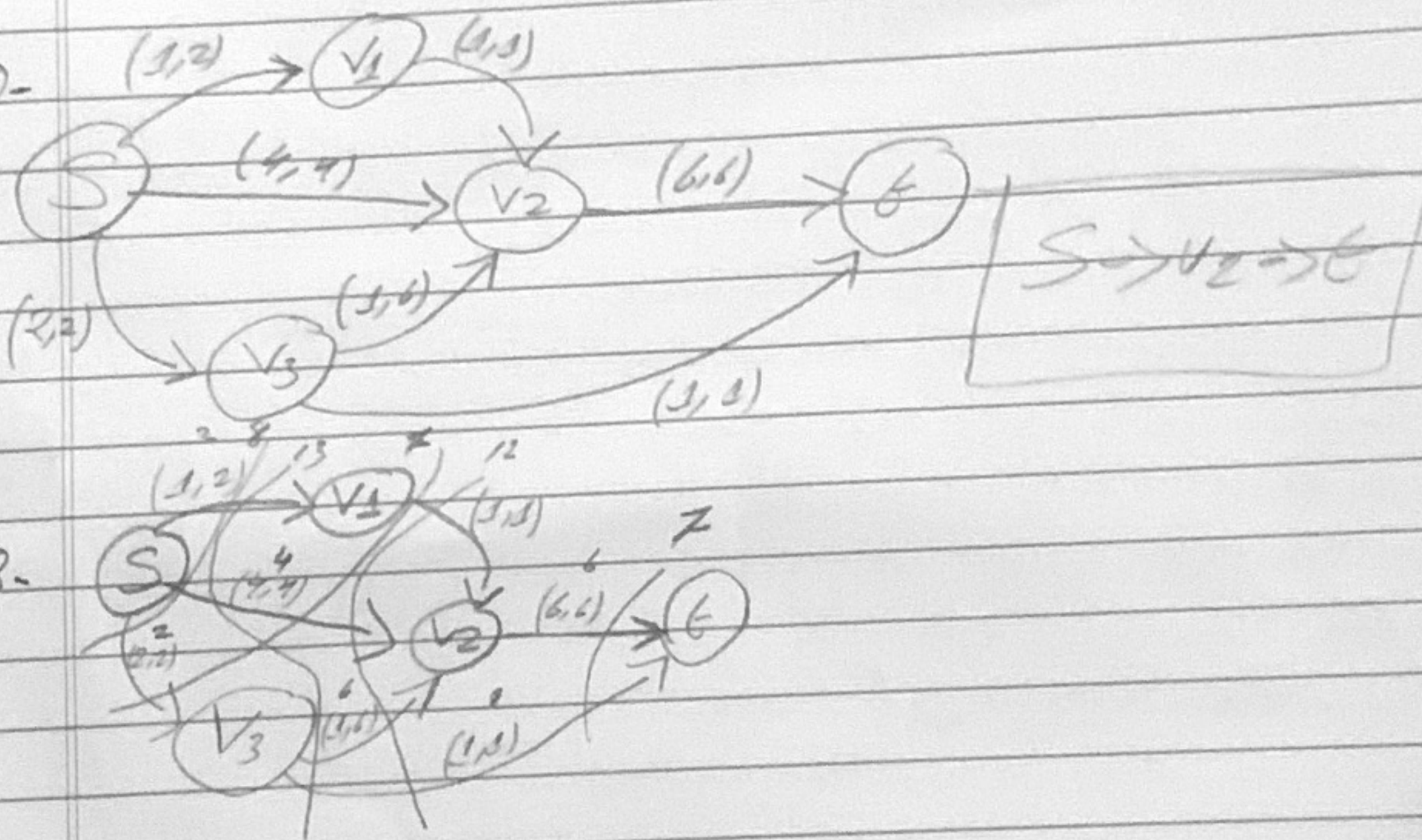
(Q5)



1-



2-



3-

using min cut, max flow theorem the min cut is Z.

9)

1-
```
function minCoins(coins, n){
    if n==0
        return 0
    minCoins = Infinity

    for ( coin of coins){
        if (coin <= n){
            subCoins = minCoins(coins, n - coins[i])
            if ( subCoins != -1) {
                minCoins = min(minCoins, subCoins + 1)
            }
        }
    }
```

return minCoins if not == Infinity else return -1;

This follows the DP principle of breaking down problems in smaller subproblems and combining solutions to solve the original problem. The function checks all k possible first coins d_i to start making changes for n cents. For each d_i, the number of coins needed to make changes for remaining $n - d_i$ cents.

The base case is 0

The optimal solution is the returned for each possible first coin d_i

9)

2- function minCoins (k, n){

Coins = new Array(n).fill(0)

first = coin[0]

for(let amount of n){
Coin[amount] = Infinity
for(let i of k){
if i <= amount
Coins[amount] = min(coins[amount], coins[amount - i]+1)
}
return coins[n]

$O(nk)$

The outer loop goes to from 1 to n and inner loop goes to 1 to k.

3- function minCoins (k, n){
coins = new Array(n).fill(0)
method = new Array(n).fill(0)

first = Coins[0]
for(let amount of n){
Coins[amount] = infinity
for(let i in k){
if(i <= amount){
if(coins[amount - i] + 1 < Coins[amount]){
coins[amount] = coins[amount - i] + 1
method[amount] = i
}

coins_used = []
amount = n

while amount > 0:
coin = method[amount]
coins_used.push(coin)
amount -= coin

return coin_used

$O(nk)$

The algorithm doesn't affect, it only involves additional array accesses and backtracking which is O(n) time.