Problem1:
Part a

X/Y		A	L	G	O	R	I	T	H	M
	0	0	0	0	0	0	0	0	0	0
A	0	1	1	1	1	1	1	1	1	1
N	0	1	1	1	1	1	1	1	1	1
A	0	1	1	1	1	1	1	1	1	1
R	0	1	1	1	1	2	2	2	2	2
C	0	1	1	1	1	2	2	2	2	2
H	0	1	1	1	1	2	2	2	3	3
I	0	1	1	1	1	2	3	3	3	3
S	0	1	1	1	1	2	3	3	3	3
M	0	1	1	1	1	2	3	3	3	4

Sub-problem:

- Let's consider two strings represented by X and Y with length n and m respectively.
- Our goal is to find the longest common subsequence of strings X and Y.
- Let us just start with computing LCS(i,j) for string X[1...i] and Y[1...j]. To compute LCS(i,j) we need to have solved smaller subproblems.
- When X[i]!=Y[j] we either have a subsequence that is ending at X[i] or Y[i]. We start with not considering Y[i] which means LCS[i,j] ends in X[i]. So to compute LCS[i,j] we will need answer to subproblem LCS[i,j-1].
- Now not considering X[i] which means LCS[i,j] ends in Y[i]. So to compute LCS[i,j] we will need answer to subproblem LCS[i-1,j].
- When X[i] == Y[i] means LCS[i,j] end at x[i] and y[i]. LCS[i,j] = LCS[i-1,j-1] + 1. This means get the LCS for strings X and Y of length i-1 and j-1 and add current character to that subsequence.
- To solve each sub problem of form LCS[i,j] we form a 2-d matrix.
- The smallest sub problem in our case is both the strings have length 0 LCS[0,0] = 0, string X of length i and Y length 0 L[i,0] = 0 and similarly L[0,j] = 0. These form our base cases. Representing first row and column in our table.

Recurrence:

From our subproblems we can build the following recurrence relationship

LCS[i,j] = L[i-1,j-1] + 1, if X[i] = Y[j]

LCS[i,j] = max(LCS[i-1,j],LCS[i,j-1]), if X[i]!=Y[j]

Correctness:

- Consider our example string "ALGORITHM" and "ANARCHISM". We start with initializing LCS[i,0]=0 and LCS[0,j]=0 and LCS[0,0]=0.
- Let us consider i = 1 and j =1. We've X[1]==Y[1] and as per our recurrence relation we have LCS[1,1] = LCS[0,0] +1 = 0 + 1 = 1.

- Let us now consider case for i = 1 and j=2. We have X[1]!=Y[2] and as per our recurrence relation LCS[1,2] = max(LCS[0,2],LCS[1,1]) = max(0,1)=1 which means LCS of strings "A" and "AL" is "A" of length 1.
- So at any LCS[i,j] we would have already computed sub problems required to solve LCS[i,j] which are LCS[i-1,j-1],LCS[i-1,j] and LCS[i,j-1].
- Hence the recurrence relation guarantees that the end of all iterations we will have the longest common subsequence of X and Y in LCS[n,m].

Algorithm:
Input - strings X and Y of length n and m.
Output - longest common subsequence between X and Y

Algorithm:
```
Consider a 2-D array L[n+1][m+1]
for i in range(0,n):
   L[i][0] = 0
for j in range(0,m):
   L[0][j] = 0

for i in range(1,n):
   for j in range(1,m):
         If X[i] == Y[j]:
                  LCS[i][j] = LCS[i-1][j-1] + 1
         else:
                  LCS[i][j] = max(LCS[i-1][j], LCS[i][j-1])
return LCS[n][m]
```

Running time:
We compare each character in X with each character in Y. Outer loop runs for n times and for each iteration of the outer loop we run inner loop for m times. Hence running time is upper bounded by O(nm).

Space complexity:
We require O(n) and O(m) to store X and Y respectively. We then create a 2-D matrix of LCS to hold the solutions for subproblems which is of size (n+1)*(m+1). Ignoring the lower order terms we've space complexity which is upper bounded by O(nm)

Part b

- From our algorithm in **part a** we have length of longest common subsequence stored in LCS[n][m].
- To return the actual LCS we need to backtrack from LCS[n][m] and need to check how we arrived at that number.
- There are three cases here we could have arrived at L[n][m] from L[n-1][m-1],L[n-1][m] or L[n][m-1]. We then move to that position till we have our LCS string
- I = n and j = m (length of X and Y respectively)

```
LCS_string = “”
While (i>0 and j>0):
        If X[i] == Y[j]:
                i--
                j--
                LCS_string = X[i]+LCS_string
        If LCS[i-1,j] > LCS[i,j-1]:
                i--
        else:
                j--
return LCS_string
```

Problem 2:

Part a.

Input: graph G, vertex s, vertex t, edge_capacity_func c, vertex_capacity_func w.

Output: maximum flow from s-t.

maxflow(graph G, vertex s, vertex t, edge_capacity_func c):

- Start with flow = 0
- Build a residual graph G' such that we have edge set that contains both the edges (u,v) and (v,u) in the graph. c' (u,v) = c(u,v) - f(u,v) and c'(v,u) = f(u,v). c' (u,v) gives us the value of how much flow we can increase over the edge(u,v) before we hit the capacity and c' (v,u) gives us the value of how much flow we can decrease.
- Find the shortest path from s-t. If no such path exists we return flow f.
- if shortest path to s-t exists we find the smallest c' (u,v) on that path. Let us call this z = min(c' (e)) $\forall$ e ε E
- Augment the flow by adding z to each edge e. If we have (u,v) in the path then we increase flow through that edge by z and if we have no such edge then we must have a backward flow (v,u) that we must decrease. f(u,v) = f(u,v) + z or f(v,u) = f(v,u) - z
- repeat

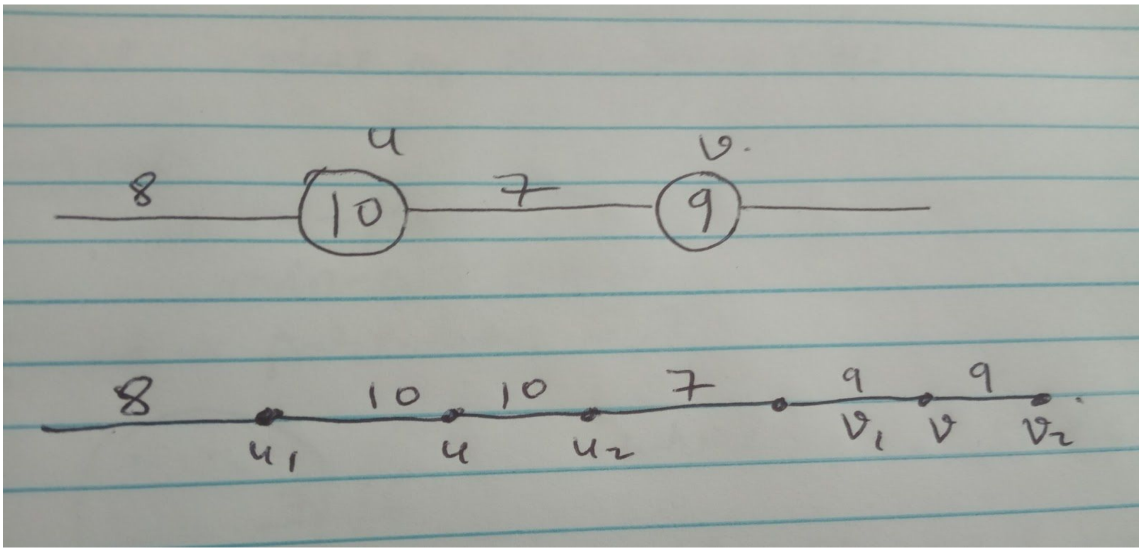
vertCapAlg(graph G, vertex s, vertex t, edge_capacity_func c, vertex_capacity_func w):

If function w exists:

Make changes to graph G(**answer to part B**)

return **maxflow(graph G, vertex s, vertex t, edge_capacity_func c)**

Part b:



We implement the code inside if condition from part a when vertex_capacity_function exists.

vertCapAlg(graph G, vertex s, vertex t, edge_capacity_func c, vertex_capacity_func w):

If vert_capacity_function w exists:

- ∀ u ε V such that w(u) has some value we split the vertex to form new edges.
- Add dummy vertices such that the net capacity remains the same i.e c(u1,u) = c(u,u2) = 10.
- E = E U (u1,u) and E = E U (u,u2)
- c = c U c(u1,u) and c = c U c(u,u2)
- We add new edges to our set of edges from the original graph G and capacities of these edges to our capacity function
- Repeat till all the vertices with weight capacity are converted to edge weight capacity representation

return **maxflow(graph G, vertex s, vertex t, edge_capacity_func c)**

- We make modification to vertCapAlg such that when we have a vertex_weight_function we add dummy vertices and edges and change the graph in such a way that we just have edge capacity function.
- In the figure above u and v have vertex capacities.
- We create vertices u1,u2,v1 and v2 such that the net capacity doesn't change.
- c(u1,u)=c(u,u2)=10 and c(v1,v)=c(v,v2)=9.
- We can then use maxflow function to get the maximum flow using just the edge weight capacity.

Correctness:

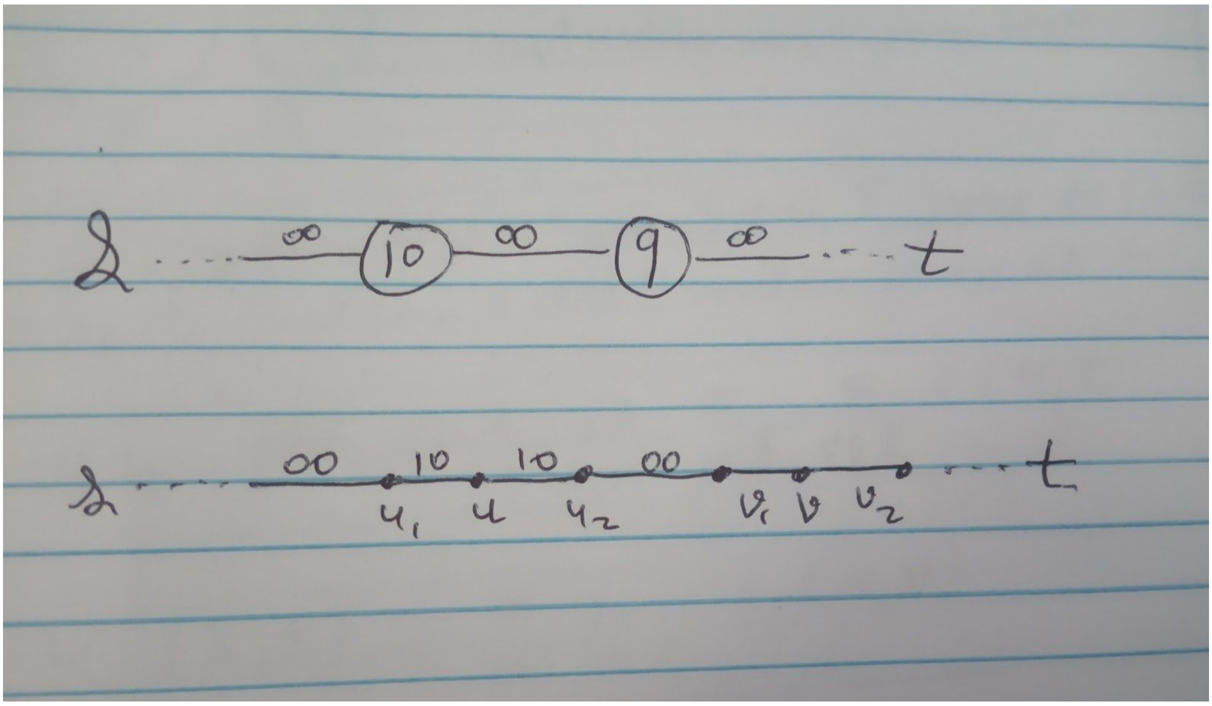
- We need to prove the correctness of function maxflow because irrespective of presence of vertex capacities we convert it to edge capacities and call maxflow function.
- At any given time in the algorithm if we have not yet returned then there is a shortest path from s-t and a smallest value along that path z such that adding f(u,v)=f(u,v) + z and f(u,v)<=c(u,v) and c'(u,v) = c(u,v) - f(u,v). If (u,v) not in E then we decrease f(v,u) by z

- We continue this till we are able to find shortest s-t path and we could increase flow on s-t path by z and adding this flow f(u,v)<=c(u,v)
- We argue that the algorithm terminates because if we are not able to find the shortest path then there must be a cut such that uεS and vεT such that f(u,v) - f(v,u) = c(u,v). If this wasn't the case our greedy algorithm would have continued and found a shortest s-t path adding a small flow z to reach to maximum flow.
- F(S,T) = K(S,T) which means we have have flow that equals cut and thus is the maximum flow.

Running time:

- For running time we consider Edmond Karp's algorithm.
- For every edge (u,v) in E it can be critical at most |V|/2. Which means we would have to consider this edge having smallest cut c'(u,v) at most |V|/2.
- We use BFS to find shortest path which takes O(|V| + |E|)
- For each edge (u,v) in original graph we added (v,u) while constructing residual graph so we have 2E and each can be critical |V|/2 times. We can find this edge per iteration using BFS. considering E>=|V|-1
- We can say that running time is O(|V| * |E^2|)

Part c:



- Consider we've a graph G as shown above. As edge can support any amount of flow we initialize all edges c(e) = infinity
- We then call VertCapAlg on graph G,vertex s and t and edge capacity c and vertex capacity c.
- In vertCapAlg as vertex capacity function exists and there are vertex capacities,for each such vertex u we create two dummy vertices and add edges (u1,u) and (u,u2) such that the net capacity remains the same.
- We then call maxflow function on graph G,vertices s and t and edge capacity c.
- Even though edges can support any amount of flow but we end up picking the minimum flow z = min(c'(u,v)) such that when we augment it by z we satisfy the flow constraint that f(u,v)<=c(u,v)
- We argued about correctness of maxflow in **partb** and that it terminates. So when we exit maxflow function we've max-flow from s-t.
- So irrespective of some edge capacities being infinity we have to consider vertex capacities which are less than infinity and are the ones considered to get the maximum flow.

Part d:

Algorithm:

- We're given vertex weight function so we call **VertCapAlg** which converters vertex weight capacities to edge weight capacities.
- **VertCapAlg** calls **maxflow** which returns the max flow f in the graph G from vertex s to t given edge weight capacity.
- We then compute residual graph G'
- To Compute min-cut we need to find F[S,T] = K[S,T] such that set S has all vertices reachable from vertex s.
- Initially we just start with S = {s}.
- For any vertex v we can reach from S in graph G' such that f[u,v] - f[v,u] < c[u,v] we add vertex v to S. so F[S,T]<K[S,T] which is not min cut we repeat the procedure till we have minimum cut i.e F[S,T] = K[S,T].
- At end of min-cut algorithm we will two set of vertices S and T such that f[u,v]-f[v,u] = c[u,v] for any u ε S and v ε V

Problem 3:
Part a. Following is the code to greedily find minimum number of coins:

```
def greedy_coins_weight(coins, w):
    coins.sort()
    coins = coins[::-1]
    print(coins)
    length = len(coins)
    i = 0
    count = 0
    d = {}
    while w:
        if w>=coins[i]:
            w= w - coins[i]
            d[coins[i]]=d.get(coins[i],0)+1
            count+=1
        else:
            i+=1
    return d, count
```

denominations,count = greedy_coins_weight([1,5,10,25,50,100],42)
Denominations = {25: 1, 10: 1, 5: 1, 1: 2} count = 5

denominations,count = greedy_coins_weight([1,5,10,25,50,100],1728)
Denominations = {100: 17, 25: 1, 1: 3} count = 21

denominations,count = greedy_coins_weight([1, 8, 20, 30, 80, 200],42)
Denominations = {30: 1, 8: 1, 1: 4} count = 6

denominations,count = greedy_coins_weight([1, 8, 20, 30, 80, 200],1728)
Denominations = {200: 8, 80: 1, 30: 1, 8: 2, 1: 2} count = 14

Part b.
consider coins = [1, 8, 20, 30, 80, 200] and w = 42.
If we run our greedy algorithm we choose 1 coin of value 30, 1 coin of value 8 and 4 coins of value 1. Total number of coins of is 6.

But if we select 2 coins of value 20 and 2 coins of value 1 we sum to weight 42 in just 4 coins.

Thus greedy approach is not always optimal and we may end up giving more number of coins even if the total sum is the same.

Part c.

Input: coin denominations and weight w.

Output: minimum number of coins required to sum to w.

Algorithm:

Let coins be the array containing coin denominations and w be the sum.

```
Count[w+1] = infinity
Count[0] = 0
For i from 1 to w+1:
   For j from 0 to length(coins):
         if(i <= coins[j]):
             Sub_prob = count[i - coins[j]]
             If sub_prob != infinity and sub_prob + 1 < count[i]:
                   Count[ i ] = sub_prob + 1
Return count[w]
```

Correctness:

- Algorithm starts with initializing the number of coins to reach a particular weight as infinity. Count to reach weight 0 as 0 which is base case.
- Consider we have i = 1 and we have coins[j] = 1 such that coins[j] <=i.Count[i] = infinity initially. In our loop we check weight i<= coins[j] and then get the minimum number of coins we needed to get to count[i - coins[j]] which in this case is count[0] = 0.
- So when we add 1 to our sub_prob we end up with number of coins required to reach i=1 as 1, and count[1] = infinity initially we update count[1] = 1.
- Similarly as we move ahead we check if adding given denomination to our count is optimal or not thereby minimizing the number of coins used.
- Finally we have minimum number of coins to reach weight w in count[w].

Hence proved the correctness.

Running time:

We iterate over the outer for loop for w+1 times and for each weight we go through all denominations of coins. Considering we have n denominations our worst case runtime would be O(wn)

Space:

We create array count to store the value of sub problems which has size of w+1. Hence space complexity is O(w).

Part d.

```
import sys
def dynamic_coin_weight(coins,w):
    count = [0]*(w+1)
    for i in range(1,w+1):
        count[i]=sys.maxsize
    count[0] = 0
    for i in range(1,w+1):
        for j in range(len(coins)):
            if coins[j]<=i:
                sub_prob = count[i-coins[j]]
                if sub_prob!=sys.maxsize and count[i]>sub_prob + 1:
                    count[i] = sub_prob + 1

    return count[w]
print(dynamic_coin_weight([1,5,10,25,50,100],42))
print(dynamic_coin_weight([1,5,10,25,50,100],1728))
print(dynamic_coin_weight([1, 8, 20, 30, 80, 200],42))
print(dynamic_coin_weight([1, 8, 20, 30, 80, 200],1728))
```

Output:

5

21

4

12

References: Professor Bo notes and wikipedia.