Ex no:6 **GREEDY ALGORITHMS**

Date: 28.9.22

1. **Implementation of Krushkal's Algorithm**

Algorithm:

- Kruskal's algorithm also finds the minimum cost spanning tree of a graph by adding edges one-by-one.

```
enqueue edges of G in a queue in increasing order of cost.
T = ϕ ;
while(queue is not empty){
  dequeue an edge e;
  if(e does not create a cycle with edges in T)
     add e to T;
}
return T;
```

Sample Input

```
0 1 10
1 3 15
2 4 3
2 0 6
0 3 5
```

Sample Output

```
2 to 3
0 to 3
0 to 1
19
```

We use the above given algorithm**:**

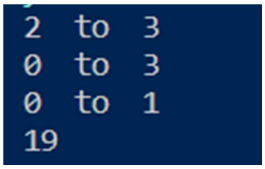
```
arr=[]
t=[int(i) for i in input().split()]
n=t[0]
que=t[1]
ind={}
graph=[[i] for i in range(n)]
for _ in range(que):
   x=[int(i) for i in input().split()]
   graph[x[0]-1].append(x[1]-1)
   if x[1]-1 in ind:
      ind[x[1]-1]+=1
   if x[1]-1 not in ind:
      ind[x[1]-1]=1
for i in range(1,n+1):
   if i not in ind:
      ind[i]=0
   # if i not in oud:
   #   oud[i]=0

def check(arr):
   l=0
```

Kaushik S
20I224

```
    c=0
    for i in arr:
        if arr[i]<0:
            c+=1
        l+=1
    return c!=l
order=[]
while check(ind):
    c=0
    l=0
    for i in range(1,n+1)[::-1]:
        if ind[i]==0:
            order.append(i)
            ind[i]-=1
            l+=1
        c+=1
    if c==l:
        break
    if l==0:
        for i in range(1,n+1):
            if ind[i]>0:
                # print(graph[i-1])
                for x in graph[i-1][1:]:
                    ind[x]-=1
                    # order.append(x)
if len(order)!=n:
    print('NOT POSSIBLE')
else:
    print(order[::-1])
```

Output:



Time Complexity Analysis:

Methods	**Time Complexity**	**Space complexity**
Kruskal alogrithm	O(ElogE) *V is the no of vertices	O(V + E), *V is the no of vertices *E is the no of edges *To store output

Result: Successfully completed the problem using Kruskal algorithm.

2. Implementation of Prims Algorithm:

Algorithm:

1. Prim's algorithm finds a minimum cost spanning tree by selecting edges from the graph one-by-one as follows:
2. It starts with a tree, T, consisting of the starting vertex, x.
3. Then, it adds the shortest edge emanating from x that connects T to the rest of the graph.
4. It then moves to the added vertex and repeats the process.

Consider a graph G=(V, E);
Let T be a tree consisting of only the starting vertex **x;**
while (T has fewer than IVI vertices)
{
find a smallest edge connecting T to G-T;
add it to T;
}

Input:

First line contains the number of vertices(n) and number of edges in the graph. next line contains two integers which denote the location j and threshold th.

Following m lines contains two integers v1 and v2 separated by space denoting the fact that vertices v1 and v2 are connected directly. Vertices are numbered from 1 to n.

Expected Output: Print the list of locations for which the WIFI is accessible (ordered by distance and location number)

Sample Input:
7
8
0 1 2
0 3 3
0 6 4
1 2 3
1 4 2
3 4 5
6 4 6
4 5 7
Sample Output:
0, 1, 4, 3, 2, 6, 5

```
e=int(input())
graph=[[0 for i in range(e+1)] for i in range(e)]
# for i in graph:
   # print(*i)

x=[]
for _ in range(int(input())):
   t=[int(i) for i in input().split()]
   graph[t[0]][t[1]]=t[2]
```

Kaushik S
20I224

```
    x.append((min(t[0],t[1]),max(t[0],t[1]),t[2]))
    graph[t[1]][t[0]]=t[2]
# print(x)

'''
7
8
0 1 2
0 3 3
0 6 4
1 2 3
1 4 2
3 4 5
6 4 6
4 5 7

'''

visited=[0]
start_index=[0]
done={}
l=0
while len(visited)+1<=e:
    # print(done)
    t=[]
    for i in x:
        if i[0] in start_index and i not in done:
                t.append(i)
    # t.sort(key=lambda x:x[2],reverse=True)
    t=sorted(t,key=lambda y:y[2])
    # print(t,visited,start_index,l)
    for i in t[::-1]:
        if i[1] in visited:
            pass
        if i[1] not in visited:
            xx=i
    # xx=t[0]
    # if xx[1] in visited:
        # continue
    start_index.append(xx[1])
    visited.append(xx[1])
    done[xx]=1
    l+=xx[2]
print(visited)
# print(start_index)
# print(l,done)

'''
9
15
0 1 4
0 7 8
1 7 11
```

1 2 8
7 8 7
7 6 1
6 5 2
6 8 6
2 8 2
2 5 4
2 3 7
2 5 4
3 5 14
3 4 9
5 4 10
'''

Output:

```
[0, 1, 4, 3, 2, 6, 5]
```

Time Complexity Analysis:

Methods	Time Complexity	Space complexity
DFS	O(V^2) *V is the no of Vertices *E is the no of Edges	O(V) *V is the no of Vertices *E is the no of Edges

Result: Successfully completed the problem using prims algorithm.

3. Implementation of Dijkstra's Shortest Path Algorithm

Algorithm:

Algorithm:

1. Assign to every node a distance value. Set it to zero for our initial node and to infinity for all other nodes.
2. Mark all nodes as unvisited. Set initial node as current.
3. For current node, consider all its unvisited neighbors and calculate their tentative distance (from the initial node). For example, if current node (A) has distance of 6, and an edge connecting it with another node (B) is 2, the distance to B through A will be 6+2=8. If this distance is less than the previously recorded distance (infinity in the beginning, zero for the initial node), overwrite the distance.
4. When all neighbors are considered for the current node, mark it as visited. A visited node will not be checked ever again; its distance recorded now is final and minimal.
5. If all nodes have been visited, finish. Otherwise, set the unvisited node with the smallest distance (from the initial node) as the next "current node" and continue from step 3.

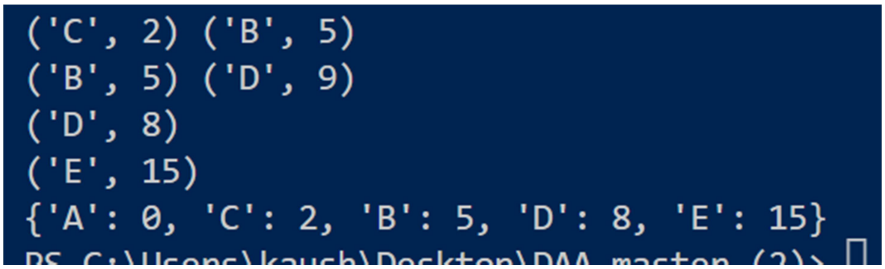
Perform Dijkstra algorithm as given above.

```
    def dijkstra(current, nodes, distances):
    unvisited = {node: None for node in nodes}
    visited = {}
    currentDistance = 0
    unvisited[current] = currentDistance
    while True:
        for neighbour, distance in distances[current].items():
            if neighbour not in unvisited:
                continue
            newDistance = currentDistance + distance
            if unvisited[neighbour] is None or unvisited[neighbour] > newDistance:
                unvisited[neighbour] = newDistance
        visited[current] = currentDistance
        del unvisited[current]
        if not unvisited:
            break
        candidates = [node for node in unvisited.items() if node[1]]
        print(*sorted(candidates, key = lambda x: x[1]))
        current, currentDistance = sorted(candidates, key = lambda x: x[1])[0]
    return visited
nodes = ('A', 'B', 'C', 'D', 'E')
distances = {
    'A': {'B': 5, 'C': 2},
    'B': {'C': 2, 'D': 3},
    'C': {'B': 3, 'D': 7},
    'D': {'E': 7},
    'E': {'D': 9}}
start = 'A'
```

Kaushik S
20I224

print(dijkstra(start, nodes, distances))

Output:



Time Complexity Analysis:

Methods	**Time Complexity**	**Space complexity**
Do Dijkstra using the provided alogirithm	O(V^2) *V is the no of Vertices	O(E) *E is the no of Edges

Result: Successfully completed the problem using the provided algorithm.

4. Implementation of Huffman Coding

Algorithm:

1. Scan text to be compressed and tally occurrence of all characters.
2. Sort or prioritize characters based on number of occurrences in text.
3. Build Huffman code tree based on prioritized list.
4. Perform a traversal of tree to determine all code words.
5. Scan text again and create new file using the Huffman codes.

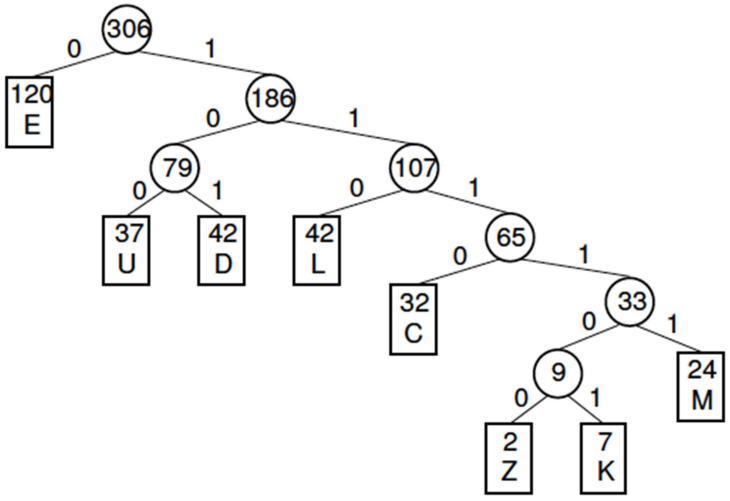
Letter freqency table

Letter	Z	K	M	C	U	D	L	E
Frequency	2	7	24	32	37	42	42	120

Huffman code

Letter	Freq	Code	Bits
E	120	0	1
D	42	101	3
L	42	110	3
U	37	100	3
C	32	1110	4
M	24	11111	5
K	7	111101	6
Z	2	111100	6

The Huffman tree (Shaffer Fig. 5.24)



Three problems:

- Problem 1: Huffman tree building
- Problem 2: Encoding
- Problem 3: Decoding

Kaushik S
20I224

Problem 2: Encoding

Encoding a string can be done by replacing each letter in the string with its binary code (the Huffman code).

Examples:
DEED 10100101 (8 bits)
MUCK 111111001110111101 (18 bits)

Problem 3: Decoding

Decoding an encoded string can be done by looking at the bits in the coded string from left to right until a letter decoded.
10100101 -> DEED

Pseudocode

Huffman(W, n)
Input: A list W of n weights.
Output: An extended binary tree T with weights
taken from W that gives the minimum weighted path length.

Procedure:
Create list F from singleton trees formed from elements of W
WHILE (F has more than one element) DO
Find W in F that have minimum values associated with their roots
Construct new tree T by creating a new node and setting T1 and T2 as its children
Let the sum of the values associated with the roots of T1 and T2 be associated with the root of T
Add T to F
OD
Huffman := tree stored in F
Input:

'z':2,'k':7,'m':24,'c':32,'u':37,'d':42,'l':42,'e':120

Expected Output:.

'z': '111100', 'k': '111101', 'm': '11111', 'c': '1110', 'u': '100', 'd': '101', 'l': '110', 'e': '0'
ded
1010101 7
100
u
We follow the given algorithm to perform and create Huffman code for the letters by grouping two letters and merge them and sort them to find the respective Huffman code:

```
letters={'z':2,'k':7,'m':24,'c':32,'u':37,'d':42,'l':42,'e':120}
r=len(letters)
# [z,0][k,0]
ans={}
while r>=2:
```

```
letters_sorted=list(sorted(letters,key=lambda x:letters[x]))
print(letters_sorted)
if len(letters)==0:
   break
l1,n1=letters_sorted[0],letters[letters_sorted[0]]
l2,n2=letters_sorted[1],letters[letters_sorted[1]]
if n1>n2:
   #n2 min
   for letter in l2:
         if letter in ans:
            ans[letter]='0'+ans[letter]
         if letter not in ans:
            ans[letter]='0'
   for letter in l1:
      if letter in ans:
         ans[letter]='1'+ans[letter]
      if letter not in ans:
         ans[letter]='1'
   letters[l1+l2]=n1+n2
else:
   for letter in l1:
         if letter in ans:
            ans[letter]='0'+ans[letter]
         if letter not in ans:
            ans[letter]='0'
   for letter in l2:
      if letter in ans:
         ans[letter]='1'+ans[letter]
      if letter not in ans:
         ans[letter]='1'

   # if l1 in ans:
   #     ans[l1]='0'+ans[l1]
   # if l1 not in ans:
   #     ans[l1]='0'

   # if l2 in ans:
   #     ans[l2]='1'+ans[l2]
   # if l2 not in ans:
   #     ans[l2]='1'
   letters[l1+l2]=n1+n2
print(l1,n1)
print(l2,n2)
del letters[l1]
del letters[l2]
```

Kaushik S
20I224

```
    r-=1
print(ans)
s=input()
a=''
for i in s:
    a+=ans[i]
print(a,len(a))
k=input()
de=''
i=0
while i<len(k):
    # print(i,ddee)
    for l in ans:
        to_con=ans[l]
        # print(to_con)
        if i+len(to_con)<=len(k):
            # print(k[i:i+len(to_con)],k,i,i+len(to_con),de,k[i:])
            if k[i:i+len(to_con)]==ans[l]:
                de+=l
                # print(de)
                i=i+len(to_con)
print(de)
```

Output:

```
{'z': '111100', 'k': '111101', 'm': '11111', 'c': '1110
', 'u': '100', 'd': '101', 'l': '110', 'e': '0'}
ded
1010101 7
100
u
```

Time Complexity Analysis:

Method	Time Complexity	Space complexity
Check DFS from all vertices	O(nlog n) *n is the no of unique characters	O(n) *n is the size the text

Result:

Successfully printed the output for the problem using DFS from all the vertices.

5. Implementation of Activity selection problem

Problem: ***Given n activities with their start and finish times. Select the maximum number of activities that can be performed by a single person, assuming that a person can only work on a single activity at a time.***

Algorithm

1. Sort the activities according to their finishing time
2. Select the first activity from the sorted array and print it.
3. Do following for remaining activities in the sorted array.

If the start time of this activity is greater than or equal to the finish time of previously selected activity then select this activity and print it.

Example 1 : Consider the following 3 activities sorted by by finish time.
start[] = {10, 12, 20};
finish[] = {20, 25, 30};
A person can perform at most **two** activities. The maximum set of activities that can be executed is {0, 2} [These are indexes in start[] and finish[]]

Example 2 : Consider the following 6 activities sorted by by finish time.
start[] = {1, 3, 0, 5, 8, 5};
finish[] = {2, 4, 6, 7, 9, 9};
A person can perform at most **four** activities. The maximum set of activities that can be executed is {0, 1, 3, 4} [These are indexes in start[] and finish[]]

Sort it according to the finish time of the jobs now we Select the start job and check if the finishing time of curr job is less than the starting time of the next jobs and add profit and assign the new job as curr job and continue

```
arr = [[60, 10], [100, 20], [120, 30]]
weight = 50
for i in range(len(arr)):
        arr[i].append(arr[i][0]/arr[i][1])
    arr.sort(key = lambda x: x[2] ,reverse = True)
    total_profit = 0
    for i in range(len(arr)):
      if weight >= arr[i][1]:
        weight -= arr[i][1]
        total_profit += arr[i][0]
      else:
        total_profit += (arr[i][2] * weight)
        weight = 0
      if weight == 0:
        break
print(total_profit)
```

Kaushik S
20I224

Output:

```
-------
240.0
```

Time Complexity Analysis:

Methods	Time Complexity	Space complexity
Sort it according to the finish time of the jobs now we Select the start job and check if the finishing time of curr job is less than the starting time of the next jobs and add profit and assign the new job as curr job and continue	O(nlogn)	O(1)

Result:

Successfully printed the output for the problem using greedy algorithm.

6. Implementation of Job Sequencing Problem

Problem: Given an array of jobs where every job has a deadline and associated profit if the job is completed before the deadline. It is also given that every job takes single unit of time, so the minimum possible deadline for any job is 1. How to maximize total profit if only one job can be scheduled at a time.

Algorithm

1) Sort all jobs in decreasing order of profit.

2) Initialize the result sequence as first job in sorted jobs.

3) Do following for remaining n-1 jobs

If the current job can fit in the current result sequence without missing the deadline, add current job to the result.

Else ignore the current job.

Example:

Input: Five Jobs with following deadlines and profits

JobID	Deadline	Profit
a	2	100
b	1	19
c	2	27
d	1	25
e	3	15

Output: Following is maximum profit sequence of jobs

c, a, e

Greedily choose the jobs with maximum profit first, by sorting the jobs in decreasing order of their profit. This would help to maximize the total profit as choosing the job with maximum profit for every time slot will eventually maximize the total profit obtained:

Output:

```
5
a 2 100
b 1 19
c 2 27
d 1 25
e 3 15
['a', 2, 100] ['c', 2, 27] ['d', 1, 25] ['b', 1, 19] ['e', 3, 15]
c a e -1 -1
```

Time Complexity Analysis:

Methods	**Time Complexity**	**Space complexity**
Greedily choose the jobs with maximum profit first, by sorting the jobs in decreasing order of their profit. This would help to maximize the total profit as choosing the job with maximum profit for every time slot will eventually maximize the total profit	O(N^2) *N is the no of jobs	O(N) *N is the no of jobs

Result:

Successfully printed the output for the problem using the algorithm.

7. Implementation of Greedy Algorithm to find Minimum number of Coins

Problem: Given a value V, if we want to make change for V Rs, and we have infinite supply of each of the denominations in Indian currency, i.e., we have infinite supply of { 1, 2, 5, 10, 20, 50, 100, 500, 1000} valued coins/notes, what is the minimum number of coins and/or notes needed to make the change?

Examples:

Input: V = 70

Output: 2

We need a 50 Rs note and a 20 Rs note.

Input: V = 121

Output: 3

We need a 100 Rs note, a 20 Rs note and a

1 Rs coin.

Start from largest possible denomination and keep adding denominations while remaining value is greater than 0.

Algorithm:

1) Initialize result as empty.
2) find the largest denomination that is
   smaller than V.
3) Add found denomination to result. Subtract
   value of found denomination from V.
4) If V becomes 0, then print result.

Else repeat steps 2 and 3 for new value of V

We follow the given algorithm:

Output:

```
123
123
4
```

Time Complexity Analysis:

Methods	Time Complexity	Space complexity
Following the given algorithm we get the output	O(V) *V is the no of Coins	O(V) *V is the no of Coins

Result:

Successfully printed the output for the problem using greedy algorithm.

8. **Implementation of Minimum Number of Platforms Required for a Railway/Bus Station**

Problem: Given arrival and departure times of all trains that reach a railway station, find the minimum number of platforms required for the railway station so that no train waits.

arr[] = {9:00, 9:40, 9:50, 11:00, 15:00, 18:00}

dep[] = {9:10, 12:00, 11:20, 11:30, 19:00, 20:00}

All events sorted by time.

Total platforms at any time can be obtained by subtracting total departures from total arrivals by that time.

Time	Event Type	Total Platforms Needed at this Time
9:00	Arrival	1
9:10	Departure	0
9:40	Arrival	1
9:50	Arrival	2
11:00	Arrival	3
11:20	Departure	2
11:30	Departure	1
12:00	Departure	0
15:00	Arrival	1
18:00	Arrival	2
19:00	Departure	1
20:00	Departure	0

Minimum Platforms needed on railway station = Maximum platforms needed at any time = 3

Time Complexity: O(nLogn), assuming that a O(nLogn) sorting algorithm for sorting arr[] and dep[].

Sort all the times and if arrival add one to count and departure remove one and make a global max count:

```
arr= [9.00,  9.40, 9.50,  11.00, 15.00, 18.00]
dep= [9.10, 12.00, 11.20, 11.30, 19.00, 20.00]
x=[]
for i in arr:
   x.append((i,'a'))
for i in dep:
   x.append((i,'d'))
ans_s=sorted(x,key=lambda x:x[0])
print(ans_s)
curr=0
glob_max=-9999
for i in ans_s:
   if i[1]=='d':
```

```
        curr-=1
    if i[1]=='a':
        curr+=1
    glob_max=max(glob_max,curr)
print(glob_max)
```

Output:

```
[(9.0, 'a'), (9.1, 'd'), (9.4, 'a'), (9.5, 'a'), (11.0, 'a'), (11.
2, 'd'), (11.3, 'd'), (12.0, 'd'), (15.0, 'a'), (18.0, 'a'), (19.0
, 'd'), (20.0, 'd')]
3
```

Time Complexity Analysis:

Methods	**Time Complexity**	**Space complexity**
Sort all the times and if arrival add one to count and departure remove one and make a global max count	O(nLogn)	O(nLogn)

Result:

Successfully printed the output for the problem using basic algorithm.

9. Implementation of Graph coloring

Unfortunately, there is no efficient algorithm available for coloring a graph with minimum number of colors as the problem is a known NP Complete problem. Greedy Algorithm to assign colors doesn't guarantee to use minimum colors, but it guarantees an upper bound on the number of colors. The basic algorithm never uses more than d+1 colors where d is the maximum degree of a vertex in the given graph.

Algorithm:

1. Color first vertex with first color.
2. Do following for remaining V-1 vertices.

a) Consider the currently picked vertex and color it with the lowest numbered color that has not been used on any previously colored vertices adjacent to it. If all previously used colors appear on vertices adjacent to v, assign a new color to it.

Time Complexity: O(V^2 + E) in worst case.

Do dfs and start from the starting vertex now find all the connected and add it into a stack and now if every connected node is not coloured we use and assign the colour to a dictionary now we pop the connected vertex and then start from colour 1 if adjacent colours are not used we use the colour to colour the vertex:

```
edges = [(0, 1), (0, 4), (0, 5), (4, 5), (1, 4), (1, 3), (2, 3), (2, 4)]

   # total number of nodes in the graph (labelled from 0 to 5)
n = 6
uncoloured=[0,1,2,3,4,5]
coloured={}
start_stack=[0]
while True:
   for i in uncoloured:
      if i in uncoloured:
         curr=i
         near=[]
         for i in edges:
            if i[0]==curr:
               start_stack.append(i[1])
               near.append(i[1])
         for i in near:
            m=1
            for i in coloured:
                  m=max(m,coloured[i])
```

```
                coloured[curr]=m+1
                if i not in coloured:
                    coloured[curr]=1
            # print(uncoloured,curr)
            try:
                uncoloured.remove(curr)
                start_stack.remove(curr)
            except:
                pass
            # print()
            break
    if len(uncoloured)==0:
        break
# print(coloured)
d=set()
for i in coloured:
    d.add(coloured[i])
print(len(d))
```

Output:

```
4
```

Time Complexity Analysis:

Methods	**Time Complexity**	**Space complexity**
Do dfs and start from the starting vertex now find all the connected and add it into a stack and now if every connected node is not coloured we use and assign the colour to a dictionary now we pop the connected vertex and then start from colour 1 if adjacent colours are not used we use the colour to colour the vertex	O(V^2 + E) *V is the no of vertices *E is the no of edges	O(V) *to store colour mapping

Result:

Successfully printed the output for the problem using DFS and sequencing

10. Implementation of Fractional Knapsack Problem

Problem: Given weights and values of n items, we need to put these items in a knapsack of capacity W to get the maximum total value in the knapsack.

In the 0-1 Knapsack problem, we are not allowed to break items. We either take the whole item or don't take it.

Input:

Items as (value, weight) pairs

arr[] = {{60, 10}, {100, 20}, {120, 30}}

Knapsack Capacity, W = 50;

Output:

Maximum possible value = 220

by taking items of weight 20 and 30 kg for 0-1 Knapsack

Output :

Maximum possible value = 240
By taking full items of 10 kg, 20 kg and
2/3rd of last item of 30 kg

Calculate the ratio value/weight for each item and sort the item on basis of this ratio. Then take the item with the highest ratio and add them until we can't add the next item as a whole and at the end add the next item as much as we can. Which will always be the optimal solution to this problem. As main time taking step is sorting, whole problem can be solved in O(n log n) only.

Sample Input

```
60 10
100 20
120 30
```

Sample Output:

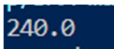
240.0

Sort by using cost by weight and from max to min then get the max amount from reverse and decreasing the weight till it becomes <=0.

```
# n=int(input())
#arr=[]
# for i in range(n):
#     arr.append([int(i) for i in input().split()])
# w=int(input())
arr=[[60,10],[100,20],[120,30]]
w=50
arr.sort(key=lambda x: (x[0]/x[1]), reverse=True)
```

```
value=0
for i in arr:
   if i[1]<=w:
      w-=i[1]
      value+=i[0]
   else:
      value+=i[0]*w/i[1]
      break
print(value)
```

Output:



Time Complexity Analysis:

Methods	Time Complexity	Space complexity
Sort by using cost by weight and from max to min then get the max amount from reverse and decreasing the weight till it becomes <=0.	O(nLogn)	O(1)

Result:

Successfully printed the output for the problem using greedy cost maximizing.

11. Implementation of Minimize Cash Flow among a given set of friends who have borrowed money from each other

Problem: Given a number of friends who have to give or take some amount of money from one another. Design an algorithm by which the total cash flow among all the friends is minimized.

Algorithm

Do following for every person Pi where i is from 0 to n-1 for the below given example.
1) Compute the net amount for every person. The net amount for person 'i' can be computed be subtracting sum of all debts from sum of all credits.

2) Find the two persons that are maximum creditor and maximum debtor. Let the maximum amount to be credited maximum creditor be maxCredit and maximum amount to be debited from maximum debtor be maxDebit. Let the maximum debtor be Pd and maximum creditor be Pc.

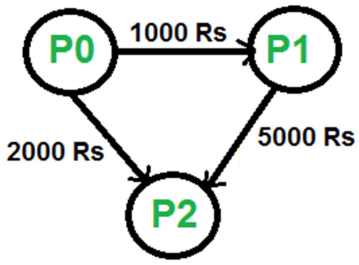
3) Find the minimum of maxDebit and maxCredit. Let minimum of two be x. Debit 'x' from Pd and credit this amount to Pc

4) If x is equal to maxCredit, then remove Pc from set of persons and recur for remaining (n-1) persons.

5) If x is equal to maxDebit, then remove Pd from set of persons and recur for remaining (n-1) persons.

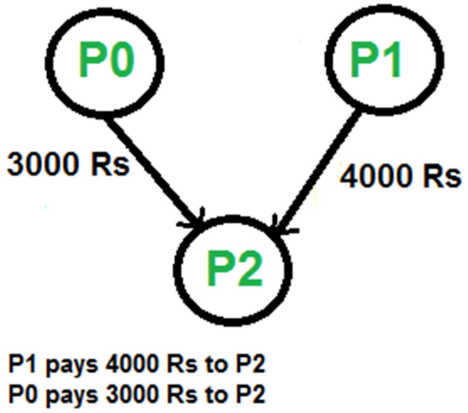
Example:

Following diagram shows input debts to be settled.



P0 has to pay 1000 Rs to P1
P0 also has to pay 2000 Rs to P2
P1 has to pay 5000 Rs to P2.

Above debts can be settled in following optimized way



Time Complexity: $O(N^2)$ where N is the number of persons.

Find code cycle and do mathematical sum and difference in the pathway of the nodes ad cycle and add and subtract accordingly.

```
N = 3
def getMin(arr):
        minInd = 0
        for i in range(1, N):
                if (arr[i] < arr[minInd]):
                        minInd = i
        return minInd
def getMax(arr):
        maxInd = 0
        for i in range(1, N):
                if (arr[i] > arr[maxInd]):
                        maxInd = i
        return maxInd
def minOf2(x, y):
        return x if x < y else y
def minCashFlowRec(amount):
        mxCredit = getMax(amount)
        mxDebit = getMin(amount)
        if (amount[mxCredit] == 0 and amount[mxDebit] == 0):
                return 0
        min = minOf2(-amount[mxDebit], amount[mxCredit])
        amount[mxCredit] -=min
        amount[mxDebit] += min
        print("Person " , mxDebit , " pays " , min, " to " , "Person " , mxCredit)
        minCashFlowRec(amount)
def minCashFlow(graph):
```

```
        amount = [0 for i in range(N)]
        for p in range(N):
                for i in range(N):
                        amount[p] += (graph[i][p] - graph[p][i])
        minCashFlowRec(amount)
graph = [ [0, 1000, 2000],
                [0, 0, 5000],
                [0, 0, 0] ]
minCashFlow(graph)
```

Output:

```
Person 1 pays 4000 to Person 2
Person 0 pays 3000 to Person 2
```

Time Complexity Analysis:

Methods	Time Complexity	Space complexity
Find code cycle and do mathematical sum and difference in the pathway of the nodes ad cycle and add and subtract accordingly.	O(N^2)	O(N)

Result:

Successfully printed the output for the problem using DFS code cycle.

12. Implementation of Find minimum time to finish all jobs with given constraints

Problem: Given an array of jobs with different time requirements. There are K identical assignees available and we are also given how much time an assignee takes to do one unit of job. Find the minimum time to finish all jobs with following constraints.

- An assignee can be assigned only contiguous jobs. For example, an assignee cannot be assigned jobs 1 and 3.
- Two assignees cannot share (or co-assigned) a job, i.e., a job cannot be partially assigned to one assignee and partially to other.

Input :

K: Number of assignees available.

T: Time taken by an assignee to finish one unit of job

job[]: An array that represents time requirements of different jobs.

Example:

Input: k = 2, T = 5, job[] = {4, 5, 10}

Output: 50

The minimum time required to finish all jobs is 50. There are 2 assignees available. We get this time by assigning {4, 5} to first assignee and {10} to second assignee.

Input: k = 4, T = 5, job[] = {10, 7, 8, 12, 6, 8}

Output: 75

We get this time by assigning {10} {7, 8} {12} and {6, 8}

Find the one set with minimum sum and take it first and pop it out of the job array and then assign it to the output to find the max length and multiply it with no of jobs to find the total time.

```
def getMax(arr, n):
   result = arr[0]
   for i in range(1, n):
      if arr[i] > result:
         result = arr[i]
   return result
def isPossible(time, K, job, n):
   cnt = 1
   curr_time = 0
   i = 0
   while i < n:
```

```
        if curr_time + job[i] > time:
            curr_time = 0
            cnt += 1
        else:
            curr_time += job[i]
            i += 1
    return cnt <= K
def findMinTime(K, T, job, n):
    end = 0
    start = 0
    for i in range(n):
        end += job[i]
    ans = end
    job_max = getMax(job, n)
    while start <= end:
        mid = int((start + end) / 2)
        if mid >= job_max and isPossible(mid, K, job, n):
            ans = min(ans, mid)
            end = mid - 1
        else:
            start = mid + 1
    return ans * T
if __name__ == '__main__':
    job = [10, 7, 8, 12, 6, 8]
    n = len(job)
    k = 4
    T = 5
    print(findMinTime(k, T, job, n))
```

Output:

```
75
```

Time Complexity Analysis:

Methods	Time Complexity	Space complexity
Find pair sum of jobs and pop it from jobs array and assign it find max time required.	O(n^2)	O(n)

Result:

Successfully printed the output for the problem using assigning the jobs greedily.