TAP C2: Greedy

1. Ionuț este student integralist in anul 3 la fmi unde se dorește înlocuirea carnetului de note cu o aplicație web. Aplicația este în etapa de testare și încă are bug-uri. Ionuț observă ca notele sale sunt memorate corect, dar că nu sunt asociate materiilor la care au fost obținute. Spre exemplu nota 10 la TAP este înregistrată pentru un examen de anul 1. Este cunoscut *n* numărul examenelor din anul 1, *m* numărul examenelor din anul 2 și *p > (n + m)* numărul examenelor la care Ionuț a obținut note, precum și notele obținute de Ionuț, incluzând examene din anul 3. Fiind lacom în ceea ce privește mediile, Ionuț dorește să asocieze notele anilor de studiu astfel încât suma mediilor din anul 1 și anul 2 să fie maximă.

```
# input p n m
# 12 3 6
# 10 5 8 9 6 7 8 6 4 4 5 9
# output
# 16.0
```

medie anul 1 (în anul 1 notele 10, 9, 9): 28/3 = 9,33
medie anul 2 (în anul 2 notele 8, 8 , 7, 6, 6, 5): 40/6 =6,67

```
line = input().split()
p, n, m = int(line[0]), int(line[1]), int(line[2])
s, v = 0, [None] * p

line = input().split()
for i in range(p):
    v[i] = int (line[i])
v = sorted(v, reverse = True)

n, m = min(n, m), max(n, m)
for i in range(n):
    s += v[i]/n

for i in range(n, n + m):
    s += v[i]/m
print(s)
```

2. Se consideră A, o mulțime de *n* activități. Fiecare activitate *i* are un timp de start s_i si un timp de terminare t_i. Să se selecteze o mulțime de activități compatibile (intervalele de desfășurare disjuncte două câte două) de cardinal maxim.

```
task = [(1,6),(7,11),(7,9),(10,12),(6,8)]
task = sorted(task, key = lambda t:t[1])
print(task[0])
fin, count = task[0][1], 1
```

```
for i in range(1,len(task)):
    if task[i][0] > fin:
        print(task[i])
        fin = task[i][1]
        count += 1
print(count)
```

3. Se cunosc *G*, greutatea unui rucsac și $g = (g_1, \dots, g_n)$, $c = (c_1, \dots, c_n)$ greutățile, respectiv costurile a *n* obiecte. Fiecare obiect poate fi încărcat parțial în rucsac. c_i este costul obținut dacă obiectul *i* este încărcat în întregime în rucsac.
Să se obțină o încărcare optimă: costul obiectelor încărcate să fie maxim, astfel încât greutatea totală încărcată în rucsac să nu depășească G.

```
class Object:
    def __init__(self, id, cost, weight):
        self.id = id
        self.cost = cost
        self.weight = weight
    def __repr__(self):
        return "object " + str(self.id) + ":(" + str(self.cost) +
" " + str(self.weight) + ")"

n, G = 4, 10
object = [None] * n
costs = [10, 9, 7, 5]
weights = [5, 4, 7, 3]

for i in range(n):
    object[i] = Object(i + 1, int(costs[i]), int(weights[i]))

object =sorted(object,key=lambda t:t.cost/t.weight,reverse=True)

g, i, totalcost = 0, 0, 0

while i < n and g < G:
    if object[i].weight + g <= G:
        g += object[i].weight
        totalcost += object[i].cost
        print(object[i])
        i += 1
    else:
        costu = object[i].cost * (G - g)/object[i].weight
        print(Object(object[i].id, costu, G - g))
        totalcost += costu
        g = G

print(totalcost) #20.66
```

4. Se consideră o mulțime A cu n elemente și *p* submulțimi ale lui A. Să se obțină, dacă este posibil o acoperire a lui A care sa utilizeze un număr minim de submulțimi.

```
U = set(['a','b','c','d','e','f','g','h','i','j'])
S = [set(['a','c','e','g']),
        set(['c','e','g','h']),
        set(['b','d','f','g']),
        set(['h','i','a']),
        set(['h','j']),
        set(['a','c','j'])]


covered = set(e for s in S for e in s)
if covered != U:
    print('No solution')

covered = set()
while covered != U:
    subset = max(S, key=lambda s: len(s - covered))
    covered |= subset #union
    print(subset)
```

5. Ionuț dorește să dea o petrecere. Un coleg poate fi invitat doar dacă este invitat grupul lui de prieteni. Ionuț are o listă unde sunt memorate prieteniile (x, y): x poate fi invitat doar dacă va fi invitat și y. y poate fi invitat doar dacă va fi invitat și x. Să se obțină numărul minim de prieteni (>1) pe care îi poate invita Ionuț.

```
line = input().split()
n,m = int(line[0]), int(line[1])
parent, count = [], []

def find(x):
    if (parent[x] == x):
        return x
    else:
        return find(parent[x])

def union(x , y):
    x1 = find(x)
    y1 = find(y)
    if (x1 != y1):
        if count[x1] < count[y1]:
            parent[x1] = y1
            count[y1] += count[x1]
        else:
            parent[y1] = x1
            count[x1] += count[y1]

parent.append(0)
count.append(1)
for i in range(n):
    parent.append(i + 1)
    count.append(1)

for i in range(m):
    line = input().split()
    x,y = int(line[0]), int(line[1])
    union(x , y)

min = n

for i in range(1,n):
    if i == find(i):
        if min > count[i]:
            min = count[i]

print(min)
```