Programarea Algoritmilor
– SEMINAR NR. 7 –
(grupa 131)

- Programare Dinamică -

1. Se dau n perechi (x, y) cu proprietate x < y. Se cere lungimea maximă k a unui lanț de perechi de forma (x_1, y_1),…, (x_i, y_i), (x_{i+1}, y_{i+1}), …, (x_k, y_k) astfel încât $y_i < x_{i+1}$.

   Exemplu: Pentru n = 6 și lp = [(12, 15), (5, 7), (20, 30), (6, 8), (9, 11), (13, 18)], lungimea maximă k a unui lanț cu proprietatea cerută este egală cu 4, iar un posibil astfel de lanț este (5, 7), (9, 11), (12, 15), (20, 30). Atenție, soluția nu este unică!

   Indicație de rezolvare: Se sortează perechile crescător după prima componentă, iar apoi se determină un subșirul crescător maximal al listei de perechi lp.

   Algoritm de complexitate $O(n^2)$ - varianta înainte:

```python
def cmpPerechi(t):
    return t[0]


lp = [(12, 15), (5, 7), (20, 30), (6, 8), (9, 11), (13, 18)]
n = len(lp)
lp.sort(key=cmpPerechi)

lmax = [1] * n
succ = [-1] * n

lmax[n-1] = 1
for i in range(n-2, -1, -1):
    for j in range(i+1, n):
        if lp[i][1] <= lp[j][0] and lmax[i] < 1 + lmax[j]:
            succ[i] = j
            lmax[i] = 1 + lmax[j]

pmax = 0
for i in range(1, n):
    if lmax[i] > lmax[pmax]:
        pmax = i

print("Lungimea maxima a unui subsir crescator: " + str(lmax[pmax]))
print("Un subsir crescator maximal: ")
i = pmax
sol = []
while i != -1:
    sol.append(lp[i])
    i = succ[i]

print(*sol)
```

2. Să se calculeze câte subșiruri strict crescătoare există într-o listă.

   Exemplu: v = [3, 2, 4, 5, 4] => 14 subșiruri strict crescătoare

   Indicație de rezolvare:
   Dacă notăm cu $nsc[i]$ = numărul subșirurilor strict crescătoare care se termină cu $v[i]$, atunci avem

$$nsc[i] = 1 + \sum_{\substack{j<i \\ v[j]<v[i]}} nsc[j]$$

   Numărul total de subșiruri strict crescătoare va fi egal cu suma elementelor din nsc. Pentru exemplu de mai sus vom obține nsc = [1, 1, 3, 6, 3], deci numărul total de subșiruri strict crescătoare este egal cu 14.

   Algoritm de complexitate O(n²), n = |v| - varianta înapoi:

```
v = [3, 2, 4, 5, 4]
n = len(v)

nsc = [1] * n
for i in range(1, n):
    for j in range(0, i):
        if v[j] < v[i]:
            nsc[i] = nsc[i] + nsc[j]

print("Numarul total de subsiruri crescatoare: " + str(sum(nsc)))
```

3. Partiționarea unei mulțimi în două submulțimi cu sume cât mai apropiate (problemă NP-completă)

   Exemplu: A = [5, 3, 5, 4, 5] => A1 = [5, 5], A2 = [5, 4, 3] (suma(A1) = 10, suma(A2) = 12 => diferența minimă = 2)

   ***Indicație de rezolvare*:** Vom construi o matrice sp cu n linii (n = |A|) și t coloane (t = sum(A)//2), în care un element sp[i][j] va fi egal cu True dacă suma j poate fi obținută folosind primele i elemente din mulțimea A sau False în caz contrar. Se observă faptul că problema are soluție doar în cazul în care t = sum(A) este un număr par!

   Algoritm de complexitate pseudo-polinomială O(n*t):

```
import sys

ls = [5, 3, 5, 4, 5]

n = len(ls)
t = sum(ls)

if t % 2 != 0:
    print("Imposibil!")
    sys.exit()
```

```
t = t // 2
sp = [[False for j in range(t+1)] for i in range(n+1)]

for i in range(n+1):
    sp[i][0] = True

for i in range(1, n+1):
    for j in range(1, t+1):
        sp[i][j] = sp[i-1][j]
        if ls[i-1] <= j:
            sp[i][j] = sp[i][j] or sp[i-1][j-ls[i-1]]

for j in range(t, -1, -1):
    if sp[n][j] == True:
        i = n
        sm_1 = []
        sm_2 = []
        while i > 0:
            if sp[i][j] != sp[i-1][j]:
                sm_1.append(ls[i-1])
                j = j - ls[i - 1]
            else:
                sm_2.append(ls[i-1])
            i = i-1
        print("Prima submultime: " + str(sm_1))
        print("A doua submultime: " + str(sm_2))
        print("Diferenta minima: " + str(abs(sum(sm_1) - sum(sm_2))))
        break
```

4. **Waiting for a ticket** (Balcaniada de Informatică 1997 - http://delab.csd.auth.gr/contests/boi_problems/421.html)

Grupul de rock U2 va da un concert în Nicosia. Un grup de n≤100.000 fani așteaptă la coadă în scopul de a cumpăra bilete de la singura casierie deschisă. Fiecare persoană vrea să cumpere numai un bilet, iar casierul poate vinde unei persoane cel mult două bilete. Casierul folosește T[i] unități de timp pentru a servi al i-lea fan (1≤i≤n). Totuși, este posibil ca doi fani așezați unul după altul să se înțeleagă astfel încât numai unul dintre ei să rămână la coadă, iar celălalt să plece, dacă timpul R[j] (1≤j≤n-1) în care casierul servește al j-lea și al (j+1)-lea fan este mai mic decât T[j]+T[j+1]. Deci, pentru a minimiza timpul de lucru al casierului, fiecare fan din coadă încearcă să negocieze cu predecesorul și cu succesorul său, ceea ce va conduce, în final, la o servire mai rapidă.
Fiind date numerele naturale nenule n (pe prima linie din fișierul de intrare), T[1],…,T[n] (pe a doua linie din fișierul de intrare) și R[1],…,R[n-1] (pe a treia linie din fișierul de intrare), se cere să se minimizeze timpul total de lucru al casierului, grupând în mod convenabil perechi de fani aflați unul după altul la coadă. Atenție, nu este obligatoriu ca un fan să se grupeze cu fanul aflat înainte sa sau după el, ci el poate fi servit și individual!

Exemplu:

concert.in	concert.out
7 5 4 3 2 1 4 4 7 3 4 2 2 4	Timpul minim: 14 Ordinea de servire: 1 2 + 3 4 + 5 6 + 7

Algoritm de complexitate O(n):

```
f = open("concert.in")

n = int(f.readline())
T = [int(x) for x in f.readline().split()]
R = [int(x) for x in f.readline().split()]

f.close()

T1 = [0] * n
T2 = [0] * n

T1[0] = T[0]
T1[1] = T[0] + T[1]

T2[0] = float("inf")
T2[1] = R[0]

for i in range(2, n):
    T1[i] = T[i] + min(T1[i-1], T2[i-1])
    T2[i] = R[i-1] + min(T1[i-2], T2[i-2])

print("Timpul minim: " + str(min(T1[n-1], T2[n-1])))

sol = []
k = n-1
while k >= 0:
    if T1[k] < T2[k]:
        sol.append(str(k+1))
        k -= 1
    else:
        sol.append(str(k) + " " + str(k+1))
        k -= 2

print(*sol[::-1], sep="\n")
```