TEHNICA PROGRAMĂRII DINAMICE
(continuare)

1. Subșir crescător maximal (LIS = Longest Increasing Subsequence)

Considerăm un șir t format din n numere întregi $t = (t_0, t_1, \dots, t_{n-1})$. Să se determine un subșir crescător de lungime maximă al șirului dat t.

Șir:

$$t = (9, 7, 3, 6, 2, 8, 5, 4, 5, 8, 7, 10, 4)$$

Secvență: elemente aflate pe poziții consecutive în șirul dat

$$t = (9, 7, 3, 6, 2, 8, 5, 4, 5, 8, 7, 10, 4)$$

Determinarea secvenței crescătoare maximale (de lungime maximă):

```
#include <fstream>
#include <iostream>

using namespace std;

int main()
{
    ifstream f("numere.txt");

    int n, lcrt, lmax, pmax, i, *v;

    f >> n;

    v = new int[n];
    for(i = 0; i < n; i++)
        f >> v[i];

    f.close();

    lcrt = lmax = 1;
    pmax = 0;
    for(i = 0; i < n-1; i++)
        if(v[i] <= v[i+1])
        {
            lcrt++;
            if(lcrt > lmax)
            {
                lmax = lcrt;
                pmax = i+1;
```

```
            }

        }
        else
            lcrt = 1;

    cout << "Lungimea maxima a unei secvente crescatoare: " << lmax << endl;

    cout << "O secventa crescatoare de lungime maxima: " << endl;
    for(i = pmax-lmax+1; i <= pmax; i++)
        cout << v[i] << " ";

    delete []v;

    return 0;
}
```

Subșir: elemente aflate pe poziții strict crescătoare (nu sunt neapărat poziții consecutive, dar elementele sunt în ordinea din șirul dat)

$$t = (9, 7, 3, 6, 2, 8, 5, 4, 5, 8, 7, 10, 4)$$

$$subșir = (7, 3, 8, 7, 4)$$

Observație: Orice secvență este un subșir, dar reciproca nu este neapărat adevărată!

Submulțime: elemente care nu păstrează neapărat ordinea

$$t = (9, 7, 3, 6, 2, 8, 5, 4, 5, 8, 7, 10, 4)$$

$$submulțime = (5, 7, 7, 6, 4)$$

De exemplu, în șirul $t = (9, 7, 3, 6, 2, 8, 5, 4, 5, 8, 7, 10, 4)$ un subșir crescător maximal este $(3, 6, 8, 8, 10)$. Soluția nu este unică, un alt subșir crescător maximal fiind $(3, 4, 5, 8, 10)$.

Se observă faptul că problema are întotdeauna soluție, chiar și în cazul în care șirul dat este strict descrescător (orice element al șirului este un subșir crescător maximal de lungime 1)!

Algoritmii de tip Greedy nu rezolvă corect această problemă în orice caz. De exemplu, pentru fiecare element din șirul dat, am putea încerca să construim un subșir crescător maximal selectând, de fiecare dată, cel mai apropiat element mai mare sau egal decât ultimul element din subșirul curent și să reținem subșirul crescător de lungime maximă astfel obținut. Aplicând acest algoritm pentru șirul $t = (7, 3, 9, 4, 5)$ vom obține subșirurile

(7, 9), (3, 9), (9), (4, 5) și (5), deci soluția furnizată de algoritmul Greedy ar fi unul dintre cele 3 subșiruri crescătoare de lungime 2. Evident, soluția ar fi incorectă, deoarece soluția optimă este subșirul (3, 4, 5), de lungime 3!

Un algoritm de tip Backtracking ar trebui să genereze toate submulțimile strict crescătoare de indici (combinări) cu $1, 2, \ldots, n$ elemente, pentru fiecare submulțime de indici să testeze dacă subșirul asociat este crescător și, în caz afirmativ, să rețină subșirul de lungime maximă.

$$t = (9, 7, 3, 6, 2, 8, 5, 4, 5, 8, 7, 10, 4)$$

$indici$ = (2, 3, 6, 9, 10) => $subșir = (3, 6, 5, 8, 7) - nu\ este\ crescător$!

Algoritmul este corect, dar ineficient, deoarece numărul submulțimilor generate și testate ar fi egal cu $C_n^1 + C_n^2 + \cdots + C_n^n = 2^n - 1$, deci algoritmul are avea o complexitate exponențială!

În continuare, vom prezenta un algoritm pentru rezolvarea acestei probleme folosind tehnica programării dinamice (un algoritm de tip Divide et Impera s-ar baza pe aceeași idee, însă fără a utiliza tehnica memoizării).

Pentru a determina un subșir crescător maximal, vom calcula, într-o manieră ascendentă, lungimea maximă a unui subșir crescător care se termină cu $t[0]$, apoi lungimea maximă a unui subșir crescător care se termină cu $t[1]$, ..., respectiv lungimea maximă a unui subșir crescător care se termină cu $t[n-1]$, iar valorile obținute (optimele locale) le vom păstra într-un tablou $lmax$ cu n elemente, respectiv $lmax[i]$ va memora lungimea maximă a unui subșir crescător care se termină cu $t[i]$.

Pentru a calcula lungimea maximă a unui subșir crescător care se termină cu elementul $t[i]$, vom lua în considerare, pe rând, toate subșirurile care se termină cu elementele $t[0], t[1], \ldots, t[i-1]$, deoarece cunoaștem deja lungimile maxime $lmax[0], lmax[1], \ldots, lmax[i-1]$ ale subșirurilor crescătoare care se termină cu ele, și vom încerca să alipim elementul $t[i]$ la fiecare dintre ele. Dacă acest lucru este posibil, respectiv dacă $t[i] \geq t[j]$ pentru un indice $j \in \{0,1, \ldots, i-1\}$, vom compara lungimea subșirului care s-ar obține, egală cu $1 + lmax[j]$, cu lungimea maximă $lmax[i]$ găsită până în acel moment și, dacă noua lungime este mai mare, vom actualiza valoarea lui $lmax[i]$.

Se observă ușor faptul că problema dată îndeplinește atât *condiția de substructură optimală* (calcularea valorii $lmax[i]$ depinde de valorile $lmax[0], lmax[1], \ldots, lmax[i-1]$), cât și *condiția de superpozabilitate* (valoarea $lmax[i]$ va fi utilizată în calculul valorilor $lmax[i+$

1], … , $lmax[n-1]$), iar relația de recurență care caracterizează substructura optimală a problemei, în formă memoizată, este următoarea:

$$lmax[i] = \begin{cases} 1, \text{ pentru } i = 0 \\ 1 + \max\limits_{0 \le j < i} \{lmax[j] | t[i] \ge t[j]\}, \text{ pentru } 1 \le i \le n-1 \end{cases}$$

Pentru a reconstitui mai simplu un subșir crescător maximal al șirului inițial, vom utiliza un tabloul auxiliar unidimensional $pred$, în care un element $pred[i]$ va conține valoarea -1 dacă elementul $t[i]$ nu a putut fi alipit la niciunul dintre subșirurile crescătoare maximale care se termină cu $t[0], t[1], \dots, t[i-1]$ sau va conține indicele $j \in \{0,1, \dots, i-1\}$ al subșirului crescător maximal $t[j]$ la care a fost alipit elementul $t[i]$, adică indicele j pentru care s-a obținut $\max\limits_{0 \le j < i} \{lmax[j] | t[i] \ge t[j]\}$.

Considerând tabloul t din exemplul inițial, vom obține următoarele valori:

i	0	1	2	3	4	5	6	7	8	9	10	11	12
t	9	7	3	6	2	8	5	4	5	8	7	10	4
lmax	1	1	1	2	1	3	2	2	3	4	4	5	3
pred	-1	-1	-1	2	-1	3	4	4	6	5	8	9	7

i	0	1	2	3	4	5	6	7	8	9	10	pmax 11	12
t	9	7	3	6	2	8	5	4	5	8	7	10	4
lmax	1	1	1	2	1	3	2	2	3	4	4	5	3
pred	-1	-1	-1	2	-1	3	2	2	6	5	8	9	7

$Subșir = (10, 8, 8, 6, 3)$

Valorile din tablourile $lmax$ și $pred$ au fost calculate astfel:

- $lmax[0] = 1$ și $pred[0] = -1$, deoarece este evident faptul că lungimea maximă a unui subșir care se termină cu $t[0]$este egală cu 1;
- $lmax[1] = 1$ și $pred[1] = -1$, deoarece elementul $t[1] = 7$ nu poate fi alipit la un subșir crescător maximal care se termină cu $t[0] = 9 > 7$;
- $lmax[2] = 1$ și $pred[2] = -1$, deoarece elementul $t[2] = 3$ nu poate fi alipit nici la un subșir crescător maximal care se termină cu $t[0] = 9 > 3$ și nici la un subșir crescător maximal care se termină cu $t[1] = 7 > 3$;

- $lmax[3] = 2$ şi $pred[3] = 2$, deoarece elementul $t[3] = 6$ poate fi alipit la un subşir crescător maximal care se termină cu $t[2] = 3 \leq 6$, deci $lmax[3] = 1 + lmax[2] = 2$;
- $lmax[4] = 1$ şi $pred[4] = -1$, deoarece elementul $t[4] = 2$ nu poate fi alipit nici la un subşir crescător maximal care se termină cu $t[0], t[1], t[2]$ sau $t[3]$;
- $lmax[5] = 3$ şi $pred[5] = 3$, deoarece elementul $t[5] = 8$ poate fi alipit la oricare dintre subşirurile crescătoare maximale care se termină cu $t[1], t[2], t[3]$ sau $t[4]$, dar lungimea maximă se obţine când îl alipim la subşirul crescător maximal care se termină cu $t[3]$, deoarece $lmax[3]$ este cea mai mare dintre valorile $lmax[1]$, $lmax[2], lmax[3]$ şi $lmax[4]$;
- .....

Lungimea maximă a unui subşir crescător maximal al şirului iniţial este dată de valoarea maximă din tabloul $lmax$, iar pentru a reconstitui un subşir crescător maximal vom utiliza informaţiile din tabloul $pred$ şi faptul că un subşir crescător maximal se termină cu elementul $t[pmax]$, unde $pmax$ este poziţia pe care se află valoarea maximă din tabloul $lmax$.

În cazul exemplului de mai sus, valoarea maximă din tabloul $lmax$ este $lmax[11] = 5$, deci $pmax = 11$, ceea ce înseamnă că un subşir crescător maximal se termină cu $t[11]$. Pentru afişarea unui subşir crescător maximal vom proceda astfel:

- iniţializăm un indice i cu $pmax$, deci $i = 11$;
- $i = 11 \neq -1$, deci afişăm elementul $t[i] = t[11] = 10$ şi indicele i devine egal cu $pred[11] = 9$;
- $i = 9 \neq -1$, deci afişăm elementul $t[i] = t[9] = 8$ şi indicele i devine egal cu $pred[9] = 5$;
- $i = 5 \neq -1$, deci afişăm elementul $t[i] = t[5] = 8$ şi indicele i devine egal cu $pred[5] = 3$;
- $i = 3 \neq -1$, deci afişăm elementul $t[i] = t[3] = 6$ şi indicele i devine egal cu $pred[3] = 2$;
- $i = 2 \neq -1$, deci afişăm elementul $t[i] = t[2] = 3$ şi indicele i devine egal cu $pred[2] = -1$;
- $i = -1$, deci am terminat de afişat un subşir crescător maximal inversat şi ne oprim.

Pentru a afişa subşirul crescător maximal reconstituit neinversat, fie îl salvăm într-o structură de date auxiliară şi apoi îl afişăm invers, fie utilizăm o funcţie recursivă (variant[utilizată în implementarea de mai jos).

În continuare, vom prezenta implementarea acestui algoritm în limbajul C, considerând faptul că datele de intrare se citesc din fişierul text `sir.txt`, care conţine pe prima linie dimensiunea n a şirului, iar pe următoarea linie se află cele n elemente ale şirului:

```
#include <iostream>
#include <fstream>

using namespace std;

void afisare(int t[], int pred[], int pozCrt)
{
    if (pozCrt != -1)
    {
        afisare(t, pred, pred[pozCrt]);
        cout << t[pozCrt] << " ";
```

```
    }
}

int main()
{
    int n;
    ifstream fin("fisier.in");
    fin >> n;
    int *tablou = new int[n];

    for (int i = 0; i < n; i++)
    {
        fin >> tablou[i];
    }
    fin.close();
    int *lmax = new int[n];
    int *pred = new int[n];

    lmax[0] = 1;
    pred[0] = -1;

    for (int i = 1; i < n; i++)
    {
        lmax[i] = 1;
        pred[i] = -1;
        for (int j = 0; j < i; j++)
            if (tablou[i] >= tablou[j] && 1 + lmax[j] > lmax[i])
            {
                lmax[i] = 1 + lmax[j];
                pred[i] = j;
            }
    }
    int pozMax = 0;

    for (int i = 0; i < n; i++)
        if (lmax[i] > lmax[pozMax])
            pozMax = i;

    cout << "Lungimea maxima a subsirului crescator este: " <<
           lmax[pozMax] << endl;
    cout << "Un subsir crescator maximal: " << endl;
    afisare(tablou, pred, pozMax);

    delete[] tablou;
    delete[] lmax;
    delete[] pred;

    return 0;
}
```

Algoritmul prezentat utilizează varianta înapoi a tehnicii programării dinamice (deoarece pentru a calcula soluția optimă $lmax[i]$ a subproblemei i am utilizat soluțiile

optime $lmax[0], lmax[1], \dots, lmax[i-1]$ ale subproblemelor $0,1, \dots, i-1)$, iar complexitatea sa este egală cu $\mathcal{O}(n^2)$.

În continuare, vom prezenta un algoritm care utilizează varianta înainte a tehnicii programării dinamice, respectiv vom calcula soluția optimă $lmax[i]$ a subproblemei i utilizând soluțiile optime $lmax[i+1], lmax[i+2], \dots, lmax[n-1]$ ale subproblemelor $i+1, i+2, \dots, n-1$. În acest scop, vom calcula, într-o manieră ascendentă, lungimea maximă a unui subșir crescător care începe cu $t[n-1]$, apoi lungimea maximă a unui subșir crescător care începe cu $t[n-2]$, ..., respectiv lungimea maximă a unui subșir crescător care începe cu $t[n-1]$, iar valorile obținute (optimele locale) le vom păstra în tabloul $lmax$ cu n elemente, respectiv $lmax[i]$ va memora lungimea maximă a unui subșir crescător care începe cu $t[i]$.

Pentru a calcula lungimea maximă a unui subșir crescător care începe cu elementul $t[i]$, vom lua în considerare, pe rând, toate subșirurile care încep cu elementele $t[i+1], t[i+2], \dots, t[n-1]$, deoarece cunoaștem deja lungimile maxime $lmax[i+1], lmax[i+2], \dots, lmax[n-1]$ ale subșirurilor crescătoare care încep cu ele, și vom încerca să adăugăm elementul $t[i]$ înaintea fiecăruia dintre ele. Dacă acest lucru este posibil, respectiv dacă $t[i] \leq t[j]$ pentru un indice $j \in \{i+1, i+2, \dots, n-1\}$, vom compara lungimea subșirului care s-ar obține, egală cu $1 + lmax[j]$, cu lungimea maximă $lmax[i]$ găsită până în acel moment și, dacă noua lungime este mai mare, vom actualiza valoarea lui $lmax[i]$.

Se observă ușor faptul că problema dată îndeplinește atât *condiția de substructură optimală* (calcularea valorii $lmax[i]$ depinde de valorile $lmax[i+1], lmax[i+2], \dots, lmax[n-1]$), cât și *condiția de superpozabilitate* (valoarea $lmax[i]$ va fi utilizată în calculul valorilor $lmax[0], \dots, lmax[i-1]$), iar relația de recurență care caracterizează substructura optimală a problemei, în formă memoizată, este următoarea:

$$lmax[i] = \begin{cases} 1, & \text{pentru } i = n-1 \\ 1 + \max\limits_{i+1 \leq j < n} \{lmax[j] | t[i] \leq t[j]\}, & \text{pentru } 0 \leq i < n-1 \end{cases}$$

Pentru a reconstitui mai simplu un subșir crescător maximal al șirului inițial, vom utiliza un tabloul auxiliar unidimensional *succ*, în care un element $succ[i]$ va conține valoarea -1 dacă elementul $t[i]$ nu a putut fi adăugat înaintea niciunuia dintre subșirurile crescătoare maximale care încep cu $t[i+1], t[i+2], \dots, t[n-1]$ sau va conține indicele $j \in \{i+1, i+2, \dots, n-1\}$ al subșirului crescător maximal $t[j]$ înaintea căruia a fost adăugat elementul $t[i]$, adică indicele j pentru care s-a obținut $\max\limits_{i+1 \leq j < n} \{lmax[j] | t[i] \leq t[j]\}$.

Considerând tabloul t din exemplul inițial, vom obține următoarele valori:

i	0	1	2	3	4	5	6	7	8	9	10	11	12
t	9	7	3	6	2	8	5	4	5	8	7	10	4
lmax	2	4	5	4	5	3	4	4	3	2	2	1	1
succ	11	5	3	5	6	9	8	8	9	11	11	-1	-1

$Subșir = (3, 6, 8, 8, 10)$

Valorile din tablourile $lmax$ şi $succ$ au fost calculate astfel:

- $lmax[12] = 1$ şi $succ[12] = -1$, deoarece este evident faptul că lungimea maximă a unui subşir care începe cu $t[12]$este egală cu 1;
- $lmax[11] = 1$ şi $succ[11] = -1$, deoarece elementul $t[11] = 10$ nu poate fi adăugat înaintea unui subşir crescător maximal care începe cu $t[12] = 4 < 10$;
- $lmax[10] = 2$ şi $succ[10] = 11$, deoarece elementul $t[10] = 7$ poate fi adăugat înaintea unui subşir crescător maximal care începe cu $t[11] = 10 \geq 7$, deci $lmax[10] = 1 + lmax[11] = 2$;
- $lmax[9] = 2$ şi $succ[9] = 11$, deoarece elementul $t[9] = 8$ poate fi adăugat înaintea unui subşir crescător maximal care începe cu $t[11] = 10 \geq 8$, deci $lmax[9] = 1 + lmax[11] = 2$;
- $lmax[8] = 3$ şi $succ[8] = 9$, deoarece elementul $t[8] = 5$ poate fi adăugat înaintea oricăruia dintre subşirurile crescătoare maximale care încep cu $t[9], t[10]$ sau $t[11]$, dar lungimea maximă se obţine când îl alipim la subşirul crescător maximal care începe cu $t[9]$, deoarece $lmax[9]$ este cea mai mare dintre valorile $lmax[9]$, $lmax[10]$ şi $lmax[11]$;
- .....

Lungimea maximă a unui subşir crescător maximal al şirului iniţial este dată de valoarea maximă din tabloul $lmax$, iar pentru a reconstitui un subşir crescător maximal vom utiliza informaţiile din tabloul $succ$ şi faptul că un subşir crescător maximal începe cu elementul $t[pmax]$, unde $pmax$ este poziţia pe care se află valoarea maximă din tabloul $lmax$.

În cazul exemplului de mai sus, valoarea maximă din tabloul $lmax$ este $lmax[2] = 5$, deci $pmax = 2$, ceea ce înseamnă că un subşir crescător maximal începe cu $t[2]$. Pentru afişarea unui subşir crescător maximal vom proceda astfel:

- iniţializăm un indice i cu $pmax$, deci $i = 2$;
- $i = 2 \neq -1$, deci afişăm elementul $t[i] = t[2] = 3$ şi indicele i devine egal cu $succ[2] = 3$;
- $i = 3 \neq -1$, deci afişăm elementul $t[i] = t[3] = 6$ şi indicele i devine egal cu $succ[3] = 5$;
- $i = 5 \neq -1$, deci afişăm elementul $t[i] = t[5] = 8$ şi indicele i devine egal cu $succ[5] = 9$;
- $i = 9 \neq -1$, deci afişăm elementul $t[i] = t[9] = 8$ şi indicele i devine egal cu $succ[9] = 11$;
- $i = 11 \neq -1$, deci afişăm elementul $t[i] = t[11] = 10$ şi indicele i devine egal cu $succ[11] = -1$;
- $i = -1$, deci am terminat de afişat un subşir crescător maximal şi ne oprim.

În continuare, vom prezenta implementarea acestui algoritm în limbajul C, considerând faptul că datele de intrare se citesc din fişierul text `sir.txt`, care conţine pe prima linie dimensiunea n a şirului, iar pe următoarea linie se află cele n elemente ale şirului:

```
#include <iostream>
#include <fstream>

using namespace std;

void afisare(int t[], int pred[], int pozCrt)
{
    if (pozCrt != -1)
    {
        cout << t[pozCrt] << " ";
        afisare(t, pred, pred[pozCrt]);
    }
}

int main()
{
    int n;

    ifstream fin("fisier.in");

    fin >> n;

    int *tablou = new int[n];

    for (int i = 0; i < n; i++)
    {
        fin >> tablou[i];
    }

    fin.close();

    int *lmax = new int[n];
    int *succ = new int[n];

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

    for (int i = n-2; i >= 0; i--)
    {
        lmax[i] = 1;
        succ[i] = -1;
        for (int j = i+1; j < n; j++)
            if (tablou[i] <= tablou[j] && 1 + lmax[j] > lmax[i])
            {
                lmax[i] = 1 + lmax[j];
                succ[i] = j;
            }
    }
    int pozMax = 0;

    for (int i = 0; i < n; i++)
        if (lmax[i] > lmax[pozMax])
            pozMax = i;
```

```
    cout << "Lungimea maxima a subsirului crescator este: " <<
          lmax[pozMax] << endl;

    cout << "Un subsir crescator maximal: " << endl;
    afisare(tablou, succ, pozMax);

//    for(int i = pozMax; i != -1; i = succ[i])
//        cout << tablou[i] << " ";

    delete[] tablou;
    delete[] lmax;
    delete[] succ;

    return 0;
}
```

Algoritmul prezentat are complexitatea egală tot cu $\mathcal{O}(n^2)$, aceasta nefiind însă minimă. O rezolvare cu complexitatea $\mathcal{O}(n \log_2 n)$ se poate obține utilizând căutarea binară: https://www.geeksforgeeks.org/longest-monotonically-increasing-subsequence-size-n-log-n/.

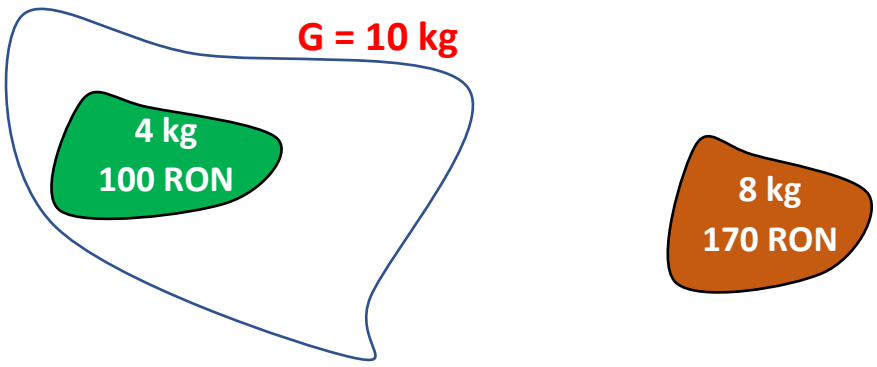
2. Problema rucsacului (varianta discretă)

Considerăm un rucsac având capacitatea maximă G și n obiecte $O_1, O_2, \dots, O_n$ pentru care cunoaștem greutățile lor $g_1, g_2, \dots, g_n$ și câștigurile $c_1, c_2, \dots, c_n$ obținute prin încărcarea lor în rucsac. Știind faptul că toate greutățile și toate câștigurile sunt numere naturale nenule, iar orice obiect poate fi încărcat doar complet în rucsac (nu poate fi "tăiat"), să se determine o modalitate de încărcare a rucsacului astfel încât câștigul total obținut să fie maxim.

De exemplu, considerând $G = 10$ kg și $n = 5$ obiecte O_1, O_2, O_3, O_4, O_5 având câștigurile $c = (80, 50, 400, 60, 70)$ RON și greutățile $g = (5, 2, 20, 3, 4)$ kg, putem obține un câștig maxim egal cu 190 RON, încărcând obiectele O_1, O_2 și O_4.

Astfel, câștigurile unitare ale obiectelor din exemplul de mai sus vor fi $u = (16, 25, 20, 20, 17.5)$ RON/kg. Astfel, algoritmul Greedy ar selecta obiectele O_2, O_4 și O_5, deoarece obiectele nu pot fi "tăiate" în varianta discretă a problemei rucsacului, și ar obține un câștig total egal cu 180 RON, evident mai mic decât cel maxim de 190 RON!

Se observă foarte ușor faptul că varianta discretă a problemei rucsacului nu are întotdeauna soluție!



cmax[2] = c[2] + cmax[G-g[2]] -> cu obiectul 1!!!

cmax[2] = 170 + cmax[2] = 170 + 0 = 170 (încarc obiectul 2) > 100 (nu încarc obiectul 2)

În concluzie, pentru a rezolva problema utilizând tehnica programării dinamice, trebuie să cunoaștem toate câștigurile maxime care se pot obține folosind primele i obiecte ($i \in \{0,1,\dots,n\}$), în limita oricărei capacități j cuprinse între 0 și G, deci, aplicând tehnica memoizării, vom considera un tablou bidimensional $cmax$ cu $n+1$ linii și $G+1$ coloane în care un element $cmax[i][j]$ va memora câștigul maxim care se poate obține folosind primele i obiecte în limita a j kilograme. Astfel, relația de recurență care caracterizează substructura optimală a problemei este următoarea:

$$cmax[i][j] = \begin{cases} 0, & \text{dacă } i = 0 \text{ sau } j = 0 \\ cmax[i-1][j], & \text{dacă } g[i] > j \\ max\begin{Bmatrix} cmax[i-1][j], \\ c[i] + cmax[i-1][j-g[i]] \end{Bmatrix}, & \text{dacă } g[i] \leq j \end{cases}$$

pentru fiecare $i \in \{0,1,\dots,n\}$ și fiecare $j \in \{0,1,\dots,G\}$. În plus față de modalitatea de calcul a elementului $cmax[i][j]$ descrisă mai sus, am adăugat cazurile particulare $cmax[0][j] = cmax[i][0] = 0$ (evident, câștigul maxim $cmax[0][j]$ care se poate obține folosind 0 obiecte în limita oricărei capacități j este 0 și câștigul maxim $cmax[i][0]$ care se poate obține folosind primele i obiecte în limita unei capacități nule este tot 0).

G = 10 kg

$c = (80, 50, 400, 60, 70)$ RON

$g = (5, 2, 20, 3, 4)$ kg

$$cmax[i][j] = \begin{cases} 0, & \text{dacă i} = 0 \text{ sau } j = 0 \\ cmax[i-1][j], & \text{dacă } g[i] > j \\ max\begin{Bmatrix} cmax[i-1][j], \\ c[i] + cmax[i-1][j - g[i]] \end{Bmatrix}, & \text{dacă } g[i] \leq j \end{cases}$$

	c_i	g_i	i/j	0	1	2	3	4	5	6	7	8	9	10
			0	0	0	0	0	0	0	0	0	0	0	0
O_1	80	5	1	0	0	0	0	0	80	80	80	80	80	80
O_2	50	2	2	0	0	50	50	50	80	80	130	130	130	130
O_3	400	20	3	0	0	50	50	50	80	80	130	130	130	130
O_4	60	3	4	0	0	50	60	60	110	110	130	140	140	190
O_5	70	4	5	0	0	50	60	70	110	120	130	140	180	190

cmax[1][5] = maxim {cmax[1-1][5], 80 + cmax[1-1][5-5]} =maxim{0, 80}

cmax[2][5] = maxim {cmax[2-1][5], 50 + cmax[2-1][5-2]} =maxim{80, 50}

cmax[4][7] = maxim {cmax[4-1][7], 60 + cmax[4-1][7-3]} = maxim{130, 60+50}
cmax[4][8] = maxim {cmax[4-1][8], 60 + cmax[4-1][8-3]} = maxim{130, 60+80}

În cazul exemplului de mai sus, avem $cmax[5][10] = 190$, deci profitul maxim care se poate obține este de 190 RON, iar pentru reconstituirea unei modalități optime de încărcare a rucsacului vom urma traseul marcat cu roșu în matricea $cmax$, obiectele care se vor încărca în rucsac corespunzând liniilor pe care se află elementele încadrate cu un dreptunghi, respectiv obiectele O_1, O_2 și O_4:

	c_i	g_i	i/j	0	1	2	3	4	5	6	7	8	9	10
			0	0	0	0	0	0	0	0	0	0	0	0
O_1	80	5	1	0	0	0	0	0	80	80	80	80	80	80
O_2	50	2	2	0	0	50	50	50	80	80	130	130	130	130
O_3	400	20	3	0	0	50	50	50	80	80	130	130	130	130
O_4	60	3	4	0	0	50	60	60	110	110	130	140	140	190
O_5	70	4	5	0	0	50	60	70	110	120	130	140	180	190

Algoritmul prezentat utilizează varianta înapoi a tehnicii programării dinamice, iar complexitatea sa este una de tip pseudo-polinomial, fiind egală cu $\mathcal{O}(nG) \approx \mathcal{O}\left(n2^{[\log_2 G]}\right)$.

Implementarea algoritmului în limbajul C:

```
#include <fstream>
#include <iostream>

using namespace std;

int main()
{
    int G, n, **cmax, *c, *g;

    ifstream fin("obiecte.txt");

    fin >> G;
    fin >> n;

    c = new int[n + 1];
    g = new int[n + 1];

    for (int i = 1; i <= n; i++)
        fin >> c[i] >> g[i];

    fin.close();

    cmax = new int*[n + 1];
    for(int i = 0; i <= n; i++)
        cmax[i] = new int[G + 1];
```

```
    for (int i = 0; i <= n; i++)
        cmax[i][0] = 0;

    for (int j = 0; j <= G; j++)
        cmax[0][j] = 0;

    for(int i = 1; i <= n; i++)
        for(int j = 1; j <= G; j++)
            if(g[i] > j)
                cmax[i][j] = cmax[i-1][j];
            else
                if(c[i] + cmax[i-1][j-g[i]] > cmax[i-1][j])
                    cmax[i][j] = c[i] + cmax[i-1][j-g[i]];
                else
                    cmax[i][j] = cmax[i-1][j];

    cout << "Castigul maxim: " << cmax[n][G] << " RON" << endl;

    cout << "Obiectele incarcate: " << endl;

    int i = n;
    int j = G;
    while(i != 0)
        if(cmax[i][j] != cmax[i-1][j])
        {
            cout << "Obiectul " << i << endl;
            j = j - g[i];
        }
        else
            i--;


    delete[] c;
    delete[] g;

    for(int i = 0; i <= n; i++)
        delete[] cmax[i];
    delete[] cmax;

    return 0;
}
```