CURS 13
TEHNICA PROGRAMĂRII DINAMICE

1. Problema rucsacului (varianta discretă)

Considerăm un rucsac având capacitatea maximă G și n obiecte $O_1, O_2, \ldots, O_n$ pentru care cunoaștem greutățile lor $g_1, g_2, \ldots, g_n$ și câștigurile $c_1, c_2, \ldots, c_n$ 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.

În capitolul dedicat tehnicii de programare Greedy am văzut faptul că varianta fracționară a acestei probleme poate fi rezolvată corect utilizând tehnica respectivă. În cazul variantei discrete, tehnica Greedy nu va mai furniza o soluție corectă întotdeauna. 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, respectiv în cazul în care greutatea celui mai mic obiect este strict mai mare decât capacitatea G a rucsacului, în timp ce varianta fracționară ar avea soluție în acest caz (ar "tăia" din obiectul cu cel mai mare câștig unitar o bucată cu greutatea G).

Pentru a rezolva problema folosind metoda programării dinamice, vom proceda într-un mod asemănător cu cel utilizat pentru a rezolva problema plății unei sume folosind un număr minim de monede, astfel:

- considerăm faptul că am analizat, pe rând, obiectele $O_1, O_2, \ldots, O_{n-1}$ și am calculat câștigul maxim pe care îl putem obține folosindu-le (nu neapărat pe toate!) în limita întregii capacități G a rucsacului, deci mai trebuie să calculăm doar câștigul maxim pe care îl putem obține folosind și ultimul obiect O_n;
- dacă obiectul O_n nu încape în rucsac (deci $g_n > G$), înseamnă că nu-l putem folosi deloc, deci câștigul maxim rămâne cel pe care l-am obținut deja utilizând obiectele $O_1, O_2, \ldots, O_{n-1}$;
- dacă obiectul O_n încape în rucsac (deci $g_n \leq G$), înseamnă că trebuie să decidem dacă este rentabil să-l încărcăm sau nu, comparând câștigul maxim deja obținut folosind obiectele $O_1, O_2, \ldots, O_{n-1}$ în limita întregii capacități G a rucsacului cu câștigul care s-ar obține prin încărcarea obiectului O_n, respectiv cu suma dintre c_n și câștigul maxim care se poate obține folosind obiectele $O_1, O_2, \ldots, O_{n-1}$ în limita capacității $G - g_n$ rămase în rucsac. Deoarece $1 \leq g_n \leq G$, rezultă că trebuie să cunoaștem câștigurile maxime care se pot obține folosind obiectele $O_1, O_2, \ldots, O_{n-1}$ în limita oricărei capacități cuprinse între 0 și G-1, la care se adaugă câștigul

maxim care se poate obține folosind obiectele $O_1, O_2, \dots, O_{n-1}$ în limita întregii capacități G a rucsacului (pentru cazul anterior), deci, de fapt, trebuie să cunoaștem câștigurile maxime care se pot obține folosind obiectele $O_1, O_2, \dots, O_{n-1}$ în limita oricărei capacități cuprinse între 0 și G!

- pentru a calcula câștigurile maxime care se pot obține folosind primele $n-1$ obiecte $O_1, O_2, \dots, O_{n-1}$ în limita oricărei capacități cuprinse între 0 și G vom repeta raționamentul anterior pentru obiectul O_{n-1} și obiectele $O_1, O_2, \dots, O_{n-2}$, apoi pentru obiectul O_{n-2} și obiectele $O_1, O_2, \dots, O_{n-3}$ și așa mai departe, până când vom calcula câștigurile maxime care se pot obține folosind doar primul obiect O_1 în limita oricărei capacități cuprinse între 0 și G.

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:

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

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). De asemenea, am considerat tablourile c și g ca fiind indexate de la 1, pentru a păs

Considerând exemplul dat, vom obține următoarele valori pentru elementele matricei $cmax$:

	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

Elementele evidențiate în matricea $cmax$ au fost calculate astfel:

- $cmax[1][1] = cmax[1][2] = cmax[1][3] = cmax[1][4] = 0$, deoarece obiectul O_1 are greutatea $g_1 = 5$, deci poate fi încărcat doar în cazul în care capacitatea j a rucsacului este cel puțin egală cu 5 (de exemplu, folosind relația de recurență, obținem $cmax[1][2] = cmax[0][2] = 0$), caz în care am obținut $cmax[1][5] = \cdots = cmax[1][10] = 80$ (de exemplu, folosind relația de recurență, obținem $cmax[1][9] = \max\{cmax[0][9], c[1] + cmax[0][9-5]\} = \max\{0, 80 + 0\} = 80$);
- $cmax[2][7] = \max\{cmax[1][7], c[2] + cmax[1][7-2]\} = \max\{80, 50 + 80\} = 130$, deoarece în limita a $j = 7$ kg încap ambele obiecte O_1 și O_2;
- linia 3 este egală cu linia 2, deoarece $g_3 = 20$ kg $> G = 10$ kg, deci obiectul O_3 nu se poate încărca în niciun caz în rucsac;
- $cmax[4][10] = \max\{cmax[3][10], c[4] + cmax[3][10-3]\} = \max\{130, 60 + 130\} = 190$, deoarece în limita a $j = 10$ kg se poate adăuga obiectul O_4 la obiectele O_1 și O_2 care au fost încărcate pentru a obține câștigul maxim folosind primele $i = 3$ obiecte în limita a $j = 7$ kg;
- $cmax[5][9] = \max\{cmax[4][9], c[5] + cmax[4][9-4]\} = \max\{140, 70 + 110\} = 180$, deoarece în limita a $j = 9$ kg este mai rentabil să încărcăm obiectul O_5 alături de obiectele O_2 și O_4 (pentru care s-a obținut câștigul maxim de 110 RON folosind primele $i = 4$ obiecte în limita a $j = 5$ kg) decât să nu-l încărcăm, caz în care am păstra câștigul maxim de 140 RON obținut prin încărcarea obiectelor O_1 și O_4 dintre primele $i = 4$ obiecte în limita a $j = 9$ kg.

Câștigul maxim care se poate obține folosind toate cele n obiecte este dat de valoarea elementului $cmax[n][G]$, iar pentru a reconstitui o modalitate optimă de încărcare a rucsacului vom utiliza informațiile din matricea $cmax$, astfel:

- considerăm doi indici $i = n$ și $j = G$;
- dacă $cmax[i][j] = cmax[i-1][j]$, înseamnă fie că obiectul O_i nu încape în rucsac, fie încape, dar nu ar fi fost rentabil să-l încărcăm. Indiferent de motiv, obiectul O_i nu a fost încărcat în rucsac (nu face parte din soluția optimă), deci trecem la următorul obiect O_{i-1}, decrementând valoarea indicelui i;
- dacă $cmax[i][j] \neq cmax[i-1][j]$, înseamnă că a fost rentabil să încărcăm obiectul O_i în limita a j kg, deci îl afișăm și trecem la reconstituirea soluției optime pentru restul de $j - g[i]$ kg folosind obiectele $O_1, O_2, \ldots, O_{i-1}$, scăzând din indicele j valoarea $g[i]$ și decrementând indicele i.

Se observă faptul că obiectele se vor afișa în ordinea descrescătoare a indicilor lor (în "sens invers"), deci trebuie utilizată o structură de date auxiliară sau o funcție recursivă pentru a le afișa în ordinea crescătoare a indicilor lor!

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

În continuare, vom prezenta implementarea acestui algoritm în limbajul C, considerând faptul că datele de intrare se citesc din fișierul text `rucsac.txt`, care conține pe prima linie capacitatea G a rucsacului, pe a doua linie numărul n de obiecte, iar pe fiecare dintre următoarele n linii se află greutatea și câștigul câte unui obiect:

```
#include<stdio.h>

int main()
{
    int n, i, j, G, c[101], g[101], cmax[101][1001];

    FILE* f = fopen("rucsac.txt", "r");

    fscanf(f, "%d", &G);
    fscanf(f, "%d", &n);

    for(i = 1; i <= n; i++)
        fscanf(f, "%d %d", &g[i], &c[i]);

    fclose(f);

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

    for(i = 1; i <= n; i++)
        for (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];

    printf("Castig maxim: %d\n", cmax[n][G]);
```

```
    printf("Obiectele selectate:\n");
    i = n;
    j = G;
    while(i >= 1)
    {
        if(cmax[i][j] != cmax[i - 1][j])
        {
            printf("%d\n", i);
            j = j - g[i];
        }
        i--;
    }

    return 0;
}
```

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

2. Planificarea proiectelor cu bonus maxim

Considerăm n proiecte $P_1, P_2, \ldots, P_n$ pe care poate să le execute o echipă de programatori într-o anumită perioadă de timp (de exemplu, o lună), iar pentru fiecare proiect se cunoaște un interval de timp în care acesta trebuie executat (exprimat prin numerele de ordine a două zile din perioada respectivă), precum și bonusul pe care îl va obține echipa dacă proiectul este finalizat la timp (altfel, echipa nu va obține niciun bonus pentru proiectul respectiv). Să se determine o modalitate de planificare a unor proiecte care nu se suprapun astfel încât bonusul obținut de echipă să fie maxim. Vom considera faptul că un proiect care începe într-o anumită zi nu se suprapune cu un proiect care se termină în aceeași zi!

Exemplu:
Vom considera faptul că datele de intrare se citesc din fișierul text *proiecte.in*, care conține pe prima linie numărul n de proiecte, iar fiecare dintre următoarele n linii conține intervalul de timp în care proiectul trebuie executat și bonusul acordat. De exemplu, a doua linie din fișierul de intrare conține informațiile despre proiectul P_1, respectiv intervalul [7, 13] în care acesta trebuie efectuat pentru ca echipa să obțină bonusul de 850 RON. Datele de ieșire se vor scrie în fișierul text *proiecte.out*, în forma indicată mai jos:

proiecte.in	**proiecte.out**
`8`	`Proiectul 4: 02-06 ->   650 RON`
`7    13   850`	`Proiectul 1: 07-13 ->   850 RON`
`4    12   800`	`Proiectul 5: 13-18 ->  1000 RON`
`1    3    250`	`Proiectul 7: 25-27 ->   300 RON`
`2    6    650`	
`13   18   1000`	`Bonusul maxim al echipei: 2800 RON`
`4    16   900`	
`25   27   300`	
`15   22   900`	

Deși problema este asemănătoare cu *problema planificării unor proiecte cu profit maxim*, prezentată în capitolul dedicat tehnicii de programare Greedy, în care intervalele de executare ale proiectelor sunt restrânse la o singură zi, o strategie de tip Greedy nu va furniza întotdeauna o soluție corectă. De exemplu, dacă am planifica proiectele în ordinea descrescătoare a bonusurilor, atunci un proiect $P_1([1,10], 1000\text{ RON})$ cu bonus mare și durată mare ar fi programat înaintea a două proiecte $P_2([1,5], 900\text{ RON})$ și $P_3([6,9], 800\text{ RON})$ cu bonusuri și durate mai mici, dar având suma bonusurilor mai mare decât bonusul primului proiect (900+800 = 1700 > 1000). Într-un mod asemănător se pot găsi contraexemple și pentru alte criterii de selecție bazate pe ziua de început, pe ziua de sfârșit, pe durată sau pe raportul dintre bonusul și durata unui proiect!

Pentru a rezolva problema folosind metoda programării dinamice, vom proceda în următorul mod:

- considerăm proiectele $P_1, P_2, \dots, P_n$ ca fiind sortate în ordine crescătoare după ziua de sfârșit (vom vedea imediat de ce);
- considerăm faptul că am calculat bonusurile maxime $bmax_1, bmax_2, \dots, bmax_{i-1}$ pe care echipa le poate obține planificând o parte dintre primele i proiecte $P_1, P_2, \dots, P_{i-1}$ (sau chiar pe toate!), iar acum trebuie să calculăm bonusul maxim $bmax_i$ pe care echipa îl poate obține luând în considerare și proiectul P_i;
- înainte de a calcula $bmax_i$, vom determina cel mai mare indice $j \in \{1,2, \dots, i-1\}$ al unui proiect P_j după care poate fi planificat proiectul P_i (i.e., ziua de început a proiectului P_i este mai mare sau egală decât ziua în care se termină proiectul P_j) și vom nota acest indice j cu ult_i (dacă nu există nici un proiect P_j după care să poată fi planificat proiectul P_i, atunci vom considera $ult_i = 0$);
- calculăm $bmax_i$ ca fiind maximul dintre bonusul pe care îl echipa poate obține dacă nu planifică proiectul P_i, adică $bmax_{i-1}$, și bonusul pe care îl echipa poate obține dacă planifică proiectul P_i după proiectul P_{ult_i}, adică $bonus_i + bmax_{ult_i}$, unde prin $bonus_i$ am notat bonusul pe care îl primește echipa dacă finalizează proiectul P_i la timp.

Se observă faptul că ult_i se poate calcula mai ușor dacă proiectele sunt sortate crescător după ziua de terminare, deoarece ult_i va fi primul indice $j \in \{i-1, i-2, \dots, 1\}$ pentru care ziua de început a proiectului P_i este mai mare sau egală decât ziua în care se termină proiectul P_j. De asemenea, se observă faptul că valorile ult_i trebuie păstrate într-un tablou, deoarece sunt necesare pentru reconstituirea soluției.

Folosind observațiile și notațiile anterioare, precum și tehnica memoizării, relația de recurență care caracterizează substructura optimală a problemei este următoarea:

$$bmax[i] = \begin{cases} 0, & \text{dacă } i = 0 \\ \max\{bmax[i-1], bonus[i] + bmax[ult[i]]\}, & \text{dacă } i \geq 1 \end{cases}$$

Bonusul maxim pe care îl poate obține echipa este dat de valoarea elementului $bmax[n]$, iar pentru a reconstitui o modalitate optimă de planificare a proiectelor vom utiliza informațiile din matricea $bmax$, astfel:

- considerăm un indice $i = n$;
- dacă $bmax[i] \neq bmax[i-1]$, înseamnă că proiectul P_i a fost utilizat în planificarea optimă, deci îl afișăm și trecem la reconstituirea soluției optime care se termină cu proiectul $P_{ult[i]}$ după care a fost planificat proiectul P_i, respectiv indicele i ia valoarea $ult[i]$;

- dacă $bmax[i] = bmax[i-1]$, înseamnă că proiectul P_i nu a fost utilizat în planificarea optimă, deci trecem la următorul proiect P_{i-1}, decrementând valoarea indicelui i.

Se observă faptul că proiectele se vor afișa invers, deci trebuie utilizată o structură de date auxiliară sau o funcție recursivă pentru a le afișa în ordinea intervalelor în care trebuie executate!

Pentru exemplul dat, vom obține următoarele valori pentru elementele tablourilor ult și $bmax$ (informațiile despre proiectele $P_1, P_2, \dots, P_n$ ale echipei vor fi memorate într-un tablou pe cu elemente de tip structură și sortare crescător în funcție de ziua de sfârșit):

i	0	1	2	3	4	5	6	7	8
pe	—	P_3	P_4	P_2	P_1	P_6	P_5	P_8	P_7
		1 3	2 6	4 12	7 13	4 16	13 18	15 22	25 27
		250	650	800	850	900	1000	900	300
ult	—	0	0	1	2	1	4	4	7
bmax	0	250	650	1050	1500	1500	2500	2500	2800
		0	250	650	1050	1500	1500	2500	2500
		250	650	800+250	850+650	900+250	1000+1500	900+850	300+2500

Valorile din tabloul $bmax$ sunt cele scrise cu roșu și au fost calculate ca fiind maximul dintre cele două valori scrise cu albastru, determinate folosind relația de recurență. De exemplu, $bmax[4] = \max\{bmax[3],\ bonus[4] + bmax[ult[4]]\} = \max\{1050,\ 850 + bmax[2]\} = \max\{1050,\ 850 + 650\} = 1500$.

Pentru exemplul considerat, bonusul maxim pe care îl poate obține echipa este $bmax[8] = 2800$ RON, iar pentru a reconstitui o planificare optimă vom utiliza informațiile din tablourile $bmax$ și ult, astfel:

- inițializăm un indice $i = n = 8$;
- $bmax[i] = bmax[8] = 2800 \neq bmax[i-1] = bmax[7] = 2500$, deci proiectul $pe[8] = P_7$ a fost programat și îl afișăm, după care indicele i devine $i = ult[i] = ult[8] = 7$;
- $bmax[i] = bmax[7] = 2500 = bmax[i-1] = bmax[6] = 2500$, deci proiectul $pe[7] = P_8$ nu a fost programat și indicele i devine $i = i - 1 = 6$;
- $bmax[i] = bmax[6] = 2500 \neq bmax[i-1] = bmax[5] = 1500$, deci proiectul $pe[6] = P_5$ a fost programat și îl afișăm, după care indicele i devine $i = ult[i] = ult[6] = 4$;
- $bmax[i] = bmax[4] = 1500 \neq bmax[i-1] = bmax[3] = 1050$, deci proiectul $pe[4] = P_1$ a fost programat și îl afișăm, după care indicele i devine $i = ult[i] = ult[4] = 2$;
- $bmax[i] = bmax[2] = 650 \neq bmax[i-1] = bmax[1] = 250$, deci proiectul $pe[2] = P_4$ a fost programat și îl afișăm, după care indicele i devine $i = ult[i] = ult[2] = 0$;
- $i = 0$, deci am terminat de afișat o modalitate optimă de planificare a proiectelor și ne oprim.

În continuare, vom prezenta implementarea acestui algoritm în limbajul C, considerând faptul că datele de intrare se citesc din fișierul text `proiecte.txt`, care conține pe prima linie numărul de proiecte n, iar pe fiecare din următoarele n linii se află informațiile despre un proiect, în ordinea ziua inițială, ziua finală și bonusul (ID-ul proiectului este dat de numărul său de ordine):

```
#include<stdio.h>
#include<stdlib.h>

//structura Proiect este utilizata pentru memorarea
//informatiilor despre un proiect
typedef struct
{
    int ID, zi_initiala, zi_finala, bonus;
} Proiect;

//functie comparator utilizata pentru sortarea crescatoare
//a proiectelor în ordinea zilei de terminare (zi_finala)
int cmpProiecte(const void *p1, const void *p2)
{
    Proiect vp1 = *(Proiect *)p1;
    Proiect vp2 = *(Proiect *)p2;

    return vp1.zi_finala - vp2.zi_finala;
}

int main()
{
    Proiect pe[101], sol[101];
    int i, j, n, ult[101], bmax[101];


    FILE *fin, *fout;

    //citim datele de intrare din fisierul de intrare proiecte.in
    fin = fopen("proiecte.in", "r");

    fscanf(fin, "%d", &n);

    //consideram in mod artificial faptul ca inaintea primului proiect
    //exista un proiect care se termina in ziua 0
    pe[0].zi_finala = 0;
    for(i = 1; i <= n; i++)
    {
        pe[i].ID = i;
        fscanf(fin, "%d %d %d", &pe[i].zi_initiala, &pe[i].zi_finala,
                                                      &pe[i].bonus);
    }

    fclose(fin);

    //sortam proiectele in ordinea crescatoarea a zilei de terminare
    qsort(pe + 1, n, sizeof(Proiect), cmpProiecte);
```

```
    //calculam valorile elementelor tablourilor bmax si ult
    bmax[0] = 0;
    for (i = 1; i <= n; i++)
    {
        ult[i] = 0;
        for (j = i-1; j >= 1; j--)
            if (pe[j].zi_finala <= pe[i].zi_initiala)
            {
                ult[i] = j;
                break;
            }

        if(pe[i].bonus + bmax[ult[i]] > bmax[i-1])
            bmax[i] = pe[i].bonus + bmax[ult[i]];
        else
            bmax[i] = bmax[i-1];
    }

    //reconstituim o planificare optima in tabloul auxiliar sol
    i = n;
    j = 0;
    while(i >= 1)
        if(bmax[i] != bmax[i-1])
        {
            sol[j++] = pe[i];
            i = ult[i];
        }
        else
            i--;

    //scriem solutia in fisierul de iesire proiecte.out
    fout = fopen("proiecte.out", "w");


    for(i = j-1; i >= 0; i--)
        fprintf(fout, "Proiectul %d: %02d-%02d -> %5d RON\n",
           sol[i].ID, sol[i].zi_initiala, sol[i].zi_finala,
           sol[i].bonus);
    fprintf(fout, "\nBonusul maxim al echipei: %d RON\n", bmax[n]);

    fclose(fout);

    return 0;
}
```

Complexitatea algoritmului prezentat este $\mathcal{O}(n^2)$ și poate fi scăzută la $\mathcal{O}(n\log_2 n)$ dacă utilizăm o căutare binară modificată pentru a calcula valoarea $ult[i]$: https://www.geeksforgeeks.org/weighted-job-scheduling-log-n-time/.