TEHNICA DE PROGRAMARE "GREEDY"

1. Problema rucsacului (varianta continuă/fracționară)

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 completă în rucsac. Știind faptul că orice obiect poate fi încărcat și fracționat (doar o parte din el), să se determine o modalitate de încărcare a rucsacului astfel încât câștigul total obținut să fie maxim. Dacă un obiect este încărcat fracționat, atunci vom obține un câștig direct proporțional cu fracțiunea încărcată din el (de exemplu, dacă vom încărca doar o treime dintr-un obiect, atunci vom obține un câștig egal cu o treime din câștigul integral asociat obiectului respectiv).

În afara variantei continue/fracționare a problemei rucsacului, mai există și varianta discretă a sa, în care un obiecte poate fi încărcat doar complet. Varianta respectivă nu se poate rezolva corect utilizând metoda Greedy, ci există alte metode de rezolvare, pe care le vom prezenta în cursul dedicat metodei programării dinamice.

Se observă foarte ușor faptul că varianta fracționară a problemei rucsacului are întotdeauna soluție (evident, dacă $G > 0$ și $n \geq 1$), chiar și în cazul în care cel mai mic obiect are o greutate strict mai mare decât capacitatea G a rucsacului (deoarece putem să încărcăm și fracțiuni dintr-un obiect), în timp ce varianta discretă nu ar avea soluție în acest caz.

Deoarece dorim să găsim o rezolvare de tip Greedy pentru varianta fracționară a problemei rucsacului, vom încerca să încărcăm obiectele în rucsac folosind unul dintre următoarele criterii:

a) în ordinea descrescătoare a câștigurilor integrale (cele mai valoroase obiecte ar fi primele încărcate);
b) în ordinea crescătoare a greutăților (cele mai mici obiecte ar fi primele încărcate, deci am încărca un număr mare de obiecte în rucsac);
c) în ordinea descrescătoare a greutăților.

Analizând cele 3 criterii propuse mai sus, putem găsi ușor contraexemple care să dovedească faptul că nu vom obține o soluții optime. De exemplu, criteriul c) ar putea fi corect doar presupunând faptul că, întotdeauna, un obiect cu greutate mare are asociat și un câștig mare, ceea ce, evident, nu este adevărat! În cazul criteriului a), considerând $G = 10$ kg și 3 obiecte având câștigurile $(100, 90, 80)$ RON și greutățile $(10, 5, 5)$ kg, vom încărca în rucsac primul obiect (deoarece are cel mai mare câștig integral) și nu vom mai putea încărca niciun alt obiect, deci câștigul obținut va fi de 100 RON. Totuși, câștigul maxim de 170 RON se obține încărcând în rucsac ultimele două obiecte! În mod asemănător (de exemplu, modificând câștigurilor obiectelor anterior menționate în

$(100, 9, 8)$ RON) se poate găsi un contraexemplu care să arate faptul că nici criteriul b) nu permite obţinerea unei soluţii optime în orice caz.

Se poate observa faptul că primele două criterii nu conduc întotdeauna la soluţia optimă deoarece ele iau în considerare fie doar câştigurile obiectelor, fie doar greutăţile lor, deci criteriul corect de selecţie trebuie să le ia în considerare pe ambele. Intuitiv, pentru a obţine un câştig maxim, trebuie să încărcăm mai întâi în rucsac obiectele care sunt cele mai "eficiente", adică au un câştig mare şi o greutate mică. Această "eficienţă" se poate cuantifica prin intermediul *câştigului unitar* al unui obiect, adică prin raportul $u_i = c_i/g_i$.

Algoritmul Greedy pentru rezolvarea variantei fracţionare a problemei rucsacului este următorul:

- sortăm obiectele în ordinea descrescătoare a câştigurilor unitare;
- pentru fiecare obiect testăm dacă încape integral în spaţiul liber din rucsac, iar în caz afirmativ îl încărcăm complet în rucsac, altfel calculăm fracţiunea din el pe care trebuie să o încărcăm astfel încât să umplem complet rucsacul (după încărcarea oricărui obiect, actualizăm spaţiul liber din rucsac şi câştigul total);
- algoritmul se termină fie când am încărcat toate obiectele în rucsac (în cazul în care $g_1 + g_2 + \cdots + g_n \leq G$), fie când nu mai există spaţiu liber în rucsac.

De exemplu, să considerăm un rucsac în care putem să încărcăm maxim $G = 53$ kg şi $n = 7$ obiecte, având greutăţile $g = (10, 5, 18, 20, 8, 40, 20)$ kg şi câştigurile integrale $c = (30, 40, 36, 10, 16, 30, 20)$ RON. Câştigurile unitare ale celor 7 obiecte sunt $u = \left(\frac{30}{10}, \frac{40}{5}, \frac{36}{18}, \frac{10}{20}, \frac{16}{8}, \frac{30}{40}, \frac{20}{20}\right) = (3, 8, 2, 0.5, 2, 0.75, 1)$ RON/kg, deci sortând descrescător obiectele în funcţie de câştigul unitar vom obţine următoarea ordine a lor: $O_2, O_1, O_3, O_5, O_7, O_6, O_4$. Prin aplicarea algoritmului Greedy prezentat anterior asupra acestor date de intrare, vom obţine următoarele rezultate:

Obiectul curent	**Fracţiunea încărcată din obiectul curent**	**Spaţiul liber în rucsac**	**Câştigul total**
—	—	53	0
O_2: $c_2 = 40, g_2 = 5 \leq 53$	1	53 – 5 = 48	0 + 40 = 40
O_1: $c_1 = 30, g_1 = 10 \leq 48$	1	48 – 10 = 38	40 + 30 = 70
O_3: $c_3 = 36, g_3 = 18 \leq 38$	1	38 – 18 = 20	70 + 36 = 106
O_5: $c_5 = 16, g_5 = 8 \leq 20$	1	20 – 8 = 12	106 + 16 = 122
O_7: $c_7 = 20, g_7 = 20 > 12$	12/20 = 0.6	0	122 + 0.6*20 = 134

În concluzie, pentru a obţine un câştig maxim de 134 RON, trebuie să încărcăm integral în rucsac obiectele O_2, O_1, O_3, O_5 şi o fracţiune de $0.6 = \frac{3}{5}$ din obiectul O_7.

Înainte de a demonstra corectitudinea algoritmului prezentat, vom face următoarele observaţii:

- vom considera obiectele $O_1, O_2, \dots, O_n$ ca fiind sortate strict descrescător în funcţie de câştigurile lor unitare, respectiv $\frac{c_1}{g_1} > \frac{c_2}{g_2} > \cdots > \frac{c_n}{g_n}$ (acest lucru se obţine grupând obiectele care au acelaşi câştig unitar, ceea ce nu afectează corectitudinea algoritmului Greedy deoarece obiectele pot fi fracţionate şi vor fi încărcate în rucsac în aceeaşi ordine);
- o soluţie a problemei va fi reprezentată sub forma unui tuplu $X = (x_1, x_2, \dots, x_n)$, unde $x_i \in [0,1]$ reprezintă fracţiunea selectată din obiectul O_i;
- în toate formulele vom considera implicit indicii ca fiind cuprinşi între 1 şi n;
- câştigul asociat unei soluţii a problemei de forma $X = (x_1, x_2, \dots, x_n)$ îl vom nota cu $C(X) = \sum c_i x_i$;
- dacă $g_1 + g_2 + \cdots + g_n \leq G$, atunci soluţia vom obţine soluţia banală $X = (1, \dots, 1)$, care este evident optimă, deci vom considera faptul că $g_1 + g_2 + \cdots + g_n > G$, precum şi faptul că în orice soluţie optimă rucsacul va fi umplut complet (adică $\sum g_i x_i = G$).

Fie $X = (x_1, x_2, \dots, x_n)$ soluţia furnizată de algoritmul Greedy prezentat, deci rucsacul va fi umplut complet (i.e., $\sum g_k x_k = G$). Presupunem că soluţia X nu este optimă, deci există o altă soluţie optimă $Y = (y_1, y_2, \dots, y_n)$, diferită de soluţia X, obţinută folosind un alt algoritm (nu neapărat unul de tip Greedy). Deoarece Y este o soluţie optimă, obţinem imediat următoarele două relaţii: $\sum g_i y_i = G$ şi câştigul $C(Y) = \sum c_i y_i$ este maxim.

Deoarece $X \neq Y$, rezultă că există un cel mai mic indice i pentru care $x_i \neq y_i$. Mai mult, datorită faptului că algoritmul Greedy încarcă din fiecare obiect cea mai mare fracţiune posibilă, rezultă că $x_i > y_i$. Totuşi, soluţia Y este optimă, deci $C(Y) > C(X)$, ceea ce înseamnă că $\exists j > i$ astfel încât $x_j < y_j$ (altfel, respectiv dacă $x_k \geq y_k$ pentru orice indice $k > i$, am avea $C(X) > C(Y)$, ceea ce ar contrazice optimalitatea soluţiei Y).

Considerăm acum soluţia $Z = (z_1, z_2, \dots, z_i, \dots, z_j, \dots, z_n)$, obţinută din soluţia Y prin mutarea unei fracţiuni ε de la obiectul j la obiectul i (i.e., vom încărca în rucsac mai puţin din obiectul j şi mai mult din obiectul i, deoarece obiectul i are un câştig unitar strict mai mare decât cel al obiectului j) astfel încât rucsacul să rămână umplut complet:

- $z_k = y_k$ pentru orice $k \neq i, j$;
- $z_j = y_j - \varepsilon$;
- $z_i = y_i + \varepsilon \frac{g_j}{g_i}$.

Valoarea fracţiunii z_i se poate calcula astfel: dacă din obiectul j vom încărca o fracţiune mai mică cu ε (i.e., fracţiunea $y_j - \varepsilon$), atunci greutatea totală a obiectelor din rucsac va scădea cu $\varepsilon \cdot g_j$ unităţi de masă (e.g., kilograme), deci pentru a păstra rucsacul încărcat complet trebuie să luăm mai mult cu $\varepsilon \cdot g_j$ unităţi de masă din obiectul i, ceea ce înseamnă o fracţiune de $\frac{\varepsilon \cdot g_j}{g_i} = \varepsilon \frac{g_j}{g_i}$ din masa sa. Evident, valoarea fracţiunii ε trebuie să fie aleasă în mod convenabil, respectiv $0 < \varepsilon \leq y_j$ şi astfel încât $z_i \leq 1$!

În continuare, calculăm câștigul asociat soluției Z:

$$C(Z) = \sum z_k c_k = \sum_{k \neq i,j} y_k c_k + z_i c_i + z_j c_j = \sum_{k \neq i,j} y_k c_k + \left(y_i + \varepsilon \frac{g_j}{g_i}\right) c_i + (y_j - \varepsilon) c_j$$

$$= \underbrace{\sum_{k \neq i,j} y_k c_k + y_i c_i + y_j c_j}_{C(Y)} - \varepsilon c_j + \varepsilon \frac{g_j}{g_i} c_i = C(Y) + \underbrace{\varepsilon \left(\frac{c_i g_j - c_j g_i}{g_i}\right)}_{>0}$$

Expresia $\varepsilon \left(\frac{c_i g_j - c_j g_i}{g_i}\right)$ este strict pozitivă deoarece $\frac{c_i}{g_i} > \frac{c_j}{g_j} \Leftrightarrow c_i g_j - c_j g_i > 0$, iar ε și g_i sunt evident strict pozitive, deci rezultă că $C(Z) > C(Y)$, ceea ce contrazice optimalitatea soluției Y, așadar presupunerea că ar exista o soluție optimă Y diferită de soluția X furnizată de algoritmul Greedy este falsă.

În continuare, vom prezenta implementarea în limbajul Python a algoritmului Greedy pentru rezolvarea variantei fracționare a problemei rucsacului:

```
# functie folosita pentru sortarea descrescătoare a obiectelor
# în raport de câștigul unitar (cheia)
def cheieCâștigUnitar(ob):
    return ob[2] / ob[1]
# citim datele de intrare din fișierul text "rucsac.in"
fin = open("rucsac.in")
# de pe prima linie citim capacitatea G a rucsacului
G = float(fin.readline())
# fiecare dintre următoarele linii conține
# greutatea și câștigul unui obiect
obiecte = []
crt = 1
for linie in fin:
    aux = linie.split()
    # un obiect este un tuplu (ID, greutate, câștig)
    obiecte.append((crt, float(aux[0]), float(aux[1])))
    crt += 1
fin.close()

# sortăm obiectele descrescător în funcție de câștigul unitar
obiecte.sort(key=cheieCâștigUnitar, reverse=True)
# n reprezintă numărul de obiecte
n = len(obiecte)
# solutie este o listă care va conține fracțiunile încărcate
# din fiecare obiect
soluție = [0] * n
# inițial, spațiul liber din rucsac este chiar G
spațiu_liber_rucsac = G
# considerăm, pe rând, fiecare obiect
for i in range(n):
    # dacă obiectul curent încape complet în spațiul liber
```

```
    # din rucsac, atunci îl încărcăm complet
    if obiecte[i][1] <= spațiu_liber_rucsac:
        spațiu_liber_rucsac -= obiecte[i][1]
        soluție[i] = 1
    else:
        # dacă obiectul curent nu încape complet în spațiul liber
        # din rucsac, atunci calculăm fracțiunea din el necesară
        # pentru a încărca complet rucsacul și algoritmul se termină
        soluție[i] = spațiu_liber_rucsac / obiecte[i][1]
        break

# calculăm câștigul maxim
câștig = sum([soluție[i] * obiecte[i][2] for i in range(n)])

# scriem datele de ieșire în fișierul text "rucsac.out"
fout = open("rucsac.out", "w")
fout.write("Castig maxim: " + str(câștig) + "\n")
fout.write("\nObiectele incarcate:\n")
i = 0
while i < n and soluție[i] != 0:
    # trunchiem procentul încărcat din obiectul curent
    # la două zecimale
    procent = format(soluție[i]*100, '.2f')
    fout.write("Obiect "+str(obiecte[i][0])+": "+procent+"%\n")
    i = i + 1
fout.close()
```

Pentru exemplul de mai sus, fișierele text de intrare și de ieșire sunt următoarele:

rucsac.in	rucsac.out
53 10 30 5 40 18 36 20 10 8 16 40 30 20 20	Castig maxim: 134.0 Obiectele incarcate: Obiect 2: 100.00% Obiect 1: 100.00% Obiect 3: 100.00% Obiect 5: 100.00% Obiect 7: 60.00%

Citirea datelor de intrare are complexitatea $\mathcal{O}(n)$, sortarea are complexitatea $\mathcal{O}(n \log_2 n)$, selectarea și încărcarea obiectelor în rucsac are cel mult complexitatea $\mathcal{O}(n)$, iar afișarea câștigului maxim obținut $\mathcal{O}(1)$, deci complexitatea algoritmului este $\mathcal{O}(n \log_2 n)$.

2. Problema interclasării optime

Se consideră n şiruri de numere sortate crescător. Ştiind faptul că interclasarea a două şiruri de lungimi p şi q are complexitatea $\mathcal{O}(p+q)$, să se determine o modalitate de interclasare a celor n şiruri astfel încât complexitatea totală să fie minimă.

Exemplu:
Fie 4 şiruri sortate crescător având lungimile 30, 20, 10 şi 25. Interclasarea primelor două şiruri necesită 30+20=50 de operaţii elementare şi obţinem un nou şir de lungime 50, deci mai trebuie să interclasăm 3 şiruri cu lungimile 50, 10 şi 30. Interclasarea primelor două şiruri necesită 50+10=60 de operaţii elementare şi numărul total de operaţii elementare devine 50+60=110, după care obţinem un nou şir de lungime 60, deci mai trebuie să interclasăm două şiruri cu lungimile 60 şi 25. Interclasarea celor două şiruri necesită 60+25=85 de operaţii elementare, iar numărul total de operaţii elementare devine 110+85=195, după obţinem şirul final de lungime 85.

Numărul total minim de operaţii elementare se obţine interclasând de fiecare dată cele mai mici două şiruri din puncte de vedere al lungimilor (i.e., primele două minime). Astfel, pentru exemplul de mai sus, prima dată interclasăm şirurile cu lungimile 10 şi 20 (primele două minime) folosind 10+20=30 de operaţii elementare şi obţinem un nou şir de lungime 30, deci mai trebuie să interclasăm 3 şiruri cu lungimile 30, 25 şi 30. Interclasăm acum şirurile cu lungimile 25 şi 30 folosind 25+30=55 de operaţii elementare şi numărul total de operaţii elementare devine 30+55=85, după care obţinem un nou şir de lungime 55, deci mai trebuie să interclasăm două şiruri cu lungimile 30 şi 55. Interclasarea celor două şiruri necesită 30+55=85 de operaţii elementare, iar numărul total de operaţii elementare devine 85+85=170 şi obţinem şirul final de lungime 85.

Pentru a implementa algoritmul optim prezentat vom utiliza o structură de date numită *coadă cu priorităţi* (*priority queue*), deoarece aceasta permite realizarea operaţiilor de inserare în coadă în funcţie de prioritate şi, respectiv, de extragere a valorii cu prioritate minimă cu complexitatea $\mathcal{O}(\log_2 n)$, unde n reprezintă numărul de elemente din coada cu priorităţi. Elementele unei cozi cu priorităţi sunt, de obicei, tupluri de forma (*prioritate*, *valoare*), dar pot şi valori simple, caz în care valoarea este considerată şi prioritate.

În limbajul Python, o structură de date de tip coadă cu priorităţi este implementată în clasa `PriorityQueue` din pachetul `queue`. Principalele metode ale acestei clase sunt:

- `PriorityQueue()` – creează o coadă cu priorităţi cu lungimea maximă nelimitată;
- `PriorityQueue(max)` – creează o coadă cu priorităţi cu lungimea maximă `max` ;
- `put(elem)` – inserează în coadă elementul `elem` în funcţie de prioritatea sa;
- `get()` – extrage din coadă unul dintre elementele cu prioritate minimă;
- `qsize()` – furnizează numărul de elemente existente în coada cu priorităţi;
- `empty()` – furnizează `True` dacă respectiva coadă este vidă sau `False` în caz contrar;
- `full()` – furnizează `True` dacă respectiva coadă este plină sau `False` în caz contrar.

Exemplu:

```
import queue

# se creeaza o coada cu priorități de lungime "infinită"
pq = queue.PriorityQueue()

# inseram elemente în coada cu priorități
pq.put(7)
pq.put(10)
pq.put(3)
pq.put(6)
pq.put(7)
pq.put(10)

# afisam elementele cozii prin extragerea repetată a valorii minime,
# deci în ordine crescătoare: 3 6 7 7 10 10
while not pq.empty():
    print(pq.get(), end=" ")
```

Atenție, o coadă cu priorități nu este o structură de date iterabilă, deoarece ea este implementată, de obicei, folosind *arbori binari de tip heap* (https://en.wikipedia.org/wiki/Binary_heap)! Un arbore binar de tip heap este un arbore binar complet cu proprietatea că valoarea din orice nod neterminal este mai mică decât valorile fiilor săi, deci valoarea din rădăcina sa va fi minimul tuturor valorilor din arbore, iar extragerea valorii minime dintr-un arbore binar de tip heap presupune ștergerea rădăcinii sale și refacerea structurii sale de heap.

Pentru a rezolva problema prezentată, folosind metoda Greedy, vom utiliza o coadă cu priorități având elemente de forma (*lungime șir*, *șir*), deci prioritatea unui șir va fi dată de lungimea sa. La fiecare pas, vom extrage din coadă două șiruri având lungimile minime, le vom interclasa, după care vom introduce în coadă șirul obținut prin interclasare. Vom repeta acest procedeu până când coada va avea un singur element, respectiv șirul rezultat prin interclasarea tuturor șirurilor date.

```
import queue

# Funcție pentru interclasarea a două șiruri sortate crescător
def interclasare(a, b):
    i = j = 0

    rez = []
    while i < len(a) and j < len(b):
        if a[i] <= b[j]:
            rez.append(a[i])
            i += 1
        else:
            rez.append(b[j])
            j += 1
```

```
    rez.extend(a[i:])
    rez.extend(b[j:])

    return rez


f = open("siruri.txt")

# fiecare linie din fișierul text de intrare siruri.txt
# conține câte un șir ordonat crescător
siruri = queue.PriorityQueue()

for linie in f:
    aux = [int(x) for x in linie.split()]
    siruri.put((len(aux), aux))

f.close()

# t = numărul total de operații elementare efectuate
t = 0
while siruri.qsize() != 1:
    # extragem primele două șiruri a și b cu lungimi minime
    a = siruri.get()
    b = siruri.get()
    # interclasăm șirurile a și b
    r = interclasare(a[1], b[1])
    # actualizăm numărul total de operații elementare
    t = t + len(r)
    # introducem în coada cu priorități șirul r rezultat
    # prin interclasarea șirurilor a și b
    siruri.put((len(r), r))

# afișăm rezultatele
print("Numar minim de operații elementare:", t)
r = siruri.get()[1]
print("Șirul obținut prin interclasarea tuturor șirurilor:", *r)
```

Estimarea complexității acestui algoritm este dificilă, deoarece complexitatea operației de interclasare a celor două șiruri cu lungimile minime depinde de lungimile celor două șiruri, deci nu poate calculată doar în funcție de dimensiunea datelor de intrare, respectiv numărul n de șiruri.

3. Planificarea unor proiecte cu profit maxim

Se consideră n proiecte, pentru fiecare proiect cunoscându-se profitul, un termen limită (sub forma unei zi din lună) şi faptul că implementarea sa durează exact o zi. Să se găsească o modalitate de planificare a unor proiecte astfel încât profitul total să fie maxim.

Exemplu*:*

proiecte.in	proiecte.out
BlackFace 2 800 Test2Test 5 700 Java4All 1 150 BestJob 2 900 NiceTry 1 850 JustDoIt 3 1000 FileSeeker 3 950 OzWizard 2 900	Ziua 1: BestJob 900.0 Ziua 2: FileSeeker 950.0 Ziua 3: JustDoIt 1000.0 Ziua 5: Test2Test 700.0 Profit maxim: 3550.0

Rezolvare:
În primul rând, vom observa faptul că dacă termenul limită al unui proiect este strict mai mare decât numărul n de proiecte, atunci putem să-l înlocuim chiar cu n. Aplicând această observaţie, putem afirma că soluţia prezentată în seminarul anterior, având complexitatea $\mathcal{O}(n \cdot zi\_max)$, poate fi uşor modificată într-una cu complexitatea $\mathcal{O}(n^2)$!

Rezolvarea optimă a acestei probleme (i.e., cu complexitatea $\mathcal{O}(n \log_2 n)$) constă în următorii paşi:
- fiecare termen limită strict mai mare decât numărul n de proiecte îl vom înlocui cu n;
- sortăm proiectele descrescător după termenul limită;
- pentru fiecare zi $zcrt$ de la n la 1 procedăm în următorul mod:
  - introducem într-o coadă cu priorităţi toate proiectele care au termenul limită $zcrt$, considerând prioritatea unui proiect ca fiind dată de profitul său;
  - extragem proiectul cu profit maxim şi îl planificăm în ziua $zcrt$.

Pentru exemplul dat, vom obţine o planificare optimă a proiectelor astfel:

Proiectele (sortate descrescător după termenul limită)	Ziua curentă (zcrt)	Coada cu priorităţi	Planificarea optimă
Test2Test 5 700 JustDoIt 3 1000 FileSeeker 3 950 BlackFace 2 800 BestJob 2 900 OzWizard 2 900 Java4All 1 150 NiceTry 1 850	5	(700, Test2Test, 5)	Ziua 5: Test2Test
	4	–	Ziua 5: Test2Test Ziua 4: –
	3	(1000, JustDoIt, 3) (950, FileSeeker, 3)	Ziua 5: Test2Test Ziua 4: – Ziua 3: JustDoIt
	2	(950, FileSeeker, 3) (900, BestJob, 2) (900, OzWizard, 2) (800, BlackFace, 2)	Ziua 5: Test2Test Ziua 4: – Ziua 3: JustDoIt Ziua 2: FileSeeker

	1	(900, BestJob, 2) (900, OzWizard, 2) (850, NiceTry, 1) (800, BlackFace, 2) (150, Java4All, 1)	Ziua 5: Test2Test Ziua 4: – Ziua 3: JustDoIt Ziua 2: FileSeeker Ziua 1: BestJob

Deoarece clasa `PriorityQueue` din pachetul `queue` implementează o coadă cu priorități care permite extragerea minimului, vom considera prioritatea unui proiect ca fiind egală cu `–profit`.

În continuare, prezentăm implementarea în limbajul Python a algoritmului Greedy prezentat mai sus:

```
import queue

# funcție care furnizează cheia necesară sortării
# proiectelor descrescător după termenul limită
def cheieTermenLimitaProiect(p):
    return p[1]


fin = open("proiecte.in")
# citim toate liniile din fișier pentru a afla simplu
# numărul n de proiecte
toate_liniile = fin.readlines()
fin.close()

n = len(toate_liniile)
# lsp = lista cu toate proiectele
lsp = []
for linie in toate_liniile:
    aux = linie.split()
    # un proiect va fi un tuplu (-profit, termen limită, denumire)
    # pentru a-l putea insera direct într-o coadă de priorități,
    # iar cheia este -profitul deoarece clasa PriorityQueue
    # implementează o coadă care permite doar extragerea minimului
    lsp.append((-float(aux[2]), min(int(aux[1]), n), aux[0]))

fin.close()

# sortăm proiectele descrescător după termenul limită
lsp.sort(key=cheieTermenLimitaProiect, reverse=True)

# planificarea optimă o vom memora într-un dicționar
# cu intrări de forma zi:proiect
planificare = {k: None for k in range(1, n + 1)}

# coada cu priorități (prioritatea = -profit)
coada = queue.PriorityQueue()
```

```
k = 0
profitmax = 0
for zcrt in range(n, 0, -1):
    # încărcăm în coadă toate proiectele care au
    # termenul limită egal cu zcrt
    while k <= n-1 and lsp[k][1] == zcrt:
        coada.put(lsp[k])
        k += 1

    # extragem din coadă proiectul cu profit maxim și
    # îl planificăm în ziua zcrt
    if coada.qsize() > 0:
        planificare[zcrt] = coada.get()
        profitmax += abs(planificare[zcrt][0])

# scriem o planificare optimă în fișierul text proiecte.out
fout = open("proiecte.out", "w")

for z in planificare:
    if planificare[z] != None:
        fout.write("Ziua " + str(z) + ": " + planificare[z][2] + " "
                   + str(abs(planificare[z][0])) + "\n")
fout.write("\nProfit maxim: " + str(profitmax))

fout.close()
```

Așa cum deja am menționat, complexitatea acestui algoritm este $\mathcal{O}(n \log_2 n)$.