Laborator 02 : Greedy

Responsabili:

- Darius-Florentin Neațu (2017-2021) [mailto:neatudarius@gmail.com]
- Radu Nichita (2021) [mailto:radunichita99@gmail.com]
- Cristian Olaru (2021) [mailto:cristianolaru99@gmail.com]
- Miruna-Elena Banu (2021) [mailto:mirunaelena.banu@gmail.com]
- Mara-Ioana Nicolae (2021) [mailto:maraioana9967@gmail.com]
- Ștefan Popa (2018-2020) [mailto:stefanpopa2209@gmail.com]

Autori:

- Alex Rotaru (2018) [mailto:rotarualexandruandrei94@gmail.com]
- Darius-Florentin Neațu (2018) [mailto:neatudarius@gmail.com]
- Radu Vișan (2018) [mailto:visanr95@gmail.com]
- Cristian Banu (2018) [mailto:cristb@gmail.com]

Obiective laborator

- Înțelegerea noțiunilor de bază legate de tehnica greedy
- Însușirea abilităților de implementare a algoritmilor bazați pe greedy

Precizări inițiale

Toate exemplele de cod se găsesc pe pagina pa-lab::demo/lab02 [https://github.com/acs-pa/pa-lab/tree/main/demo/lab02].

Exemplele de cod apar încorporate și în textul laboratorului pentru a facilita parcurgerea cursivă a acestuia. **ATENȚIE!** Varianta actualizată a acestor exemple se găsește întotdeauna pe GitHub.

- Toate bucățile de cod prezentate în partea introductivă a laboratorului (înainte de exerciții) au fost testate. Cu toate acestea, este posibil ca din cauza mai multor factori (formatare, caractere invizibile puse de browser etc.) un simplu copy-paste să nu fie de ajuns pentru a compila codul.
- Vă rugăm să compilați **DOAR** codul de pe GitHub. Pentru raportarea problemelor, contactați unul dintre maintaineri.
- Pentru orice problemă legată de conținutul acestei pagini, vă rugăm să dați e-mail unuia dintre responsabili.

Importanță - aplicații practice

În general tehnicile de tip greedy sau programare dinamică (următoarele laboratoare) sunt folosite pentru rezolvarea problemelor de optimizare. Acestea pot adresa probleme în sine sau pot fi subprobleme dintr-un algoritm mai mare. De exemplu, algoritmul Dijkstra pentru determinarea drumului minim pe un graf alege la fiecare pas un nod nou urmărind algoritmul greedy.

Există însă probleme care ne pot induce în eroare. Astfel, există probleme în care urmărind criteriul greedy

nu ajungem la soluția optimă. Este foarte important să identificăm cazurile când se poate aplica greedy și cazurile când este nevoie de altceva. Alteori această soluție neoptimă este o aproximare suficientă pentru ce avem nevoie. Problemele NP-complete necesita multă putere de calcul pentru a găsi optimul absolut. Pentru a optimiza aceste calcule mulți algoritmi folosesc decizii greedy și găsesc un optim foarte aproape de cel absolut.

Greedy

“greedy” = “lacom”. Algoritmii de tip greedy vor să construiască într-un mod cât mai rapid soluția unei probleme. Ei se caracterizează prin luarea unor decizii rapide care duc la găsirea unei potențiale soluții a problemei. Nu întotdeauna asemenea decizii rapide duc la o soluție optimă; astfel ne vom concentra atenția pe identificarea acelor anumite tipuri de probleme pentru care se pot obține soluții optime. În general există mai multe soluții posibile ale problemei. Dintre acestea se pot selecta doar anumite soluții optime, conform unor anumite criterii. Algoritmii greedy se numără printre cei mai direcți algoritmi posibili. Ideea de bază este simplă: având o problemă de optimizare, de calcul al unui cost minim sau maxim, se va alege la fiecare pas decizia cea mai favorabilă, fără a evalua global eficiența soluţiei. Scopul este de a găsi una dintre acestea sau dacă nu este posibil, atunci o soluție cât mai apropiată, conform criteriului optimal impus.

Trebuie înțeles faptul că rezultatul obținut este optim doar dacă un optim local conduce la un optim global. În cazul în care deciziile de la un pas influențează lista de decizii de la pasul următor, este posibilă obținerea unei valori neoptimale. În astfel de cazuri, pentru găsirea unui optim absolut se ajunge la soluții supra-polinomiale. De aceea, dacă se optează pentru o astfel de soluție, algoritmul trebuie însoțit de o demonstrație de corectitudine. Descrierea formală a unui algoritm greedy este următoarea:

```
// C este mulțimea candidaților
function greedy(C) {
    S ← ∅ // în S construim soluția

    while !solutie(C) and C ≠ ∅
        x ← un element din C care minimizează/maximizează select(x)
        C ← C \ {x}
        if fezabil( S ∪ {x}) then S ← S∪{x}

    return S
}
```

Este ușor de înțeles acum de ce acest algoritm se numește ”greedy”: la fiecare pas se alege cel mai bun candidat de la momentul respectiv, fără a studia alternativele disponibile în momentul respectiv şi viabilitatea acestora în timp.

Dacă un candidat este inclus în soluție, rămâne acolo, fără a putea fi modificat, iar dacă este exclus din soluție, nu va mai putea fi niciodată selectat drept un potențial candidat.

Exemple

Simple task

Enunț

Fie un șir de N numere pentru care se cere determinarea unui subșir de numere cu suma maximă. Un subșir al unui șir este format din elemente (nu neapărat consecutive) ale șirului respectiv, în ordinea în care acestea apar în șir.

Pentru numerele $1, -5, 6, 2, -2, 4$ răspunsul este $1, 6, 2, 4$ (suma 13).

Soluție

Se observă că tot ce avem de făcut este să verificăm fiecare număr dacă este pozitiv sau nu. În cazul pozitiv, îl introducem în subșirul soluție.

Dacă toate numerele sunt negative, soluția este dată de cel mai mare numar negativ (cel mai mic în modul).

Problema spectacolelor

Enunț

Se dau mai multe spectacole, prin timpii de start și timpii de final. Se cere o planificare astfel încât o persoană să poată vedea cât mai multe spectacole.

Soluție

Rezolvarea constă în **sortarea** spectacolelor crescător după timpii de final, apoi la fiecare pas se alege **primul spectacol** care are timpul de start mai mare decât ultimul timp de final. Timpul inițial de final este inițializat la $-\inf$ (spectacolul care se termină cel mai devreme va fi mereu selectat, având timpul de start mai mare decât timpul inițial).

Implementare

```
bool end_hour_comp (pair<int, int>& e1, pair<int, int>& e2) {
    // comparam doar dupa ora de sfarsit
    return (e1.second < e2.second);
}

vector<pair<int, int>> plan(vector<pair<int, int> >& intervals) {
    vector<pair<int, int>> plan;
    // se sorteaza intervalele pe baza orei de sfarsit a spectacolelor
    sort(intervals.begin(), intervals.end(), end_hour_comp);

    // se ia ultimul spectacol ca terminat la -oo pt a putea incepe cu
    // cel mai devreme
    int last_end = INT_MIN; // -oo a.k.a -infinit
    for (auto interval : intervals) {
        // daca inceputul intervalului curent este dupa sfarsitul ultimului
        // spectacol (last_end) il adaugam in lista de spectacole la care
        // se participa
        if (interval.first >= last_end)
        {
            plan.push_back(interval);
            // dupa ce am adaugat un spectacol, updatam ultimul sfarsit de spectacol
            last_end = interval.second;
        }
    }
    return plan;
}
```

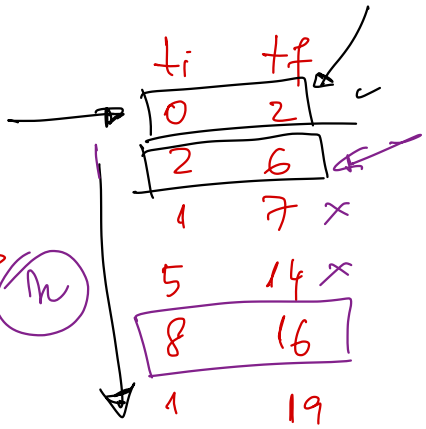
Complexitate

Soluția va avea următoarele complexități:

$n = 6$

	t_i	t_f	
1)	1	19	
2)	0	2	(1)
3)	1	7	(3)
4)	2	6	(2)
5)	5	14	(4)
6)	8	16	

Sort → n



3

(0, 2)
(2, 6)
(8, 16)

Timp: $\underbrace{O(n \log n)}_{\text{sortare}} + \underbrace{O(n)}_{\text{parcurgere}} = O(n + n \log n) = \boxed{O(n \log n)}$

- **complexitate temporală** : $T(n) = O(n * log(n))$
  - explicație
    - sortarea are $O(n * log(n))$
    - facem încă o parcurgere în $O(n)$
- **complexitate spațială** : depinde de algoritmul de sortare folosit.

Problema florarului

Enunț

Se dă un grup de k oameni care vor să cumpere împreună n flori. Fiecare floare are un preț de bază, însă prețul cu care este cumpărată variază în funcție de numărul de flori cumpărate anterior de persoana respectivă. De exemplu dacă George a cumparat 3 flori (diferite) și vrea să cumpere o floare cu prețul 2, el va plăti $(3 + 1) * 2 = 8$. Practic el va plăti un preț proporțional cu numărul de flori cumpărate până atunci tot de el.

Cerința: Se cere pentru un număr k de oameni și n flori să se determine care este costul minim cu care grupul poate să achiziționeze toate cele n flori o singură dată.

Observație: Un tip de floare se cumpără o singură dată. O persoană poate cumpăra mai multe tipuri de flori. În final în grup va exista un singur exemplar din fiecare tip de floare.

Formal avem k număr de oameni, n număr de flori, $c[i]$ = pretul florii de tip i, costul de cumpărare i va fi $(x + 1) * c[i]$, unde x este numărul de flori cumpărate anterior de persoana respectivă.

n=3 k=3 c=[2 5 6]
Cost minim = 13

Explicație: Fiecare individ cumpără câte o floare, deci acestea se cumpăra la prețul nominal.

Soluție

Se observă că prețul efectiv de cumpărare va fi mai mare cu cât cumpărăm acea floare mai tarziu. Dacă considerăm cazul în care avem o singură persoană în grup observăm că are sens să cumpărăm obiectele în ordine descrescătoare (deoarece vrem să minimizăm costul fiecărui tip de floare, acesta crește cu cât cumpărăm floarea mai tarziu).

De aici, gândindu-ne la versiunea cu k persoane, observăm că ar fi mai ieftin dacă am repartiza următoarea cea mai scumpa floare la alt individ. Deci împărțim florile sortate descrescător după pret în grupuri de câte k, fiecare individ luând o floare din acest grup și ne asigurăm că prețul va creste doar in funcție de numărul de grupuri anterioare.

Implementare

```
struct greater_comparator
{
    template<class T>
    bool operator()(T const &a, T const &b) const { return a > b; }
};

int minimum_cost(int k, vector<int>& costs) {
    // sortam vectorul de preturi in ordine descrescatoare
```

k=3 n = 5

c = [2 5 4 3 7]

1 : 5 14

2 : 3 8

3 : 2

S = 32

c = [7 5 4 | 3 2]

1	2	3
7	5	4
3	2	

→ 7 + 5 + 4

→ 6 + 4

S = 26

(i % K == 0)

sort → c = [20 15 13 | 0 7 4 | 3 2]

k k ≤ k

$S_1 = (20 + 15 + 13) \cdot 1$ $S_2 = 2 \times 21$ $S_3 = 3 \times 6$

```
    sort(costs.begin(), costs.end(), greater_comparator());

    // numarul de flori cumparate de fiecare individ din grup la un moment dat
    int x = 0;
    // o varianta mai putin eficienta spatial ar fi fost sa retinem pt fiecare
    // individ din grup numarul de flori cumparate intr-un hashmap
    // costul total
    int total_cost = 0;

    // parcurgem fiecare pret de floare si o "asignam" unui individ din grup
    // pretul acesteia fiind proportional cu numarul de achizitii facut pana acum
    // de acesta (x)
    for (int idx = 0; idx < costs.size(); idx++) {
        int customer_idx = idx % k;
        total_cost += (x + 1) * costs[idx];
        // in momentul in care ultimul individ a cumparat o floare din grupul curent
        // incrementam numarul de flori achizitionate de fiecare
        if (customer_idx == k - 1) {
            x += 1;
        }
    }
    return total_cost;
}
```

Complexitate

Soluția va avea următoarele complexități:

- **complexitate temporală** : $T(n) = O(n * log(n))$
  - explicație
    - sortarea are $O(n * log(n))$
    - facem încă o parcurgere în $O(n)$
- **complexitate spațială** : depinde de algoritmul de sortare folosit. Fără partea de sortare, spațiul este constant (nu se ia în considerare vectorul de elemente).

Problema cuielor

Enunț

Fie N scânduri de lemn, descrise ca niște **intervale închise** cu capete reale. Găsiți o mulțime **minimă** de cuie astfel încât fiecare scândură să fie bătută de cel puțin un cui. Se cere poziția cuielor.

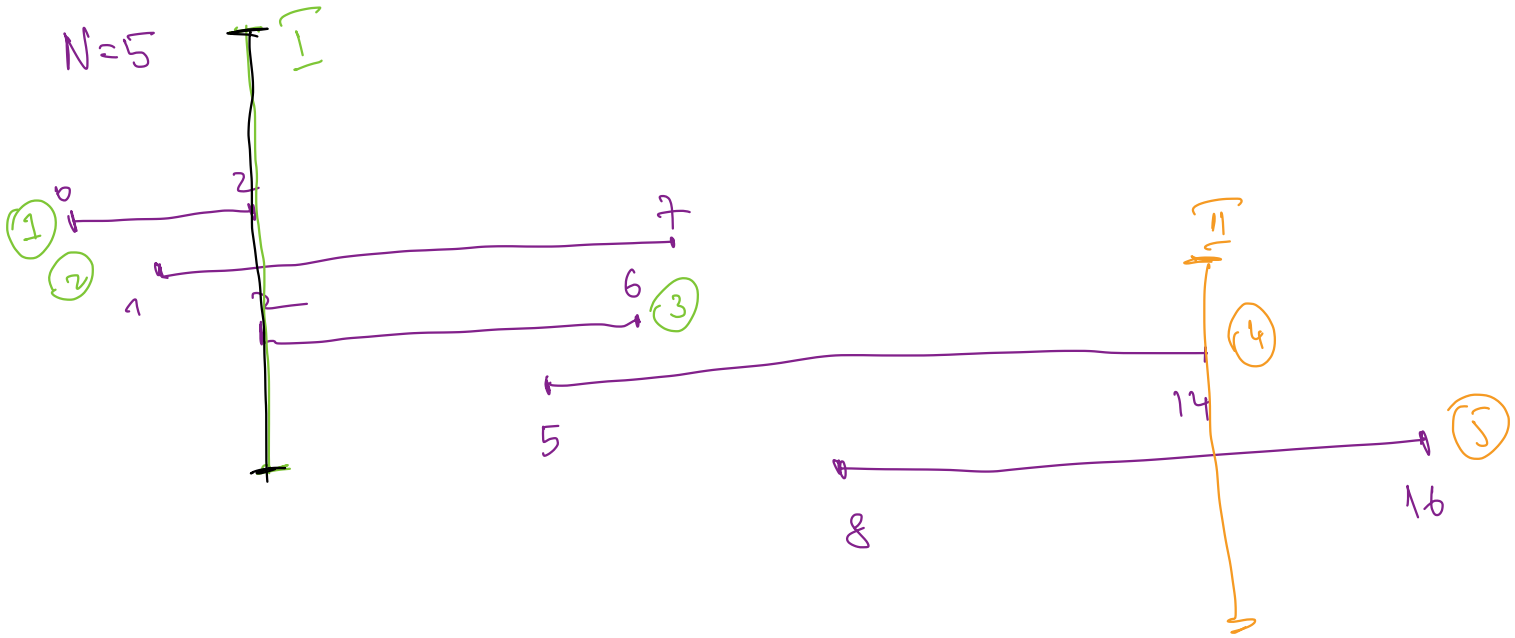
Formulat matematic: găsiți o mulțime de puncte de cardinal **minim** M astfel încât pentru orice interval [a_i, b_i] din cele N, să existe un punct x din M care să aparțină intervalului [a_i, b_i].

- intrare: N = 5, intervalele: [0, 2], [1, 7], [2, 6], [5, 14], [8, 16]
- ieșire: M = {2, 14}
- explicație: punctul 2 se află în primele 3 intervale, iar punctul 14 în ultimele 2

Soluție

Se observă că dacă x este un punct din M care nu este capăt dreapta al niciunui interval, o translație a lui x la dreapta care îl duce în capătul dreapta cel mai apropiat nu va schimba intervalele care conțin punctul. Prin urmare, există o mulțime de cardinal minim M pentru care toate punctele x sunt capete dreapta.

Astfel, vom crea mulțimea M folosind numai capete dreapta în felul următor:



2 cuie

→ 1)	0	(2)
2)	2	6 ✓
3)	1	7 ✓
4)	5	(14)
5)	8	16 ✓

- sortăm intervalele dupa capătul dreapta
- iterăm prin fiecare interval și dacă intervalul curent nu conține ultimul punct introdus în mulțime atunci îl adăugam pe acesta la mulțime

Implementare

```
bool point_in_interval(const pair<int, int>&interval, int point) {
    return point >= interval.first && point <= interval.second;
}

bool right_edge_comparator (pair<int, int>& e1, pair<int, int>& e2) {
    // comparam scandurile dupa capatul drepata
    return (e1.second < e2.second);
}

vector<int> cover_intervals_greedy(vector<pair<int, int>>& intervals) {
    vector<int> nails; // pozitiile cuielor, a.k.a multimea M
    // ultimul punct inserat
    int last_point = INT_MIN;

    //sortam invervalele dupa capatul drepata
    sort(intervals.begin(), intervals.end(), right_edge_comparator);

    for (auto interval : intervals) {
        // daca intervalul nu contine ultimul punct adaugat
        if (!point_in_interval(interval, last_point)) {
            // il adaugam in multimea M
            nails.push_back(interval.second);
            // updatam ultimul punct inserat
            last_point = interval.second;
        }
    }

    return nails;
}
```

Complexitate

Soluția va avea următoarele complexități:

- **complexitate temporală** : $T(n) = O(n * log(n))$
    - explicație
        - sortare: $O(n * logn)$
        - parcurgerea intervalelor: $O(n)$
- **complexitate spațială** : depinde de algoritmul de sortare folosit.

Concluzii şi observații

Aspectul cel mai important de reținut este că soluțiile găsite trebuie să reprezinte optimul global, ci nu doar local. Se pot confunda ușor problemele care se rezolvă cu greedy cu cele care se rezolvă prin programare dinamică (vom vedea săptămâna viitoare).

Exercitii

Scheletul de laborator se găsește pe pagina pa-lab::skel/lab02 [https://github.com/acs-pa/pa-

lab/tree/main/skel/lab02].

Rucsac

Fie un set cu n obiecte (care pot fi **tăiate** - varianta continuă a problemei). Fiecare obiect i are asociată o pereche (w_i, p_i) cu semnificația:

- $w_i = weight_i$ = greutatea obiectului cu numărul i
- $p_i = price_i$ = prețul obiectului cu numărul i
  - $w_i >= 0$ si $p_i > 0$

Gigel are la dispoziție un rucsac de **volum infinit**, dar care suportă o **greutate maximă** (notată cu W - weight knapsack).

El vrea să găsească **o submulțime de obiecte** (nu neapărat întregi) pe care să le bage în rucsac, astfel încât suma profiturilor să fie **maximă**.

Dacă Gigel bagă în rucsac obiectul i, caracterizat de (w_i, p_i), atunci profitul adus de obiect este p_i (presupunem că îl vinde cu cât valorează).

În această variantă a problemei, Gigel poate tăia oricare dintre obiecte, obținând o proporție din acesta. Dacă Gigel alege doar x din greutatea w_i a obiectului i, atunci el câștigă doar $\frac{x}{w_i} * p_i$.

Task-uri:

- Să se determine profitul maxim pentru Gigel.
- Care este complexitatea soluției (timp + spatiu)? De ce?

obiecte:

index	0	1	2
greutate	60	100	120
valoare	10	20	30

greutate = 50

Output: 12.5 Explicație: avem 50 capacitate și toate obiectele au o greutatate mai mare, decidem să luăm cât putem din produsul cu raportul valoare / greutate cel mai mare. profit = 30 / 120 * 50 = 12.5

obiecte:

index	0	1	2
greutate	20	50	30
valoare	60	100	120

greutate = 50

Output: 180 Explicație: Sortăm obiectele după raportul valoare / profit si avem în ordine: {30, 120}, {20, 60}, {50, 100}. Introducem obiecte până când umplem sacul ⇒ intră primele 2 obiecte. Calculăm profitul 120 + 60 = 180

Distanțe

Considerăm 2 localități A și B aflate la distanța D. Între cele 2 localități avem un număr de n benzinării, date prin distanța față de localitatea A. Mașina cu care se efectuează deplasarea între cele 2 localități poate parcurge maxim m kilometri având rezervorul plin la început. Se dorește parcurgerea drumului cu un număr minim de opriri la benzinării pentru realimentare (după fiecare oprire la o benzinărie, mașina pleacă cu rezervorul plin).

Distanțele către benzinării se reprezintă printr-o listă de forma $0 < d1 < d2 <. . . < dn$ unde di ($1 <= i <= n$) reprezintă distanța de la A la benzinăria i. Pentru simplitate, se consideră că localitatea A se află la 0, iar $dn = D$ (localitatea B se află in același loc cu ultima benzinărie).

Se garantează că există o planificare validă a opririlor astfel încât să se poată ajunge la localitatea B.

$n = 5$

$m = 10$

$d = (2, 8, 15, 25, 30)$

Răspunsul este 3, efectuând 3 opriri la a 2-a, a 3-a, respectiv a 4-a benzinărie.

Teme la ACS

Pe parcursul unui semestru, un student are de rezolvat n teme (nimic nou până aici...). Se cunosc enunțurile tuturor celor n teme de la **începutul semestrului**.

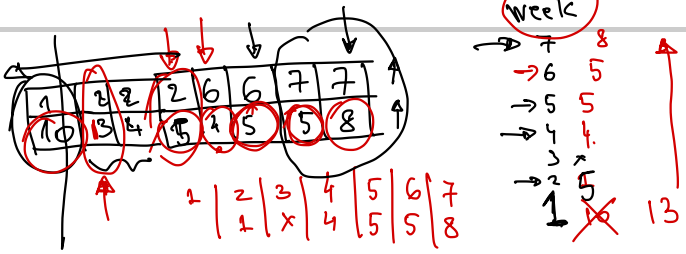
Timpul de rezolvare pentru oricare dintre teme este de **o săptămână** și **nu** se poate lucra la mai multe teme în același timp. Pentru fiecare temă se cunoaște un termen limită $d[i]$ (exprimat în săptămâni - deadline pentru tema i) și un punctaj $p[i]$.

Nicio fracțiune din punctaj nu se mai poate obține după expirarea termenului limită.

Task-uri:

- Să se definească o planificare de realizare a temelor, în așa fel încât punctajul obținut să fie **maxim**.
- Care este complexitatea soluției (timp + spațiu)? De ce?

index	0	1	2	3	4
deadline	6	6	2	7	7
punctaj	5	4	1	5	8



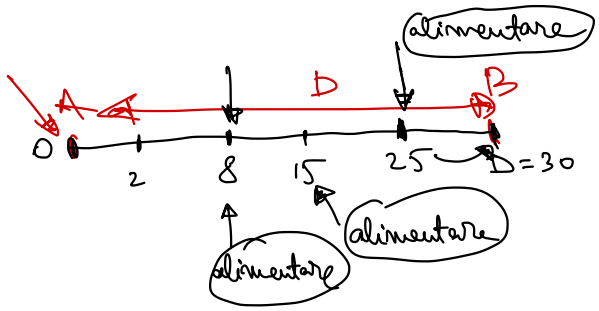
Output: $1 + 4 + 5 + 5 + 8 = 23$

Explicație: Putem face toate temele deoarece până ajungem la deadline-urile lor avem suficiente unități de timp.

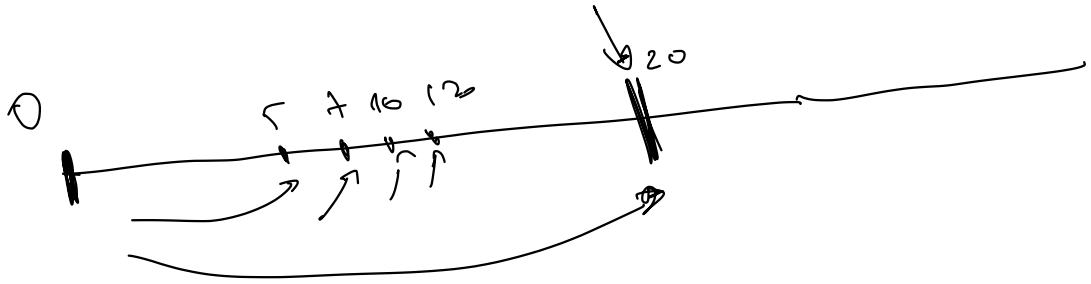
index	0	1	2	3	4	5	6	7
deadline	3	3	3	3	9	11	11	11
punctaj	4	5	6	9	10	2	4	6

Output: $5 + 6 + 9 + 10 + 2 + 4 + 6 = 42$

Explicație: Până în deadline 3 avem la dispoziție 3 unități de timp și 4 teme. Deci sortăm după punctaj și le includem pe cele mai valoroase: 5, 6, 9. Până la deadline 9 avem la dispoziție 6 unități de timp și 4 teme. Le includem pe toate.



$m = 10\,km$



$m = 20$

BONUS

Rezolvați problema Dishonest Sellers [http://codeforces.com/problemset/problem/779/C].

Hint: aici [http://codeforces.com/blog/entry/50724].

Extra

Incercați problema MaxSum [https://www.hackerrank.com/contests/test-practic-pa-2017-v1-plumbus/challenges/1-1-usoare] de la test PA 2017.
Problema 1 de la tema PA 2017. Puteți descărca enunțul și checkerul de aici [https://ocw.cs.pub.ro/courses/_media/pa/teme/pa2017_tema1.zip].
Problema 3 de la tema PA 2017. Puteți descărca enunțul și checkerul de aici [https://ocw.cs.pub.ro/courses/_media/pa/teme/pa2017_tema1.zip].
Puteți rezolva această problemă pe leetcode [https://leetcode.com/problems/two-city-scheduling/]

Referințe

[0] Chapter **Greedy Algorithms**, “Introduction to Algorithms”, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein

pa/laboratoare/laborator-02.txt · Last modified: 2021/03/09 09:47 by radu.nichita