Laborator 3: Programare Dinamică (1/2)

Obiective laborator

- înțelegerea noțiunilor de bază despre programare dinamică
- însușirea abilităților de implementare a algoritmilor bazați programare dinamică.

Precizări inițiale

Toate exemplele de cod se găsesc pe pagina pa-lab::demo/lab03 [https://github.com/acs-pa/pa-lab/tree/main/demo/lab03].

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 rugam să compilați **DOAR** codul de pe GitHub. Pentru raportarea problemelor, contactați unul dintre maintaineri.
- Pentru orice problemă legată de conținutul acestei pagini, vă rugam să dați e-mail unuia dintre responsabili.

Ce este DP?

Similar cu greedy, tehnica de programare dinamică este folosită în general pentru rezolvarea **problemelor de optimizare**. În continuare vom folosi acronimul **DP (dynamic programming)**.

De asemenea, DP se poate folosi și pentru probleme în care nu căutăm un optim, cum ar fi **problemele de numărare** (vom exemplifica în lab04).

Aplicatii DP

Programarea dinamică are un **câmp larg de aplicare**, însă la PA ne vom rezuma la câteva aplicații care vor fi menționate pe parcursul laboratoarelor 3 și 4. De asemenea, această tehnică va fi folosită și în laboratoarele de grafuri (ex. algoritmul Floyd-Warshall - pe care îl veți implementa și la PA; algoritmi pe arbori etc).

Programarea dinamică presupune rezolvarea unei probleme prin **descompunerea ei în subprobleme** şi rezolvarea acestora. Spre deosebire de divide et impera, subproblemele nu sunt disjuncte, ci **se suprapun**.

Pentru a evita recalcularea porțiunilor care se suprapun, rezolvarea se face pornind de la cele mai mici subprobleme şi folosindu-ne de rezultatul acestora calculăm subproblema imediat mai mare. Cele mai mici subprobleme sunt numite subprobleme unitare, acestea putând fi rezolvate într-o complexitate constantă, ex: cea mai mare subsecvență dintr-o mulțime de un singur element.

Pentru a nu recalcula soluțiile subproblemelor ce ar trebui rezolvate de mai multe ori, pe ramuri diferite, se reține soluția subproblemelor folosind o tabelă (matrice uni, bi sau multi-dimensională în funcție de problemă) cu rezultatul fiecărei subprobleme. Această tehnică se numește **memoizare**.

Această tehnică determină ”valoarea” soluției pentru fiecare din subprobleme. Mergând de la subprobleme mici la subprobleme din ce în ce mai mari ajungem la soluția optimă, la nivelul întregii probleme. ”Valoarea” își schimbă înțelesul logic de la o problemă la alta. În problemele de minimizarea costului, ”valoarea” este reprezentată de costul minim. În probleme care presupun identificarea unei componente maxime, ”valoarea” este caracterizată de dimensiunea componentei.

După calcularea valorii pentru toate subproblemele se poate determina efectiv mulțimea de elemente care compun soluția. „Reconstrucția” soluţiei se face mergând din subproblemă în subproblemă, începând de la problema cu valoarea optimă și ajungând în subprobleme unitare. Metoda și recurența variază de la problemă la problemă, dar în urma unor exerciții practice va deveni din ce în ce mai facil să le identificați.

Cei curioși pot citi aici [https://en.wikipedia.org/wiki/Dynamic_programming#History] adevărul despre numele acestei tehnici. 😎

Ce determina DP?

Aplicând această tehnică determinăm **una din soluțiile optime**, problema putând avea mai multe soluții optime. În cazul în care se dorește determinarea tuturor soluțiilor optime, algoritmul trebuie combinat cu unul de backtracking în vederea construcției soluțiilor.

În cazul problemelor de numărare, această tehnică ne va furniza **numărul căutat**.

Tipar general DP

Aplicarea acestei tehnici de programare poate fi descompusă în următoarea secvență de pași:

1. Identificarea structurii și a metricilor utilizate în caracterizarea soluției optime;
2. Determinarea unei metode de calcul recursiv pentru a afla valoarea fiecărei subprobleme;
3. Calcularea “bottom-up” a acestei valori (de la subproblemele cele mai mici la cele mai mari);
4. Reconstrucția soluției optime pornind de la rezultatele obținute anterior.

Exemple clasice

Programarea Dinamică este cea mai flexibilă tehnică din programare. Cel mai ușor mod de a o înțelege presupune parcurgerea cât mai multor exemple.

Propunem câteva categorii de recurențe, pe care le vom grupa astfel:

- recurențe de tip **SSM** (Subsecvența de Sumă Maximă)
- recurențe de tip **RUCSAC**
- recurențe de tip **PODM** (Parantezare Optimă de Matrici)
- recurențe de tip **numărat**
- recurențe pe **grafuri**

Pentru o problemă dată, este **posibil** să găsim **mai multe recurențe corecte** (mai multe soluții posibile). Evident, criteriul de alegere între acestea va fi cel bazat pe complexitate.

Categoria 1: SSM

Aceste recurențe au o oarecare asemănare cu problema SSM (enunț + soluție).

ATENȚIE! Rețineți diferența între următoarele 2 noțiuni!

* **subsecvență** (substring [https://en.wikipedia.org/wiki/Substring] în engleză) pentru un vector **v** înseamnă un alt vector $u = [v[i], v[i+1], \ldots, v[j]]$ unde $i <= j$.

* **subșir** (subsequence [https://en.wikipedia.org/wiki/Subsequence] în engleză) pentru un vector **v** înseamnă un alt vector $u = [v[i_1], v[i_2], \ldots, v[i_k]]]$ unde $i_1 < i_2 < \ldots < i_k$.

SSM

Enunț

Fie un vector v cu n elemente întregi. O subsecvență de numere din șir este de forma: $v_i, v_{i+1}, \ldots, v_j$ ($i <= j$), având suma asociată $s_{ij} = v_i + v_{i+1} + \ldots + v_j$. O subsecvență **nu** poate fi vidă.

Cerință

Să se determine subsecvența de sumă maximă (notată **SSM**).

Exemple

n = 6

i	1	2	3	4	5	6
v[i]	-10	2	3	-1	2	-3

Răspuns: SSM este între 2 și 5 (poziții). Are suma +6. ($SSM = 2, 3, -1, 2$)

Explicație: avem numere pozitive, deci există o soluție simplă în care putem să alegem doar un număr pozitiv/mai multe numere pozitive de pe poziții alăturate (adică încercăm să evităm numere negative). Cele mai lungi subsecvențe cu numere pozitive sunt 2,3 și 2. Observăm că dacă extindem $2, 3$ la $2, 3, -1, 2$, deși am inclus un număr negativ, suma secvenței crește.

n = 4

i	1	2	3	4
v[i]	10	20	30	40

Răspuns: SSM este între 1 și 4 (poziții). Are suma 100. ($SSM = 10, 20, 30, 40$)

Explicație: deoarece toate numerele sunt **pozitive**, SSM cuprinde toate numerele.

n = 4

i	1	2	3	4
v[i]	-10	-20	-30	-40

Răspuns: SSM este între 1 si 1 (poziții). Are suma -10. ($SSM = -10$)

Explicație: deoarece toate numerele sunt **negative**, SSM cuprinde doar cel mai mare număr.

Rezolvare

Tiparul acestei probleme ne sugerează că o soluție este obținută incremental, în sensul că **putem** privi problema astfel: găsim cea mai bună soluție folosind **primele** $i-1$ elemente din șir, apoi încercăm să o **extindem** folosind elementul **i** (adică ne extindem la dreapta ~CU~ $v[i]$).

Numire recurență

Întrucât la fiecare pas trebuie sa reținem **cea mai bună soluție** folosind un **prefix** din vectorul v, soluția va fi salvată într-un tablou auxiliar definit astfel:

$dp[i]$ = suma subsecvenței de sumă maximă (**suma SSM**) folosind **doar primele i** elemente din vectorul v și care **se termină pe poziția i**

Mențiuni

- Pentru a menține o convenție, toate tablourile de acest tip din laborator vor fi notate cu **dp** (dynamic programming).
- Ca să rezolvăm problema dată, trebuie să rezolvăm o mulțime de subprobleme
  - $dp[i]$ reprezintă **soluția** pentru problema $v[1], \ldots, v[i]$ și care se termină cu $v[i]$
- Soluția pentru problema inițială este maximul din vectorul $dp[i]$ - a.k.a. **max(dp[i])**.

Găsire recurență

Întrucât dorim ca această problemă să fie rezolvabilă printr-un algoritm/bucată de cod, trebuie să descriem o metodă concretă prin care vom calcula $dp[i]$.

- **Cazul de bază**
  - În general în probleme putem avea mai multe cazuri de bază, care în principiu se leagă de valori extreme are dimensiunilor subproblemelor.
  - În cazul SSM, avem un singur caz de bază, când avem un singur element în prefix: $dp[1] = v[1]$.
  - Explicație: dacă avem un singur element, atunci acesta formează singura subsecvență posibilă, deci $SSM = v[1]$

- **Cazul general**
- presupune inductiv că avem rezolvate toate subproblemele mai mici
- în cazul SSM, presupunem că avem calculat $dp[i-1]$ și dorim sa calculăm $dp[i]$ (cunoaștem cea mai bună soluție folosind primele i-1 elememente și vedem dacă elementul de pe poziția i o poate îmbunătăți)
- la fiecare pas avem de ales dacă $v[i]$ extinde cea mai bună soluție care se termină pe $v[i-1]$ sau se începe o nouă secvență cu $v[i]$
- decidem în funcție de $dp[i-1]$ și $v[i]$
  - **dacă** $dp[i-1] >= 0$ (cea mai bună soluție care se termină pe i - 1 are cost nenegativ)
    - extindem secvență care se termină cu v[i-1] folosind elementul v[i]: $dp[i] = dp[i-1] + v[i]$
    - Explicație: $dp[i-1] + v[i] >= v[i]$ (încă are rost să extind)
  - **dacă** $dp[i-1] < 0$ (cea mai bună soluție care se termină pe i - 1 are cost negativ)
    - vom începe o nouă secvență cu $v[i]$, adică $dp[i] = v[i]$
    - Explicație: $v[i] > dp[i-1] + v[i]$, deci prin extindere nu obțin soluție maximă!

Implementare recurență

În majoritatea problemelor de DP, găsirea recurenței ocupă cea mai mare parte a timpului de rezolvare (lucru adevărat și în cazul problemelor de la PA). De aceea, faptul că ați reușit să scrieți pe foaie lucruri foarte complicate poate fi un indiciu ca ați pornit pe o cale greșită.

Problema se poate testa pe infoarena: Subsecvență de sumă maximă [https://www.infoarena.ro/problema/ssm].

Mai jos se află un exemplu simplu de implementare a recurenței găsite în C++.

```
// găsește SSM pentru vectorul v cu n elemente
// pentru a menține convenția din explicații:
//      - elementele sunt indexate de la 0, dar le folosesc doar pe cele care incep de la 1
//                                          => v[1], ..., v[n]
int SSM(int n, vector<int> &v) {
        vector<int> dp(n + 1);    // vector cu n + 1 elemente (indexarea începe de la 0)
                                  // am nevoie de dp[1], ..., dp[n]

        // caz de bază
        dp[1] = v[1];

        // caz general
        for (int i = 2; i <= n; ++i) {
                if (dp[i - 1] >= 0) {
                        // extinde la dreapta cu v[i]
                        dp[i] = dp[i - 1] + v[i];
                } else {
                        // încep o nouă secvență
                        dp[i] = v[i];
                }
        }

        // soluția e maximul din vectorul dp
        int sol = dp[1];
        for (int i = 2; i <= n; ++i) {
                if (dp[i] > sol) {
                        sol = dp[i];
                }
        }

        return sol; // aceasta este suma asociată cu SSM
}
```

Dacă dorim să afișăm și indicii între care apare SSM, putem să stocăm și poziția de start pentru fiecare soluție intermediară. . Hint: definiți **start[i]** = poziția pe care a început subsecvența care dă soluția cu cost dp[i].

Mențiuni

Întrucât această soluție presupune calculul iterativ (coloană cu coloană) a matricei dp, complexitatea este liniară. De asemenea, se mai parcurge o dată dp pentru a găsi maximul.

- **complexitate temporală** : $T = O(n)$
- **complexitate spațială** : $S = O(n)$
  - desigur că pentru problema SSM, nu era nevoie sa reținem, tablourile dp/start în memorie.
  - puteam sa construim element cu element și maximul din dp în aceleași timp (întrucât ne trebuie ultima valoare la fiecare pas și maximul global).
  - în acest caz complexitatea spațială devine $S = O(1)$

Pentru a ilustra toți pașii posibili într-o astfel de problemă, totul a fost prezentat cât mai simplu (NU în toate problemele putem facem simplificări de tipul "NU am nevoie să stochez tabloul dp").

SCMAX

Fie un vector v cu n elemente întregi. Un subșir de numere din șir este de forma: $v_{i_1}, v_{i_2}, \ldots, v_{i_k}$. Un subșir **nu** poate fi vid ($k >= 1$).

Cerința

Să se determine subșirul crescător maximal (notat **SCMAX**) - un subșir ordonat strict crescător și are lungime maximă (dacă sunt mai multe soluții, să se gasească una oarecare).

Exemple

n = 6

i	1	2	3	4	5	6
v[i]	100	12	13	-1	15	-30

Răspuns: $SCMAX = 12, 13, 15$ ($SCMAX = v[2], v[3], v[5]$).

Explicație: Toate subșirurile ordonate strict crescător sunt:

- 100
- 12
- $12, 13$
- $12, 13, 15$
- $12, 15$
- 13
- $13, 15$
- -1
- $-1, 15$
- 15
- -30

Cel menționat este singurul de lungime 3.

n = 6

i	1	2	3	4	5	6
v[i]	100	12	13	-1	15	14

Răspuns:

- $SCMAX = 12, 13, 15$ ($SCMAX = v[2], v[3], v[5]$).
- $SCMAX = 12, 13, 14$ ($SCMAX = v[2], v[3], v[6]$).

Explicație: Toate subșirurile ordonate strict crescător sunt:

- 100
- 12
- $12, 13$
- $12, 13, 15$
- $12, 13, 14$
- 13

- $13, 14$
- -1
- $-1, 15$
- $-1, 14$
- 15
- 14

Cele 2 soluții indicate au ambele lungime maximă.

Rezolvare

Tipar

Verificăm dacă se aplică tiparul de la SSM: **găsim cea mai buna soluție folosind primele $i-1$ elemente din șir, apoi încercăm să o extindem folosind elementul i (adică ne extindem la dreapta ~CU~ $v[i]$)**.

- Dacă avem cea mai bună soluție pentru intervalul $1, 2, \ldots, i-1$ și care se termină cu $v[i-1]$, atunci încercăm să extindem soluția cu $v[i]$ (putem dacă $v[i-1] < v[i]$)
- Altfel.. Unde am putea să îl punem pe $v[i]$?
    - Păi am putea sa încercăm să îl punem la finalul soluției care se termină pe $v[i-2]$, $v[i-3]$, ... sau $v[1]$

Numire recurență

$dp[i]$ = lungimea celui mai lung subșir(**lungime SCMAX**) folosind (doar o parte) din primele i elemente din vectorul v și care se termină pe poziția i

Mențiuni

- Ca să rezolvăm problema dată, trebuie să rezolvăm o mulțime de subprobleme
    - $dp[i]$ reprezintă **soluția** pentru problema $v[1], \ldots, v[i]$ și care se termină cu $v[i]$
- Soluția pentru problema inițială este maximul din vectorul $dp[i]$.

Găsire recurență

- **Cazul de bază**
    - Și în problema SCMAX, cazul pentru $i = 1$ este caz de bază.
        - dacă avem un singur element, atunci avem o singură subsecvență de lungime 1, ea este soluția
        - $dp[1] = 1$

- **Cazul general**
    - presupune inductiv că avem rezolvate toate subproblemele mai mici
    - în cazul SCMAX, presupunem că avem calculate $dp[1], dp[2], \ldots, dp[i-1]$ și dorim să calculăm $dp[i]$ (cunoaștem cea mai bună soluție folosind primele j elemente și vedem dacă elementul de pe poziția i o poate îmbunătăți - $j = 1 : i-1$)
    - deoarece nu știm unde e cel mai bine să îl pune pe $v[i]$ (după care v[j]?), încercăm pentru toate valorile posibile ale lui j (unde $j = 1 : i-1$)
        - **dacă $v[j] < v[i]$**, atunci subșirul crescător care se termină pe poziția j, poate fi extins la dreapta cu elementul v[i], generând lungimea **dp[j] + 1**

        - deci dp[i] = max(dp[j] + 1), $j = 1 : i - 1$ (dacă nu există un astfel de j, valoarea lui max(…) este 0)
    - Ce se întamplă totuși dacă nu există un j care să îndeplinească condiția de mai sus? Atunci $v[i]$ va forma singur un subșir crescător de lungime 1 (care poate fi folosit la un pas ulterior)

Reunind cele spuse mai sus:

- $dp[1] = 1$
- $dp[i] = 1 + max(dp[j])$, unde $j = 1 : i - 1$ **și** $v[j] < v[i]$; $i = 2 : n$

Implementare recurență

Problema se poate testa pe infoarena: Subșir crescător maximal [https://www.infoarena.ro/problema/scmax].

Mai jos se află un exemplu simplu de implementare a recurentei găsite în C++.

```
// n   = numărul de elemente din vector
// v   = vectorul dat (v[1], v[2], ..., v[n] - indexare de la 1 ca în explicații)

void scmax(int n, vector<int> &v) {
        vector<int> dp(n + 1);   // în explicații indexarea începe de la 1

        // caz de bază
        dp[1] = 1;   // [ v[1] ] este singurul subșir (crescător) care se termină pe 1

        // caz general
        for (int i = 2; i <= n; ++i) {
                dp[i] = 1;   // [ v[i] ] - este un subșir (crescător) care se termină pe i

                // încerc să îl pun pe v[i] la finalul tuturor soluțiilor disponibile
                // o soluție se termină cu un element v[j]
                for (int j = 1; j < i; ++j) {
                        // soluția trivială: v[i]
                        if (v[j] < v[i]) {
                                // din (..., v[j]) pot obține (..., v[j], v[i])
                                // (caz în care prec[i] = j)

                                // voi alege j-ul curent, când alegerea îmi găsește o soluție mai bună decât ce am deja
                                if (dp[j] + 1 > dp[i]) {
                                        dp[i] = dp[j] + 1;
                                }
                        }
                }
        }

        // soluția e maximul din vectorul dp
        int sol = dp[1], pos = 1;
        for (int i = 2; i <= n; ++i) {
                if (dp[i] > sol) {
                        sol = dp[i];
                        pos = i;
                }
        }

        return sol;
}
```

Problema se poate testa pe infoarena: Subșir crescător maximal [https://www.infoarena.ro/problema/scmax].

În pa-lab::demo/lab03/02-scmax [https://github.com/acs-pa/pa-lab/tree/main/demo/lab03/02-scmax] găsiți un exemplu de implementare care arată și cum puteți reconstitui SCMAX. Față de implementarea anterioară, în această versiune se folosește un tablou auxiliar prec.

$prec[i]$ = indicele j al elementului v[j], pentru care $dp[j] + 1 == dp[i]$ (adică acel j pentru care subșirul crescător maximal care se termină cu $v[i]$ este extinderea cu un element a celui care se termină cu $v[j]$.

- dacă nu există un astfel de j, atunci $prec[i] = 0$ (prin convenție)

Mențiuni

Întrucât această soluție presupune calculul iterativ (coloană cu coloană) a matricei dp, complexitatea este polinomială (pătratică - pentru fiecare element din tabloul, facem o trecere prin elementele deja calculate).

- **complexitate temporală** : $T = O(n^2)$
  - se poate obține o soluție în complexitate $T = O(nlogn)$ dacă se folosește o căutare binară pentru a găsi elementul j dorit (ex. implemetare [https://www.infoarena.ro/job_detail/1248867?action=view-source]).
- **complexitate spațială** : $S = O(n)$
  - NU putem obține o complexitate spatială mai bună, întrucât avem nevoie să stocăm cel puțin vectorul dp (stocăm și vectorul **prec** dacă avem nevoie să reconstituim SCMAX)

Categoria 2: RUCSAC

Aceste recurențe au o oarecare asemănare cu problema RUCSAC - varianta discretă (enunț + soluție).

RUCSAC

Enunț

Fie un set (vector) cu n obiecte (care nu pot fi tăiate - varianta discretă 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** pe care sa 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 obiectul).

Cerință

Să se determine **profitul maxim** pentru Gigel.

Exemple

$n = 5$ si $W = 10$

Processing math: 100%					
w	3	3	1	1	2

p	6	3	2	8	5

Răspuns: **24** (profitul maxim)

Explicație: va alege toate obiectele :D.

$n = 5$ și $W = 3$

	1	2	3	4	5
w	3	3	1	1	2
p	6	3	2	8	5

Răspuns: **13** (profitul maxim)

Explicație: va alege obiectele cu indicii 4 si 5 (profit: 8 + 5)

Rezolvare

Tipar

Cum am transpune tiparul de la SSM/SCMAX în problema RUCSAC?

- știm care este profitul maxim pe care îl obține dacă folosim
  - doar primul element
  - doar primele 2 elemente
  - …
  - doar primele $i - 1$ elemente
- ajung să mă gândesc la obiectul (elementul) i
  - este posibil ca acesta să nu apară neapărat în soluția cea mai bună, caz în care nu îl folosesc, deci soluția maximă se găsește între cele menționate mai sus
  - dacă folosesc elementul i caracterizat de (w_i, p_i), în primul rând acesta trebuie să încapă în ghiozdan…
    - cum verific acest lucru?
    - o recurență de tipul $dp[i] = \dots$ nu va fi suficientă, pentru că în această problemă am 2 dimensiuni: **obiectele** (submulțimile de indici) și **greutățile** (asociate cu obiectele / submulțmile de obiecte).

Numire recurență

Întrucât la fiecare pas trebuie să reținem **cea mai bună soluție** folosind un **prefix** din vectorul de obiecte, dar pentru că trebuie să punem și **o restricție de greutate** necesară (ocupată in rucsac), soluția va fi salvată într-un tablou auxiliar definit astfel:

$dp[i][cap]$ = profitul maxim (**profit RUCSAC**) obținut folosind (doar o parte) din primele i obiecte și având un rucsac de **capacitate maximă cap**

Observații:

- **NU** există restricție în folosirea obiectului i în soluția menționată de $dp[i][cap]$ (a.k.a. se poate folosi sau se poate ignora).
- Soluția problemei se găsește în $dp[n][W]$ (profitul maxim folosind (doar o parte) din primele n elemente - adică soluția bazată pe inspectarea tuturor obiectelor; capacitatea maximă folosită este W - adică soluția bazată pe ghiozdanul de capacitate maximă).

Găsire recurență

- **Cazul de bază**
  - Dacă avem o submulțime vidă de obiecte selectate.
    - $dp[0][cap] = 0$
    - Explicație: Dacă nu alegem obiecte, atunci profitul este 0 indiferent de capacitate.

- **Cazul general**
  - $dp[i][cap] = ?$
  - presupune inductiv că avem rezolvate toate subproblemele mai mici
    - subprobleme mai mici înseamnă să folosească mai puține obiecte sau un rucsac cu capacitatea mai mică
    - vedem dacă prin folosirea obiectului i, obținem cea mai bună soluție in $dp[i][cap]$
      - **NU folosesc obiectul i**
        - în acest caz, o să alegem cea mai bună soluție formată cu celelalte $i-1$ elemente și aceeași capacitate a rucsacului
        - soluția generată de acest caz: $dp[i][cap] = dp[i-1][cap]$
      - **folosesc obiectul i**
        - dacă îl folosesc, înseamnă că pentru el trebuie să am rezervată în rucsac o capacitate egală cu w_i
          - adică când am selectat dintre primele $i-1$ elemente, nu trebuia să ocup mai mult de $cap - w_i$ din capacitatea rucsacului
          - față de subproblema menționată, câștig în plus p_i (profitul pe care îl aduce acest obiect
        - soluția generată de acest caz: $dp[i][cap] = dp[i-1][cap - w_i] + p_i$

Reunind cele spuse mai sus, obținem:

- $dp[0][cap] = 0$, pentru $cap = 0 : W$
- $dp[i][cap] = max(dp[i-1][cap], dp[i-1][cap - w_i] + p_i)$
  - pentru $i = 1 : n$, $cap = 0 : W$

Implementare recurență

Problema se poate testa pe infoarena: Problema rucsacului [https://www.infoarena.ro/problema/rucsac].

Mai jos se află un exemplu simplu de implementare a recurenței găsite in C++.

```
// n   = numărul de obiecte din colecție
// W   = capacitatea maximă a rucsacului
// (w[i], p[i]) = caracteristicile obiectului i ($i = 1 : n)

int rucsac(int n, int W, vector<int> &w, vector<int> &p) {
    // dp este o matrice de dimensiune (n + 1) x (W + 1)
    // pentru că folosim dp[0][*] pentru mulțimea vidă
    //                   dp[*][0] pentru situația în care ghiozdanul are capacitate 0
    vector< vector<int> > dp(n + 1, vector<int>(W + 1, 0));

    // cazul de bază
    for (int cap = 0; cap <= W; ++cap) {
        dp[0][cap] = 0;
    }

    // cazul general
    for (int i = 1; i <= n; ++i) {
        for (int cap = 0; cap <= W; ++cap) {
```

```
            // nu folosesc obiectul i => e soluția de la pasul i - 1
            dp[i][cap] = dp[i-1][cap];

            // folosesc obiectul i, deci trebuie să rezerv w[i] unități în rucsac
            // înseamnă ca înainte trebuie să ocup maxim cap - w[i] unități
            if (cap - w[i] >= 0) {
                int sol_aux = dp[i-1][cap - w[i]] + p[i];

                dp[i][cap] = max(dp[i][cap], sol_aux);
            }
        }
    }

    return dp[n][W];
}
```

Mențiuni

Întrucât această soluție presupune calculul iterativ (linie cu linie) a matricei dp, complexitatea este polinomială.

- **complexitate temporală** : $T = O(n * W)$
- **complexitate spațială** : $S = O(n * W)$
  - dacă nu ne interesează să reconstituim soluția (să afișăm submulțimea efectiv), atunci putem să NU stocăm toată matricea dp
  - ca să calculăm o linie, avem nevoie doar de ultima linie
  - putem să stocăm la orice moment de timp doar ultima linie și linia curentă
  - complexitatea spațială se reduce astfel la $S = O(W)$

Exercitii

Scheletul de laborator se găsește pe pagina pa-lab::skel/lab03 [https://github.com/acs-pa/pa-lab/tree/main/skel/lab03].

1. Not again!

Gigel are o colectie impresionanta de monede. El ne spune ca are **n tipuri** de monede, avand un numar nelimitat de monede din fiecare tip. Cunoscand aceasta informatie (data sub forma unui vector **v** cu **n** elemente), el se intreaba care este numarul minim de monede cu care poate plati o **suma S**.

Task-uri:

- 1.1 **Determinati numarul minim de monede** (din cele pe care le are) cu care Gigel poate forma suma S.
- 1.2 Care este complexitatea solutiei (timp + spatiu)? De ce?

Este posibil ca pentru anumite valori ale lui S si v, aceasta problema sa nu aiba solutie. In acest caz raspunsul este **-1**.

$n = 4$ si $S = 12$

i	1	2	3	4
v	1	2	3	6

Raspuns: 2

Explicatie: Avem 4 tipuri de monede: 1 euro, 2 euro, 3 euro si 6 euro (lui Gigel nu ii mai place sa foloseasca RON). Avem la dispozitie oricate monede din fiecare tip. Suma 12 poate fi obtinuta in

urmatoarele moduri:

- $12 = 6 + 6$
- $12 = 6 + 3 + 3$
- $12 = 6 + 3 + 2 + 1$
- $12 = 6 + 2 + 2 + 2$
- $12 = 6 + 3 + 3$
- $12 = 3 + 3 + 3 + 3$
- $12 = 3 + 3 + 3 + 2 + 1$
- $12 = 3 + 3 + 2 + 2 + 2$
- $12 = 3 + 2 + 2 + 2 + 2 + 1$
- $12 = 2 + 2 + 2 + 2 + 2 + 2$
- … (ati inteles ideea :D)

Solutia cu numar minim de monede se obtine pentru modul $6 + 6$.

$n = 3$ si $S = 11$

i	1	2	3
v	1	2	5

Raspuns: 3

Explicatie: Avem 3 tipuri de monede: 1 euro, 2 euro si 5 euro (lui Gigel nu ii mai place sa foloseasca RON). Avem la dispozitie oricate monede din fiecare tip. Suma 11 poate fi obtinuta in urmatoarele moduri:

- $11 = 5 + 5 + 1$
- $11 = 5 + 2 + 2 + 2$
- $11 = 5 + 2 + 2 + 1 + 1$
- $11 = 5 + 2 + 1 + 1 + 1 + 1$
- $11 = 5 + 1 + 1 + 1 + 1 + 1 + 1$
- $11 = 2 + 2 + 1 + 1 + 1 + 1 + 1 + 1 + 1$
- $11 = 2 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1$
- $11 = 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1$

Solutia cu numar minim de monede se obtine pentru modul $5 + 5 + 1$.

$n = 3$ si $S = 11$

i	1	2	3
v	2	4	6

Raspuns: -1

Explicatie: Nu putem forma suma 11 folosind tipurile (valorile) 2, 4, 6.

2. CMLSC

Fie doi vectori cu numere intregi: **v** cu **n** elemente si **w** cu **m** elemente. Sa se gaseasca **cel mai lung subsir comun** (notat **CMLSC**) care apare in cei doi vectori. Se cere o solutie de complexitate optima. Daca exista mai multe solutii, se poate gasi oricare.

- 2.1 **Determinare lungime** CMLSC. (Hint: DP)
- 2.2 **Reconstituire** CMLSC (afisati si care sunt termenii CMLSC).
- 2.3 Care este **complexitatea** solutiei (timp + spatiu)? De ce?

Rezolvati in ordine task-urile.

subsir (**subsequence** in engleza) pentru un vector **v** inseamna un alt vector $u = [v[i_1], v[i_2], \ldots, v[i_k]]]$ unde $i_1 < i_2 < \ldots < i_k$.

$n = 3$ si $m = 5$

i	1	2	3
v	6	-1	9

j	1	2	3	4	5
w	0	6	2	9	8

Raspuns: $lungime = 2$, $CMLSC = [6, 9]$

Explicatie: Toate subsirurile comune posibile sunt:

- $[6]$
- $[6, 9]$
- $[9]$

Solutia mentionata are lungime maxima.

$n = 8$ si $m = 5$

i	1	2	3	4	5	6	7	8
v	2	1	5	3	4	5	2	7

j	1	2	3	4	5
w	1	5	5	7	4

Raspuns: $lungime = 4$, $CMLSC = [1, 5, 5, 7]$

Explicatie: Toate subsirurile comune posibile sunt (duplicatele vor fi mentionate o singura data):

- $[1]$
- $[1, 5]$
- $[1, 7]$
- $[1, 4]$
- $[1, 5, 5]$
- $[1, 5, 7]$
- $[1, 5, 4]$
- $[1, 5, 5, 7]$
- $[5]$
- $[5, 5]$
- $[5, 7]$
- $[5, 4]$
- $[4]$

Solutia mentionata are lungime maxima.

$n = 8$ si $m = 5$

i	1	2	3	4	5	6	7	8
v	2	1	5	3	4	5	2	7

j	1	2	3	4	5
w	1	5	7	-5	4

Raspuns: $lungime = 3$, $CMLSC = [1, 5, 7]$ (exemplu de solutie)

Explicatie: Toate subsirurile comune posibile sunt (duplicatele vor fi mentionate o singura data):

- $[1]$
- $[1, 5]$
- $[1, 7]$
- $[1, 4]$
- $[1, 5, 7]$
- $[1, 5, 4]$
- $[5]$
- $[5, 7]$
- $[5, 4]$
- $[4]$

Solutii pot fi: $[1, 5, 7]$ si $[1, 5, 4]$. Pentru $[1, 5, 7]$, se observa ca sunt 2 astfel de subsiruri in vectorul v. Oricare este bun.

BONUS

Rezolvati pe infoarena problema custi [https://infoarena.ro/problema/custi].

Extra

Modificati solutia de la Rucsac prezentata in laborator pentru a obtine o complexitate spatiala mai buna (se va retine un numar minim de linii din matrice). Puteti testa solutia voastra pe infoarena la problema rucsac [https://infoarena.ro/problema/rucsac].

Se da o matrice de dimensiuni $n * m$ si Q intrebari de forma: "Care este suma din submatricea care are coltul stanga-sus (x, y) si coltul dreapta-jos (p,q)?"

Se considera proprietatea: Q este mult mai mare decat dimensiunile matricei.

Sa se raspunda in mod eficient la cele Q intrebari.

Rezolvatie pe infoarena problema joctv [https://infoarena.ro/problema/joctv].

Solutie: Se fixeaza 2 linii pentru zona dreptunghiulara. Se reduce problema la SSM in O(n). Complexitate: $O(n^3)$.

Rezolvati pe leetcode problema Interleaving String [https://leetcode.com/problems/interleaving-string/description/].

https://www.geeksforgeeks.org/cutting-a-rod-dp-13/ [https://www.geeksforgeeks.org/cutting-a-rod-dp-13/] Cutting rod [https://www.geeksforgeeks.org/cutting-a-rod-dp-13/] [0]
Partition Problem [https://www.geeksforgeeks.org/partition-problem-dp-18/]

Referințe

[0] Chapter **Dynamic Programming**, "Introduction to Algorithms", Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein

[1] http://infoarena.ro/problema/ssm [http://infoarena.ro/problema/ssm]

[2] http://infoarena.ro/problema/scmax [http://infoarena.ro/problema/scmax]

[3] http://infoarena.ro/problema/rucsac [http://infoarena.ro/problema/rucsac]