Laborator 3: Programare Dinamică

Responsabili:

- Darius Neațu [mailto:neatudarius@gmail.com]
- Radu Vișan [mailto:visanr95@gmail.com]
- Cristian Banu [mailto:cristb@gmail.com]
- Răzvan Chițu [mailto:razvan.ch95@gmail.com]

Obiective laborator

- Ințelegerea noțiunilor de bază despre programare dinamică
- Insușirea abilităților de implementare a algoritmilor bazați programare dinamică.

Precizari initiale

Toate exemplele de cod se gasesc in demo-lab03.zip.

Acestea apar incorporate si in textul laboratorului pentru a facilita parcurgerea cursiva a laboratorului.

- Toate bucatile de cod prezentate in partea introductiva a laboratorului (inainte de exercitii) 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 sa nu fie de ajuns pentru a compila codul.
- Va rugam sa incercati si codul din arhiva **demo-lab03.zip**, inainte de a raporta ca ceva nu merge.
- Pentru orice problema legata de continutul acestei pagini, va rugam sa dati email unuia dintre responsabili.

Ce este DP?

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

De asemenea, DP se poate folosi si pentru probleme in care nu cautam un optim, cum ar fi **problemele de numarare** (vom exemplifica in lab04).

Aplicatii DP

Programarea dinamică are un **câmp larg de aplicare**, insa la PA ne vom rezuma la cateva aplicatii care vor fi mentionate pe parcursul laboratoarelor 3 si 4. De asemenea, aceasta tehnica va fi folosita si in laboratoarele de grafuri (ex. algoritmul Floyd-Warshall - pe care il veti implementa si la PA; algoritmi pe arbori etc).

Programare 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. Aceasta tehnica se numește **memoizare**.

Aceasta tehnică determina ”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” is schimba înțelesul logic de la o problema la alta. În probleme de minimizarea costului, ”valoarea” este reprezentata de costul minim. In 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 curiosi pot citi aici [https://en.wikipedia.org/wiki/Dynamic_programming#History] adevarul despre numele acestei tehnici.

Ce determina DP?

Aplicând aceasta 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.

In cazul problemelor de numarare, aceasta tehnica ne va furniza **numarul cautat**.

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 tehnica din programare. Cel mai ușor mod de a o înțelege presupune parcurgerea cât mai multor exemple.

Propunem cateva categorii de recurente, pe care le vom grupa astfel:

- recurente de tip **SSM** (Subsecventa de Suma Maxima)
- recurente de tip **RUCSAC**
- recurente de tip **PODM** (Parantezare Optima de Matrici)
- recurente de tip **numărat**
- recurente pe **grafuri**

Pentru o problema data, este **posibil** sa gasim **mai multe recurente corecte** (mai multe solutii posibile). Evident, criteriul de alegere intre acestea va fi cel bazat pe complexitate.

Categoria 1: SSM

Aceste recurente au o oarecare asemanare cu problema SSM (enunt + solutie).

SSM

Enunt

Fie un vector v cu n elemente intregi. O subsecventa de numere din sir este de forma: $s_i, s_{i+1}, \ldots, s_j$ $(i <= j)$, avand suma asociata $s_{ij} = s_i + s_{i+1} + \ldots + s_j$. O subsecventa **nu** poate fi vida.

Cerinta

Sa se determine subsecventa de suma maxima (notata **SSM**).

Exemple

n = 6

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

Raspuns: SSM este intre 2 si 5 (pozitii). Are suma +6. ($SSM = 2, 3, -1, 2$)

Explicatie: avem numere pozitive, deci exista o solutie simpla in care putem sa alegem doar un numar pozitiv/mai multe numere pozitive de pe pozitii alaturate (adica incercam sa evitam numere negative). Cele mai lungi subsecvente cu numere pozitive sunt

2,3 si 2. Observam ca daca extindem $2, 3$ la $2, 3, -1, 2$, desi am inclus un numar negativ, suma secventei creste.

n = 4

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

Raspuns: SSM este intre 1 si 4 (pozitii). Are suma 100. ($SSM = 10, 20, 30, 40$)

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

n = 4

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

Raspuns: SSM este intre 1 si 1 (pozitii). Are suma -10. ($SSM = -10$)

Explicatie: deoarece toate numerele sunt **negative**, SSM cuprinde doar cel mai mare numar.

Rezolvare

Tipar

Tiparul acestei probleme ne sugereaza ca o solutie este obtinuta incremental, in sensul ca **putem** privi problema astfel: gasim cea mai buna solutie folosind **primele** $i - 1$ elemente din sir, apoi incercam sa o **extindem** folosind elementul **i** (adica ne extindem la dreapta ~CU~ $v[i]$).

Numire recurenta

Intrucat la fiecare pas trebuie sa retinem **cea mai buna solutie** folosind un **prefix** din vectorul v, solutia va fi salvata intr-un tablou auxiliar definit astfel:

$dp[i]$ = suma subsecventei de suma maxima (**suma SSM**) folosind **doar primele i** elemente din vectorul v si care se termina pe pozitia i

Mentiuni

- Pentru a mentine o conventie, toate tablourile de acest tip din laborator vor fi notate cu **dp** (dynamic programming).
- Ca sa rezolvam problema data, trebuie sa rezolvam o multime de subprobleme
  - $dp[i]$ reprezinta **solutia** pentru problema $v[1], \ldots, v[i]$ si care se termina cu $v[i]$
- Solutia pentru problema initiala este maximul din vectorul $dp[i]$.

Gasire recurenta

Intrucat dorim ca aceasta problema sa fie rezolvabila printr-un algoritm/bucata de cod, trebuie sa descriem o metoda concreta prin care vom calcula $dp[i]$.

- **Cazul de baza**
  - In general in probleme putem avea mai multe cazuri de baza, care in principiu se leaga de valori extreme are dimensiunilor subproblemelor.
  - In cazul SSM, avem un singur caz de baza, cand avem un singur element in prefix: $dp[1] = v[1]$.
  - Explicatie: daca avem un singur element, atunci acesta formeaza singura subsecventa posibila, deci $SSM = v[1]$

- **Cazul general**
- presupune inductiv ca avem rezolvate toate subproblemele mai mici
- in cazul SSM, presupunem ca avem calculat $dp[i-1]$ si dorim sa calculam $dp[i]$ (cunoastem cea mai buna solutie folosind primele i-1 elememente si vedem daca elementul de pe pozitia i o poate imbunatati)
- la fiecare pas avem de ales daca $v[i]$ extinde cea mai buna solutie care se termina pe $v[i-1]$ sau se incepe o noua secventa cu $v[i]$
- decidem in functie de $dp[i-1]$ si $v[i]$
  - **daca** $dp[i-1] >= 0$ (cea mai buna solutie care se termina pe i - 1 are cost nenegativ)
    - extindem secventa care se termina cu v[i-1] folosind elementul v[i]: $dp[i] = dp[i-1] + v[i]$

    - Explicatie: $dp[i-1] + v[i] >= v[i]$ (inca are rost sa extind)
  - **daca** $dp[i-1] < 0$ (cea mai buna solutie care se termina pe i - 1 are cost negativ)
    - vom incepe o noua secventa cu $v[i]$, adica $dp[i] = v[i]$
    - Explicatie: $v[i] > dp[i-1] + v[i]$, deci prin extindere nu obtin solutie maxima!

Implementare recurenta

In majoritatea problemelor de DP, gasirea recurentei ocupa cea mai mare parte a timpului de rezolvare (lucru adevarat si in cazul problemelor de la PA). De aceea, faptul ca ati reusit sa scrieti pe foaie lucruri foarte complicate poate fi un indiciu ca ati pornit pe o cale gresita.

Mai jos se afla un exemplu simplu de implementare a recurentei gasite in C++.

```
// gaseste SSM pentru vectorul v cu n elemente
// pentru a mentine conventia din explicatii:
//      - 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 incepe de la 0)
                                  // am nevoie de dp[1], ..., dp[n]

        // caz de baza
        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 {
                        // incep o noua secventa
                        dp[i] = v[i];
                }
        }

        // solutia 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 asociata cu SSM
}
```

Daca dorim sa afisam si indicii intre care apare SSM, putem sa stocam si pozitia de start pentru fiecare solutie intermediara. Gasiti aceasta solutie in **demo-lab03.zip**. Hint: definiti **start[i]** = pozitia pe care a inceput subsecventa care da solutia cu cost dp[i].

Mentiuni

Intrucat aceasta solutie presupune calculul iterativ (coloana cu coloana) a matricei dp, complexitatea este liniara. De asemenea, se mai parcurge o data dp pentru a gasi maximul.

- **complexitate temporala** : $T = O(n)$
- **complexitate spatiala** : $S = O(n)$
  - desigur ca pentru problema SSM, nu era nevoie sa retinem, tablourile dp/start in memorie.
  - puteam sa construim element cu element si maximul din dp in aceleasi timp (intrucat ne trebuie ultima valoare la fiecare pas si maximul global).
  - in acest caz complexitatea spatiala devine $S = O(1)$

Pentru a ilustra toti pasii posibili intr-o astfel de problema, totul a fost prezentat cat mai simplu (NU in toate problemele putem facem simplificari de tipul “NU am nevoie sa stochez tabloul dp”).

SCMAX

Enunt

Fie un vector v cu n elemente intregi. Un subsir de numere din sir este de forma: $s_{i_1}, s_{i_2}, ..., s_{i_k}$. Un subsir **nu** poate fi vid ($k >= 1$).

Cerinta

Sa se determine subsirul crescator maximal (notat **SCMAX**) - un subsir ordonat strict crescator si are lungime maxima (daca sunt mai multe solutii, sa se gaseasca una oarecare).

Exemple

n = 6

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

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

Explicatie: Toate subsirurile ordonate strict crescator sunt:

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

Cel mentionat este singurul de lungime 3.

n = 6

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

Raspuns:

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

Explicatie: Toate subsirurile ordonate strict crescator sunt:

- 100
- 12
- $12, 13$
- $12, 13, 15$
- $12, 13, 14$
- 13
- $13, 15$
- $13, 14$
- -1
- $-1, 15$
- $-1, 14$
- 15
- 14

Cele 2 solutii indicate au ambele lungime maxima.

Rezolvare

Tipar

Verificam daca se aplica tiparul de la SSM: **gasim cea mai buna solutie folosind primele $i-1$ elemente din sir, apoi incercam sa o extindem folosind elementul i (adica ne extindem la dreapta ~CU~ $v[i]$).**

- Daca avem cea mai buna solutie pentru intervalul $1, 2, .., i-1$ si care se termina cu $v[i-1]$, atunci incercam sa extindem solutia cu $v[i]$ (putem daca $v[i-1] < v[i]$)
- Altfel.. Unde am putea sa il punem pe $v[i]$?
  - Pai am putea sa incercam sa il punem la finalul solutiei care se termina pe $v[i-2]$, $v[i-3]$, ... sau $v[1]$

Numire recurenta

$dp[i]$ = lungimea celui mai lung subsir(**lungime SCMAX**) folosind (doar o parte) din primele i elemente din vectorul v si care se termina pe pozitia i

Mentiuni

- Ca sa rezolvam problema data, trebuie sa rezolvam o multime de subprobleme
  - $dp[i]$ reprezinta **solutia** pentru problema $v[1], ..., v[i]$ si care se termina cu $v[i]$
- Solutia pentru problema initiala este maximul din vectorul $dp[i]$.

Gasire recurenta

- **Cazul de baza**
  - Si in problema SCMAX, cazul pentru $i = 1$ este caz de baza.
    - daca avem un singur element, atunci avem o singura subsecventa de lungime 1, ea este solutia
    - $dp[1] = 1$

- **Cazul general**
  - presupune inductiv ca avem rezolvate toate subproblemele mai mici
  - in cazul SCMAX, presupunem ca avem calculate $dp[1], dp[2], ..., dp[i-1]$ si dorim sa calculam $dp[i]$ (cunoastem cea mai buna solutie folosind primele j elemente si vedem daca elementul de pe pozitia i o poate imbunatati - $j = 1 : i-1$)
  - deoarece nu stim unde e cel mai bine sa il pune pe $v[i]$ (dupa care v[j]?), incercam pentru toate valorile posibile ale lui j (unde $j = 1 : n-1$)
    - **daca** $v[j] < v[i]$, atunci subsirul crescator care se termina pe pozitia j, poate fi extins la dreapta cu elementul v[i], generand lungimea **dp[j] + 1**
      - deci dp[i] = max(dp[j] + 1), $j = 1 : i-1$ (daca nu exista un astfel de j, valoarea lui max(…) este 0)
    - Ce se intampla totusi daca nu exista un j care sa indeplineasca conditia de mai sus? Atunci $v[i]$ va forma singur un subsir crescator de lungime 1 (care poate fi la un pas ulterior)

Reunind cele spuse mai sus:

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

Implementare recurenta

Mai jos se afla un exemplu simplu de implementare a recurentei gasite in C++.

```
// n   = numarul de elemente din vector
// v   = vectorul dat (v[1], v[2], ..., v[n] - indexare de la 1 ca in explicatii)

void scmax(int n, vector<int> &v) {
        vector<int> dp(n + 1);   // in explicatii indexarea incepe de la 1

        // caz de baza
        dp[1] = 1;   // [ v[1] ] este singurul subsir (crescator) care se termina pe 1

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

                // incerc sa il pun pe v[i] la finalul tuturor solutiilor disponibile
                // o solutie se termina cu un element v[j]
                for (int j = 1; j < i; ++j) {
                        // solutia triviala: v[i]
                        if (v[j] < v[i]) {
```

```
                                // din (..., v[j]) pot obtine (..., v[j], v[i])
                                // (caz in care prec[i] = j)

                                // voi alege j-ul curent, cand alegerea imi gaseste o solutie mai buna decat ce am deja
                                if (dp[j] + 1 > dp[i]) {
                                        dp[i] = dp[j] + 1;
                                }
                        }
                }
        }

        // solutia 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;
}
```

In **demo-lab03.zip** gasiti un exemplu de implementare care arata si cum puteti reconstitui SCMAX. Fata de implementarea anterioara, in aceasta versiune se foloseste un tablou auxiliar prec.

$prec[i]$ = indicele j al elementului v[j], pentru care $dp[j] + 1 == dp[i]$ (adica acel j pentru care subsirul crescator maximal care se termina cu $v[i]$ este extinderea cu un element a celui care se termina cu $v[j]$.

- daca nu exista un astfel de j, atunci $prec[i] = 0$ (prin conventie)

Mentiuni

Intrucat aceasta solutie presupune calculul iterativ (coloana cu coloana) a matricei dp, complexitatea este polinomiala (patratica - pentru fiecare element din tabloul, facem o trecere prin elementele deja calculate).

- **complexitate temporala** : $T = O(n^2)$
  - se poate obtine o solutie in complexitate $T = O(nlogn)$ daca se foloseste o cautare binara pentru a gasi elementul j dorit.
- **complexitate spatiala** : $S = O(n)$
  - NU putem obtine o complexitate spatiala mai buna, intrucat avem nevoie sa stocam cel putin vectorul dp (stocam si vectorul prec daca avem nevoie sa reconstituim SCMAX)

Categoria 2: RUCSAC

Aceste recurente au o oarecare asemanare cu problema RUCSAC - varianta discreta (enunt + solutie).

RUCSAC

Enunt

Fie un set (vector) cu n obiecte (care nu pot fi taiate - varianta discreta a problemei). Fiecare obiect i are asociata o pereche (w_i, p_i) cu semnificatia:

- w_i = $weight_i$ = greutatea obiectului cu numarul i
- p_i = $price_i$ = pretul obiectului cu numarul i
  - $w_i >= 0$ si $p_i > 0$

Gigel are la dispozitie un rucsac de **volum infinit**, dar care suporta o **greutate maxima** (notata cu W - weight knapsack).

El vrea sa gaseasca **o submultime de obiecte** pe care sa le bage in rucsac, astfel incat **suma profiturilor sa fie maxima**.

Daca Gigel baga in rucsac obiectul i, caracterizat de (w_i, p_i), atunci profitul adus de obiect este p_i (presupunem ca il vinde cu cat valoreaza obiectul).

Cerinta

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

Exemple

$n = 5$ si $W = 10$

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

Raspuns: **24** (profitul maxim)

Explicatie: va alege toate obiectele :D.

$n = 5$ si $W = 3$

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

Raspuns: **13** (profitul maxim)

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

Rezolvare

Tipar

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

- stim care este profitul maxim pe care il obtine daca folosim
  - doar primul element
  - doar primele 2 elemente
  - …
  - doar primele $i - 1$ elemente
- ajung sa ma gandesc la obiectul (elementul) i
  - este posibil ca acesta sa nu apara neaparat in solutia cea mai buna, caz in care nu il folosesc, deci solutia maxima se gaseste intre cele mentionate mai sus
  - daca folosesc elementul i caracterizat de (w_i, p_i), in primul rand acesta trebuie sa incapa in ghiozdan…
    - cum verific acest lucru?
    - o recurenta de tipul $dp[i] = ...$ nu va fi suficienta, pentru ca in aceasta problema am 2 dimensiuni: **obiectele** (submultimile de indici) si **greutatile** (asociate cu obiectele / submultimile de obiecte).

Numire recurenta

Intrucat la fiecare pas trebuie sa retinem **cea mai buna solutie** folosind un **prefix** din vectorul de obiecte, dar pentru ca trebuie sa punem si **o restrictie de greutate** necesara (ocupata in rucsac), solutia va fi salvata intr-un tablou auxiliar definit astfel:

$dp[i][cap]$ = profitul maxim (**profit RUCSAC**) obtinut folosind (doar o parte) din primele i obiecte si avand un rucsac de **capacitate maxima cap**

Observatii:

- NU exista restrictie daca in solutia mentionata de $dp[i][cap]$ este folosit OBLIGATORIU elementul i
- Solutia problemei se gaseste in $dp[n][W]$ (profitul maxim folosind (doar o parte) din primele n elemente - adica toate; capacitatea maxima folosita este W - adica capacitatea maxima a rucsacului).

Gasire recurenta

- **Cazul de baza**
  - Daca avem o submultime vida de obiecte selectate.
    - $dp[0][cap] = 0$
    - Explicatie: Daca nu alegem obiecte, atunci profitul este 0 indiferent de capacitate.
- **Cazul general**

- $dp[i][cap] = ?$
- presupune inductiv ca avem rezolvate toate subproblemele mai mici
  - subprobleme mai mici inseamna sa foloseasca mai putine obiecte sau un rucsac cu capacitatea mai mica
  - vedem daca prin folosirea obiectului i, obtinem cea mai buna solutie in $dp[i][cap]$
    - **NU folosesc obiectul i**
      - in acest caz, o sa alegem cea mai buna solutie formata cu celelalte $i-1$ elemente si aceeasi capacitate a rucsacului
      - solutia generata de acest caz: $dp[i][cap] = dp[i-1][cap]$
    - **folosesc obiectul i**
      - daca il folosesc, inseamna ca pentru el trebuie sa am rezervata in rucsac o capacitate egala cu w_i
        - adica cand am selectat dintre primele $i-1$ elemente, nu trebuia sa ocup mai mult de $cap - w_i$ din capacitatea rucsacului
        - fata de subproblema mentionata, castig in plus p_i (profitul pe care il aduce acest obiect
      - solutia generata de acest caz: $dp[i][cap] = dp[i-1][cap - w_i] + p_i$

Reunind cele spuse mai sus, obtinem:

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

Implementare recurenta

Mai jos se afla un exemplu simplu de implementare a recurentei gasite in C++.

```
// n   = numarul de obiecte din colectie
// W   = capacitatea maxima 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 ca folosim dp[0][*] pentru multimea vida
    //                   dp[*][0] pentru situatia in care ghiozdanul are capacitate 0
    vector< vector<int> > dp(n + 1);
    for (int i = 0; i <= n; ++i) {
        dp[i].resize(W + 1);
    }

    // cazul de baza
    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 obiectu i => e solutia de la pasul i - 1
            dp[i][cap] = dp[i-1][cap];

            // folosesc obiectul i, deci trebuie sa rezerv w[i] unitati in rucsac
            // inseamna ca inainte trebuie sa ocup maxim cap - w[i] unitati
            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];
}
```

Mentiuni

Intrucat aceasta solutie presupune calculul iterativ (linie cu linie) a matricei dp, complexitatea este polinomiala.

- **complexitate temporala** : $T = O(n * W)$
- **complexitate spatiala** : $S = O(n * W)$

- daca nu ne intereseaza sa reconstituim solutia (sa afisam submultimea efectiv), atunci putem sa NU stocam toata matricea dp
- ca sa calculam o linie, avem nevoie doar de ultima linie
- putem sa stocam la orice moment de timp doar ultima linie si linia curenta
- complexitatea spatiala se reduce astfel la $S = O(W)$

Exercitii

In laboratorul de astazi, implementarea exercitiilor nu va fi punctata. Cu toate acestea, daca doriti sa implementati problemele propuse spre rezolvare, puteti folosi scheletul de laborator din arhiva skel-lab03.zip.

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.

Problema preluata de aici [https://leetcode.com/problems/coin-change/description/]. Solutia este aici [https://leetcode.com/problems/coin-change/solution/].

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.

Task-uri:

- 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.

Problema preluata de aici [https://infoarena.ro/problema/cmlsc].

- $dp[i][j]$ = lungimea celui mai lung subsir comun pentru sub-vectorii $v[1..i]$ si $w[1..j]$
- $dp[i][0] = 0$; $dp[0][j] = 0$
- $dp[i][j] = (v[i] == w[j]\ ?\ dp[i-1][j-1] + 1 : max(dp[i][j-1], dp[i-1][j])$

Solutia este $dp[n][m]$. Reconstituirea se face pe baza matricei dp. Din starea (i, j) se poate trece in starea $(p, q) = (i-1, j-1) sau (i-1, j) sau (i, j-1)$, pe baza relatiei dintre valorile din dp. Obtinem astfel un CMLSC inversat. Complexitate: $O(n * m)$.

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] Capitolul **Dynamic Programming** din **Introductions to Algorithms** de către T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. 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]

[4] Colecție de probleme de programare dinamică (geeksforgeeks.com) [https://www.geeksforgeeks.org/dynamic-programming/]

[5] Probleme de interviu care se rezolvă cu PD [https://practice.geeksforgeeks.org/explore/?category%5B%5D=Dynamic%20Programming&page=1&sortBy=accuracy]

[6] Sniedovich, Moshe. Dynamic programming: foundations and principles. CRC press, 2010.