# Programare dinamică
***pentru nivelul începător și mediu***

Programarea diamică este o tehnică (metodă) de programare. Acest lucru înseamnă că sunt probleme care dacă îndeplinesc anumite condiții se pot încadra ca rezolvabile cu această tehnică.

*La problemele de programare dinamică se cere de regulă un optim.*

*"Să se determine costul minim pentru a realiza o anumită sarcină"*

Dacă reușim să exprimăm sarcina în funcție de alte sarcini mai simple rezolvabile anterior, într-un mod aciclic, atunci problema este candidată de a fi rezolvabilă cu programare dinamică.

Când spunem "sarcini rezolvabile una din alta într-un mod aciclic" ne gândim că dacă o sarcină `A` se poate rezolva dacă avem rezolvată deja o altă sarcină `B`, să nu fie nevoie ca sarcina `B` să aibă nevoie direct sau indirect de sarcina `A`.

Ideea anterioară, a aciclicității dependențelor, este fundamentală când vorbim de programare dinamică.

Astfel dacă sarcina de rezolvat este un element al unui șir, adesea se caută o formulă de recurență în care elementul șirului se scrie în funcție de alte elemente ale șirului (sarcinile de care depinde cea de rezolvat). Această recurență este aceeași la fiecare element al șirului și apoi ne mai rămâne să calculăm aceste elemente cu o repetiție în ordinea dependențelor. Bineînțeles, este nevoie de etapa de inițializare, adică pentru primele elemente, care nu au de cine să fie exprimate recurent anterior, să identificăm valorile lor. Exact ca la relațiile de recurență.

Totodată, ce am descris anterior respectă principiile recursivității, adică, pentru a rezolva sarcina curentă putem apela recursiv în sarcinile de care depinde. Este însă esențială memorizarea. Adică, dacă facem recursiv, odată ce rezolvăm o sarcină, notăm valoarea ei într-o structură, iar dacă vreun alt apel recursiv are nevoie de acea valoare, să verifice mai întâi în structură.

De regulă putem împărți problemele de programare dinamică în două:

- Probleme de optim (să se determine costul minim pentrui a realiza o anumită sarcină).
- Probleme de numărare (să se determine în câte moduri putem realiza o anumită sarcină).

În toate cazurile, avem nevoie de o structură (vector, matrice) în elementele căreia să calculăm rezultatele sarcinilor. Adică fiecare sarcină să aibă un element al structurii în care se stochează valoarea sa după ce de calculează.

Practic, pașii de rezolvare a unei probleme cu programare dinamică sunt:

1. Identificăm stările (sarcinile) apoi stabilim structura în care calculăm informațiile despre stări (aici este esențial să definim riguros, anterior, semnificația elementelor stucturii).
2. Stabilim care sunt elementele de început ale structurii, adică acelea a căror valoare o calculăm direct, dar și care este elementul structurii care reprezintă starea finală, cea asociată cu cerința problemei.
3. Căutăm o relație de recurență între elementele stucturii.
4. Transcriem în program relația de recurență.

Adesea pasul `4` este cel mai simplu, iar o sursă de programare dinamică este de obicei scurtă. Dar tocmai pașii anteriori necesită un efort mare de analiză și gândire. Dacă ne dăm seama că semnificația pe care am dat-o elementelor stucturii nu ne duce și la găsirea unei relații de recurență, va trebui să căutăm altă structură, sau altă semnificație. Modalități de a alege această semnificație vom discuta la concret în timp ce vom prezenta problemele rezolvate.

# I. Probleme la care optimul curent depinde de un numar constant de optime anterioare.

Chiar dacă am numit capitolul folosind termenul de “optim”, vom încadra aici și multe probleme de numărare.

**Subsecvența de sumă maximă.**

*Enunț*

Avem un șir de numere (întregi) și dorim să găsim o secvență de elemente a căror sumă să fie cât mai mare.

*Analiză și soluții*

Deseori la problemele rezolvabile prin dinamică se caută mai întâi o soluție greedy. Aici vom prezenta și demonta o idee pe care am observat că o spun repede mulți elevi: căutăm secvențe maximale de elemente pozitive și determinăm suma celei mai mari. Dar nu este suficient, pentru că putem avea, de exemplu, între două astfel de secvențe o sumă de elemente negative mai mică (în modul) decât fiecare dintre cele două pozitive și atunci e de preferat să luăm o secvență cu toate (ceva de genul: `2 7 -1 -3 2 3 2`; aici suma tuturor este mai mare decât suma din fiecare dintre secvențele de elemente pozitive).

Această problemă are în primul rând o soluție de complexitate $n^3$: fixăm toate secvențele posibile cu două foruri (unul pentru începuturi și unul pentru finaluri) și apoi facem un al treilea for pentru a calcula suma dintre cei doi indici fixați.

Există și soluție cu timp de calcul de ordin $n^2$ (pe măsură ce avansăm cu al doilea for extindem secvența de la pasul anterior - cu același început fixat de primul for). Sau cu sume parțiale: folosim primele două foruri ca la soluția în $n^3$ și înlocuim al treilea for cu o diferență de sume parțiale.

Soluția cu programare dinamică are complexitate de ordin `n`, și se obținem astfel:

Într-un vector `S`, cu aceeași dimensiune ca și vectorul dat `V`, calculăm `S[i]` = *suma maximă a unei secvențe cu finalul exact la poziția* `i`.

Observăm că pentru a calcula valoarea `S[i]` avem două variante:

- Fie lipim pe `V[i]` la o secvență terminată la poziția anterioară (și acum este evident că ne convine să considerăm cea mai bună secvență terminată la poziția anterioară, și ne ajută faptul că aceasta este deja în `S[i-1]`, deci, cantitatea pe care o vom lua în calcul va fi `S[i-1] + V[i]`)
- Fie începem o nouă secvență cu `V[i]`.

Includem oricum pe `V[i]` întrucât, conform definiției lui `S[i]`, trebuie să avem o secvență care îl conține pe `V[i]`.

Deci `S[i]` depinde doar de elementul din S de pe poziția anterioară. Astfel, la început trebuie să cunoaștem doar pe `S[1]`. Care evident, conform definiției este chiar `V[1]`.

Implementarea ar fi așa:

```
S[1] = V[1];
for (i=2;i<=n;i++)
     if (S[i-1] + V[i] > V[i])
          S[i] = S[i-1]+V[i];
     else
          S[i] = V[i];
```

Noi trebuie să afișăm maximul din `S`. Astfel obținem valoarea sevenței de sumă maximă. Dacă reținem și poziția maximului din `S` avem deci și finalul secvenței de sumă maximă.

Fiind la început, e util să identificăm formal cum am identificat la această problemă cei `4` pași despre care am spus anterior că este o bună practică să îi urmărim:

| | |
|---|---|
| Identificăm stările apoi stabilim structura în care calculăm informațiile despre stări (aici este esențial să definim riguros, anterior, semnificația elementelor stucturii) | `S[i]` = *suma maximă a unei secvențe cu finalul exact la poziția* `i` |
| Stabilim care sunt elementele de început ale structurii, adică acelea a căror valoare o calculăm direct, dar și care este elementul structurii care reprezintă starea finală, cea asociată cu cerința problemei | `S[1]` = `V[1]` *este inițializarea*<br>*Maximul din* `S` *este rezultatul cerut.* |

| Căutăm o relație de recurență între elementele stucturii | `S[i] = maxim(S[i-1]+V[i], V[i])` |
|---|---|
| Transcriem în program relația de recurență | *Codul anterior.* |

Alegerea între `S[i-1]+V[i]` și `V[i]` mai poate avea semnificația: Dacă suma maximă de la pasul anterior este pozitivă, lipim pe `V[i]` la ea, altfel, ne convine să îl luăm pe `V[i]` singur.

O altă analiză care trebuie făcută este cea despre memorie: pentru cetința enunțată chiar ne trebuie vectorul `S`?

Observăm că un element din `S` se determină folosind elementul anterior. Astfel, am putea folosi doar două varicbile `sCurent` și `sAnterior` și avea:

```
sAnterior = V[1];
for (i=2;i<=n;i++) {
      sCurent = max(sAnterior + V[i], V[i])
      sAnterior = sCurent;
}
```

Sau chiar o singură variabilă s care se actualizează:

```
s = V[1];
for (i=2;i<=n;i++) {
      s = max(s + V[i], V[i])
}
```

Recomandăm implementarea algoritmului pe infoarena, unde se cere și identificarea poziției subsecvenței de sumă maximă:

https://infoarena.ro/problema/ssm

**Alte probleme care se rezolvă folosind subsecvența de sumă maximă**

***Buline*** *(https://www.infoarena.ro/problema/buline)*

*Pe scurt, această problemă cere determinarea subsecvenței de sumă maximă pe un șir circular de numere întregi.*

Considerând șirul circular memorat într-un tablou unidimensional cu `n` elemente (în care elementele de pe pozițiile `1` și `n` sunt “vecine”), observația estențială este că subsecvența de sumă maximă poate fi:

- Undeva între pozițiile `1` și `n` ca și cum șirul nu ar fi considerat circular (aceasta se identifică folosind algoritmul obișnuit)

- Să înceapă spre finalul șirului și să se continue cu începutul. În acest caz, restul elementelor, de prin mijloc, trebuie să formeze o secvență de sumă cât mai mică. Așadar, în acest caz avem de determinat subsecvența de sumă minimă pe șirul dat, neconsiderat circulat. Rezolvarea este la fel ca și cea pentru subsecvența de sumă maximă.

***JocTV*** *(https://www.infoarena.ro/problema/joctv)*

*Pentru o matrice pătratică de dimensiune* `n`*, cu elemente întregi, avem de determinat o submatrice de sumă maximă. Vom prezenta o rezolvare cu timp de calcul de ordin* $n^3$*.*

Fixăm în toate modurile linia de sus și linia de jos a submatricei candidat. Aceasta necesită două foruri și avem deci timp de calcul de ordin $n^2$ pentru asta.

Acum avem de determinat coloana stângă și cea dreaptă. Considerăm un vector, care are câte un element pentru fiecare coloană, astfel:

`C[i]` = suma elementelor de pe coloana `i`, aflate între cele două linii fixate.

Pentru a construi rapid vectorul `C` pentru fiecare poziționare de linii `liniaSus`/`liniaJos`, putem face de la început sume parțiale pe coloane, și când acem nevoie, obținem repede `C[i] = Sp[liniaJos][i] - Sp[liniaSus-1][i]`.

Odată acest vector construit, suma unei secvențe din el este de fapt suma unei submatrice care are coloana din stânga pe poziția de început a secvenței și coloana din dreapta pe poziția de final a secvenței. Așadar ne interesează subsecvența de sumă maximă din acest vector.

Alte probleme care folosesc subsecvența de sumă maximă:

***Șotron*** *(https://www.infoarena.ro/problema/sotron)*

***Peri*** *(http://campion.edu.ro/arhiva/index.php?page=problem&action=view&id=935)*

## Probleme de numărare

Aceste probleme cer să determinăm numărul de elemente pentru mulțimi cu anumite proprietăți. De multe ori pentru aceste probleme soluția este o formulă. Însă nu tot timpul acest lucru este posibil și în acest caz cunoașterea unei abordări prin programare dinamică poate fi soluția.

Iată un exemplu:

a) Să se determine câte șiruri avem cu proprietățile:
- Au lungimea o valoare `n` dată
- Au ca elemente `0`, `1`, `2`

b) Să se determine câte șiruri avem cu proprietățile:
- Au lungimea o valoare `n` dată
- Au ca elemente `0`, `1`, `2`
- Nu putem avea `0` și `1` vecine

c) Să se determine câte șiruri avem cu proprietățile:
- Au lungimea o valoare `n` dată
- Au ca elemente `0`, `1`, `2`, …, `9`
- Se mai cunosc anumite perechi de cifre `i`,`j` care nu pot fi vecine

***Problema a)*** are clar soluția $3^n$.

O altă soluție care duce la acest rezultat se obține cu următorul raționament.

Notăm cu `D[i]` = *câte șiruri de lungime* `i`*, cu* `0`*,* `1`*,* `2` *putem avea*.

Fiecare șir de lungime `i-1` se poate transforma în trei moduri diferite într-un șir de lungime `i` (un mod adăugându-i la final un `0`, alt mod adăugându-i la final un `1` și alt mod adăugându-i la final un `2`). Dar noi avem `D[i-1]` șiruri de lungime `i-1` și am arătat că fiecare produce `3` șiruri diferite de lungime `i`. Așadar:

```
D[i] = D[i-1]*3
```

Ce avem este tocmai o recurență care necesită un termen anterior.
Ne mai trebuie elementul de la început: `D[1] = 3` (sunt `3` șiruri de lungime `1`).
Avem așadar tot ce trebuie pentru o soluție de programare dinamică:

```
cin>>n;
D[1] = 3;
for (i=2;i<=n;i++)
     D[i] = D[i-1]*3;
cout<<D[i];
```

Noi de fapt înmulțim `3` cu el însuși de `n` ori, așadar ajungem la formula depistată de la început $3^n$.

Și aici observăm că nu este necesar tot vectorul `D`, ci o singură variabilă pe care doar o actualizăm.

***Problema b)***

Aici avem constrângerea că nu putem avea `0` și `1` vecine. Dacă avem `D[i-1]` șiruri corecte de dimensiune `i-1`, acum nu mai este clar că din fiecare astfel de șir putem obține `3` diferite (de exemplu, dintr-un șir de lungime `i-1` terminat cu `3` putem obține `3` diferite de lungime `i`, că putem adăuga după `2` oricare dintre cele `3` cifre, dar dacă șirul se termină cu `0` sau cu `1`, avem doar două cifre pe care le putem adăuga).

Din analiza anterioară rezultă că nu este suficient să ținem `D[i]` cu semnificația de mai sus, ci să despărțim în `3` structuri:

`Z[i]` = *numărul de șiruri de lungime* `i` *terminate cu* `0`
`U[i]` = *numărul de șiruri de lungime* `i` *terminate cu* `1`
`D[i]` = *numărul de șiruri de lungime* `i` *terminate cu* `2`

Astfel:

`Z[i] = Z[i-1] + D[i-1]` (obținem șir de lungime `i` terminat cu `0`, adăugând un `0` la finalul unui șir terminat cu `0` sau terminat cu `2`)

`U[i] = U[i-1] + D[i-1]` (obținem șir de lungime `i` terminat cu `1`, adăugând un `1` la finalul unui șir terminat cu `1` sau terminat cu `2`)

`D[i] = Z[i-1] + U[i-1] + D[i-1]` (obținem șir de lungime `i` terminat cu `2`, adăugând un `2` la finalul unui șir terminat cu `0` sau terminat cu `1` sau terminat cu `2`)

La început ne trebuie: `Z[1] = 1`; `U[1] = 1`; `D[1] = 1`;

Soluția finală este `Z[n]+U[n]+D[n]`

O modalitate mai elegantă de scriere a codului și care este un pas pentru generalizare (necesară la problema c)) este următoarea:

Folosim o matrice cu `3` linii și `n` coloane (linia `0` este pentru șiruri terminate cu `0`, linia `1` pentru șiruri terminate cu `1`, linia `2` pentru șiruri terminate cu `2)` .

Astfel: `D[c][i]` = *numărul de șiruri de lungime* `i` *terminate cu cifra* `c`.

```
for (c=0;c<=2;c++)
      D[c][1] = 1;
for (i=2;i<=n;i++) {
      D[0][i] = D[0][i-1]+D[2][i-1];
      D[1][i] = D[1][i-1]+D[2][i-1];
      D[2][i] = D[0][i-1]+D[1][i-1]+D[2][i-1];
}
```

Soluția este `D[0][n] + D[1][n] + D[2][n]`;

Am putea face economie de memorie, observând că pentru calculului celor `3` valori de la poziția `i` sunt necesare doar cele `3` valori de la poziția `i-1`.

***Problema c)***

Presupunem că aici avem la intrare o matrice `C`, de `10X10` în care `C[i][j] = 1` dacă cifrele `i` și `j` nu pot fi vecine.

În acest caz, avem de asemenea nevoie să cunoaștem câte șiruri de lungime `i` avem, dar și cifra în care se termină (pentru a ști ce este permis să adăugăm).

Este dificil acum să luăm `10` variabile diferite, ca la prima variantă a soluției anterioare (adică ar fi complicat de codat) și ne rămâne doar varianta cu o matrice cu `10` linii astfel:

`D[c][i]` = *numărul de șiruri de lungime* `i` *terminate cu cifra* `c`.

```
for (c=0;c<=9;c++)
      D[c][i] = 1;
for (i=2;i<=n;i++) {
      for (c=0;c<=9;c++) {
            /// aici dorim să determinăm D[c][i].
            /// Avem deci de adăugat cifra c la finalul unui
            /// sir de lungime i-1. Aici ne folosim de
            /// matricea C pentru a identifica
            /// cifre cif după care poate fi pusă cifra c
            D[c][i] = 0;
            for (cif=0;cif<=9;cif++)
                  if (C[cif][c] == 0)
                        D[c][i] += D[cif][i-1];
      }
}
```

Suma de pe ultima coloană a matricei `D` este soluția problemei.

O problemă pe ideea de mai sus este ***aceasta***: https://www.infoarena.ro/problema/nrcuv

De ***asemenea***: https://www.infoarena.ro/problema/div3

La problema a) a fost evident că putem găsi o formulă pentru rezultat, mai ales uitându-ne la faptul că rezultatul depinde de un singur număr, `n`.

La problema b) a mai fost și o constrângere și atunci găsirea unei formule începe să devină complicată. Totuși, și aici se poate, noi depindem doar de `n` și de o singură constrângere.

La problema c) însă, avem până la `100` de constrângeri așa că numărul de parametri de care ar depinde formula este foate mare. În acest caz soluția rămâne doar dinamica.

Ca și concluzie despre problemele de numărare, dacă avem doar puțini parametri de care depinde rezultatul, avem toate șansele să găsim o formulă, având apoi și varianta implementării acesteia folosind noțiuni de combinatorică. Dacă însă rezultatul depinde de mulți parametri, cum ar fi configurația unei structuri cu multe stări, nu are sens să căutăm formule.

Soluțiile prezentate au timpul de calcul de ordin `n`, înmulțit cu o constantă care depinde de numărul de constrângeri.

Observație. Unele dintre problemele de numărare de acest tip se pot rezolva folosind exponențierea de matrice, înlocuind factorul `liniar o(n)` cu unul `logaritmic o(logn)`. Această tehnică nu face obiectul acestei prezentări.

O altă idee de bază care trebuie urmărită la problemele de numărare. Trebuie să ne asigurăm de două lucruri:

- Să numărăm toate soluțiile;
- Să nu numărăm o soluție de mai multe ori sau să ratăm numărarea unora, sau chiar să numărăm soluții incorecte;

De exemplu, la problema b) noi putem avea șiruri terminate cu `0`, `1`, `2` și avem grijă să le luăm în calcul pe toate. Deci numărăm tot. Pe de altă parte, dintr-un șir dat, dacă punem la final valori diferite, obținem soluții diferite. Deci nu numărăm de mai multe ori același lucru.

Un alt lucru esențial în care gândim problemele de dinamică: noi ținem date despre o stare și ne gândim că la pasul următor mai adăugăm ceva pentru a obține altă stare. Așa că mai mereu spunem "***care se termină cu: …***" - pentru a avea informații despre cum putem obține noua stare prin adăugare.

La problemele de numărare se întâmplă adesea ca rezultatul să fie o valoare mare. Astfel, trebuie să estimăm care este tipul de date necesar pentru a memora corect valoarea lui. Uneori ne putem da seama că nu avem un tip de date dedicat memorării rezultatului și atunci trebuie să implementăm operații pe numere mari. În unele cazuri enunțul ne ajută să evităm operații cu numere mari, indicând ca rezultatul să se afișeze "modulo o valoare precizată". În acest ultim caz va trebui să cunoaștem regulile de aritmetică modulară (referindu-ne pe scurt la cest lucru, spunem doar că partea puțin mai dificilă este la împărțire, când trebuie să calculăm "invers modularul", în rest este în general suficient să aplicăm operatorul "modulo" după fiecare operație).

### ***Scara***

*Avem o scara cu* `n` *trepte si suntem jos (treapta* `0`*). La un pas pot urca una sau două trepte. Se cere să determinăm cate moduri distincte de a urca scara sunt. Dacă vom codifica o modalitate de urcare printr-un șir de* `1` *și* `2` *conform ordinii în care facem pașii, două modalități diferă dacă există cel puțin o poziție pe care valorile din cele două codificări diferă.*

Vom calcula: `D[i]` = in cate moduri pot ajunge pe treapta i.

- Pot ajunge acolo de pe treapta `i-1` urcand o treapta deodata. Fiecare mod de a ajunge pe treapta `i-1` devine un mod de a ajunge pe treapta `i`.
- Pot ajunge acolo de pe treapta `i-2` urcand doua treapte deodata. Fiecare mod de a ajunge pe treapta `i-2` devine un mod de a ajunge pe treapta `i`.

Astfel:

```
D[i] = D[i-1]+D[i-2]
```

În acest caz este nevoie să cunoaștem două valori la început (recurența folosește doi termeni anteriori).

`D[1] = 1` (urc direct o treaptă)
`D[2] = 2` (o variantă este să fac direct un pas de două trepte iar cealaltă este să facem succesiv doi pași de câte o treaptă).

Problema este echivalentă cu a determina toate modurie de a scrie numărul `n` ca sumă de termeni `1` și `2`, unde ordinea termenilor contează.

```
N = 5
1 1 1 1 1, 1 1 1 2, 1 1 2 1, 1 2 1 1, 1 2 2, 2 1 1 1,  2 1 2, 2 2 1
```

Pe scurt, problema cere să determinăm termenul `n` din șirul lui *Fibonacci*.

### *Sumele partiale*

Folosim des tehnica "sumelor parțiale" pentru a face precalculări utile mai departe în optimizare. Această tehnică poate fi prezentată și folosind principiile programării dinamice:

Pentru un șir `V` cu `n` elemente, notăm `S[i]` = *suma primelor* `i` *elemente din* `S`. Calculând elementele lui `S` în ordinea crescătoare a indicilor, pentru a obține `S[i]` ne folosim de `S[i-1]` (care deja memorează suma primelor `i-1` elemente), la care adunăm `V[i]`.

Astfel avem recurența cunoscută:

```
S[1] = V[1];
S[i] = S[i-1] + V[i];
```

### *Maxime parțiale*

Chiar determinarea maximului dintr-un vector, pot fi interpretată și prin prisma programarii dinamice.

Noi determinăm maximul dintr-un vector, în mod tradițional astfel:

```
maxim = v[1];
for (i=2;i<=n;i++)
      if (v[i] > maxim)
            maxim = v[i];
```

Am putea privi lucrurile și astfel: notam `M[i]` = *cel mai mare dintre primele i elemente ale vectorului*. Valoarea `M[i]` se calculează în funcție de elementul curent, `V[i]` și de `M[i-1]`, tocmai calculat la pasul anterior.

```
M[1] = v[1];
for (i=2;i<=n;i++)
      M[i] = max(v[i], M[i-1])
```

Codul inițial este doar optimizarea de memorie, adică nu mai este necesar vectorul `M` și suficientă o singură variabila, `maxim`.

***Sume partiale pe matrice***

Extinderea la două dimensiuni a sumelor parțiale pe vector presupune ca într-o matrice `A` să calculăm:

`S[i][j]` = *suma din submatricea cu coluri opuse (*`1,1`*) (*`i,j`*)*. S se calculează clasic, crescător după linii și apoi crescător după coloane, astfel:

`S[i][j] = S[i-1][j] + S[i][j-1] - S[i-1][j-1] + A[i][j];`

## Probleme pe matrice

***Clărire*** (https://www.pbinfo.ro/probleme/393/cladire1)

*Avem o matrice cu* `n` *linii și* `m` *coloane. Unele celule sunt blocate și le marcăm cu* `1`*. Să se determine în câte moduri putem face un traseu din colțul (*`1,1`*) în colțul (*`n,m`*) care să treacă doar prin celule libere și în care fiecare pas se poate face în celula vecină din dreapta sau de jos (dacă aceasta este liberă).*

Calculăm `D[i][j]` = *numărul de moduri prin care ajung în linia* `i`*, coloana* `j`*, pornind din (*`1,1`*).*
`D[i][j] = 0`, *dacă este inaccesibilă celula* `i,j`
`D[i][j] = D[i-1][j] + D[i][j-1]`, *în caz contrar*

Interpretarea formulei de mai sus este:
Orice mod de a ajunge în `i-1,j` se transformă într-un mod de a ajunge în `i,j`
Orice mod de a ajunge in `i,j-1` se transformă într-un mod de a ajunge in `i,j`

Initializare:
`D[1][1] = 1;`
Solutia este in `D[n][m]`

```
#include <fstream>
#define MOD 9901
#define DIM 1002
```

```
using namespace std;
ifstream fin ("cladire1.in");
ofstream fout("cladire1.out");

int A[DIM][DIM];
int D[DIM][DIM];
int n, m, b, k, i, j;

int main () {
    fin>>n>>m;
    fin>>b;
    for (k=1;k<=b;k++) {
        fin>>i>>j;
        A[i][j] = 1;
    }
    D[1][1] = 1;
    for (i=1;i<=n;i++) {
        for (j=1;j<=m;j++) {
            if (i == 1 && j == 1)
                continue;
            if (A[i][j] == 1)
                D[i][j] = 0;
            else
                D[i][j] = (D[i-1][j] + D[i][j-1]) % MOD;
        }
    }
    fout<<D[n][m];
}
```

Pentru a evita cazuri particulare considerăm că elementele din `D` de pe linia `0` și cele de pe coloana `0` sunt egale cu valoarea `0`.

Astfel calculez matricea `D` cu doua foruri.

Optimizare de memorie: la linia curentă din `D` am nevoie doar de linia anterioară din `D` și astfel pot ține doar doi vectori, `liniaCurenta` si `liniaAnterioara`.

Astfel, la pasul `i`, calculez orice element `j` din `liniaCurenta` astfel:
`liniaCurenta[j] = liniaCurenta[j-1] + liniaAnterioara[j]`

Când termin de calculat o linie, înainte să trec la următoarea, copiez vectorul `linieCurenta` în vectrorul `linieAnterioara`.

Soluția se va găsi în: `linieCurenta[m]`.

```
#include <fstream>
#define MOD 9901
#define DIM 1002
using namespace std;
```

```
ifstream fin ("cladire1.in");
ofstream fout("cladire1.out");

int A[DIM][DIM];
int linieAnterioara[DIM], linieCurenta[DIM];

int n, m, b, k, i, j;

int main () {
    fin>>n>>m;
    fin>>b;
    for (k=1;k<=b;k++) {
        fin>>i>>j;
        A[i][j] = 1;
    }
    linieAnterioara[1] = 1;
    for (j=2;j<=m;j++)
        if (A[1][j] == 1)
            linieAnterioara[j] = 0;
        else
            linieAnterioara[j] = linieAnterioara[j-1];

    for (i=2;i<=n;i++) {
        for (j=1;j<=m;j++) {
            if (A[i][j] == 1)
                linieCurenta[j] = 0;
            else
                linieCurenta[j] = (linieAnterioara[j] +
                                   linieCurenta[j-1]) % MOD;
        }
        for (j=1;j<=m;j++) {
            linieAnterioara[j] = linieCurenta[j];
        }
    }
    fout<<linieCurenta[m];

}
```

***Sumtri*** ([https://www.pbinfo.ro/probleme/385/sumtri](https://www.pbinfo.ro/probleme/385/sumtri))

*Date fiind numere naturale sub forma triunghiului de sub diagonala principală de la o matrice pătratică, se cere să determinăm un drum cu proprietățile:*

- *Pornește din elementul (*`1,1`*);*
- *La fiecare pas trece în elementul de sub sau în cel de sub, aflat în diagonală, spre dreapta;*
- *Are suma maximă;*

Citim valorile date într-o martice `A`, astfel:

```
fin>>n;
for (i=1;i<=n;i++)
      for (j=1;j<=i;j++)
            fin>>a[i][j];
```

Calculăm:
`D[i][j]` = *suma maximă a unui drum care pleacă din din* `1,1` *și ajunge în* `i,j`

Avem:
`D[1][1] = A[1][1];`
`D[i][j] = A[i][j] + maxim(D[i-1][j],D[i-1][j-1])`

Maximul de pe linia `n` a matricei `D` este soluția.

Observăm că la această problemă putem lucra tot cu ultimele două linii. Putem chiar renunța la matricea `A`, făcând operațiile în timpul citirii.

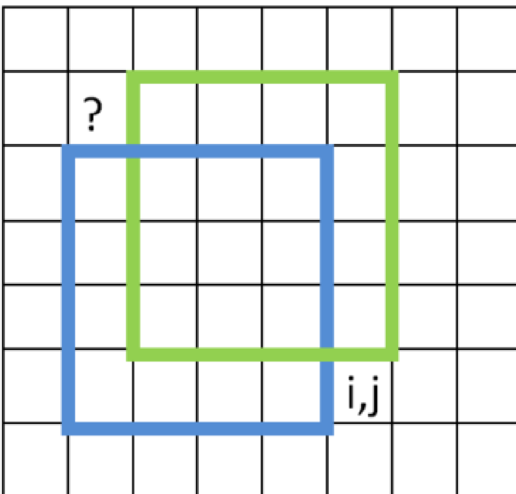
***Cuști*** ([https://www.infoarena.ro/problema/custi](https://www.infoarena.ro/problema/custi))

*Dată fiind o matrice cu elemente* `0` *și* `1`*, se cere să determinăm câte pătrate pline cu* `0` *sunt, pentru fiecare lungime posibilă a laturii, de la* `1` *până lungimea maximă a laturii vreunui pătrat.*

Vom rezolva problema în două etape.

*Etapa 1.*

Determinăm pentru fiecare poziție `i,j` *latura maximă a unui pătrat care are colțul din dreapta jos în acea poziție*. Stocăm aceste informații într-o matrice `D`.

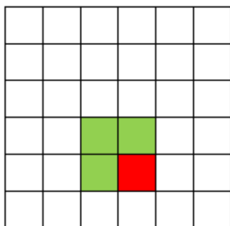


În figura de mai jos, pentru a afla latura maximă a unui pătrat cu colțul din dreapta jos în (`i,j`), adică `D[i][j]`, observăm că ne putem folosi de `D[i-1][j]` (pătratul cu chenarul verde) și de `D[i][j-1]`, cel cu chenarul albastru. Dacă au aceeași dimensiune și sunt pline de `1`, ne-ar mai interesa valoarea din celula marcată cu `?` a lui `A`.

Dacă cele două pătrate evidențiate nu au laturi egale, ne interesează valoarea minimă dintre ele (de exemplu dacă ar fi mai mic cel verde, nu ne-am mai putea extinde spre stânga pe linia de jos). Altă observație este că, pentru a acoperi și celula marcată cu `?` putem verifica valoarea `D[i-1][j-1]`.

Astfel: `D[i][j] = minim(D[i-1][j], D[i][j-1], D[i-1][j-1] + 1`, dacă `A[i][j] = 1` și `D[i][j] = 0` în caz contrar.

Aceasta este un caz tipic de dinamică simplă pe matrice, în care un element se calculează în funcție de cele `3` “vecine anterioare”, ca în figura următoare:



```
for (i=1;i<=n;i++)
     for (j=1;j<=n;j++)
          if (A[i][j] == 0)
               D[i][j] = 0;
          else
               D[i][j] = minim(D[i-1][j],D[i][j-],D[i-1][j-1])+1;
```

De exemplu, în această etapă, pentru o matrice `A` ca aceea din celula stângă de mai jos, obținem o matrice `D` ca aceea din celula dreaptă.

| A | D |
|---|---|
| 1 1 1 0 0<br>1 1 0 1 1<br>1 1 1 1 1<br>1 1 1 0 0<br>1 1 1 0 0 | 1 1 1 0 0<br>1 2 0 1 1<br>1 2 1 1 2<br>1 2 2 0 0<br>1 2 3 0 0 |

Am putea să folosim un vector de frecvență și, de fiecare dată când calculăm un element nenul `D[i][j]`, să facem `F[D[i][i]] ++;`

Am obține:
`F[1] = 11, F[2] = 6, F[3] = 1`

Observăm acum că `F[i]` = câte matrice pline cu `1` au latura `i` și sunt maxime dintre cele care au colțul dreapta jos în același loc.

*Etapa a doua*

Observăm că acolo unde scrie `3` trebuie numărată și o matrice de latură `2` și o alta de latură `1`. De asemenea, peste tot pe unde scrie `2` trebuie numărată o matrice de latură `1`.

Astfel, `F[3]` trebuie adunat la `F[2]`, `F[2]` așa actualizat trebuie adunal la `F[1]`. Deci ce avem de făcut este sume maxime în vectorul `F`, invers, de la poziția maximă nenulă până la poziția `1`.

Pentru a obtine solutia problemei, valorile anterioare `F[i]` trebuie adunate si la `F[i-1]` si la `F[i-2]` ...

```
for (i=maximDinD-1;i>=1;i --)
     F[i] += F[i+1];
```

Vectorul `F` este acum solutia problemei.

***Cel mai lung subșir cumun*** (https://www.infoarena.ro/problema/cmlsc)

Această problemă clasică am încadrat-o tot la categoria celor care necesită calcularea elementelor unei matrice și pentru un element este nevoie de cele `3` “vecine anterioare”.

*Avem deci un șir* `A` *cu* `n` *elemente și un șir* `B` *cu* `m` *elemente. Dorim să determinăm un șir care să fie subșir în* `A`*, subșir și în* `B` *și să aibă lungimea maximă.*

Vom calcula o matrice `D` cu semnificația:

`D[i][j]` = *lungimea maximă a unui subșir comun care folosește elemente dintre primele din* `A` *și dintre primele din* `B`.

Observăm că definiția noastră nu implică să avem un șubșir terminat obligatoriu cu `A[i]` și cu `B[j]`.

La calculul lui `D[i][j]` avem variantele

- `A[i] == B[j]`     În acest caz, `D[i][j] = 1 + D[i-1][j-1]`
- `A[i] != B[j]`, deci nu putem extinde cu un caracter comun un subșir existent, și în acest caz vom alege cea mai bună variantă dintre `D[i-1][j]` și `D[i][j-1]`

Lungimea maximă cerută este deci `D[n][m]`.

Suplimentar, la această problemă avem de determinat și efectiv un subșir. Reconstrucția unui astfel de drum se face de regulă invers, identificând pentru o poziție modul în care am ajuns acolo.

În acest caz, vom porni din poziția (n,m) , adică inițializăm i=n și j=m.

Dacă A[i] == B[j] înseamnă că am ajuns în poziția (i,j) extinzând subșirul cu un element, așadar stocăm la soluție elementul comun și decrementăm atât pe i cât și pe j.

Dacă A[i] este diferit de B[j] vom vedea de unde am ajuns în (i,j): Dacă D[i-1][j] > D[i][j-1] facem doar i --, în caz contrar doar j --. Pe acest caz nu stocăm nimic la soluție.

De remarcat că este este corectă reconstrucția soluției doar dacă mergem invers, din locul în care am obținut optimul final. Dacă am fi pornit de la început, nu am ști care dintre ramurile pe care mergem este cea care duce în final la soluție.

La implementarea de mai jos, în loc să stocăm elementele subșirului soluție (și apoi să le afișăm invers), am folosit o abordare recursivă, tipărind după autoapel.

```
#include <fstream>
using namespace std;

int d[1030][1030], a[1030], b[1030];
int n, m, i, j;

ifstream fin ("cmlsc.in");
ofstream fout("cmlsc.out");

void drum(int i, int j, int k) {
    if (k!=0) {
        if (a[i] == b[j]) {
            drum(i-1, j-1, k-1);
            fout<<a[i]<<" ";
        } else
            if (d[i-1][j] > d[i][j-1])
                drum(i-1, j, k);
            else
                drum(i, j-1, k);
    }
}

int main () {

    fin>>n>>m;
    for (i=1;i<=n;i++) {
        fin>>a[i];
    }
    for (i=1;i<=m;i++) {
        fin>>b[i];
    }

    for (i=1;i<=n;i++)
        for (j=1;j<=m;j++)
```

```
            if (a[i] == b[j]) {
                d[i][j] = 1 + d[i-1][j-1];
            } else {
                d[i][j] = max(d[i-1][j], d[i][j-1]);
            }

    fout<<d[n][m]<<"\n";
    drum(n, m, d[n][m]);
    return 0;
}
```

Iată și o observație care poate face implementarea și mai ușoară: în loc să facem dinamica din `1,1` și apoi să afișăm invers șirul recostituit începând din (`n`,`m`), am putea face dinamica începând din (`n`,`m`) și să tipărim apoi direct soluția pornind din (`1`,`1`). Asta pentru că, dacă ne imaginăm șirurile oglindite, subșirul comun maximal este același, obținut invers.

***Matrix*** (https://www.infoarena.ro/problema/matrix)

*Avem o matrice pătratică `A` (de dimensiune `n`), în care vom căuta apariții ale unei anumite submatrice pătratice date `B` (de dimensiune `m`, cu `m <= n`), doar că o potrivire a lui `B` se consideră corectă și dacă elementele sale sunt în submatricea din `A` în altă ordine decât în matricea dată `B`.*

Prima observație este că nu este necesară memorarea matricei `B` ci calcularea unui vector de frecvență în care notăm doar de câte ori apare un element în matricea `B`. Elementele celor două matrice sunt litere mici ale alfabetului, așadar vectorul de frecvență are `26` de componente.

Acum, dacă vom fixa o poziție (`i`,`j`) în `A`, pentru a verifica dacă submatricea de dimensiune `mXm` a lui `A`, cu colțul din dreapta jos în (`i`,`j`) este o "potrivire" a lui `B` în sensul din enunț, trebuie ca fiecare literă să apară în submatricea de testat de același număr de ori ca în vectorul de frecvență.

La prima vedere am avea nevoie de `26` de matrice `nXn`, pentru a face verificarea pentru fiecare literă în parte. În realiate vom folosi doar o matrice suplimentară ok, booleană, în care în `ok[i][j]` presupunem la început că este o potrivire a lui `B` cu colțul dreapta jos în `i`,`j`, și dacă testul eșurează pentru cel puțin o literă, anulăm presupunerea pentru poziția `i`,`j`.

Pentru o anume literă `ch`, facem testul în felul următor:

- Într-o matrice `D`, marcăm cu `1` doar pe pozițiile unde avem litera `ch`.
- Calculăm apoi sume parțiale pe matricea `D`
- Fixăm toate pozițiile (`i`,`j`) unde poate fi colțul dreapta jos al unei potriviri a lui `B`, și pentru a verifica dacă ch apare de câte ori trebuie acolo, testă, doar dacă `F[ch] = D[i][j] - D[i-m][j] - D[i][j-m] + D[i-m][j-m]`. Adică suma din

submatricea `mXm` cu colțul din dreapta jos în (`i,j`) unde elementele `1` sunt doar în poziții în care apare `ch` în `A`.

- Când trecem la altă literă dintre cele `26` folosim aceeași matrice `D` pe care o reconstruim.
- Pozițiile pe care rămâne `1` în `ok` sunt cele de potrivire.

Timpul de calcul este cel mult pătratic la fiecare dintre aceste etape.

```
#include <fstream>
using namespace std;

char c, a[1010][1010];
int d[1010][1010], sol, i, j, n, m, ok[1010][1010], f[130];
int main (){
    ifstream fin ("matrix.in");
    ofstream fout("matrix.out");

    fin>>n>>m;
    for (i=1;i<=n;i++)
        for (j=1;j<=n;j++)
            fin>>a[i][j];
    for (i=1;i<=m;i++)
        for (j=1;j<=m;j++) {
            fin>>c;
            f[c] ++;
        }

    for (i=m;i<=n;i++)
        for (j=m;j<=n;j++)
            ok[i][j] = 1;

    for (c = 'a'; c <= 'z'; c++){
        for (i=1;i<=n;i++)
            for (j=1;j<=n;j++)
                d[i][j] = (a[i][j] == c)
                        + d[i-1][j] + d[i][j-1] - d[i-1][j-1];
        for (i=m;i<=n;i++)
            for (j=m;j<=n;j++)
                if (d[i][j]-d[i-m][j]-d[i][j-m]+d[i-m][j-m] !=
f[c])
                    ok[i][j] = 0;
    }

    for (i=m;i<=n;i++)
        for (j=m;j<=n;j++)
            sol += ok[i][j];

    fout<<sol;
    return 0;
}
```

# II. Probleme la care optimul curent depinde de optime aflate oriunde în urmă.

La problemele de la capitolul I, când structura aleasă a fost un vector, în formula recurenței, `D[i]` depindea de `D[i-1]` sau cel mult și de `D[i-2]`. În cazul matricelor, `D[i][j]` depindea de regulă de cele "3 *elemente vecine anterioare*", adică `D[i-1][j]`, `D[i][j-1]`, `D[i-1][j-1]`. Consecința era că la vectori obțineam de regulă un algoritm de calcul liniar, iar la matrice unul pătratic (practic determinăm fiecare element al structurii în timp constant).

În acest capitol vom prezenta câteva probleme în care elementul curent are nevoie de toate cele aflate înaintea lui.

***Subșirul crescător maximal*** (https://www.infoarena.ro/problema/scmax)

*Am un șir de valori, naturale, să se găsească un subșir crescător de lungime maximă.*

De regulă suntem tentați să încercăm diverse abordări greedy, dar la o analiză atentă se poate arăta că nu avem soluție cu această tehnică. Avem însă mai multe abordări folosind programarea dinamică.

*Soluția cu timp de calcul de ordin* $n^2$

`D[i]` = *lungimea maximă a unui subșir crescător terminat la poziția* `i` *(deci care conține pe* `V[i]`*)*. Pentru a găsi relația de recurență, ne gândim că vrem să alipim pe `V[i]` la un subșir format deja cu elemente aflate pe poziții anterioare. Așadar ne interesează poziții `j`, cu `j < i` și `V[j] < V[i]` (asta în cazul în care dorim să determinăm un subșir strict crescător, sau `V[j] <= V[i]`, dacă sunt permise în subșir și elemente vecine egale).

Dintre toate pozițiile `j` care îndeplinesc această proprietate, alegem evident pe cele care au `D[j]` maxim. Astfel obținem:

`D[i]` = `1` + `maxim`(`D[j]`, *pentru* `j` *de la* `1` *la* `i-1` *și* `V[j]` < `V[i]`)

Pentru a construi și o soluție, când calculăm `D[i]`, reținem în alt vector `T`, poziția din fața lui `i` unde am determinat valoarea maximă care a contribuit la `D[i]`.

Apoi, după ce determinăm valoarea maximă din `D`, și poziția sa, ne putem întoarce prin pozițiile obținute începând de acolo, prin intermediul lui `T`, și obținem soluția. Este necesară o abordare recursivă, cu afișare la întoarcere, sau stocarea pozițiilor determinate într-un vector care apoi se afișează invers.

```
#include <fstream>
#define DIM 100010
using namespace std;
int v[DIM], t[DIM];
```

```
int d[DIM];
int n, i, j, maxim, sol, p, u, k;

ifstream fin ("scmax.in");
ofstream fout("scmax.out");

void drum(int u) {
    if (u!=0) {
        drum(t[u]);
        fout<<v[u]<<" ";
    }
}

int main () {
    fin>>n;
    for (i=1;i<=n;i++)
        fin>>v[i];

    d[1] = 1;
    for (i=2;i<=n;i++) {
        maxim = 0;
        for (j=1;j<i;j++)
            if (v[j] < v[i] && d[j] > maxim) {
                maxim = d[j];
                p = j;
            }
        d[i] = maxim + 1;
        if (d[i] != 1)
            t[i] = p;
        else
            t[i] = 0;

        if (d[i] > sol) {
            sol = d[i];
            u = i;
        }
    }

    fout<<sol<<"\n";
    drum(u);
    return 0;
}
```

Revenind la construcția drumului (a subșirului efectiv) facem câteva observații utile în general când este nevoie să reconstituim drumul.

Ne putem imagina elementele vectorului ca fiind nodurile unui graf iar când calculăm valoarea `D` a unui nod, notăm ca părinte al său (în `T`) nodul anterior cu `D` maxim din care am obținut valoarea sa.

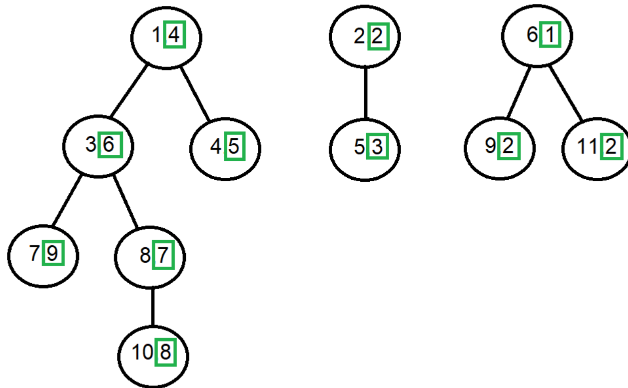
Se obține astfel o pădure de arbori și orice traseu între un nod și rădăcină este un drum minim de la rădăcină la acel nod.

Pentru un nod dat putem reconstitui drumul doar plecând invers, în sus pe arbore, de la el la rădăcină, pentru că dacă am porni din rădăcină nu am ști pe ce cale trebuie să mergem că să ajungem la nodul dorit.

Vorbim de "*pădure de arbori*" și nu doar de unul singur, pentru că fiecare nod care nu are pe nimeni mai mic în față va deveni rădăcina unui astfel de arbore.

Să vedem un exemplu (ne gândim la subșir strict crescător):

```
Indice: 1 2 3 4 5 6 7 8 9 10 11
     V: 4 2 6 5 3 1 9 7 2  8  2
     D: 1 1 2 2 2 1 3 3 2  4  2
     T: 0 0 1 1 2 0 3 3 6  8  6
```




***Numărul de subșiruri maximale*** (https://www.infoarena.ro/problema/subsiruri)

La această problemă, pe lângă valoarea `D`, care ne indică lungimea maximă a vreunui șir terminat la fiecare poziție, mai ținem și un șir `S` cu semnificația:

`S[i]` = *numărul de șiruri de lungime maximă terminate la poziția* `i` *(adică de lungime* `D[i]`*)*

Pentru a actualiza valoarea `S[i]`, atunci când calculăm `D[i]`, dacă avem mai multe poziții anterioare `j` care oferă maximul pentru `D[i]`, vom aduna la `S[i]` toate aceste valori `S[j]`.

Astfel această problemă devine un bun exemplu de a avea în același loc și o cerință de optim și una de numărare.

*Subșir crescător maximal cu timp de calcul de ordin n log n*

Vom schimba modul în care definim structura de optime.

Astfel după analizarea primelor `i` elemente din `V`, dacă, presupunând că lungimea maximă a vreunui subșir determinat până acum este m, vom ține într-un vector următoarea informație:

Pentru fiecare indice `j` de la `1` la `m`, ne interesează valoarea minimă în care se poate termina un subșir crescător de lungime `j`. La implementare vom ține de fapt indicele din `V` al acestei valori minime (indicele este util pentru reconstrucția drumului, și oricum prin intermediul lui avem accest și la valoare).

Vom avea deci:

`D[j]` = *indicele din* `V` *al celei mai mici valori în care se poate termina un subșir de lungime* `j`.

Prima observație este că valorile corespunzătoare indicilor din `V` sunt în ordine crescătoare (evident, prin reducere la absurd: dacă `7` ar fi valoarea minimă de final a unui subșir de lungime `2`, nu ar putea fi de exemplu `5` (`<7`) valoarea minimă în care se termină un subșir de lungime `3`).

O altă observație: indicii din `D` nu sunt neapărat în ordine crescătoare, adică soluția nu este dată, de exemplu de valorile finale din `D`.

Iată un exemplu.

```
            i
5  9  3  8  5  7  2  …
```

Așadar, suntem la poziția `i=7`, adică am analizat primele `7` elemente din șirul dat (pot să urmeze și altele). Șirul D ar fi:

```
Indice:    1    2     3
     D:  7(2)  5(5)  6(7)
```

Între paranteze am pus valorile din `V` (care sunt în ordine crescătoare) și înainte am pus indicii lor din `V`, care nu sunt în ordine crescătoare (de exemplu, e clar că valoarea minimă întâlnită este cea mai mică valoare în care se termină un subșir de lungime `1`, iar ea poate fi oriunde în șir, chiar și la final …)

Să presupunem că următoarea valoare care apare în șir este `6`

```
              i
5  9  3  8  5  7  2  6
```

E clar că ea va contribui la un subșir de lungime `3` (de exemplu `3 5 6`). Până acum valoarea minimă în care se termina un subșir de lungime `3` era `7`. Acum ea va deveni `6`. Această actualizare trebuie făcută în `D` și observăm că este suficient să căutăm binar prima poziție din `D` unde se află o valoare mai mare decât `6` și să înlocuim această poziție cu indicele corespunzător lui `6`. Dacă nu există o astfel de poziție, înseamnă că noua valoare provoacă o nouă lungime maximă, așa că o adăugăm la `D` (mărim `m` cu `1` și facem `D[m] = i`).

Lungimea maximă determinată este valoarea lui `m` după ce vom fi analizat în ordine, toare cele `n` elemente din `V`.

```
#include <fstream>
#define DIM 100010
using namespace std;

int V[DIM], T[DIM], D[DIM];
int n, m, i, p, u, mid;
/**
D[i] = indicele din V al celei mai mici valori in care se poate
termina un sir de lungime i (de fapt D[i] = poZitia din V a unei
astfel de valori) folosind elementele deja parcurse
**/

ifstream fin("scmax.in");
ofstream fout("scmax.out");

void sol(int i) {
    if (i) {
        sol(T[i]);
        fout<<V[i]<<" ";
    }
}

int main() {
    fin>>n;
    for (i=1;i<=n;i++)
        fin>>V[i];

    m = 1;
    D[1] = 1;
    for (i=2;i<=n;i++) {
        p = 1; ///caut pozitia ultimei valori D[poz] din
               /// D pentru care V[i] > V[ D[poz] ]
        u = m;
        while (p<=u) {
            mid = p+(u-p)/2;
            if (V[ D[mid] ] >= V[i])
                u = mid - 1;
            else
                p = mid + 1;
        }
        /**
        p ramane pe pozitia primei valori
        mai mari sau egale cu ce caut
        u ramane pe pozitia ultimei valori
        strict mai mici decat ce caut
        **/
        if (p > m) {
            m++;
```

```
            D[m] = i;
            T[i] = D[p-1];
        } else {
            D[p] = i;
            T[i] = D[p-1];/// extind sirul terminat pe pozitia 
p-1(u)
        }
    }
    fout<<m<<"\n";
    sol(D[m]); ///in T este un vector de tati pentru indici din V
    return 0;
}
```

Pentru reconstrucția soluției avem nevoie de un șir de indici *tata*. Am observat că indicii valorilor prezente la final în `D` nu reprezintă un subșir de afișat (ei nefiind neapărat în ordine crescătoare). Problema se rezolvă astfel: în momentul în care valoarea curentă `V[i]` își găsește poziția `p` unde trebuie marcată în `D`, în urma căutării binare, facem `T[i]` = `D[p-1]`; Semnificația este că valoarea de pe poziția `p-1` este un indice din `V`, al unei valori mai mici decât `V[i]`, întâlnită înaintea lui `V[i]` și care reprezintă valoare de final pentru un subșir de lungime cu `1` mai mică decât `p`, cea determinată cu final în `V[i]`. Apoi, facem reconstituirea drumului clasic, cu o funcție recursivă, ca și în alte cazuri.

*Subșir crescător maximal folosind AIB*

Există și posibilitatea de a determina subșirul crescător de lungime maximă folosind un arbore indexat binar.

În sursa următoare mai întâi normalizăm valorile din șirul dat. Astfel, chiar dacă valorile din șirul dat pot fi foarte mari, odată cu normalizarea ele vor fi maxim agale cu `n`. Astfel, vom ține un `AIB` în care indicii vectorului corespund valorilor normalizate. Când vom întâlni o valoare normalizată `x`, vom face update la poziția `x` în `AIB` cu lungimea maximă a unui subșir terminat în elementul de pe poziția lui `x` din vectorul `V`. Astfel, dacă înainte de update pentru valoarea curentă `x`, interogăm `AIB` la poziția `x-1`, obținem practic maximul parțial, care, conform celor scrise mai sus, va fi lungimea maximă a unei secvențe din șirul dat, dintre valorile mai mici decât `x` și întâlnite, evident înaintea lui `x`.

Această abordare implică deci mai mulți pasi și algoritmi și nu este cea preferată față de implementarea în `n log n` prezentată anterior, dar noi o prezentăm pentru diversitate și legarea noțiunilor.

Iată o sursă pe această idee, din care, cu siguranță să învață mult:

```
#include <fstream>
#include <algorithm>
#define DIM 100001
```

```
using namespace std;

int V[DIM], W[DIM], A[DIM], i, p, N, Maxim, Max, K, P[DIM], T[DIM],
pozMax, poz;

ifstream fin ("scmax.in");
ofstream fout("scmax.out");

void drum(int p) {
	if (p) {
		drum(T[p]);
		fout<<W[p]<<" ";
	}
}

inline int lsb(int x) {
	return x & -x;
}

int cauta(int x) {
	int p = 1, u = K, m;
	while (p<=u) {
		m = (p+u) >> 1;
		if (W[m] == x)
			return m;
		if (W[m] < x)
			p = m+1;
		else
			u = m-1;
	}
}

int query (int p, int &pp) {
	int maxim = 0;
	pp = 0;
	for (;p;p-=lsb(p)) {
		if (maxim < A[p]) {
			maxim = A[p];
			pp = P[p];
		}
	}
	return maxim;
}

void update(int p, int v) {
	int aux = p;
	for (;p<=K;p+=lsb(p))
		if (A[p] < v) {
			A[p] = v;
			P[p] = aux;
		}
```

```
}

int main() {
      fin>>N;
      for (i=1;i<=N;i++) {
            fin>>V[i];
            W[i] = V[i];
      }

      sort(W+1, W+N+1);
      K = 1;
      for (i=2;i<=N;i++)
            if (W[i] != W[K])
                  W[++K] = W[i];


      for (i=1;i<=N;i++) {
            p = cauta(V[i]);
            int Max = query(p-1, poz);
            if (1+Max > Maxim) {
                  Maxim = 1 + Max;
                  pozMax = p;
            }
            update(p, 1+Max);
            T[p] = poz;
      }

     fout<<Maxim<<"\n";
      drum(pozMax);
      return 0;
}
```

***Cuburi3*** ([https://www.infoarena.ro/problema/cuburi3](https://www.infoarena.ro/problema/cuburi3))

*Pe scurt, această problemă ne cere să determinăm un subșir crescător maximal pentru un șir de perechi. Subșirul determinat trebuie să fie crescător și când ne uităm la șirul elementelor de pe prima poziție din fiecare pereche, și când ne uităm la șirul elementelor de pe a doua poziție din fiecare pereche.*

Observația estențială este că, dacă sortăm șirul dat după una dintre valori (după cele de pe prima poziție sau după cele de pe a doua poziție), ne rămâne efectiv de aplicat algoritmul de subșir crescător maximal clasic, pe un șir de valori (celelalte, care nu au fost luate în calcul la sortare).

***Joc13*** (https://www.infoarena.ro/problema/joc13)

Avem o matrice cu `2` linii și `n` coloane și trebuie determinat un drum de sumă maximă care pleacă din celula (`1,1`), ajunge în celula (`2,n`), nu trece de două ori prin aceeași celulă și face maxim `k` pași consecutivi pe aceeași linie.

Observația principală este că drumul nu trebuie să se "întoarcă". Adică pașii nu pot fi făcuți de pe o coloană pe una cu indice mai mic. Dacă s-ar întâmpla acest lucru s-ar ajunge la celule care să fie vizitate de mai multe ori.

Ne putem gândi să calculăm `D[i][j]` = *suma maximă a valorilor din celulele vizitate pentru un drum care pleacă din (*`1,1`*) și ajunge în (*`i,j`*).*

Observăm că semnificație structurii alese este cumva în spiritul enunțului care chiar asta cere: drumul minim până la o anume celulă, iar noi alegem o structură în care indicii identifică celula în care ajungem iar valoarea identifică informația cu care ajungem în acea celulă.

Pentru a ajunge în celula (`i,j`) putem păși fie din cealaltă celulă de pe aceeași coloană, fie de pe celula de pe aceeași linie și de pe coloana anterioară. Însă în acest al doilea caz trebuie să ne asigurăm că nu facem mai mulți pași consecutivi la dreapta. Astfel, ne dăm seama că ne mai trebuie un indice la dinamică și vom avea:

`D[i][j][k]` = *suma maximă a celulelor de pe un drum care pleacă din (*`1,1`*), ajunge în (*`i,j`*) și a făcut exact* `k` *pași la dreapta pe rândul* `i` *înainte să ajungă la coloana j.*

|   | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |   |
|---|---|---|---|---|---|---|---|---|---|
| 1 |   |   |   |   |   |   |   |   |   |
| 2 |   |   |   |   |   |   |   |   |   |

De exemplu, pentru dacă drumul de mai sus ar fi cel mai bun mod de a ajunge în celul `1,8` și să fi făcut ultima dată la dreapta `3` pași, această informație o păstrăm în `D[1][8][3]`.

Pentru a calcula așadar `D[i][j][k]` ne trebuie

- Maxim dintre:

`D[i][j-1][1]` + *suma de linia* `i` *de la coloana* `j` *la coloana* `j`*,*
`D[i][j-2][2]` + *suma de linia i de la coloana* `j-1` *la coloana* `j`*,*
`D[i][j-3][3]` + *suma de linia i de la coloana* `j-2` *la coloana* `j`*,*
*…*
`D[i][j-k][k]` + *suma de linia i de la coloana* `j-k+1` *la coloana* `j`*,*

În sursa de mai jos mai folosim și artificii de sume și maxime parțiale.

```
#include <fstream>
#define INF 5000010
#define DIM 5002
using namespace std;

int a[2][DIM], s[2][DIM], d[2][DIM][11];
int n, k, i, j, x;
int main () {
    ifstream fin("joc13.in");
    ofstream fout("joc13.out");

    fin>>n>>k;
    for (i=0;i<=1;i++)
        for (j=1;j<=n;j++) {
            fin>>x;
            a[i][j] = a[i][j-1] + x;
        }
    d[0][1][1] = a[0][1];
    s[0][1] = a[0][1];

    d[0][2][2] = a[0][2];
    s[0][2] = a[0][2];

    d[1][2][2] = a[1][2] + a[0][1];
    s[1][2] = d[1][2][2];

    for (i=3; i<=n; i++) {
        s[0][i] = -INF;
        s[1][i] = -INF;
        for (j=2; j<=k && i-j+1 >= 1; j++) {
            d[0][i][j] = s[1][i-j+1] + a[0][i] - a[0][i-j];
            s[0][i] = max(s[0][i], d[0][i][j]);
            d[1][i][j] = s[0][i-j+1] + a[1][i] - a[1][i-j];
            s[1][i] = max(s[1][i], d[1][i][j]);
        }
    }
    fout<<max(s[1][n], s[0][n] + a[1][n] - a[1][n-1])<<"\n";
    return 0;
}
```

***Pali*** (https://www.infoarena.ro/problema/pali)

*Problema cere să partiționăm un șir în numar minim de secvențe palindromice.*

*Soluția 1*

Vom calcula `D[i]` = *numarul minim de palindroame pentru a acoperi elementele de la pozitia 1 la poziția* `i`. Soluția s-ar afla în `D[n]`

Pentru a calcula `D[i]`:

```
D[1] = 1;
for(i=2;i<=n;i++) {
      D[i] = n+1;
      /// numărul maxim posibil de palindroame nu
     /// poate fi mai mare decât numărul de elemente din șir

      for (j=i;j>=1;j --)
           if (secvența de la j la i e palindrom)
                 D[i] = min(D[i], 1 + D[j-1]);
                 /// ne imaginăm că am mai adaugat un
                 /// palindrom la final
```

Timp de calcul este de ordin $n^3$ deoarece verificarea dacă secvența de la `j` la `i` este palindromică necesită și ea timp de calcul liniar.

*Soluția 2*

*Păstrăm semnificația lui* `D` *dar schimbăm modul de a acționa: în loc să considerăm* `i` *ca fiind finalul unui palindrom, vom considera* `i` *ca fiind mijlocul unui palindrom. Vom considera atât cazul cu* `i` *mijloc de palindrom de lungime impară cât și cazul cu* `i` *mijloc de palindrom de lungime pară.*

Astfel, dacă avem un palindrom care începe la poziția `st`, se termină la poziția `dr` și este centrat în `i`, putem face update la poziția `D[dr]` cu valoarea `1 + D[st-1]`

Optimizarea vine din faptul că, plecând cu palindroamele centrate în `i` de la lungime mică la lungime mai mare, dacă trecem de la secvența `[st, dr]` la secvența `[st-1,dr+1]`, avem de verificat doar dacă `2` elemente sunt egale (`a[st-1]` și `a[dr+1]`). Totodată, când condiția nu mai este îndeplinită, nu mai are sens să încercăm extinderea. În continuare prezentăm o soluție pe această idee, care are timp de calcul de ordin $n^2$. Este util de remarcat că, fiind la o anume poziție `i`, nu facem actualizări neapărat la poziția `i`, ci la diverse poziții de după `i` (notate de noi `dr`), și în funcție de diverse valori calculate deja anterior la poziții de dinaintea lui `i`.

```
#include <fstream>
#include <cstring>
using namespace std;

char s[5010];
int n, st, dr;
int D[5010];
int main () {

    ifstream fin ("pali.in");
```

```
    ofstream fout("pali.out");

    fin>>s+1;
    n = strlen(s+1);
    D[0] = 0;
    for (int i=1;i<=n;i++)
        D[i] = n;
    for (int i=1;i<=n;i++) {
        st = i;
        dr = i;
        D[i] = min(D[i], D[i-1]+1);
        while (st-1 >= 1 && dr + 1 <= n && s[st-1] == s[dr+1]) {
            st--,dr++;
            D[dr] = min(D[dr], D[st-1] + 1);
        }
        st = i+1;
        dr = i;
        while (st-1 >= 1 && dr + 1 <= n && s[st-1] == s[dr+1]) {
            st--,dr++;
            D[dr] = min(D[dr], D[st-1] + 1);
        }
    }
    fout<<D[n];
    return 0;
}
```

***Scara3*** ([https://www.infoarena.ro/problema/scara3](https://www.infoarena.ro/problema/scara3))

În sursa prezentată în continuare am notat:

- `D[i]` = *numărul minim de pași pentru a ajunge pe treapta* `i`
- `C[i]` = *costul minim pentru a ajunge pe treapta* `i` *în* `D[i]` *pași*

Ca și la problema anterioară, la această problemă, ne imaginăm că suntem la o poziție `i`, la care am calculat deja valorile optime și facem actualizări ale *optimelor de după poziția* `i`, în funcție de optimul de la poziția `i`. Tratăm cele `3` moduri în care putem urca.

```
#include <fstream>
#define DIM 1210
using namespace std;
ifstream fin  ("scara3.in");
ofstream fout ("scara3.out");
int D[DIM], C[DIM], A[DIM], E[DIM];
/**
D[i] = numarul  minim de salturi sa ajung pe treapta i
       pornind de pe treapta 0
C[i] = costul minim de a ajunge pe treapta i cu D[i] salturi
**/
```

```
int n, k, L, x, y, i, j;

int main()
{
    fin>>n>>k;
    for (i=1;i<=k;i++) {
        fin>>x>>y;
        A[x] = y;
    }
    fin>>L;
    for (i=1;i<=L;i++) {
        fin>>x>>y;
        E[x] = y;
    }

    D[0] = 0;
    for (i=1;i<=n;i++)
        D[i] = n+1;
    for (i=0;i<n;i++) {
        // imi imaginez ca sunt pe treapta i si vad unde pot sari
        if (D[i+1] > D[i]+1) {
            D[i+1] = D[i] + 1;
            C[i+1] = C[i];
        } else
            if (D[i+1] == D[i]+1 && C[i+1] > C[i]) {
                C[i+1] = C[i];
            }
        if (A[i] != 0) {
            for (j=i+1;j<=n && j<=i+A[i]; j++) {
                if (D[j] > D[i]+1) {
                    D[j] = D[i] + 1;
                    C[j] = C[i];
                } else
                    if (D[j] == D[i]+1 && C[j] > C[i]) {
                        C[j] = C[i];
                    }
            }
        }
        if (E[i] != 0) {
            for (j=i+1;j<=n && j<=i+2*E[i]; j++) {
                if (D[j] > D[i]+1) {
                    D[j] = D[i] + 1;
                    C[j] = C[i] + (j-i+1)/2;
                } else
                    if (D[j] == D[i]+1 && C[j] > C[i] + (j-i+1)/2)
{
                        C[j] = C[i] + (j-i+1)/2;
                    }
            }
        }
```

```
    }
    fout<<D[n]<<" "<<C[n]<<"\n";
    return 0;
}
```

# III. Problema rucsacului

Pornim de la următoarea problemă: *date fiind* `n` *numere naturale și o valoare* `S`*, să se determine o submulțime a mulțimii celor* `n` *numere cu proprietatea că suma elementelor din submulțime este* `S`.

Prima idee este să generăm toate submulțimile mulțimii date și să vedem astfel dacă vreuna dintre ele are sumă `S`. Acesta este un algoritm exponențial, de tip backtracking și nu s-ar încadra într-un timp rezonabil decât pentru valori relativ mici ale lui `n`, maxim ceva peste `20`. Aceasta este deci o soluție corectă, dar nepractică pentru valori mai mari ale lui `n`.

Altă idee care se mai aduce în discuție este o abordare greedy: de exemplu sortăm crescător valori și încercăm să alegem cât timp am suma mai mică decât `S`. Problema este că s-ar putea să ajung la un element care sare peste `S` și atunci ar trebui să mă întorc, poate un pas, poate mai mulți, și să fac altă alegere. Astfel, dăm tot într-un algoritm de tip backtracking.

Să discutăm o soluție pentru cazul în care valoarea lui `n` este maxim `10000`, valorile elementelor din șir sunt naturale, iar `S` este maxim `10000`, de asemenea natural.

Vom folosi un vector de frecvență `D` în care notăm:

`D[i]` = `1` *dacă putem obține suma* `i`
`D[i]` = `0` *dacă nu putem obține suma* `i`
*Pentru început toate elementele lui* `D` *sunt* `0`*.*

Vom analiza elementele din șirul dat (să îl numim `V`) într-o ordine oarecare, de exemplu în ordinea citirii.

Luând în calcul doar primul element, putem obține o singură “sumă” adică chiar valoarea lui. Practic, marcăm `D[V[1]]` cu `1`.

Luând în calcul primele două elemente, vom avea `1` în `D` astfel:

```
D[V[1]] = 1
D[V[2]] = 1
D[V[1]+V[2]] = 1;
```

Să încercăm acum să generalizăm:

- am parcurs primele `i-1` elemente din `V` și avem marcat în `D` cu `1` la pozițiile care se pot obține ca sume cu oricâți termeni dintre primii `i-1`.
- Când luăm în calcul pe `V[i]`, căutăm în `D` elemente `j` cu `D[j] = 1` și, pentru fiecare dintre ele, facem `D[j + V[i]] = 1`, și în plus `D[V[i]] = 1`

Iată un exemplu:

`V: 3 2 6 8`

Înainte să analizăm pe `6`, folosind pe `3` și pe `2`, obținem sumele: `2`, `3` și `5`

Astfel, șirul `D` arată:

| Indici: | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| D: | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Când analizăm pe `6`, găsim `1` în pozițiile `2`, `3` și `5` și vom căuta să marcăm cu `1` în pozițiile: `2+6`, `3+6`, `5+6` și `6`.

Obținem:

| Indici: | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| D: | 0 | 0 | 1 | 1 | 0 | 1 | **1** | 0 | **1** | **1** | 0 | **1** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Se conturează următorul algoritm:

```
for (i=1;i<=n;i++) {
      D[V[i]] = 1;
      for (j=1;j<=S;j++)
            if (D[j] == 1)
                  D[j+V[i]] = 1;
}
```

Prima observație este că există riscul să considerăm sume în care aceeași valoare se adună de mai multe ori (de exemplu dacă avem `V[i] = 3` iar în `D` avem `1` la poziția `2`, noi căutând valori `1` crescător după indici (în codul de mai sus), găsim `D[2] = 1`, facem apoi `D[2+3] = 1`, apoi vom întâlni și pe acest `1` de pe poziția `5` setat de noi și efectul este că facem și `D[5+3] = 1`, semnificația fiind că am considerat pe `3` de două ori la suma `8`).

Pentru a preveni acest lucru, singurul lucru pe care îl avem de făcut este să căutăm pozițiile cu `1` din `D` în ordine descrescătoare a indicilor. Astfel, vom face noi marcaje mai la dreapta, dar nu le vom mai întâlni la parcurgerea pentru `V[i]` curent, care se face descrescător după indicii din `D`.

Codul devine:

```
for (i=1;i<=n;i++) {
      for (j=S;j>=1;j++)
```

```
          if (D[j] == 1)
               D[j+V[i]] = 1;
     D[V[i]] = 1;
}
```

Observăm că a fost necesar să marcăm în `D` suma pentru `V[i]`, după `for`.

Legat de marcarea sumei formată din câte un termen, pe care noi o facem separat, avem un truc care ne permite să scăpăm de tratarea diferențiată a acestui caz: marcăm încă de la început în `D` cu `1` pe poziția `0` și de câte ori căutăm indici cu `1` în `D`, mergem până la `0`. Astfel, vom mai face și o marcare în poziția `0+V[i]`, care este tocmai ce ne trebuie.

```
D[0] = 1;
for (i=1;i<=n;i++) {
     for (j=S;j>=0;j++)
          if (D[j] == 1)
               D[j+V[i]] = 1;
}
```

La prezentarea anterioară ne-am concentrat pe ideea de bază a algoritmului. Bineînțeles că trebuie ținut cont de diverse situații: de exemplu putem ignora valorile din `V` mai mari ca `S`, sau valorile `j+V[i]` care ajung să fie mai mari decât `S`.

Pe exemplul anterior, dacă am lua în calcul și următoarea valoare, `8`, observăm că ajungem să generăm din nou unele sume marcate deja cu `1` (cum ar fi suma `8` sau suma `11`).

Așadar, numărul de submulțimi posibile este de ordin $2^n$, dar noi avem nevoie doar de sume până la `S`, care trebuie să fie mai mic, pentru ca algoritmul să aibă valoare practică.

În concluzie, generăm toate sumele mai mici sau egale cu `S`, folosind valori din șirul `V`, cu timp de calcul de ordin `n*s`.

O altă observație: dacă al doilea `for` îl facem crescător, ca în varianta inițială, rezolvăm practic o altă problemă: având un șir de numere, să se determine toate sumele ce se pot obține cu valori din el, știind că este permis să folosim o valoare de mai multe ori.

În oricare dintre cazuri, se pune și problema determinării efectiv a unei submulțimi de sumă `S` (până acum noi doar am verificat că putem obține suma `S`). Pentru acest lucru, de fiecare dată când facem o marcare de forma `D[j+V[i]] = 1`, într-un vector `T` putem marca `T[j+V[i]] = i`. Practic semnificația este: `T[i]` = *indicele ultimului termen folosit pentru a obține suma* `i`. Chiar dacă facem căutarea valorilor `1` descrescător, există riscul de a nota un termen de mai multe ori la o sumă. Pentru a evita acest lucru, vom seta în `T` ca mai sus doar la poziții `j+V[i]` când se setează pentru prima dată la `1`.

```
D[0] = 1;
for (i=1;i<=n;i++) {
      for (j=S;j>=0;j++)
            if (D[j] == 1)
                  if (D[j+V[i]] == 0) {
                        D[j+V[i]] = 1;
                        T[j+V[i]] = i;
                  }
}
void drum(int suma) {
      if (suma != 0) {
            drum(suma-V[ T[suma] ];
            cout<<V[ T[suma] ];
      }
}
```

Un apel de forma `drum(S)` al funcției de mai sus ar afișa o submulțime de sumă `S` (noi am ales aici să o tipărim în ordinea în care termenii apar în `S`, adică am mers invers pornind din `S` și am făcut afișarea după autoapel).

***Jocul*** (https://www.infoarena.ro/problema/jocul)

*Problema cere să împărțim un șir dat în două subșiruri astfel încât sumele valorilor din cele două subșiruri să fie cât mai apropiate.*

Oservația esențială este că, dacă notăm cu `S` suma elementelor șirului dat, una din cele două sume determinate va fi mai mică sau egală decât `S/2` și cealaltă mai mare sau egală decât `S/2`. Astfel, generăm toate sumele posibile mai mici sau egale decât `S/2` și o alegem pe cea mai mare dintre ele, `s`. Cealaltă va fi `S-s`.

***Rucsac*** (https://www.infoarena.ro/problema/rucsac)

Această problemă aduce un factor în plus: fiecare element din șir mai are asociată și o valoare, câștigul pe care îl avem dacă o alegem. Cerința este să determinăm o submulțim cu suma maxim egală cu `G` și de câștig maxim.

În acest caz vom schimba puțin semnificația valorilor din `D`:

`D[i]` = *câștigul maxim pentru a obține suma* `i` *(sau, în contextul problemei, pentru a încărca în rucsac obiecte de valoare* `i`*)*. Vom calcula valorile din D pentru indici mai mici sau egali cu `i`.

```
using namespace std;
int D[10010];
///D[i] = profitul maxim prin care pot obtine greutatea i

int g[5001], p[5001];
int sol, i, j, n, G;
int main () {
    ifstream fin ("rucsac.in");
    ofstream fout("rucsac.out");
    fin>>n>>G;
    for(i=1;i<=n;i++)
        fin>>g[i]>>p[i];
    D[0] = 1;
    for (i=1;i<=n;i++)
        for (j=G; j>=0; j--)
            if (D[j] != 0 && j+g[i] <= G) {
                D[ j+g[i] ] = max( D[j+g[i]], D[j] + p[i] );
                sol = max(sol, D[j+g[i]]);
            }
    fout<<-1 + sol;
    return 0;
}
```

***Energii*** (https://www.infoarena.ro/problema/energii)

Problema aceasta cere să determinăm o sumă cel puțin egală cu `G` și de *cost minim* (de asemenea, elementele date sunt caracterizte prin două informații, *valoare* și *cost*).

Ca și la problema clasică, vom calcula `D[i]` = *costul minim de a obține suma valorilor ca fiind* `i`. Problema este că pare că as avea nevoie să calculez valori pentru indici din `D` mai mari decât `G`, și e dificil de prezis cât de mult mă pot duce cu acești indici. Însă, o observație simplă, reduce considerabil efortul: Pentru toți indicii mai mari sau egali ca `G` noi vom actualiza direct în `D[G]`.

Avem deci:
`D[i]` = *costul minim să obțin chiar suma* `i` *(pentru* `i<G`*)*
`D[i]` = *costul minim să obțin o sumă mai mare sau egală cu* `i` *(pentru* `i = G`*)*

Sursa:

```
#include <fstream>
#define INF 50000010
using namespace std;

int D[5010];
int C[1010], E[1010];

int n, G, i, j, k;
```

```
int main () {
    ifstream fin("energii.in");
    ofstream fout("energii.out");

    fin>>n>>G;
    for (i=1;i<=n;i++) {
        fin>>E[i]>>C[i];
    }

    for (i=1;i<=G;i++)
        D[i] = INF;

    for (i=1;i<=n;i++) {
        for (j=G-1;j>=0;j--)
            if (D[j] != INF) {
                k = j+E[i];
                if (k > G)
                    k = G;
                if (D[k] > D[j] + C[i])
                    D[k] = D[j] + C[i];
            }
    }
    if (D[G] == INF)
        fout<<-1;
    else
        fout<<D[G];
    return 0;
}
```

***Șanț*** (https://www.infoarena.ro/problema/sant)

*Avem* `C` *categorii de muncitori și din fiecare categorie putem angaja oricâți muncitori. Un muncitor din categoria* `i` *sapă exact* `L[i]` *metri de șanț și costă exact* `P[i]` *lei. Dorim să angajăm exact* `N` *muncitori care să sape exact* `S` *metri de șanț. Dorim să determinăm costul minim cu care să realizăm acest lucru.*

Datele de intrare ne permit să calculăm:

`D[i][j]` = *costul minim să angajăm exact* `i` *muncitori care sapă exact* `j` *metri de șanț.*

Nu e nimic magic în felul de a raționa pentru alegerea semnificației structurii. Pur și simplu am interpretat cerința din enunț. Dacă reușim să determinăm o recurență, vom găsi soluția în `D[N][S]`.

Așadar, fiecare linie a matricei `D` este pentru update-ul făcut la introducerea unui nou muncitor, iar fiecare coloană corespunde unei lungimi de șanț.

Când ajungem la linia `i` din `D`, ne bazăm că avem calculate liniile anterioare pentru fiecare coloană de la `1` la `S`.

Dacă mai introducem al `i`-lea muncitor și vrem să aflăm care este costul minim de a săpa exact `j` metri de șanț (adică să calculăm practic pe `D[i][j]`), încercăm să considerăm acest muncitor de fiecare categorie posibilă dintre cele `C`.

De exemplu, dacă am alege un muncitor de categoria `1`, el ar săpa `L[1]` metri de șanț și ar costa `P[1]` lei. Așadar, ne-ar interesa un mod optim să fi săpat `j-L[1]` metri de șanț cu `i-1` muncitori. Dar această valoare este deja calculată în `D[i-1][j-L[1]]`. Evident că se mai adaugă și `P[1]`, costul angajării muncitorului `1`.

Deci, `D[i][j]` = maxim dintre:

- `D[i-1][j-L[1]] + P[1]`
- `D[i-1][j-L[2]] + P[2]`

…

- `D[i-1][j-L[C]] + P[C]`

Valoarea `C` este mică, deci timpul de calcul de ordin `N*S*C` obținut, este suficient. La formulele de mai sus trebuie să ținem seama de cazurile particulare (de exemplu, valorile `D` de care avem nevoie să fie calculate, iar valorile `j-L[]` să nu fie negative sau nule).

```
#include <cstdio>
#include <algorithm>
#define INF 32000

using namespace std;
short d[101][1001],t[101][1001],p[21],l[21],sol[101];
int main()
{
    FILE *fin=fopen ("sant.in","r");
    FILE *fout=fopen ("sant.out","w");
    int s,n,c,i,j,sc,elem,k;
    fscanf (fin,"%d%d%d",&s,&n,&c);
    for (i=1;i<=s;i++)
        d[1][i]=INF;
    for (i=1;i<=c;i++){
        fscanf (fin,"%d%d",&l[i],&p[i]);
        if (l[i]<s && (d[1][l[i]]>p[i])){
            d[1][l[i]]=p[i];
            t[1][l[i]]=i;
        }
    }
    for (i=2;i<=n;i++)
        for (j=1;j<=s;j++){
            d[i][j]=INF;
```

```
            for (k=1;k<=c;k++)
                if (j>l[k] && (d[i][j]>d[i-1][j-l[k]]+p[k]) ){
                    d[i][j]=d[i-1][j-l[k]]+p[k];
                    t[i][j]=k;
                }
        }
    if (d[n][s]==INF)
        d[n][s]=0;
    fprintf (fout,"%d\n",d[n][s]);
    if (d[n][s]==0)
        return 0;
    sc=s;
    elem=0;
    for (i=n;i>0;i--){
        sol[++elem]=t[i][sc];
        sc=sc-l[t[i][sc]];
    }
    sort (sol+1,sol+elem+1);
    for (i=1;i<=elem;i++)
        fprintf (fout,"%d ",sol[i]);
    return 0;
}
```

Câteva observații legate de sursa de mai sus

- Am ales să facem separat inițializarea pentru linia `1` (adică pentru ce ar săpa optim un singur muncitor).
- Observăm că pntru calculul unei linii este necesară doar linia anterioară. Astfel, la prima vedere am putea renunța la matricea `D` și să lucrăm cu ultimele două linii, cum am mai arătat la alte probleme.
- Totuși, problema cere să reconstituim și drumul și pentru asta avem nevoie de o matrice. Noi folosim mai sus: `T[i][j]` = *categoria din care face parte al* `i`*-lea (ultimul) muncitor ales cu care să fi săpat exact* `j` *metri de șanț.* Valoarea din `T[i][j]` o actualizăm în momentul în care calculăm pe `D[i][j]`.
- Reconstrucția soluției o facem clasic, pe matricea `T`, însă nu am mai ales o abordare recursivă, pentru că ordinea în care se cer muncitorii este cea lexicografică după categorii, așa că îi salvăm într-un vector pe care îl sortăm.
- Deci legat de memorie, puteam scăpa de $n^2$ pentru matricea `D`, dar nu puteam scăpa pentru `T`.

Poate vă gândiți de ce la această dinamică a trebuit o matrice și nu a fost suficient un singur vector `D`, de lungime `S`, în care să actualizăm la fiecare nou muncitor introdus. Motivul este că după ce face actualizări al i-lea muncitor, mai rămân în `D` elemente marcate fără să fie luat în calcul acest ultim muncitor. Ori cerința problemei noastre este să folosim exact `N` muncitori.

***Comp2*** ([https://www.infoarena.ro/problema/comp2](https://www.infoarena.ro/problema/comp2))

Vom calcula o dinamică astfel:
`P[i]` = *numărul minim de persoane necesare pentru a avea* `i` *relații*
`S[i]` = *numărul minim de șefi necesari pentru a avea* `P[i]` *angajati pentru* `i` *relatii*

La un moment dat ne imaginăm că mai introducem o echipă formata din “`sefi`” șefi și “`angajati`” angajați. Această echipă va contribui cu “`relatii = sefi * angajati`” relații.
Pentru orice valoare `k` pentru care am calculat valorile optime `P[k]` și `S[k]` pentru `k` relații, încercăm să adăugăm datele acestei echipe, adică:

- Obținem `k + sefi * angajati` relații
- Cu `P[k] + sefi + angajati` oameni
- Cu `S[k] + sefi` șefi

Adică încercăm să actualizăm valorile `P` și `S` în poziția `k + sefi * angajati`.
În sursa prezentată observăm cum reținem structuri care să permită și reconstrucția soluției.

```
#include <fstream>
#define DIM 2010
#define INF 1000000
using namespace std;
int P[DIM]; // P[i] = numarul minim de persoane sa am i relatii
int S[DIM]; // S[i] = numarul minim de sefi sa am P[i] angajati
pentru i relatii
int lastPersonal[DIM];
int lastSefi[DIM];
int nr[DIM];
int e;

int main () {
    ifstream fin ("comp2.in");
    ofstream fout("comp2.out");

    fin>>e;
    for (int i=1;i<=e;i++) {
        P[i] = S[i] = INF;
    }
    P[0] = S[0] = 0;

    for (int sefi = 1; sefi <= e; sefi++) {
        int maxAngajati = e/sefi;
        for (int angajati=0; angajati<=maxAngajati; angajati++){
            // ultima companie are sefi,angajati
            int personal = sefi + angajati;
            int relatii = sefi * angajati;
            for (int k = 0; k<=e-relatii; k++)
                if (P[k] != INF) {
                    int updRelatii = k + relatii;
                    int updPersonal = P[k] + personal;
                    int updSefi = S[k] + sefi;
```

```
                    if (P[updRelatii] > P[k] + personal || (
                    P[updRelatii] == P[k] + personal &&
                    S[updRelatii] > S[k] + sefi )) {
                        P[updRelatii] = P[k] + personal;
                        S[updRelatii] = S[k] + sefi;
                        lastPersonal[updRelatii] = personal;
                        lastSefi[updRelatii] = sefi;
                        nr[updRelatii] = 1 + nr[k];
                    }
                }
            }
    }
    fout<<P[e]<<" "<<S[e]<<" "<<nr[e]<<"\n";
    while (e) {
        fout<<lastPersonal[e]<<" "<<lastSefi[e]<<"\n";
        e -= lastSefi[e] * (lastPersonal[e] - lastSefi[e]);
    }

    return 0;
}
```

# IV. Probleme de dinamică mixtă

***Parantezarea optimă la înmulțire matricelor*** (https://infoarena.ro/problema/podm)

Problema parantezării optime la înmulțirea matricelor este într-adevăr una clasică pentru acest tip de dinamică. Noi o vom reformula puțin:

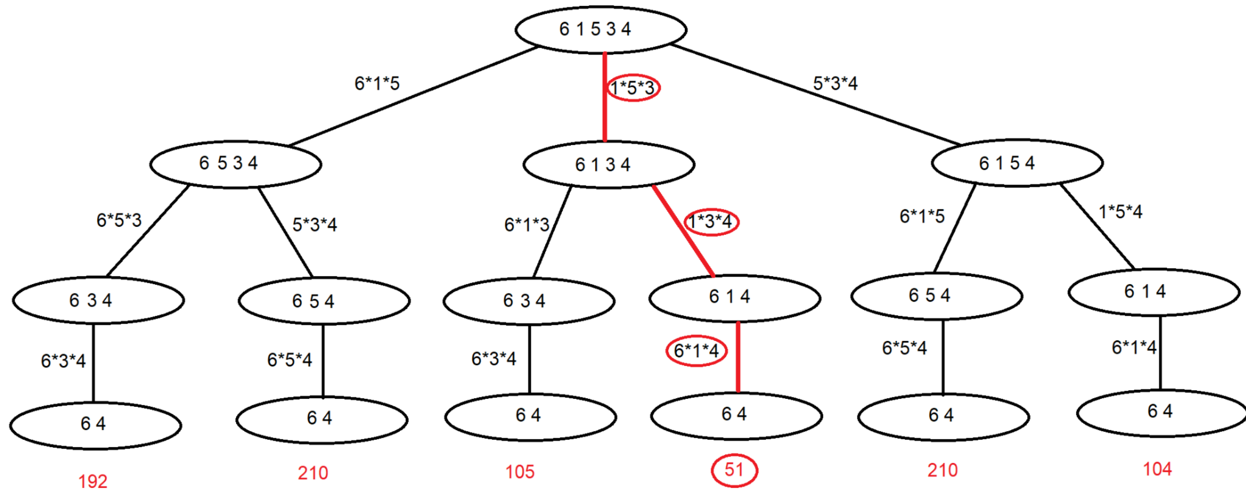
*Se dă un șir* `V` *cu* `n` *numere naturale. Trebuie să executăm de* `n-2` *ori următoarea operație: alegem* `3` *elemente vecine în acel moment în șir,* `V[i]`, `V[i+1]`, `V[i+2]` *și eliminăm elementul din mijloc. Costul acestei operații este* `V[i]*V[i+1]*V[i+2]`. *Deci, după această operație elementele aflate anterior pe pozițiile* `i` *și* `i+2` *devin vecine.*

*După* `n-2` *astfel de operații singurele elemente rămase în șir sunt primul și ultimul din șirul inițial.*

*Cerința este de a alege convenabil o ordine de a aplica operațiile astfel încât suma costurilor celor* `n-2` *operații să fie minimă.*

Exemplu

Pentru șirul `2 1 5 3 4` avem variantele:




Costul optim observăm că este cel de pe traseul de eliminări indicat cu roșu. Poate părea că o strategie greedy de a părtra mereu minimul este și optimă însă se poate demonta. Pe de o parte pot fi mai multe minime, pe de altă parte, mai depinde și de vecinii direcți, iar dacă șirul este mai mare e clar că pot fi diverse configurații.

Traseele indicate de noi sunt pentru a ajuta la înțelegerea problemei și dacă ar fi să le calculăm pe toate timpul ar fi exponențial (de fapt cumva factorial, pentru că e ca și cum am aplica eliminările în toate modurile posibile). Se observă că sunt stări în care ajungem cumva din nou.

Soluția optimă pentru această problemă este alta:

Vom nota `D[i][j]` = *costul minim pentru a elimina toate elementele din secvența de la* `i` *la* `j`*, mai puțin pe* `V[i]` *și pe* `V[j]`. La modul cum gândim rezolvarea, noi ne imaginăm că elementele neeliminate încă, rămân pe pozițiile lor originale din `V`, dar mai dispar elemente de pe anumite poziții.

Conform definiției de mai sus, căutăm la final soluția în elementul `D[1][n]`.

Pentru secvențele de lungime `2`, `D[i][i+1] = 0`. Acest lucru se datorează faptului că secvențele de două elemente conțin doar capetele, nu și altceva în interior, cum spune definiția lui `D`.

Pentru secvențe mai lungi de la poziția `i` la poziția `j`, alegem pe rând câte o poziție `k`, dintre `i` și `j`, care să indice ultimul eliminat înainte de a rămâne doar `V[i]` și `V[j]`. Ca să ajungem în starea ca între `V[i]` și `V[j]` să mai rămână doar `V[k]`, ar trebui ca toate elementele dintre `i` și `k` să fi fost eliminate și la fel toate elementele dintre `k` și `j`. Pentru aceste eliminări ne interesează variantele de cost minim, adică `D[i][k]` și `D[k][j]`.

Astfel, costul optim de a elimina tot din interiorul secvenței de la `i` la `j` și ultima dată să eliminăm pe `V[k]` este: `D[i][k] + D[k][j] + V[i]*V[k]*V[j]`.

Dar noi putem, cum spuneam, să alegem cum dorim care este elementul pe care îl eliminăm ultimul, așadar:

`D[i][j]=minim(D[i][k]+D[k][j]+V[i]*V[k]*V[j])`, *pentru orice* `k` *de la* `i+1` *la* `j-1`.

Observăm că pentru o secvență de la `i` la `j` e necesar să avem anterior calculate rezultatele optime pentru secvențe de lungime mai mică.

Astfel, rămâne să calculăm optimele secvențelor în ordine crescătoare a lungimii lor.

Algoritmul ar fi:

```
for (i=1;i<n;i++)
    D[i][i+1] = 0;
for (lungime = 3; lungime <= n; lungime++) {
/// fixam lungimea secventei

    for (i=1; i+lungime-1<=n;i++) {
    /// fixam inceputul de secventa

        j = i+lungime-1;
        D[i][j] = INF;
        for (k=i+1;k<j;k++)
            D[i][j]=min(D[i][j],D[i][k]+D[k][j]+V[i]*V[k]*V[j]);
    }
}
```

Timpul de calcul este așadar de ordin $n^3$.

Alt mod de a ne imagina rezolvarea: avem de calculat elementele matricei `D` în ordinea:

- Mai întâi diagonala paralelă cu cea principală aflată deasupra ei
- Apoi următoarea astfel de diagonală aflată și mai deasupta

…

- În final ajungem la elementul din dreapta sus, care este chiar `D[1][n]`

Elementele de pe aceeași astfel de diagonală au diferența indicilor constantă, deci reprezintă secvențe din `V` de aceeași lungime, iar pe măsură ce ne îndepărtăm de diagonala principală obținem diagonale ce reprezintă secvențe din ce în ce mai lungi.

Spunem că avem dinamică mixtă pentru că odată ce fixăm o valoare, cum este `k`, este necesar să căutăm optime aflate și de o parte și de cealaltă a acestei valori.

O astfel de problemă se poate rezolva și recursiv, dar obligatoriu folosind procedeul de memorizare (adică să stocăm într-o structură valorile calculate pentru a nu mai autoapela în ele). Dacă nu memorizăm ajungem la expandarea exponențială, așa cum reiese din arborele de exemplu de mai sus.

```
int calcul (int i, int j) {
    if (j == i+1) {
        D[i][j] = 0;
        return 0;
    } else {
        if (D[i][j] != 0)
            return D[i][j];
        int sol = INF;
        for (int k=i+1;k<j;k++) {
            int aux = calcul(i, k) + calcul(k, j) + V[i]*V[k]*V[j];
            if (aux < sol)
                sol = aux;
            D[i][j] = sol;
            return D[i][j];
        }
    }
}
```

Aici apelăm cu `calcul(1, n);`

Practic structura din spatele recursivității este chiar matricea cu semnificația anterioară. Complexitatea în timp este aceeași la ambele abordări, totuși ne așteptăm ca în astfel de cazuri soluțua nerecursivă să se comporte mai bine în practică.

***Redu*** (https://infoarena.ro/problema/redu)

Pe scurt, avem un șir și la fiecare pas putem alege două elemente vecine și să le eliminăm. Elementele fiind litere mici ale alfabetului, se dă la intrare o matrice `26X26` prin care se indică un cost pentru a elimina oricare pereche de două litere care ar fi vecine în acel moment. Trebuie să reducem tot șirul cu un cost minim.

La această problemă abordarea este următoarea:

- Calculăm `D[i][j]` = *costul minim să eliminăm toată secvența de litere de la indicele* `i` *la indicele* `j`.
- Pentru a calcula această valoarem ne gânfim care literă de la `i+1` la `j` este cea care se va împerechea cu prima (se va elimina odată cu ea). Dacă această literă este pe poziția `k`, ar trebui să avem înainte deja eliminate toate literele de la poziția `i+1` la `k-1` (cu cost optim `D[i+1][k-1]`) și apoi să eliminăm și toate literele de la poziția `k+1` la poziția `j`, optim, valoare care se află în `D[k+1][j]`

O altă observație este că literele care se elimină deodată cu prima trebuie să fie așa încât între ele este un număr par de litere în șirul inițial (literele se elimină câte două).

Deci avem:
`D[i][j]=minim(D[i+1][k-1]+D[k+1][j]+cost(V[i],V[k]))`, *cu* `k=i+1` *din* `2` *în* `2`.

Iată mai departe o sursă pe această idee:

```
#include <fstream>
#define INF 100000000
using namespace std;
char s[512];
int a[26][26];
int i, j, k, L, n;
int d[512][512];

ifstream fin ("redu.in");
ofstream fout("redu.out");

int main () {
    fin>>n;
    fin>>s;
    for (i=0;i<26;i++)
        for (j=0;j<26;j++)
            fin>>a[i][j];

    for (i=0;i<n-1;i++)
        d[i][i+1] = a[ s[i]-'a' ][ s[i+1]-'a' ];

    for (L = 4; L <= n; L+=2) {
        for (i=0; i+L-1<n;i++) {
            j = i+L-1;
            d[i][j] = INF;
            for (k=i+1;k<=j;k+=2)
                d[i][j] = min(d[i][j], d[i+1][k-1] + d[k+1][j] +
                                       a[ s[i]-'a' ][ s[k]-'a' ]);
        }
    }
    fout<<d[0][n-1];
    return 0;
}
```