

## Laborator 4: Programare Dinamică (continuare)

Responsabili:

- Darius-Florentin Neațu (2017–2021) [mailto:neatudarius@gmail.com]
- Radu Nichita (2021) [mailto:radunichita99@gmail.com]
- Cristian Olaru (2021) [mailto:cristianolaru99@gmail.com]
- Miruna-Elena Banu (2021) [mailto:mirunaelena.banu@gmail.com]
- Mara-Ioana Nicolae (2021) [mailto:maraoana9967@gmail.com]
- Ștefan Popa (2018–2020) [mailto:stefanpopa2209@gmail.com]

Autori:

- Darius-Florentin Neațu (2018) [mailto:neatudarius@gmail.com]
- Radu Vișan (2018) [mailto:visanr95@gmail.com]
- Cristian Banu (2018) [mailto:cristb@gmail.com]
- Răzvan Chițu (2018) [mailto:razvan.ch95@gmail.com]

## Obiective laborator

- Înțelegerea noțiunilor de bază despre programarea dinamică.
- Însușirea abilităților de implementare a algoritmilor bazați pe programarea dinamică.

## Precizări inițiale

Toate exemplele de cod se găsesc pe pagina pa-lab::demo/lab04 [https://github.com/acs-pa/pa-lab/tree/main/demo/lab04].

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

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

## Ce este DP?

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

De asemenea, DP se poate folosi și pentru probleme în care nu căutam un optim, cum ar fi **problemele de numărare**.

Pentru noțiunile prezentate până acum despre DP, vă rugăm să consultați pagina laboratorului 3.

## Exemple clasice

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

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

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

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

### Categoria 3: **PODM** Parantezare optimă de matrice

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

**ATENȚIE!** Acest tip de recurențe poate fi mai greu (decât celelalte). Puteți consulta **acasă** materialele puse la dispoziție pentru a înțelege mai bine această categorie.

Caracteristici:

- Acest tip de problemă presupune că o putem formula ca pe o problemă de tip **subinterval**  $[i, j]$ .
- Dacă dorim să găsim optimul pentru acest interval, va trebui să luăm în calcul toate combinațiile de 2 subprobleme care ar putea genera o soluție pentru problemele  $[i, j]$ .
- Se consideră fiecare divizare în 2 subprobleme, dată de intermediarul  $k$ , astfel:
  - Fie  $[i, k]$  și  $[k + 1, j]$  cele 2 subprobleme pentru care cunoaștem soluțiile, atunci o soluție pentru  $[i, j]$  se poate obține îmbinându-le pe cele două
  - pentru a găsi soluția cea mai bună:
    - vom itera prin toate valorile  $k$  posibile
    - o vom alege pe cea care maximizează soluția problemei  $[i, j]$

- Calculul se face de la intervale mici (probleme ușoare –  $[i, i]$  sau  $[i, i + 1]$ ) spre probleme generale (dimensiune generală –  $[i, j]$ ). În final, se ajunge și la dimensiunile inițiale ( $[1, n]$ ).
- Privind imaginea de ansamblu, adică modul în care se completează matricea dp, observăm că aceasta se completează **diagonală cu diagonală**.

## Exemple clasice

### PODM

#### Enunț

Fie un produs matriceal  $M = M_1 M_2 \dots M_n$ . Putem pune paranteze în mai multe moduri și vom obține același rezultat (înmulțire asociativă), dar este posibil să obținem un număr diferit de **înmulțiri scalare**.

Matricea  $M_i$  are (prin convenție), dimensiunile  $d_{i-1} d_i$ .

#### Cerință

Se cere să se găsească o **parantezare optimă de matrice** (PODM), adică să se găsească o parantezare care să minimizeze numărul de înmulțiri scalare.

#### Exemple

$n = 3$

i	0	1	2	3
d	2	3	4	5

Răspuns: **64** (înmulțiri scalare)

Explicație: Avem 3 matrice:

- A de dimensiuni (2, 3)
- B de (3, 4)
- C de (4, 5)

În funcție de ordinea efectuării înmulțirilor matriceale, numărul total de înmulțiri scalare poate să fie foarte diferit:

- $(AB)C \Rightarrow 24 + 40 = 64$  de înmulțiri
  - explicație:  $X = (AB)$  generează  $2 * 3 * 4 = 24$  înmulțiri,  $(XC)$  generează  $2 * 4 * 5 = 40$  de înmulțiri
- $A(BC) \Rightarrow 60 + 30 = 90$  de înmulțiri
  - explicație:  $X = (BC)$  generează  $3 * 4 * 5 = 60$  înmulțiri,  $(AX)$  generează  $2 * 3 * 5 = 30$  de înmulțiri

Rezultatul optim se obține pentru cea de a treia parantezare:  $(AB)C$ .

$n = 4$

i	0	1	2	3	4
d	2	3	4	2	3

Răspuns: **48** (înmulțiri scalare)

Explicație: Avem 4 matrice:

- A de dimensiuni (2, 3)
- B de (3, 4)
- C de (4, 2)
- D de (2, 3)

În funcție de ordinea efectuării înmulțirilor matriceale, numărul total de înmulțiri scalare poate să fie foarte diferit:

- $(AB)CD \Rightarrow 24 + 16 + 12 = 52$  înmulțiri
  - explicație:  $X = (AB)$  generează  $2 * 3 * 4 = 24$  înmulțiri scalare,  $Y = (XC)$  generează  $2 * 4 * 2 = 16$  înmulțiri scalare,  $Z = YD$  generează  $2 * 2 * 3 = 12$  înmulțiri scalare
- $A(BC)D \Rightarrow 24 + 12 + 12 = 48$  înmulțiri
  - explicație:  $X = (BC)$  generează  $3 * 4 * 2 = 24$  înmulțiri scalare,  $Y = (AX)$  generează  $2 * 3 * 2 = 12$  înmulțiri scalare,  $Z = YD$  generează  $2 * 2 * 3 = 12$  înmulțiri scalare
- $(AB)(CD) \Rightarrow 24 + 12 = 36$  înmulțiri
  - explicație:  $X = (AB)$  generează  $2 * 3 * 4 = 24$  înmulțiri scalare,  $Y = (CD)$  generează  $4 * 2 * 3 = 24$  înmulțiri scalare,  $Z = XY$  generează  $2 * 4 * 3 = 24$  înmulțiri scalare
- $A((BC)D) \Rightarrow 24 + 18 + 27 = 69$  înmulțiri
  - explicație:  $X = (BC)$  generează  $3 * 4 * 2 = 24$  înmulțiri scalare,  $Y = (XD)$  generează  $3 * 2 * 3 = 18$  înmulțiri scalare,  $Z = AY$  generează  $3 * 3 * 3 = 27$  înmulțiri scalare
- $A(B(CD)) \Rightarrow 24 + 36 + 18 = 78$  înmulțiri
  - explicație:  $X = (CD)$  generează  $4 * 2 * 3 = 24$  înmulțiri scalare,  $Y = (BX)$  generează  $3 * 4 * 3 = 36$  înmulțiri scalare,  $Z = AY$  generează  $2 * 3 * 3 = 18$  înmulțiri scalare

Rezultatul optim se obține pentru cea de a treia parantezare:  $((A(BC))D)$ .

$n = 4$

$n=3$

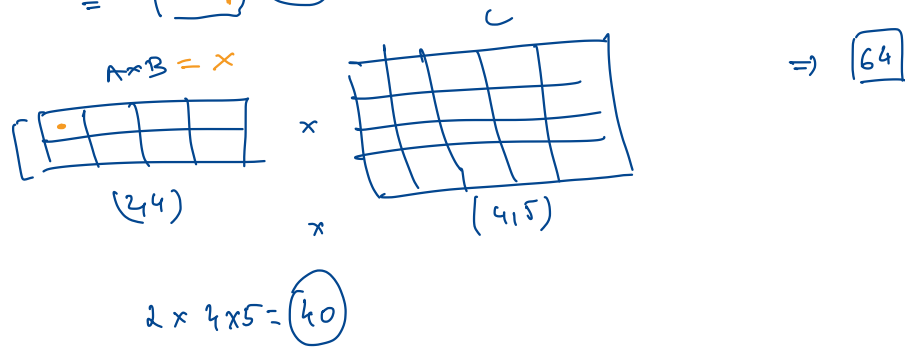
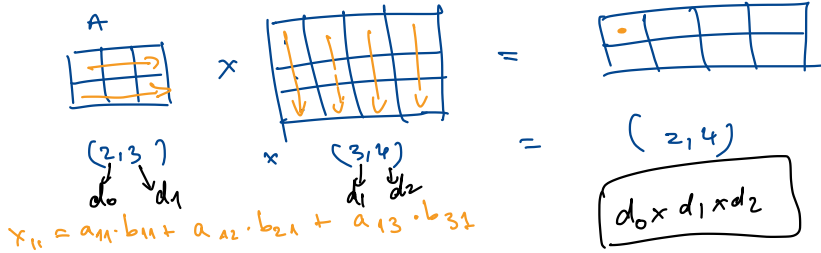
d 

0	1	2	3
2	3	4	5

 $\rightarrow$  A (2,3)  
B (3,4)  
C (4,5)

$A \times B$   
 $(a,b) \times (b,c)$

$(A \times B) \times C$



$(A \times B) \times C \rightarrow 64$  înmulțiri scalare

$A \times (B \times C) \rightarrow 90$  de înmulțiri scalare

**Recurentă**

$dp[i][j] = \text{nr. minim de înmulțiri scalare cu care se poate obține produsul } M_i \times M_{i+1} \times M_{i+2} \times \dots \times M_{j-1} \times M_j, i \leq j$

$(n) \rightarrow M_1 \times M_2 \times \dots \times M_{n-1} \times M_n$   
 $M_i \times \dots \times M_j$   
 $(M_i)$

**SOLUȚIA**:  $dp[1][n]$

	1	2	3
1	0		64
2	INF	0	
3	INF	INF	0

$i \leq j$

$M_i \times M_{i+1} \times \dots \times M_{j-1} \times M_j$

**CĂUTĂRI DE BAZĂ**

$dp[i][i] = 0 \quad (M_i), \forall i \in \overline{1:n}$

$dp[i][j] = \text{INF}, \forall i > j$

$i=1: dp[1][2]$   
 $j=2: dp[2][3]$

$dp[i][i+1] = d_{i-1} \cdot d_i \cdot d_{i+1}, \forall i$

$M_i \times M_{i+1} \rightarrow (d_{i-1}, d_i) \times (d_i, d_{i+1}) \rightarrow d_{i-1} \cdot d_i \cdot d_{i+1}$

A B

i	0	1	2	3	4
d	13	5	89	3	34

Răspuns: 2856 (înmulțiri scalare)

Explicație: Avem 4 matrice:

- A de dimensiuni (13, 5)
- B de (5, 89)
- C de (89, 3)
- D de (3, 34)

În funcție de ordinea efectuării înmulțirilor matriciale, numărul total de înmulțiri scalare poate să fie foarte diferit:

- $((AB)C)D \Rightarrow 10582$  înmulțiri
- $(AB)(CD) \Rightarrow 54201$  înmulțiri
- $(A(BC))D \Rightarrow 2856$  înmulțiri
- $A((BC)D) \Rightarrow 4055$  înmulțiri
- ...

Rezultatul optim se obține pentru cea de a treia parantezare:  $(A(BC))D$ .

## TIPAR

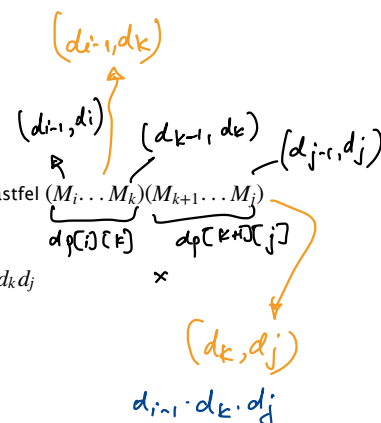
A fost descris în detaliu mai sus (când s-a vorbit de categorie).

Numire recurență

 $dp[i][j] = \text{numărul minim de înmulțiri scalare cu care se poate obține produsul } M_i * M_{i+1} * \dots * M_j$ 

Găsire recurență

- **Cazul de bază :**
  - $dp[i][i] = 0$ 
    - NU avem niciun efort dacă nu avem ce înmulți.
  - $dp[i][i+1] = d_{i-1}d_i d_{i+1}$ 
    - Dacă avem două matrice, putem doar să le înmulțim. Nu are sens să folosim paranteze.
- **Cazul general:**  $dp[i][j] = \min(dp[i][k] + dp[k+1][j] + d_{i-1}d_k d_j)$  unde  $k = i : j - 1$ 
  - dacă avem de efectuat șirul de înmulțiri  $M_i \dots M_j$ , atunci putem pune paranteze oriunde și să facem înmulțirile astfel
    - costul minim pentru  $(M_i \dots M_k)$  este  $dp[i][k]$
    - costul minim pentru  $(M_{k+1} \dots M_j)$  este  $dp[k+1][j]$
    - vom avea, în final, de înmulțit 2 matrice de dimensiune  $d_{i-1} * d_k$  și  $d_k * d_j$ , operație care are costul  $d_{i-1}d_k d_j$
  - însumăm cele 3 costuri intermediare



Implementare

Puteți rezolva și testa problema PODM pe infoarena aici [https://infoarena.ro/problema/podm].

Un exemplu de implementare în C++ se găsește mai jos.

```
// kInf este valoarea maximă - "infiniutul" nostru
const unsigned long long kInf = std::numeric_limits<unsigned long long>::max();

// T = O(n ^ 3)
// S = O(n ^ 2) - stocăm n x n întregi în tabloul dp
unsigned long long solve_podm(int n, const vector<int> &d) {
    // dp[i][j] = numărul MINIM înmulțiri scalare cu codare, poate fi calculat produsul
    // matriceal M_i * M_{i+1} * ... * M_j
    vector<vector<unsigned long long>> dp(n + 1, vector<unsigned long long> (n + 1, kInf));

    // Cazul de bază 1: nu am ce înmulți
    for (int i = 1; i <= n; ++i) {
        dp[i][i] = 0ULL; // 0 pe unsigned long long (voi folosi mai încolo și 1ULL)
    }

    // Cazul de bază 2: matrice d[i - 1] x d[i] înmulțită cu matrice d[i] x d[i + 1]
    // (matrice pe poziții consecutive)
    for (int i = 1; i < n; ++i) {
        dp[i][i + 1] = 1ULL * d[i - 1] * d[i] * d[i + 1];
    }

    // Cazul general:
    for (int len = 2; len <= n; ++len) {
        for (int i = 1; i + len - 1 <= n; ++i) {
            int j = i + len - 1;
            // Iterăm prin indicii dintre capete, spărgând șirul de înmulțiri în două (paranteze).
            for (int k = i; k < j; ++k) {
                // M_i * ... * M_j = (M_i * ... * M_k) * (M_{k+1} * ... * M_j)
                unsigned long long new_sol = dp[i][k] + dp[k + 1][j] + 1ULL * d[i - 1] * d[k] * d[j];

                // actualizăm soluția dacă este mai bună
                dp[i][j] = min(dp[i][j], new_sol);
            }
        }
    }
}
```

```
// Rezultatul se află în dp[1][n]: Numărul MINIM de înmulțiri scalare
// pe care trebuie să le facem pentru a obține produsul M_1 * ... * M_n
return dp[1][n];
}
```

Sursa a fost scrisă pentru a fi testată pe infoarena. În cazul problemei PODM [https://infoarena.ro/problema/podm], deoarece avem o sumă de foarte multe produse, rezultatul este foarte mare. Pe infoarena se cerea ca rezultatul să fie afișat așa cum e, garantându-se că încapă pe 64 biți.

Reamintim că prin înmulțirea/adunarea a două variabile de tipul **int**, rezultatul poate să nu încapă pe 32 biți. De aceea, în soluția prezentată, s-a făcut cast pe 64 biți.

**ATENȚIE!** La PA, în general, vom folosi convenția *expresie % kMOD* care va fi detaliată în capitolul următor din acest laborator.

Complexitate

Întrucat soluția presupune fixarea capetelor unui subinterval (i, j), apoi alegerea unui intermediar (k), complexitatea este dată de aceste 3 cicluri.

- **complexitate temporală:**  $T(n) = O(n^3)$
- **complexitate spațială:**  $S(n) = O(n^2)$

## Categoria 4: NUMĂRAT

Aceste recurențe au o oarecare asemănare:

- toate numără lucruri! :p
- interesante sunt cazurile când numărul căutat este foarte mare (altfel am putea apela la alte metode – ex. generarea tuturor candidaților posibili cu backtracking)
  - în acest caz, deoarece numărul poate să nu încapă pe un tip reprezentabil standard (ex. int pe 32/64 de biți), se cere (de obicei) restul împărțirii numărului căutat la un număr **MOD** (vom folosi în continuare această notație).

## Sfaturi / Reguli

- când căutați o recurență pentru o problema de numărare trebuie să aveți grijă la două aspecte:
  - 1) să **NU** numărați același obiect de două ori.
  - 2) să numărați toate obiectele în cauză.
- de multe ori, o problemă de numărare implică o partiționare a **tuturor** posibilelor soluții după un anumit criteriu (relevant). Găsirea criteriului este partea esențială pentru aflarea recurenței.

## Regulile de lucru cu clase de resturi

Reamintim câteva proprietăți matematice pe care ar trebui să le aveți în vedere atunci când implementați pentru a obține corect resturile anumitor expresii. (corect poate să însemne, de exemplu, să evitați overflow :D – lucru neintuitiv câteodată).

- proprietăți de bază:
  - $(a + b) \% MOD = ((a \% MOD) + (b \% MOD)) \% MOD$
  - $(a * b) \% MOD = ((a \% MOD) * (b \% MOD)) \% MOD$
  - $(a - b) \% MOD = ((a \% MOD) - (b \% MOD) + MOD) \% MOD$  (restul nu poate fi ceva negativ; în C++ % nu funcționează pe numere negative)
- invers modular
  - $\frac{a}{b} \% MOD = ((a \% MOD) * (b^{MOD-2} \% MOD)) \% MOD$
  - **DACĂ** MOD este prim; **DACĂ** a și b nu sunt multipli ai lui MOD

- **definiție** : **b** este inversul modular al lui **a** în raport cu **MOD** dacă  $a * b = 1(modulo MOD)$
- **utilizare** :  $\frac{a}{b} \% MOD = ((a \% MOD) * (invers(b) \% MOD)) \% MOD$
- **calculare** : deoarece la PA această discuție are sens doar în contextul posibilității implementării unei recurențe DP în care folosim resturile doar pentru a evita overflow/imposibilitatea de a reține rezultatul pe tipurile standard de tip int (adică nu ne interesează să dăm o metoda generală pentru invers modular), vom simplifica problema – **MOD este prim!!!**
- **Mica teoremă a lui Fermat**: Dacă p este un număr prim și a este un număr întreg care nu este multiplu al lui p, atunci  $a^{p-1} = 1(modulo p)$ 
  - din definiția inversului modular, reiese că **a** și **b** nu sunt multipli ai lui **MOD**
  - introducând notațiile noastre în teoremă și prelucrând obținem
    - $a^{MOD-1} = 1(modulo MOD) \Leftrightarrow a * (a^{MOD-2}) = 1(modulo MOD)$
    - deci, inversul modular al lui a (în aceste condiții specifice) este  $b = a^{MOD-2}$

Reamintim că prin înmulțirea/adunarea a două variabile de tipul int, rezultatul poate să nu încapă pe 32 biți. E posibil să trebuiască să combinăm regulile de la resturi cu următoarele:

- C++
  - **1LL / 1ULL** – constanta 1 pe 64 biți cu semn / fără semn
  - **1LL \* a \* b** – am grijă ca rezultatul să nu dea overflow și să se stocheze direct pe 64 biți (cu semn)
- Java
  - **1L** – constanta 1 pe 64 biți cu semn (în Java nu există unsigned types)
  - **1L \* a \* b** – am grijă ca rezultatul să nu dea overflow și să se stocheze direct pe 64 biți (cu semn)

## Gardurile lui Gigel

Enunt

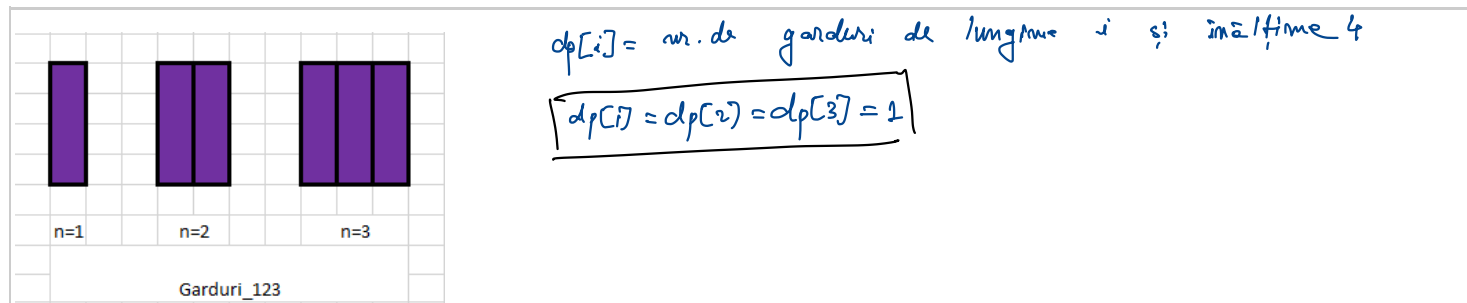
Gigel trece de la furat obiecte cu un rucsac la numarat garduri (fiecare are micile lui plăceri :D). El dorește să construiască un gard folosind în mod repetat **un singur tip de piesă**.

O piesă are dimensiunile **4 x 1** (o unitate = 1m). Din motive irelevante pentru această problemă, orice gard construit trebuie să aibă **înălțime 4m** în orice punct.

O piesă poate fi pusă în poziție **orizontală** sau în poziție **verticală**.

### Cerinta

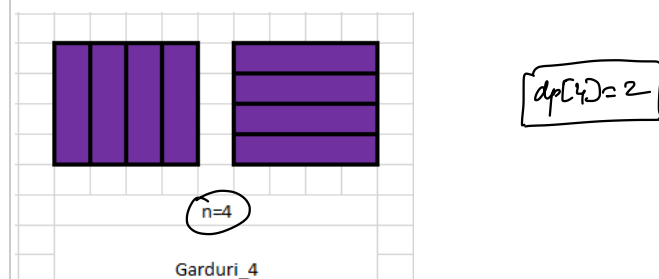
Gigel se întreabă **câte garduri de lungime n și înălțime 4** există? Deoarece celălalt prenume al lui este Bulănel, el intuiește că acest număr este foarte mare, de aceea va cere **restul împărțirii** acestui număr la **1009**.



$n = 1$  sau  $n = 2$  sau  $n = 3$

Răspuns: 1 (un singur gard)

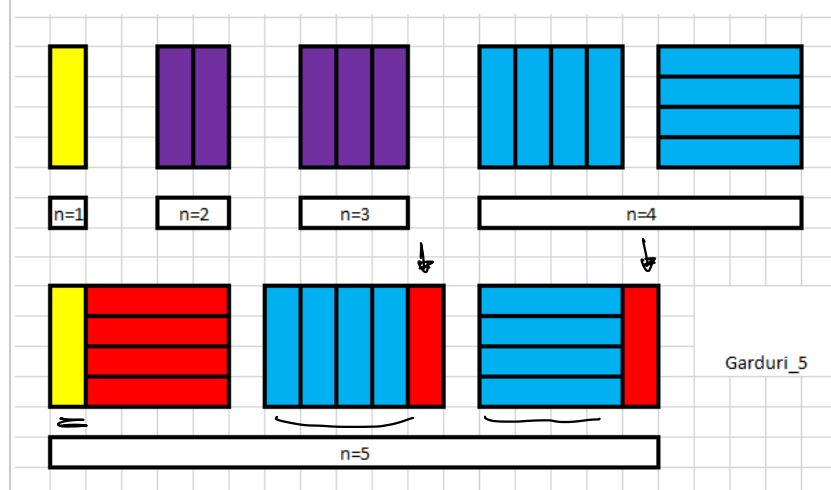
Explicație: Se poate forma un singur gard în fiecare caz, după cum este ilustrat și în figura **Garduri\_123**.



$n = 4$

Răspuns: 2

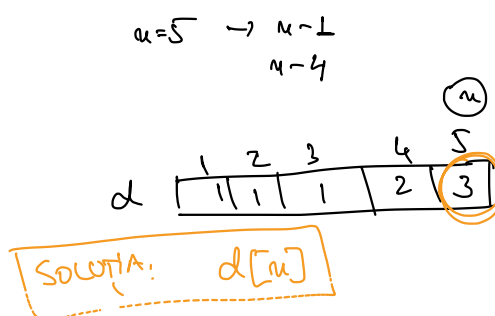
Explicație: Se pot forma 2 garduri, în funcție de cum așezăm piesele, după cum este ilustrat și în figura **Garduri\_4**. Observăm că de fiecare dată când punem o piesă în poziție orizontală, de fapt suntem obligați să punem 4 piese, una peste alta!

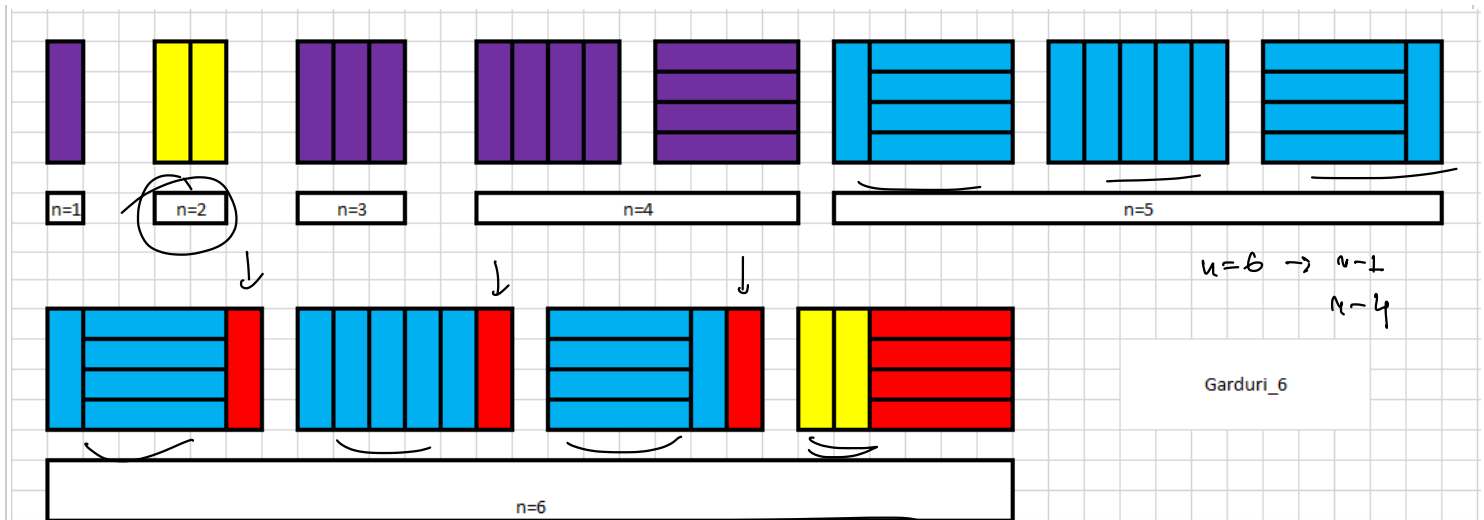


Răspuns: 3

Explicație: Se pot forma 3 garduri, în funcție de cum așezăm piesele, după cum este ilustrat și în figura **Garduri\_5**.

- dacă dorim ca acest gard să se termine cu 4 piese în poziție **orizontală** (una peste alta – marcat cu roșu), atunci la stanga mai rămâne de completat **un subgard de lungime 1**, în toate modurile posibile
- dacă dorim ca acest gard să se termine cu o piesă în poziție **verticală** (marcat cu roșu), atunci la stanga mai rămâne de completat **un subgard de lungime 4**, în toate modurile posibile





$n = 6$

Raspuns: 4

$$dp[i] = (dp[i-1] + dp[i-4]) \% MOD \quad \text{RECURENTA GENERALA}$$

Explicatie: Se pot forma 4 garduri, in functie de cum asezam piesele, dupa cum este ilustrat si in figura **Garduri\_6**.

- daca dorim ca acest gard sa se termine cu o piesa in pozitie **verticala** (marcat cu rosu), atunci la stanga mai ramane de completat **un subgard de lungime 5**, in toate modurile posibile
- daca dorim ca acest gard sa se termine cu 4 piese in pozitie **orizontala** (una peste alta – marcat cu rosu), atunci la stanga mai ramane de completat **un subgard de lungime 2**, in toate modurile posibile

## Recurenta

Numire recurenta

$dp[i]$  = numarul de garduri de lungime  $i$  si inaltime 4 (nimic special – exact ceea ce se cere in enunt)

Raspunsul la problema este  $dp[n]$ .

Gasire recurenta

- Caz de baza**
  - $dp[1] = dp[2] = dp[3] = 1$ ,  $dp[4] = 2$
- Caz general**
  - atunci dorim sa formam un gard de lungime  $i$  ( $i \geq 5$ ) am vazut ca putem alege cum sa punem ultima/ultimele piese
    - DACA** alegem ca ultima piesa sa fie pusa in pozitie verticala, atunci la stanga mai ramane de completat **un subgard de lungime  $i - 1$** 
      - numarul de moduri in care putem face acest subgard este  $dp[i - 1]$
    - DACA** alegem ca ultima piesa sa fie in pozitie orizontala (de fapt punem 4 piese in pozitie orizontala), atunci la stanga mai ramane de completat **un subgard de lungime  $i - 4$** 
      - numarul de moduri in care putem face acest subgard este  $dp[i - 4]$
  - $dp[i] = (dp[i - 1] + dp[i - 4]) \% MOD$

Asa cum am zis in sectiunea de sfaturi si reguli (<http://ocw.cs.pub.ro/courses/pa/laboratoare/laborator-04?#sfaturireguli>) vrem sa facem o **partitionare** dupa un anumit **criteriu**, in cazul problemei de fata criteriul de partitionare este daca gardul se termina cu o scandura verticala sau orizontala.

De asemenea tot in sectiunea sfaturi si reguli (<http://ocw.cs.pub.ro/courses/pa/laboratoare/laborator-04?#sfaturireguli>) am precizat ca nu vrem **sa numaram un obiect** (un mod de a construi gardul) **de doua ori**. Recurenta noastra ( $dp[i] = dp[i-1] + dp[i-4]$ ) nu ia un obiect de doua ori pentru ca orice solutie care vine din  $dp[i-4]$  e diferita de alta care vine din  $dp[i-1]$  pentru ca difera in cel putin ultima scandura asezata)

Implementare recurenta

Aici puteti vedea un exemplu simplu de implementare in C++.

```
#define MOD 1009
int gardurile_lui_Gigel(int n) {
    // cazurile de baza
    if (n <= 3) return 1;
    if (n == 4) return 2;

    vector<int> dp(n + 1); // pastrez indexarea de la 1 ca in explicatii

    // cazurile de baza
    dp[1] = dp[2] = dp[3] = 1;
    dp[4] = 2;

    // cazul general
    for (int i = 5; i <= n; ++i) {
        dp[i] = (dp[i - 1] + dp[i - 4]) % MOD;
    }

    return dp[n];
}
```

Mentionez ca am folosit expresia  $dp[i] = (dp[i - 1] + dp[i - 4]) \% MOD$  în loc de  $dp[i] = ((dp[i - 1] \% MOD) + (dp[i - 4] \% MOD)) \% MOD$  deoarece pe valorile anterior calculate în dp a fost deja aplicată operația.

Am plecat cu numerele 1, 1, 1, 2 și la fiecare pas rezultatul stocat este  $\% MOD$ , deci tot ce este stocat **deja** în dp este un rest în raport cu MOD. NU mai era nevoie deci să aplică  $\%$  și pe termenii din paranteză.

Complexitate

- **complexitate temporală:**  $T = O(n)$ 
  - explicație: avem o singură parcurgere în care construim tabloul dp
  - se poate obține  $T = O(\log n)$  folosind exponențiere pe matrice!
- **complexitate spațială:**  $S = O(n)$ 
  - explicație: stocăm tabloul dp
  - se poate obține  $S = O(1)$  folosind exponențiere pe matrice!

## Tehnici folosite în DP

De multe ori este nevoie să folosim câteva tehnici pentru a obține performanță maximă cu recurență găsită.

În laboratorul 3 se menționează tehnica de memoizare (în prima parte a laboratorului). În acesta ne vom rezuma la cum putem folosi cunoștințele de lucru matricial pentru a favoriza implementarea unor anumite tipuri de recurențe.

## Exponențiere pe matrice pentru recurențe liniare

### Recurențe liniare

O recurență liniară în contextul laboratorului de DP este de forma:

- $dp[i] = \sum_{k=1}^{KMAX} c_k * dp[i - k]$ 
  - pentru **KMAX o constantă**
  - de obicei, KMAX este foarte mică comparativ cu dimensiunea  $n$  a problemei
  - $c_k$  constante reale (unele pot fi nule)

O astfel de recurență ar însemna că pentru a calcula **costul problemei  $i$** , înbinăm costurile problemelor  $i - 1, i - 2, \dots, i - k$  fiecare contribuind cu un anumit coeficient  $c_1, c_2, \dots, c_k$ .

Presupunând că nu mai există alte specificații ale problemei și că având cele KMAX cazuri de bază (primele KMAX valori ar trebui știute/deduse prin alte reguli), atunci un algoritm poate implementa recurența de mai sus folosind 2 cicluri de tip for (for  $i = 1 : n$ , for  $k = 1 : KMAX \dots$ ).

- **complexitatea temporală:**  $T = O(n * KMAX) = O(n)$ 
  - reamintim că acea valoare KMAX este o constantă foarte mică în comparație cu  $n$  (ex.  $KMAX < 100$ )
- **complexitatea spațială:**  $S = O(n)$
- am presupus că avem nevoie să reținem doar tabloul dp

## Exponențiere pe matrice

Facem următoarele notații:

- $S_i$  = starea la pasul  $i$ 
  - $S_i = (dp[i - k + 1], dp[i - k + 2], \dots, dp[i - 1], dp[i])$
- $S_k$  = starea inițială (în care cunoaște cele  $k$  cazuri de bază)
  - $S_k = (dp[1], dp[2], \dots, dp[k - 1], dp[k])$
- $C$  = matrice cu coeficienți constanți
  - are dimensiune  $KMAX * KMAX$
  - putem pune constante în clar
  - putem pune constantele  $c_k$  care tin de problema curentă

Algoritm naiv

Putem formula problema astfel:

- $S_k$  = este starea inițială
- pentru a obține starea următoare, aplicăm algoritmul următor
  - $S_i = S_{i-1} C$

Determinare  $C$

Pentru a determina elementele matricei  $C$ , trebuie să ne uităm la înmulțirea matriceală de mai sus și să alegem elementele lui  $C$  astfel încât prin înmulțirea lui  $S_{i-1}$  cu  $C$  să obținem elementele din  $S_i$ .



$$\begin{bmatrix} dp[i-k+1] & \dots & dp[i-1] & dp[i] \end{bmatrix} = \begin{bmatrix} dp[i-k] & \dots & dp[i-2] & dp[i-1] \end{bmatrix} \begin{bmatrix} 0 & 0 & \dots & 0 & 0 & c_k \\ 1 & 0 & \dots & 0 & 0 & c_{k-1} \\ 0 & 1 & \dots & 0 & 0 & c_{k-2} \\ \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & 1 & 0 & c_2 \\ 0 & 0 & \dots & 0 & 1 & c_1 \end{bmatrix}$$

- ultima coloana contine toti coeficientii  $c_k$  intrucat  $dp[i] = \sum_{k=1}^{KMAX} c_k * dp[i-k]$
- celelalte coloane contin doar cate o valoare nenula
  - pe coloana  $j$  vom avea valoarea 1 pe linia  $j+1$  ( $j = 1 : KMAX - 1$ )
    - cum obtinem, de exemplu,  $dp[i-1]$ ?
    - pai avem  $dp[i-1]$  chiar si in starea  $S_{i-1}$ , deci trebuie sa il copiam in starea  $S_i$ 
      - copierea se realizeaza prin inmultirea cu 1
      - daca  $dp[i-1]$  era pe ultima pozitiei (pozitia  $k$ ) in starea  $S_{i-1}$ , in noua stare  $S_i$  este pe penultima pozitie (pozitia  $k-1$ )
        - deci s-a deplasat la stanga cu o pozitie!
    - in noua stare, noua pozitie este deplasata cu o unitate la stanga fata de starea precedenta
      - de aceea pe coloana  $j$ , vrem sa avem elementul 1 pe linia  $j+1$  ( $j = 1 : KMAX - 1$ )
      - cand inmultim  $S_{i-1}$  cu coloana  $C_j$  **dorim sa**
        - ce copiam?
          - valoarea  $dp[i - KMAX + j]$  din  $S_{i-1}$  in  $S_i$
          - adica sa copiam a  $j$ -a valoare de pe linie
        - unde copiam?
          - de pe pozitia  $j+1$  pe pozitia  $j$

Exponentiere logaritmica pe matrice

Algoritmul naiv de mai sus are dezavantajul ca are tot o complexitate temporală  $O(n)$ .

Sa executam cativa pasi de inductie pentru a vedea cum este determinat  $S_i$ .

$$S_i = S_{i-1}C$$

$$S_i = S_{i-2}C^2$$

$$S_i = S_{i-3}C^3$$

...

$$S_i = S_k C^{i-k}$$

In laboratorul 2 (Divide et Impera) am invatat ca putem calcula  $x^n$  in timp logaritmic. Deoarece si inmultirea matricilor este asociativa, putem calcula  $C^n$  in timp logaritmic.

Obtinem astfel o solutie cu urmatoarele complexitati:

- complexitate temporală** :  $T = O(KMAX^3 * \log(n))$ 
  - explicatie
    - facem doar  $O(\log n)$  pasi, dar un pas implica inmultire de matrice
    - o inmultire de matrice patratica de dimensiune  $KMAX$  are  $KMAX^3$  operatii
    - aceasta metoda este eficienta cand  $KMAX \ll n$  ( $KMAX$  este mult mai mic decat  $n$ )
- complexitatea spatiala** :  $S = O(KMAX^3)$ 
  - explicatie
    - este nevoie sa stocam cateva matrici

Observatie! In ultimele calcule nu am sters contanta  $KMAX$ , intrucat apare la puterea a 3-a!  $KMAX = 100$  implica  $KMAX^3 = 10^6$ , valoare care nu mai poate fi ignorata in practica ( $KMAX^3$  poate fi comparabil cu  $n$ ).

### Gardurile lui Gigel (optimizare)

Dupa cum am vazut mai sus, in problema cu garduri data de Gigel solutia este o recurenta liniara:

- $dp[1] = dp[2] = dp[3] = 1$ ,  $dp[4] = 2$ ;
- $dp[i] = dp[i-1] + dp[i-4]$  pentru  $i > 4$

Exponentiere rapida

- $k = 4$
- $S_4 = (dp[1], dp[2], dp[3], dp[4]) = (1, 1, 1, 2)$
- $S_i = (dp[i-3], dp[i-2], dp[i-1], dp[i])$
- Raspunsul se afla efectuand operatia  $S_n = S_4 * C^{n-4}$ , unde  $C$  are urmatorul continut:

$$C = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix}$$

Mai jos se afla o implementare simplista in C++ care cuprinde toate etapele pe care trebuie sa le realizati in cod, dupa ce stiti cum arata recurenta sub forma matriceala.

```
#define MOD 1009
#define KMAX 4

// C = A * B
void multiply_matrix(int A[KMAX][KMAX], int B[KMAX][KMAX], int C[KMAX][KMAX]) {
    int tmp[KMAX][KMAX];

    // tmp = A * B
    for (int i = 0; i < KMAX; ++i) {
        for (int j = 0; j < KMAX; ++j) {
            unsigned long long sum = 0; // presupun ca suma incapa pe 64 de biti

            for (int k = 0; k < KMAX; ++k) {
                sum += 1LL * A[i][k] * B[k][j];
            }

            tmp[i][j] = sum % MOD;
        }
    }

    // C = tmp
    memcpy(C, tmp, sizeof(tmp));
}

// R = C^p
void power_matrix(int C[KMAX][KMAX], int p, int R[KMAX][KMAX]) {
    // tmp = I (matricea identitate)
    int tmp[KMAX][KMAX];
    for (int i = 0; i < KMAX; ++i) {
        for (int j = 0; j < KMAX; ++j) {
            tmp[i][j] = (i == j) ? 1 : 0;
        }
    }

    while (p != 1) {
        if (p % 2 == 0) {
            multiply_matrix(C, C, C); // C = C*C
            p /= 2; // ramane de calculat C^(p/2)
        } else {
            // reduc la cazul anterior:
            multiply_matrix(tmp, C, tmp); // tmp = tmp*C
            --p; // ramane de calculat C^(p-1)
        }
    }

    // avem o parte din rezultat in C si o parte in tmp
    multiply_matrix(C, tmp, R); // rezultat = tmp * C
}

int garduri_rapide(int n) {
    // cazurile de baza
    if (n <= 3) return 1;
    if (n == 4) return 2;

    // construiesc matricea C
    int C[KMAX][KMAX] = { {0, 0, 0, 1},
                          {1, 0, 0, 0},
                          {0, 1, 0, 0},
                          {0, 0, 1, 1} };

    // vreau sa aplic formula S_n = S_4 * C^(n-4)

    // C = C^(n-4)
    power_matrix(C, n - 4, C);

    // sol = S_4 * C = dp[n] (se afla pe ultima pozitie din S_n,
    // deci voi folosi ultima coloana din C)
    int sol = 1 * C[0][3] + 1 * C[1][3] + 1 * C[2][3] + 2 * C[3][3];
    return sol % MOD;
}
```

Remarcati faptul ca in functia de inmultire se foloseste o matrice temporara *tmp*. Motivul este ca vrem sa apelam functia *multiply(C, C, C)* unde C joaca atat rol de intrare cat si de iesire. Daca am pune rezultatele direct in C, atunci am strica inputul inainte sa obtinem rezultatul.

Putem spune ca acea functie este **matrix\_multiply\_safe**, in sensul ca pentru orice A,B,C care respecta dimensiunile impuse, functia va calcula corect produsul.

In arhiva **demo-lab04.zip** gasiti o sursa completa in care se realizeaza:

- o verificare a faptului ca cele 2 implementari ( **gardurile\_lui\_Gigel** si **garduri\_rapide**) produc aceleasi rezultate
- un benchmark in care cele 2 implementari sunt comparate
  - pe sistem uzual (laptop) s-au obtinut urmatoarele rezultate:

```
test case: varianta simpla
n = 100000000 sol = 119; time = 0.984545 s
test case: varianta rapida
n = 100000000 sol = 119; time = 0.000021 s
```

```
test case: varianta simpla
n = 1000000000 sol = 812; time = 9.662377 s
test case: varianta rapida
n = 1000000000 sol = 812; time = 0.000022 s
```

- se observa clar diferenta intre cele 2 solutii (am confirmat ceea ce spunea si teoria:  $O(n)$  vs  $O(\log(n))$ ); aceasta tehnica imbunatateste drastic o solutie gasita relativ usor.

## Exercitii

Scheletul de laborator se găsește pe pagina pa-lab::skel/lab04 [https://github.com/acs-pa/pa-lab/tree/main/skel/lab04].

## DP or math?

Fie un sir de **numere naturale strict pozitive**. Cate **subsiruri** (submultimi nevide) au suma numerelor **para**?

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

Task-uri:

- Se cere o **solutie folosind DP**.
- Inspectand recurenta gasita la punctul precedent, incercati sa o inlocuiti cu o **formula matematica**.
- Care este **complexitatea** pentru fiecare solutie (timp + spatiu)? Care este mai buna? De ce? :D

Deoarece rezultatul poate fi prea mare, se cere **restul impartirii** lui la  $1000000007(10^9 + 7)$ .

Pentru punctaj maxim pentru aceasta problema, este necesar sa rezolvati toate subpunctele (ex. nu folositi direct formula, gasiti mai intai recurenta DP). Trebuie sa implementati **cel putin** solutia cu DP.

$n = 3$

i	1	2	3
v	2	6	4

Raspuns: 7

Explicatie: Toate subsirurile posibile sunt

- [2] → 2 P
- [2, 6] → 8 P
- [2, 6, 4] → 12 P
- [2, 4] → 6 P
- [6] → 6 P
- [6, 4] → 10 P
- [4] → 4 P

Toate subsirurile de mai sus au suma para.

Handwritten notes:  $n=3 \rightarrow 2^3 - 1 = 7$

$n = 3$

i	1	2	3
v	2	1	3

Raspuns: 3

Explicatie: Toate subsirurile posibile sunt

- [2] → 2 P
- [2, 1] → 3 P
- [2, 1, 3] → 6 P
- [2, 3] → 5 P
- [1] → 1 P
- [1, 3] → 4 P
- [3] → 3 P

Subsirurile cu suma para sunt: [2], [2, 1, 3], [1, 3].

Handwritten notes:  $\{2\}$ ,  $\{1, 3\}$ ,  $C_2^2 = 1$

$n = 3$

i	1	2	3
v	3	2	1

Raspuns: 3

Explicatie: Toate subsirurile posibile sunt

- [3] → 3 P
- [3, 2] → 5 P
- [3, 2, 1] → 6 P
- [3, 1] → 4 P
- [2] → 2 P
- [2, 1] → 3 P
- [1] → 1 P

Subsirurile cu suma para sunt: [3, 2, 1], [3, 1], [2].

Morala: există probleme pentru care găsim o soluție cu DP, dar pentru care pot exista și alte soluții mai bune (am ignorat citirea/afișarea).

In problemele de numarar, exista o **sansa** buna sa putem gasi (si) o formula matematica, care poate fi implementata într-un mod mai eficient decat o recurenta DP.

$dp[i][0] = \text{nr. de submultimi cu suma n. } \boxed{\text{pară}}$  folosind primele  $i$  nr. din şir  
 $dp[i][1] = \text{nr. de submultimi cu suma n. } \text{impară}$  folosind primele  $i$  nr. din şir

### CĂZURILE DE BAZĂ

$$dp[0][0] = 1 \quad // \emptyset, \text{ suma } = 0$$

$$dp[0][1] = 0$$

dp

0	1	0
1	2	0
2	2	2
3	4	4
4	8	8

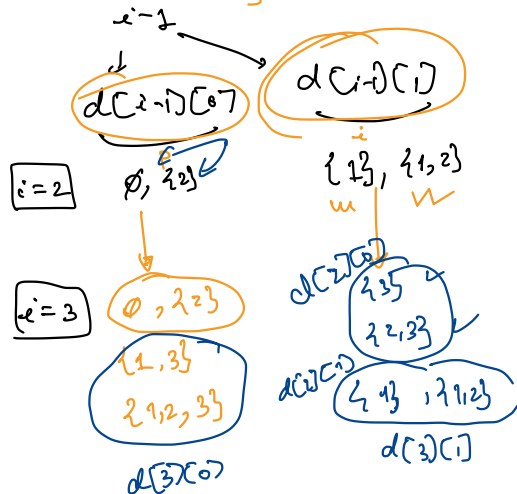
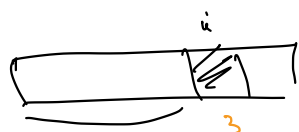
1	2	3	4
2	1	3	4

SOLUȚIA:  $dp[n][0]$

### RECURENȚĂ GENERALĂ

$$dp[i][0] = \begin{cases} 2 \times dp[i-1][0], & \text{if } i \text{ is even} \\ dp[i-1][0] + dp[i-1][1], & \text{if } i \text{ is odd} \end{cases}$$

$$dp[i][1] = \begin{cases} 2 \times dp[i-1][1], & \text{if } i \text{ is even} \\ dp[i-1][0] + dp[i-1][1], & \text{if } i \text{ is odd} \end{cases}$$



$i=4$   
 $p = \{4\}, \{1, 2\}, \{1, 3\}, \{1, 2, 3\}$   
 $p + p = \{4\}, \{1, 4\}, \{2, 4\}, \{3, 4\}, \{1, 2, 4\}, \{1, 3, 4\}, \{2, 3, 4\}, \{1, 2, 3, 4\}$

$i = \{4\}$   
 $i + p = \{4\}, \{1, 4\}, \{2, 4\}, \{3, 4\}, \{1, 2, 4\}, \{1, 3, 4\}, \{2, 3, 4\}, \{1, 2, 3, 4\}$

Cate subsiruri au suma **impara**?

Expresie booleana

~ PDDM

Se da o expresie booleana corecta cu  $n$  termeni. Fiecare din termeni poate fi unul din stringurile **true**, **false**, **and**, **or**, **xor**.

Numarati modurile in care se pot aseza paranteze astfel incat rezultatul sa fie **true**. Se respecta regulile de la logica (tabelele de adevar pentru operatiile **and**, **or**, **xor**).

Deoarece rezultatul poate fi prea mare, se cere **restul impartirii** lui la  $1000000007 (10^9 + 7)$ .

In schelet vom codifica cu valori de tip char cele 5 stringuri:

- **false**: 'F'
- **true**: 'T'
- **and**: '&'
- **or**: '|'
- **xor**: '^'

$$n = 5$$

$$T \quad F \quad F \quad \wedge \quad T$$

$$T \wedge (F \wedge T) = T \wedge F = F$$

2	0	1
0	0	0
1	0	1

1	0	1
0	0	1
1	1	1

1	0	1
0	0	1
1	1	0

Functia pe care va trebui sa o implementati voi va folosi variabilele **n** (numarul de termeni) si **expr** (vectorul cu termenii expresiei).

$n = 5$  si `expr = ['T', '&', 'F', '^', 'T']` (`expr = [ true and false xor true ]`)

Raspuns: 2

Explicatie: Exista 2 moduri corecte de a paranteza expresia astfel incat sa obtinem rezultatul **true** (1).

- `T & (F ^ T)`
  - `(T & F) ^ T`
- $$\rightarrow (T \wedge F) \wedge T = F \wedge T = F$$

Complexitate temporală dorită este  $O(n^3)$ .

Optional, se pot defini functii ajutatoare precum `**is_operand**`, `**is_operator**`, `**evaluate**`.

Pentru rezolvarea celor doua probleme ganditi-va la ce scrie in sectiunea Sfaturi / Reguli [<http://ocw.cs.pub.ro/courses/pa/laboratoare/laborator-04?#sfaturireguli>]. Pentru fiecare dintre cele doua probleme facem o **partitionare dupa un anumit criteriu**.

Pentru problema **DP or math?** partitionam toate subsirurile dupa criteriul **paritatii sumei subsirului** (cate sunt pare/impare).

Pentru problema **expresie booleana** partitionam toate parantezarile posibile dupa rezultatul lor (cate dau true/false).

## Bonus

Asistentul va alege una dintre problemele din sectiunea Extra.

Recomandam sa **NU** fie una din cele 3 probleme de la Test PA 2017. Recomandam sa le incercati dupa ce recapitulati acasa DP1 si DP2, pentru a verifica daca cunostintele acumulate sunt la nivelul asteptat.

## Extra

Rezolvati problema extraterestrii [<https://www.hackerrank.com/contests/test-practic-pa-2017-v1-plumbus/challenges/test-1-extraterestrii>] de la Test PA 2017.

Rezolvati problema Secvente [<https://www.hackerrank.com/contests/test-practic-pa-2017-v1-plumbus/challenges/test-1-secvente>] de la Test PA 2017.

Rezolvati problema PA Country [<https://www.hackerrank.com/contests/test-practic-pa-2017-v2-meeseeks/challenges/test-2-pa-country-medic>] de la Test PA 2017.

Rezolvati pe infoarena problema iepuri [<http://infoarena.ro/problema/ieipuri>].

Hint: Exponentiere logaritmica pe matrice

Solutie:

- $dp[0] = X; dp[1] = Y; dp[2] = Z;$
- $dp[i] = (A * dp[i-1] + B * dp[i-2] + C * dp[i-3]) \% 666013$

Pentru punctaj maxim, pentru fiecare test se foloseste ecuatia matriceala atasata. Complexitate:  $O(T * \log(n))$ .

Rezolvati pe leetcode problema Minimum Path Sum [<https://leetcode.com/problems/minimum-path-sum/description/#>].

Rezolvati pe infoarena problema Lacusta [<http://infoarena.ro/problema/Lacusta>].

Rezolvati pe infoarena problema Suma4 [<http://infoarena.ro/problema/Suma4>].

Rezolvati pe infoarena problema subsir [<https://www.infoarena.ro/problema/subsir>].

$F[i][j]$  = nr. de posibilități de a parcurge de la  $i$  până la  $j$  și  
expresia să fie evaluată la fals

270

0	1	2	3	4	5	6	7
11	F	^	F	^	T	2	T

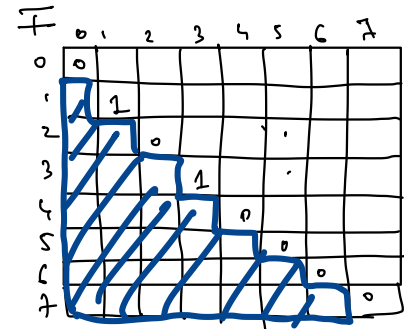
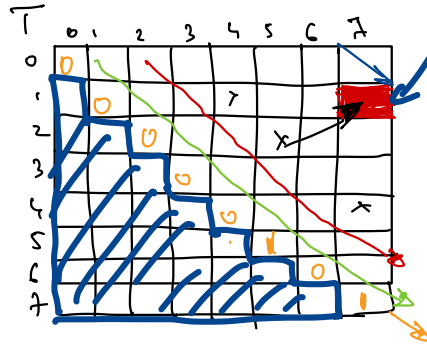
## CAZURILE DE BAZĂ

$$F[i][i] = 1, \quad \text{exp}[i] = F$$

2	0	1
0	0	0
1	0	1

11	0	1
0	0	1
1	1	1

1	0	1
0	0	1
1	1	0



(RECURRENTĂ)

$$K = \alpha' \int_{-1}^1$$

$$\exp[k] = |z|$$

$$TC[i][j] = \min_{1 \leq k \leq j-1} \{ TC[i][k] + TC[k+1][j] \}$$

$$f(i) \cdot j = f(i) \cdot (k-1) * f(k+1) \cdot j +$$

$$\exp[k] = \text{'||'}$$

$$T[i][j] = T[i][k-1] * T[k+1][j] + T[i][k] * f[k+1][j] + f[i][k-1] * T[k+1][j]$$

$$f[i][j] = f[i][k-1] * f[k+1][j]$$

$$\exp[k] = |A|$$

$$T(i) [j] = T(i) [k-1] * f[k+1] [j] +$$

$$T[i][j] = T[i][k-1] * T[k+1][j] + T[i][k-1] * T[k+1][j]$$

Rezolvati pe infoarena problema 2sah [<https://infoarena.ro/problema/2sah>].

Hint: Exponentiere logaritmica pe matrice

O descrie detaliata se afla in arhiva OJI 2015 [<http://olimpiada.info/oji2015/index.php?cid=arhiva>].

Articolul de pe leetcode [<https://leetcode.com/discuss/general-discussion/458695/Dynamic-Programming-Patterns>] conține o listă cu diverse tipuri de probleme de programare dinamică, din toate categoriile discutate la PA.

## 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/podm> [<http://infoarena.ro/problema/podm>]

[2] <http://infoarena.ro/problema/kfib> [<http://infoarena.ro/problema/kfib>]

pa/laboratoare/laborator-04.txt · Last modified: 2021/03/09 10:04 by radu.nichita