Laborator 4: Programare Dinamică (continuare)

Responsabili:

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

Obiective laborator

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

Precizari initiale

Toate exemplele de cod se gasesc in demo-lab04.zip.

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

- Toate bucatile de cod prezentate in partea introductiva a laboratorului (inainte de exercitii) au fost testate. Cu toate acestea, este posibil ca din cauza mai multor factori (formatare, caractere invizibile puse de browser etc) un simplu copy-paste sa nu fie de ajuns pentru a compila codul.
- Va rugam sa incercati si codul din arhiva **demo-lab04.zip**, inainte de a raporta ca ceva nu merge. :D
- Pentru orice problema legata de continutul acestei pagini, va rugam sa dati email unuia dintre responsabili.

Ce este DP?

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

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

Pentru restul notiunilor prezentate pana acum despre DP, va rugam sa consulati pagina laboratorului 3.

Exemple clasice

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

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

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

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

Categoria 3: PODM

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

ATENTIE! Acest tip de recurente poate fi mai greu (decat celelalte). Puteti consulta **acasa** materialele puse la dispozitie pentru intelege mai bine aceasta categorie.

Caracteristici:

- Acest tip de problema presupune ca o putem formula ca pe o problema de tip **subinterval** $[i, j]$.
- Daca dorim sa gasim optimul pentru acest interval, va trebuie sa luam in calcul toate combinatiile de 2 subproblemele care ar fi putut genera solutie pentru probleme $[i, j]$.
- Se considera fiecare divizare in 2 subprobleme, data de intermediarul k
  - $[i, k]$ si $[k + 1, j]$ sunt cele 2 subprobleme pentru care cunoastem solutiile
  - atunci o solutie pentru $[i, j]$ se poate obtine imbinandu-le pe cele doua
  - ca sa gasim solutia cea mai buna
    - vom itera prin toate valorile k posibile
    - vom alege pe cea care maximizeaza solutia problemei $[i, j]$
- Calculul se face de la intervale mici (probleme usoare - $[i, i]$ sau $[i, i + 1]$) spre probleme generale (dimensiune generala - $[i, j]$). In final se ajunge si la dimensiunile initiale ($[1, n]$).
  - Privind imaginea de ansamblu, adica cum se completeaza matricea, obervam ca matricea dp se completeaza **diagonala cu diagonala**.

Exemple clasice

PODM

Enunt

Fie un produs matricial $M = M_1 M_2 ... M_n$. Putem pune paranteze in mai multe moduri si vom obtine acelasi rezultat (inmultire asociativa), dar este posibil sa obtinem numar diferit de **inmultiri scalare** .

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

Cerinta

Se cere sa se gaseasca o **parantezare optima de matrice** (PODM), adica sa se gaseasca o parantezare care sa minimizeze numarul de inmultiri scalare.

Exemple

$n = 3$

i	0	1	2	3
d	2	3	4	5

Raspuns: **64** (inmultiri scalare)

Explicatie: Avem 3 matrici:

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

In functie de ordinea efectuarii inmultirilor matriciale , numarul total de inmultiri scalare poate sa fie foarte diferit:

- $(AB)C$ ⇒ $24 + 40 = 64$ de inmultiri
  - explicatie: $X = (AB)$ genereaza $2 * 3 * 4 = 24$ inmultiri, (XC) genereaza $2 * 4 * 5 = 40$ de inmultiri
- $A(BC)$ ⇒ $60 + 30 = 90$ de inmultiri
  - explicatie: $X = (BC)$ genereaza $3 * 4 * 5 = 60$ inmultiri, (AX) genereaza $2 * 3 * 5 = 30$ de inmultiri

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

$n = 4$

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

Raspuns: **48** (inmultiri scalare)

Explicatie: Avem 4 matrici:

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

In functie de ordinea efectuarii inmultirilor matriciale, numarul total de inmultiri scalare poate sa fie foarte diferit:

- $(AB)C)D$ ⇒ $24 + 16 + 12 = 52$ inmultiri
  - explicatie: $X = (AB)$ genereaza $2 * 3 * 4 = 24$ inmultiri scalare, $Y = (XC)$ genereaza $2 * 4 * 2 = 16$ inmultiri scalare, $Z = YD$ genereaza $2 * 2 * 3 = 12$ inmultiri scalare
- $(A(BC))D$ ⇒ $24 + 12 + 12 = 48$ inmultiri
  - explicatie: $X = (BC)$ genereaza $3 * 4 * 2 = 24$ inmultiri scalare, $Y = (AX)$ genereaza $2 * 3 * 2 = 12$ inmultiri scalare, $Z = YD$ genereaza 2 * 2 * 3$= 12$ inmultiri scalare
- $(AB)(CD)$ ⇒ $=$ inmultiri
  - explicatie: $X = (AB)$ genereaza $2 * 3 * 4 = 24$ inmultiri scalare, $Y = (CD)$ genereaza $4 * 2 * 3 =$24 inmultiri scalare, $Z = XY$ genereaza $2 * 4 * 3 = 24$ inmultiri scalare
- $A((BC)D)$ ⇒ $24 + 18 + 27 = 69$ inmultiri
  - explicatie: $X = (BC)$ genereaza $3 * 4 * 2 = 24$ inmultiri scalare, $Y = (XD)$ genereaza $3 * 2 * 3 = 18$ inmultiri scalare, $Z = AY$ genereaza $3 * 3 * 3 = 27$ inmultiri scalare
- $A(B(CD))$ ⇒ $24 + 36 + 18 = 78$ inmultiri
  - explicatie: $X = (CD)$ genereaza $4 * 2 * 3 = 24$ inmultiri scalare, $Y = (BX)$ genereaza $3 * 4 * 3 = 36$ inmultiri scalare, $Z = AY$ genereaza $2 * 3 * 3 = 18$ inmultiri scalare

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

$n = 4$

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

Raspuns: **2856** (inmultiri scalare)

Explicatie: Avem 4 matrici:

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

In functie de ordinea efectuarii inmultirilor matriciale , numarul total de inmultiri scalare poate sa fie foarte diferit:

- $((AB)C)D$ ⇒ 10582 inmultiri
- $(AB)(CD)$ ⇒ 54201 inmultiri
- $(A(BC))D$ ⇒ 2856 inmultiri
- $A((BC)D)$ ⇒ 4055 inmultiri
- ...

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

TIPAR

A fost descris in detaliu mai sus (cand s-a vorbit de categorie).

Numire recurenta

$dp[i][j]$ = **numarul minim de inmultiri scalare** cu care se poate obtine produsul $M_i * M_{i+1} * ... * M_j$

Gasire recurenta

- **Cazul de baza** :
  - $dp[i][i] = 0$
    - NU avem nici un efort daca nu avem ce inmulti.
  - $dp[i][i+1] = d_{i-1}d_i d_{i+1}$
    - Daca avem doua matrice, putem doar sa le inmultim. Nu are sens sa folosim paranteze.
    - Daca inmultim 2 matrice de dimensiuni $d_{i-1} * d_i$ si $d_i * d_{i+1}$, avem costul $d_{i-1}d_i d_{i+1}$
- **Cazul general**: $dp[i][j] = min(dp[i][k] + dp[k+1][j] + d_{i-1}d_k d_j)$, unde $k = i : j - 1$
  - daca avem de efectuat sirul de inmultiri $M_i \ldots M_j$, atunci putem pune paranteze oriunde si sa facem inmultirile astfel $(M_i \ldots M_k)(M_{k+1} \ldots M_j)$
    - costul minim pentru $(M_i \ldots M_k)$ este $dp[i][k]$
    - costul minim pentru $(M_{k+1} \ldots M_j)$ este $dp[k+1][j]$
    - vom avea in final de inmultit 2 matrice de dimensiune $d_{i-1} * d_k$ si $d_k * d_j$, operatie care are costul $d_{i-1}d_k d_j$
  - insumam cele 3 costuri intermediare

Implementare

Puteti rezolva si testa problema PODM pe infoarena aici [https://infoarena.ro/problema/podm].

Un exemplu de implementare in C++ se gaseste mai jos.

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

// T = O(n ^ 3)
// S = O(n ^ 2) - stocam n x n intregi in tabloul dp
 unsigned long long solve_podm(int n, const vector<int> &d) {
    // dp[i][j] = numarul MINIM inmultiri scalare cu codare poate fi calculat produsul
    //             matricial M_i * M_i+1 * ... * M_j
    vector<vector<unsigned long long>>  dp(n + 1, vector<unsigned long long> (n + 1, kInf));

    // Cazul de baza 1: nu am ce inmulti
    for (int i = 1; i <= n; ++i) {
        dp[i][i] = 0ULL;  // 0 pe unsigned long long (voi folosi mai incolo si 1ULL)
    }

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

    // Cazul general:
    // dp[i][j] = min(dp[i][k] + dp[k + 1][j] + d[i - 1] * d[k] * d[j]), k = i : j - 1
    for (int len = 2; len <= n; ++len) {            // fixam lungimea intervalului (2, 3, 4, ...)
        for (int i = 1; i + len - 1 <= n; ++i) {    // fixam capatul din stanga: i
            int j = i + len - 1;                    // capatul din dreapta se deduce: j

            // Iteram prin indicii dintre capete, spargand sirul de inmultiri in doua (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];

                // actualizam solutia daca este mai buna
                dp[i][j] = min(dp[i][j], new_sol);
            }
        }
    }

    // Rezultatul se afla in dp[1][n]: Numarul MINIM de inmultiri scalare
    // pe care trebuie sa le facem pentru a obtine produsul M_1 * ... * M_n
    return dp[1][n];

}
```

Sursa a fost scrisa pentru a fi testata pe infoarena. In cazul problemei PODM [https://infoarena.ro/problema/podm], deoarece avem o suma de foarte multe produse, rezultatul este foarte mare. Pe infoarena se cerea ca rezultatul sa fie afisat asa cum e, garantandu-se ca incape pe 64 biti.

Reamintim ca prin inmultirea/adunarea a doua variabile de tipul **int**, rezultatul poate sa nu incapa pe 32 biti. De aceea, in solutia prezentata, s-a facut cast pe 64biti.

ATENTIE! La PA, in general, vom folosi conventia $expresie \ \% \ kMod$, care va fi detaliata in capitolul urmatorul din acest laborator.

Complexitate

Intrucat solutia presupune fixarea capetelor unui subinterval (i, j), apoi alegerea unui intermediar (k), complexitatea este data de aceste 3 cicluri.

- **complexitate temporala**: $T(n) = O(n^3)$
- **complexitate spatiala**: $S(n) = O(n^2)$

Categoria 4: NUMARAT

Aceste recurente au o oarecare asemanare:

- toate numara lucruri! :p
- interesante sunt cazurile cand numarul cautat este foarte mare (altfel am putea apela la alte metode - ex. generarea tuturor candidatilor posibili cu backtracking)
  - in acest caz, deoarece numarul poate sa nu incapa pe un tip reprezentabil standard (ex. int pe 32/64 de biti), se cere (de obicei) restul impartirii numarului cautat la un numar **MOD** (vom folosi in continuare aceasta notatie).

Sfaturi / Reguli

- cand cautati o recurenta pentru o problema de numarare trebuie sa aveti grija la doua aspecte:
  - 1) sa **NU** numarati acelasi obiect de doua ori.
  - 2) sa numarati toate obiectele in cauza.
- de multe ori o problema de numarare implica o partitionare a **tuturor** posibilelor solutii dupa un anumit criteriu (relevant). Gasirea criteriului este partea esentiala pentru gasirea recurentei.

Regulile de lucru cu clase de resturi

Reamintim cateva proprietati matematice pe care ar trebui sa le aveti in vedere atunci in implementati pentru obtine corect resturile pentru anumite expresii. (corect poate sa insemne, de exemplu, sa evitati overflow :D - lucru neintuitiv cateodata).

- proprietati de de baza
  - $(a + b) \ \% \ MOD = ((a \ \% \ MOD) + (b \ \% \ MOD)) \ \% \ MOD$
  - $(a \ * b) \ \% \ MOD = ((a \ \% \ MOD) \ * (b \ \% \ MOD)) \ \% \ MOD$
  - $(a - b) \ \% \ MOD = ((a \ \% \ MOD) - (b \ \% \ MOD) + MOD) \ \% \ MOD$ (restul nu poate fi ceva negativ; in C++ **%** nu functioneaza pe numere negative)
- invers modular
  - $\frac{a}{b} \ \% \ MOD = ((a \ \% \ MOD) * (b^{MOD-2} \ \% \ MOD)) \ \% \ MOD)$
    - **DACA** MOD este prim; **DACA** a si b nu sunt multipli ai lui MOD

- **definitie** : **b** este inversul modular a lui **a** in raport cu **MOD** daca $a * b = 1(modulo \ MOD)$
- **utilizare** : $\frac{a}{b} \ \% \ MOD = ((a \ \% \ MOD) * (invers(b) \ \% \ MOD)) \ \% \ MOD$
- **calculare** : deoarece la PA aceasta discutie are sens doar in contextul posibilitatii implementarii unei recurente DP in care folosim resturile doar pentru a evita overflow/imposibilitatea de a retine rezultatul pe tipurile standard de tip int (adica nu ne intereseaza sa dam o metoda generala pentru invers modular), vom simplifica problema - **MOD este prim!!!**
- **Mica teorema a lui Fermat**: Daca p este un numar prim si a este un număr intreg care nu este multiplu al lui p, atunci $a^{p-1} = 1(modulo \ p)$.
  - din definitia inversului modular, reiese ca **a** si **b** nu sunt multipli ai lui **MOD**
  - introducand notatiile noastre in teorema si prelucrand obtinem
    - $a^{MOD-1} = 1(modulo \ MOD) <=> a * (a^{MOD-2}) = 1(modulo \ MOD)$
    - deci inversul modular al lui a (in aceste conditii specifice) este $b = a^{MOD-2}$

Reamintim ca prin inmultirea/adunarea a doua variabile de tipul int, rezultatul poate sa nu incapa pe 32 biti. E posibil sa trebuiasca sa combinam regulile cu resturi cu urmatoarele:

- C++
  - **1LL / 1ULL** - constanta 1 pe 64 biti cu semn / fara semn
    - **1LL * a * b** - am grija ca rezultatul sa nu dea overflow si sa se stocheze direct pe 64biti (cu semn)
- Java
  - **1L** - constanta 1 pe 64biti cu semn (in Java nu exista unsigned types)
    - **1L * a * b** - am grija ca rezultatul sa nu dea overflow si sa se stocheze direct pe 64biti (cu semn)

Gardurile lui Gigel

Enunt

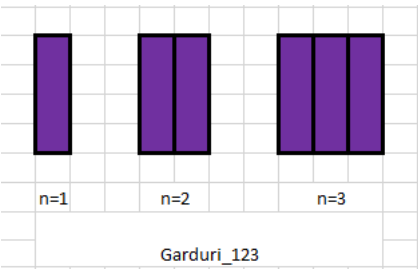
Gigel trece de la furat obiecte cu un rucsac la numarat garduri (fiecare are micile lui placeri :D). El doreste sa construiasca un gard folosind in mod repetat **un singur tip de piesa**.

O piesa are dimensiunile **4 x 1** (o unitate = 1m). Din motive irelevante pentru aceasta problema, orice gard construit trebuie sa aiba **inaltime 4m** in orice punct.

O piesa poate fi pusa in pozitie **orizontala** sau in pozitie **verticala**.

Cerinta

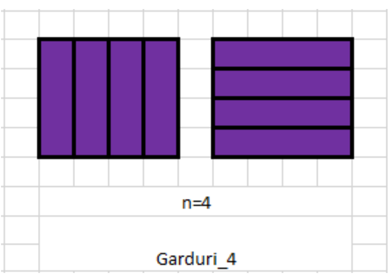
Gigel se intreaba **cate garduri de lungime n** si inaltime 4 exista? Deoarece celalalt prenume al lui este Bulănel, el intuieste ca acest numar este foarte mare, de aceea va cere **restul impartirii** acestui numar la **1009**.



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

Raspuns: **1** (un singur gard)

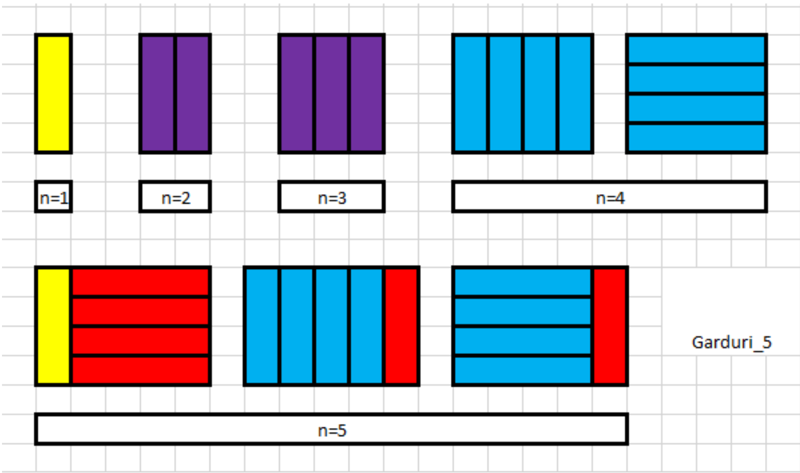
Explicatie: Se poate forma un singur gard in fiecare caz, dupa cum este ilustrat si in figura **Garduri_123**.



$n = 4$

Raspuns: **2**

Explicatie: Se pot forma 2 garduri, in functie de cum asezam piesele, dupa cum este ilustrat si in figura **Garduri_4**. Observam ca de fiecare daca cand punem o piesa in pozitie orizontala, de fapt suntem obligati sa punem 4 piese, una peste alta!

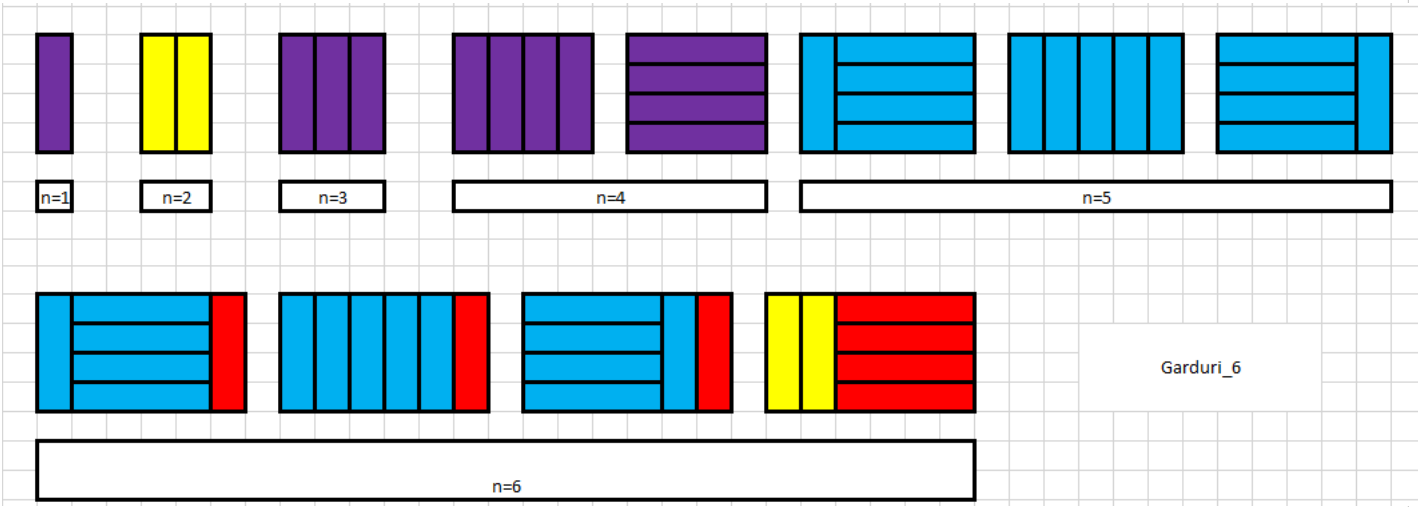


$n = 5$

Raspuns: **3**

Explicatie: Se pot forma 3 garduri, in functie de cum asezam piesele, dupa cum este ilustrat si in figura **Garduri_5**.

- 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 1**, in toate modurile posibile
- 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 4**, in toate modurile posibile



$n = 6$

Raspuns: **4**

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 ceeea 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 ($i >= 5$) 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 **partionare** dupa un anumit **criteriu**, in cazul problemei de fata criteriul de partionare 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 $dp[i] = (dp[i-1] + dp[i-4]) \ \% \ MOD$ in loc de $dp[i] = ((dp[i-1] \ \% \ MOD) + (dp[i-4] \ \% \ MOD)) \ \% \ MOD$, deoarece [e valorile anterior calcule in dp a fost deja aplicata operatia .

Am plecat cu numerele $1, 1, 1, 2$ si la fiecare pas rezultatul stocat este $\% \ MOD$, deci tot ce este stocat **deja** in dp este un rest in raport cu MOD. NU mai era nevoie deci sa aplica **%** si pe termenii din paranteza.

Complexitate

- **complexitate temporala**: $T = O(n)$
  - explicatie: avem o singura parcurgere in care construim tabloul dp
  - se poate obtine $T = O(logn)$ folosind exponentiere pe matrice!
- **complexitate spatiala**: $S = O(n)$
  - explicatie: stocam tabloul dp
  - se poate ontine $S = O(1)$ folosind exponentiere pe matrice!

Tehnici folosite in DP

De multe ori este nevoie sa folosim cateva tehnici pentru a obtine performanta maxima cu recurenta gasita.

In laboratorul 3 se mentiona tehnica de memoizare (in prima parte a laboratorului). In acesta ne vom rezuma la cum putem folosi cunostintele de lucru matricial pentru a favoriza implementarea unor anumite tipuri de recurente.

Exponentiere pe matrice pentru recurente liniare

Recurente liniare

O recurenta liniara in contextul laboratorului de DP este de forma:

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

O astfel de recurenta ar insemna ca pentru a calcula **costul problemei i** , imbinam costurile problemelor $i-1, i-2, \dots, i-k$, fiecare contribuind cu un anumit coeficient $c_1, c_2, \dots, c_k$.

Presupunand ca nu mai exista alte specificatii ale problemei si ca avand cele KMAX cazuri de baza (primele KMAX valori ar trebui stiute/deduse prin alte reguli), atunci un algoritm poate implementa recurenta de mai sus folosind 2 cicluri de tip for (for i = 1 : n, for k = 1 : KMAX …).

- **complexitatea temporala** : $T = O(n * KMAX) = O(n)$
  - reamintim ca acea valoarea KMAX este o constanta foarte mica in compartie cu n (ex. KMAX < 100)
- **complexitatea spatiala** : $S = O(n)$

- am presupus ca avem nevoie sa retinem doar tabloul dp

Exponentiere pe matrice

Facem urmatoarele notatii:

- 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 initiala (in care cunoaste cele k cazuri de baza)
  - $S_k = (dp[1], dp[2], \dots, dp[k-1], dp[k])$
- C = matrice ce coeficienti constanti
  - are dimensiune $KMAX * KMAX$
  - putem pune constante in clar
  - putem pune constantele c_k care tin de problema curenta

Algoritm naiv

Putem formula problema astfel:

- S_k = este starea initiala
- pentru a obtine starea urmatoare, aplicam algoritmul urmator
  - $S_i = S_{i-1}C$

Determinare C

Pentru a determina elementele matricei C, trebuie sa ne uitam la inmultirea matriceala de mai sus si sa alegem elementele lui C astfel incat prin inmultirealui S_{i-1} cu C sa obtinem elementele din S_i.

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

- 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 temporala $O(n)$.

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

$$S_i = S_{i-1}C$$
$$S_i = S_{i-2}C^2$$
$$S_i = S_{i-3}C^3$$
$$\dots$$
$$S_i = S_kC^{i-k}$$

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 temporala** : $T = O(KMAX^3 * log(n))$
  - explicatie
    - facem doar $O(logn)$ 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 $KMAX << n$ (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:

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

Exponentiere rapida :p

- $k = 4$
- $S_4 = (dp[1], dp[2], dp[3], dp[4]) = (1, 1, 1, 4)$
- $S_i = (dp[i-3], dp[i-2], dp[i-1], dp[i])$
- 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  incape 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 rezulate:

```
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

In acest laborator vom folosi scheletul de laborator din arhiva skel-lab04.zip.

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 $u = [v[i_1], v[i_2], \ldots, v[i_k]]]$ unde $i_1 < i_2 < \ldots < i_k$.

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 subsirule posibile sunt

- $[2]$
- $[2, 6]$
- $[2, 6, 4]$
- $[2, 4]$
- $[6]$
- $[6, 4]$
- $[4]$

Toate subsirurile de mai sus au suma para.

$n = 3$

i	1	2	3
v	2	1	3

Raspuns: 3

Explicatie: Toate subsirule posibile sunt

- $[2]$
- $[2, 1]$
- $[2, 1, 3]$
- $[2, 3]$
- $[1]$
- $[1, 3]$
- $[3]$

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

$n = 3$

i	1	2	3
v	3	2	1

Raspuns: 3

Explicatie: Toate subsirule posibile sunt

- $[3]$
- $[3, 2]$
- $[3, 2, 1]$
- $[3, 1]$
- $[2]$
- $[2, 1]$
- $[1]$

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

Morala: exista probleme pentru care gasim o solutie cu DP, dar pentru care poate exista si alte solutii mai bune (am ignorat citirea).

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

Dar cate subsiruri au suma **impara**?

Problema este preluata de aici [https://infoarena.ro/problema/azerah]. Solutia se gaseste aici [https://www.infoarena.ro/onis-2015/solutii-runda-1#azerah].

Expresie booleana

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**: '^'

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`

```
Complexitate temporală dorita 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 critieriul **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-medie] de la Test PA 2017.

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

Hint: Exponentiere logaritmica pe matrice

Solutie:

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

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].

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].

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]