Laborator 4: Programare Dinamică (2/2)

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 rugam 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

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 gasi 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 prima 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:

- $(AB)C)D \Rightarrow 24 + 16 + 12 = 52$ 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
- $(AB)(CD) \Rightarrow$ = inmulțiri
  - 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: $((A(BC))D)$.

$n = 4$

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

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

Răspunsul la problemă este **dp[1][n]** .

Găsire recurență

- **Cazul de bază** :
  - $dp[i][i] = 0$
    - NU avem niciun efort dacă nu avem ce înmulți.
  - $dp[i][i+1] = d_{i-1}d_id_{i+1}$
    - Dacă avem două matrice, putem doar să le înmulțim. Nu are sens să folosim paranteze.
    - Daca înmulțim 2 matrice de dimensiuni $d_{i-1} * d_i$ și $d_i * d_{i+1}$, avem costul $d_{i-1}d_id_{i+1}$
- **Cazul general**: $dp[i][j] = min(dp[i][k] + dp[k+1][j] + d_{i-1}d_kd_j)$, unde $k = i:j-1$
  - dacă avem de efectuat șirul de înmulțiri $M_i \ldots M_j$, atunci putem pune paranteze oriunde și să facem înmulțirile 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, în final, de înmulțit 2 matrice de dimensiune $d_{i-1} * d_k$ si $d_k * d_j$, operație care are costul $d_{i-1}d_kd_j$
  - 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.

```
// INF este valoarea maximă - "infinitul" nostru
const auto INF = 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, INF));

    // 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:
    // 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) {            // fixăm lungimea intervalului (2, 3, 4, ...)
        for (int i = 1; i + len - 1 <= n; ++i) {    // fixăm capătul din stânga: i
            int j = i + len - 1;                    // capătul din dreapta se deduce: j

            // Iterăm prin indicii dintre capete, spărgând șirul de înmulțiri in 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 inmultiri 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 asa cum e, garantându-se că încape 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 $expresie \% MOD$, 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 numară 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
        - $a^{MOD-1} = 1 (modulo\ MOD) <=> a * (a^{MOD-2}) = 1 (modulo\ MOD)$
        - 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 biti 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

Enunț

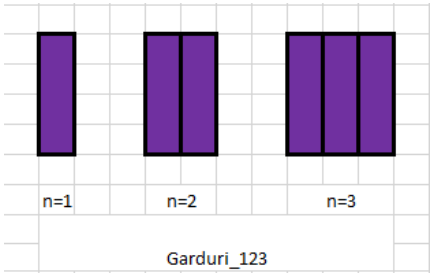
Gigel trece de la furat obiecte cu un rucsac la numărat garduri (fiecare are micile lui plăceri :D). El dorește să construiască un gard folosind în mod repetat **un singur tip de piesă**.

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

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

Cerință

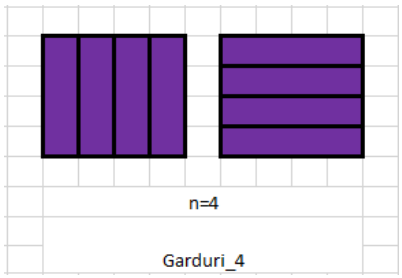
Gigel se întreabă **câte garduri de lungime n și înălțime 4** există? Deoarece celălalt prenume al lui este Bulănel, el intuiește că acest număr este foarte mare, de aceea va cere **restul împărțirii** acestui numar la **1009**.



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

Răspuns: **1** (un singur gard)

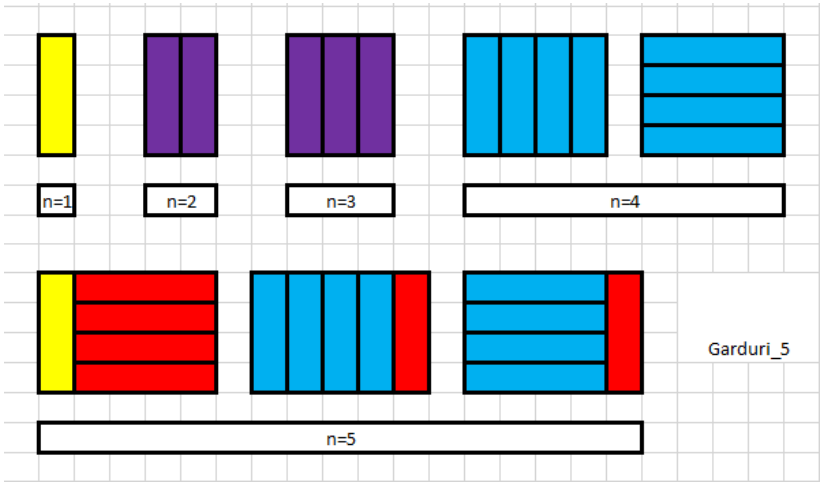
Explicație: Se poate forma un singur gard în fiecare caz, după cum este ilustrat și în figura **Garduri_123**.



$n = 4$

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!

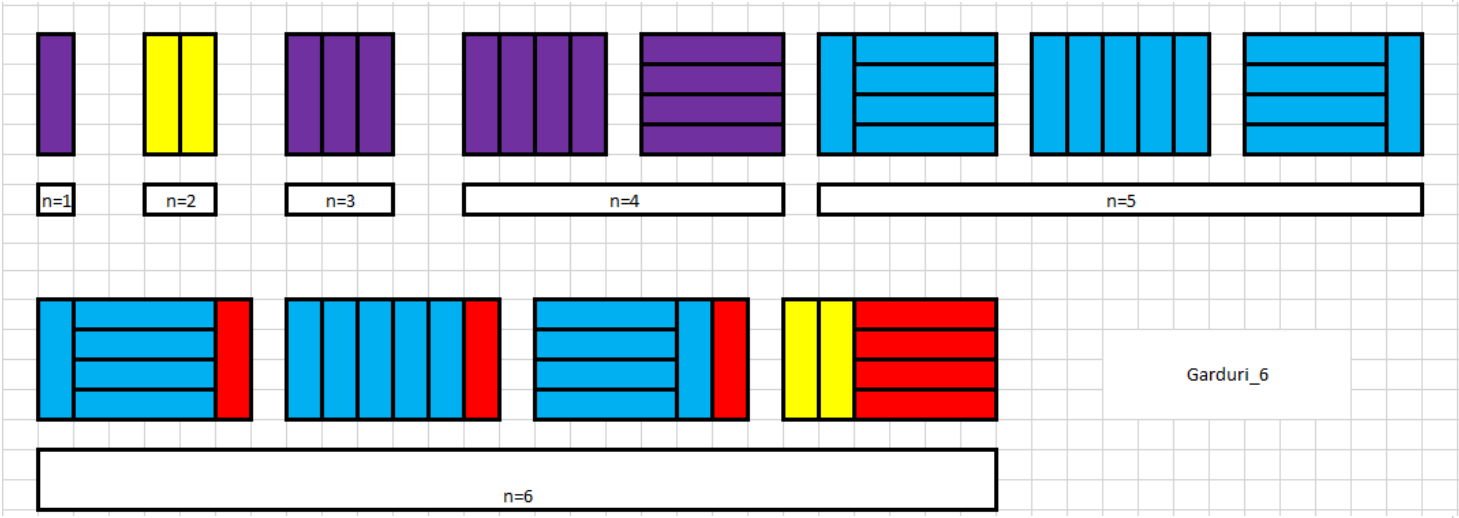


$n = 5$

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 stânga mai ramane 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 stânga mai rămâne de completat **un subgard de lungime 4**, în toate modurile posibile



$n = 6$

Răspuns: **4**

Explicație: Se pot forma 4 garduri, în funcție de cum așezăm piesele, după cum este ilustrat și în figura **Garduri_6**.

- dacă dorim ca acest gard să se termine cu o piesă în poziție **verticală** (marcat cu roșu), atunci la stânga mai rămâne de completat **un subgard de lungime 5**, în toate modurile posibile
- dacă dorim ca acest gard să se termine cu 4 piese în poziție **orizontală** (una peste alta - marcat cu roșu), atunci la stânga mai ramane de completat **un subgard de lungime 2**, în toate modurile posibile

Recurență

Numire recurență

$dp[i]$ = numărul de garduri de lungime i și înălțime 4 (nimic special - exact ceea ce se cere în enunț)

Răspunsul la problemă este $dp[n]$.

Găsire recurență

- **Caz de bază**
  - $dp[1] = dp[2] = dp[3] = 1$; $dp[4] = 2$
- **Caz general**
  - atunci când dorim să formăm un gard de lungime i ($i >= 5$) am văzut că putem alege cum să punem ultima/ultimele piese
    - **DACĂ** alegem ca ultima piesă să fie pusă în poziție verticală, atunci la stânga mai rămâne de completat **un subgard de lungime** $i - 1$
      - numărul de moduri în care putem face acest subgard este $dp[i - 1]$
    - **DACĂ** alegem ca ultima piesă să fie în poziție orizontală (de fapt, punem 4 piese în poziție orizontală), atunci la stânga mai rămâne de completat **un subgard de lungime** $i - 4$
      - numărul de moduri în care putem face acest subgard este $dp[i - 4]$
  - $dp[i] = (dp[i - 1] + dp[i - 4]) \% MOD$
  -

Așa cum am zis în secțiunea de sfaturi și reguli [http://ocw.cs.pub.ro/courses/pa/laboratoare/laborator-04?&#sfaturireguli] vrem să facem o **parționare** după un anumit **criteriu**: în cazul problemei de față, criteriul de parționare este dacă gardul se termină cu o scândură verticală sau orizontală.

De asemenea, tot în secțiunea sfaturi și reguli [http://ocw.cs.pub.ro/courses/pa/laboratoare/laborator-04?&#sfaturireguli] am precizat că nu vrem **să număram un obiect** (un mod de a construi gardul) **de două ori**. Recurența noastră (dp[i] = dp[i-1] + dp[i-4]) nu ia un obiect de două ori pentru că orice soluție care vine din dp[i-4] e diferită de alta care vine din dp[i-1] pentru că diferă în cel puțin ultima scândură așezată)

Implementare recurență

Aici puteți vedea un exemplu simplu de implementare în C++.

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

    vector<int> dp(n + 1); // păstrez indexarea de la 1 ca în explicații

    // cazurile de bază
    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];
}
```

Menționez că 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 aplicăm **%** și pe termenii din paranteză.

Complexitate

- **complexitate temporală**: $T = O(n)$
  - explicație: avem o singură parcurgere în care construim tabloul dp
  - se poate obține $T = O(logn)$ 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 performanța maximă cu recurența găsită.

În prima parte a laboratorului 3 se menționa tehnica de memoizare. În acesta, ne vom rezuma la cum putem folosi cunoștințele de lucru matriceal 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**, îmbinăm costurile problemelor $i-1, i-2, \ldots, i-k$, fiecare contribuind cu un anumit coeficient $c_1, c_2, \ldots, c_k$.

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 …).

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

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

- ultima coloană conține toți coeficienții c_k întrucât $dp[i] = \sum_{k=1}^{KMAX} c_k * dp[i-k]$
- celelalte coloane conțin doar câte o valoare nenulă
  - pe coloana j vom avea valoarea 1 pe linia $j+1$ ($j = 1 : KMAX - 1$)
    - cum obținem, de exemplu, $dp[i-1]$?
    - păi, avem $dp[i-1]$ chiar și în starea S_{i-1}, deci trebuie să îl copiem în starea S_i
      - copierea se realizează prin inmulțirea cu 1
      - dacă $dp[i-1]$ era pe ultima poziție (poziția k) în starea S_{i-1}, în noua stare S_i este pe penultima poziție (poziția $k-1$)
        - deci s-a deplasat la stânga cu o poziție!
  - în noua stare, noua poziție este deplasată cu o unitate la stânga față de starea precedentă
    - de aceea, pe coloana j, vrem să avem elementul 1 pe linia $j+1$ ($j = 1 : KMAX - 1$)
    - când înmulțim S_{i-1} cu coloana C_j **dorim să**
      - ce copiem?
        - valoarea $dp[i - KMAX + j]$ din S_{i-1} în S_i
        - adică să copiem a j-a valoare de pe linie
      - unde copiem?
        - de pe poziția $j+1$ pe poziția j

Exponențiere logaritmică pe matrice

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

Să executăm câțiva pași de inducție pentru a vedea cum este determinat S_i.

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

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

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

$$\dots$$

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

În laboratorul 2 (Divide et Impera) am învățat că putem calcula x^n în timp logaritmic. Deoarece și înmulțirea matricilor este asociativă, putem calcula C^n in timp logaritmic.

Obținem astfel o soluție cu următoarele complexități:

- **complexitate temporală** : $T = O(KMAX^3 * log(n))$
  - explicație
    - facem doar $O(logn)$ pași, dar un pas implică înmulțire de matrice
    - o înmulțire de matrice patrătică de dimensiune KMAX are $KMAX^3$ operații
  - această metodă este eficientă când $KMAX << n$ (KMAX este mult mai mic decât n)
- **complexitatea spațială** : $S = O(KMAX^3)$
  - explicație
    - este nevoie să stocăm câteva matrice

Observație! În ultimele calcule nu am șters constanta KMAX, întrucât apare la puterea a 3-a! $KMAX = 100$ implică $KMAX^3 = 10^6$, valoare care nu mai poate fi ignorată în practică ($KMAX^3$ poate fi comparabil cu n).

Gardurile lui Gigel (optimizare)

După cum am văzut mai sus, în problema cu garduri dată de Gigel, soluția este o recurență liniară:

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

Exponențiere rapidă

- $k = 4$
- $S_4 = (dp[1], dp[2], dp[3], dp[4]) = (1, 1, 1, 2)$
- $S_i = (dp[i-3], dp[i-2], dp[i-1], dp[i])$
- Răspunsul se află efectuând operația $S_n = S_4 * C^{n-4}$, unde C are următorul conținut:

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

Mai jos se află o implementare simplistă în C++ care cuprinde toate etapele pe care trebuie să le realizați în cod, după ce știți cum arată recurența sub forma matricală.

```
#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 că suma începe pe 64 de biți

            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;                       // rămâne de calculat C^(p/2)
        } else {
            // reduc la cazul anterior:
            multiply_matrix(tmp, C, tmp); // tmp = tmp*C
            --p;                          // rămâne de calculat C^(p-1)
        }
    }

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

int garduri_rapide(int n) {
    // cazurile de bază
    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 să 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 află pe ultima poziție din S_n,
   // deci voi folosi ultima coloană din C)
   int sol = 1 * C[0][3] + 1 * C[1][3] + 1 * C[2][3] + 2 * C[3][3];
   return sol % MOD;
}
```

Remarcați faptul că în funcția de înmulțire se folosește o matrice temporară *tmp*. Motivul este că vrem să apelăm funcția $multiply(C, C, C)$, unde C joacă atât rol de intrare cât și de ieșire. Dacă am pune rezultatele direct in C, atunci am strica inputul înainte să obținem rezultatul.

Putem spune că acea funcție este **matrix_multiply_safe**, în sensul că pentru orice A,B,C care respectă dimensiunile impuse, funcția va calcula corect produsul.

Pr git găsiți o sursă completă în care se realizează:

- o verificare a faptului că cele 2 implementări (**gardurile_lui_Gigel** și **garduri_rapide**) produc aceleași rezultate
- un benchmark în care cele 2 implementări sunt comparate
    - pe sistem uzual (laptop) s-au obținut următoarele rezulate:

```
test case: varianta simplă
n = 100000000 sol = 119; time = 0.984545 s
test case: varianta rapidă
n = 100000000 sol = 119; time = 0.000021 s


test case: varianta simplă
n = 1000000000 sol = 812; time = 9.662377 s
```

```
test case: varianta rapidă
n = 1000000000 sol = 812; time = 0.000022 s
```

- se observă clar diferența între cele 2 soluții (am confirmat ceea ce spunea și teoria: $O(n)$ vs $O(log(n))$); această tehnică îmbunătățește drastic o soluție gasită relativ usor.

Exerciții

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 șir de **numere naturale strict pozitive**. Câte **subșiruri** (submulțimi nevide) au suma numerelor **pară**?

subșir (**subsequence** în engleză) pentru un vector **v** înseamnă un alt vector $u = [v[i_1], v[i_2], \ldots, v[i_k]]$ unde $i_1 < i_2 < \ldots < i_k$.

Task-uri:

- Se cere o **soluție folosind DP**.
- Inspectând recurența gasită la punctul precedent, încercați să o înlocuiți cu o **formulă matematică**.
- Care este **complexitatea** pentru fiecare soluție (timp + spațiu)? Care este mai bună? De ce? :D

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

Pentru punctaj maxim pentru această problemă, este necesar să rezolvați toate subpunctele (ex. nu folosiți direct formula, găsiți mai întâi recurența DP). Trebuie să implementați **cel puțin** soluția cu DP.

$n = 3$

i	1	2	3
v	2	6	4

Răspuns: 7

Explicație: Toate subșirurile posibile sunt

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

Toate subșirurile de mai sus au suma pară.

$n = 3$

i	1	2	3
v	2	1	3

Răspuns: 3

Explicație: Toate subșirurile posibile sunt

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

Subșirurile cu sumă pară sunt: $[2]$, $[2, 1, 3]$, $[1, 3]$.

$n = 3$

i	1	2	3
v	3	2	1

Răspuns: 3

Explicație: Toate subșirurile posibile sunt

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

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

Subșirurile cu sumă pară 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).

În problemele de numărat, există o **șansă** bună să putem găsi (și) o formulă matematică, ce poate fi implementată într-un mod mai eficient decât o recurență DP.

Câte subșiruri au suma **impară**?

Expresie booleană

Se dă o expresie booleană corectă cu n termeni. Fiecare din termeni poate fi unul din stringurile **true**, **false**, **and**, **or**, **xor**.

Numărați modurile în care se pot așeza paranteze astfel încât rezultatul să fie **true**. Se respectă regulile de la logică (tabelele de adevăr pentru operațiile **and**, **or**, **xor**).

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

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

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

Funcția pe care va trebui să o implementați voi va folosi variabilele **n** (numărul de termeni) și **expr** (vectorul cu termenii expresiei).

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

Răspuns: 2

Explicație: Există 2 moduri corecte de a paranteza expresia astfel încât să obținem rezultatul **true** (1).

- `T&(F^T)`
- `(T&F)^T`

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

Opțional, se pot defini funcții ajutătoare precum **is_operand**, **is_operator**, **evaluate**.
```

Pentru rezolvarea celor două probleme gândiți-vă la ce scrise în secțiunea Sfaturi / Reguli [http://ocw.cs.pub.ro/courses/pa/laboratoare/laborator-04?&#sfaturireguli]. Pentru fiecare dintre cele două probleme facem o **partiționare după un anumit criteriu**.

Pentru problema **DP or math?** partiționăm toate subșirurile după critieriul **parității sumei subșirului** (câte sunt pare/impare).
Pentru problema **expresie booleană** partiționăm **toate parantezările posibile după rezultatul lor** (câte dau true/false).

Bonus

Asistentul va alege una dintre problemele din secțiunea Extra.

Recomandăm să **NU** fie una dintre cele 3 probleme de la Test PA 2017. Recomandăm să le incercați după ce recapitulați acasă DP1 și DP2, pentru a verifica dacă cunoștințele acumulate sunt la nivelul așteptat.

Extra

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

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

Rezolvați 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.

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

Hint: Exponențiere logaritmică pe matrice

Soluție:

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

Pentru punctaj maxim, pentru fiecare test se folosește ecuația matriceală atașată. Complexitate: $O(T * log(n))$.

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

Rezolvați pe infoarena problema Lăcusta [http://infoarena.ro/problema/Lacusta].

Rezolvați pe infoarena problema Suma4 [http://infoarena.ro/problema/Suma4].

Rezolvați pe infoarena problema subșir [https://www.infoarena.ro/problema/subsir].

Rezolvați pe infoarena problema 2șah [https://infoarena.ro/problema/2sah].

Hint: Exponențiere logaritmică pe matrice

O descriere detaliată se află în 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] Chapter **Dynamic Programming**, "Introduction to Algorithms", Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein

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

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