UNIVERSITATEA DIN
BUCUREȘTI

FACULTATEA DE
MATEMATICĂ ȘI
INFORMATICĂ

SPECIALIZAREA INFORMATICĂ

Lucrare de licență

PROGRAMARE DINAMICĂ CU MĂȘTI DE BIȚI

Absolvent
Ciobanu Dragoș

Coordonator științific
Lect. dr. Dumitran Adrian Marius

București, iunie 2025

Rezumat

Lucrarea este destinată elevilor și studenților pasionați de programare competitivă. În teză este descris preliminar modul în care numerele pot fi reprezentate și manipulate sub formă de măști de biți. Apoi, se trece la Programarea Dinamică cu Măști de Biți (Bitmask DP), care este folosită pentru a rezolva probleme complexe pe grafuri sau numere prime, ilustrate prin exemple clare. De asemenea, sunt oferite explicații suplimentare pentru tehnici precum Sum over Subsets DP, care ajută la calculul eficient al sumelor pe submăști și Plug DP, care este o tehnică utilizată în probleme care implică construcții pas cu pas. În cele din urmă, concluziile și graficele arată cât de utile sunt aceste metode pentru rezolvarea problemelor algoritmice.

Abstract

This paper is aimed at high-school and university students who are passionate about competitive programming. It begins with a basic explanation of how numbers can be represented and modified using bit masks. It then presents Bitmask Dynamic Programming (Bitmask DP), a technique for solving complex problems involving graphs or prime numbers, illustrated with clear examples. The article also explains techniques such as Sum over Subsets DP, used to efficiently compute sums over submasks, and Plug DP, a method for solving problems that require incremental construction. Finally, the conclusions and graphics demonstrate how useful these strategies are for tackling algorithmic challenges.

Cuprins

Capitolul 1

Introducere

1.1 Tipul lucrării și domeniul de încadrare

Lucrarea de față este o lucrare de licență care se încadrează în domeniul informaticii, având ca subdomeniu informatica teoretică. Tema propusă face parte din categoria metodelor de optimizare algoritmică, cu accent pe tehnicile de programare dinamică. Mai precis, lucrarea abordează o ramură specializată a acestei metode și anume programarea dinamică cu măști de biți, utilizată pentru probleme în care stările se pot reprezenta binar, iar procesarea lor se face eficient prin operații pe biți.

1.2 Prezentarea generală a temei

Programarea dinamică este o tehnică esențială pentru rezolvarea eficientă a problemelor care pot fi împărțite în subprobleme recurente, prin memorarea soluțiilor intermediare pentru a evita recalcularea acestora. O extensie importantă a acestei metode o reprezintă programarea dinamică cu măști de biți, în care stările problemei sunt codificate sub forma unor numere întregi, ale căror biți indică apartenența sau prezența anumitor elemente într-o configurație.

Această abordare se dovedește extrem de eficientă pentru probleme de optimizare pe mulțimi mici, pe grafuri de dimensiuni reduse sau în situații în care este necesară procesarea tuturor submăștilor unui set. Tehnica este frecvent utilizată în cadrul concursurilor de algoritmică, de la olimpiadele naționale de informatică, până la competiții internaționale.

1.3 Scopul și motivația alegerii temei

Scopul principal al lucrării este de a oferi o prezentare structurată și aplicată a programării dinamice cu măști de biți, completată de aplicații concrete, explicate pas cu pas, însoțite de analiza complexității algoritmilor folosiți. Motivația alegerii acestei teme vine

atât din utilitatea sa practică în concursurile de algoritmică, cât și din lipsa unor materiale concise, în limba română, care să abordeze tehnica într-un mod gradual, pornind de la noțiuni de bază și ajungând la aplicații complexe.

Experiența personală acumulată în pregătirea pentru concursuri de algoritmică a evidențiat necesitatea unui material care să faciliteze înțelegerea acestei tehnici, mai ales în cazul problemelor în care recunoașterea tiparului și alegerea unei structuri potrivite pentru stocarea stărilor reprezintă cheia obținerii unui algoritm eficient.

1.4 Contribuția proprie

Contribuția personală constă în structurarea și redactarea conținutului teoretic pe baza surselor bibliografice consultate, selecția și adaptarea problemelor studiate, dezvoltarea soluțiilor implementate și realizarea de grafice comparative pentru evidențierea diferențelor dintre soluțiile brute și cele optimizate cu ajutorul programării dinamice cu măști de biți. Fiecare capitol este structurat astfel încât să conțină partea teoretică, urmată de exemple de probleme relevante, iar la final, în Anexa A, sunt prezentate grafice comparative privind complexitatea teoretică a algoritmilor.

Graficele au rolul de a evidenția modul în care complexitatea algoritmilor variază în funcție de dimensiunea datelor de intrare, fără a include valori concrete de timp de execuție, acestea depinzând de platforma pe care se realizează rulările. Totodată, am structurat lucrarea astfel încât să urmeze o progresie logică, începând cu noțiunile introductive despre măști de biți, continuând cu aplicații pe grafuri, submăști, numere prime, tablouri de mici dimensiuni și încheind cu o concluzie finală.

1.5 Structura lucrării

Lucrarea este organizată în șapte capitole, fiecare având un rol bine definit:

- **Capitolul 1 — Introducere:** stabilește domeniul de încadrare, prezintă tema, scopul, motivația alegerii, contribuția proprie și structura generală a documentului.
- **Capitolul 2 — Preliminarii:** oferă conceptele de bază privind reprezentarea măștilor de biți, operații fundamentale pe biți, aplicații introductive și funcții utile din limbajul C++.
- **Capitolul 3 — Programare dinamică cu măști de biți aplicată pe grafuri de dimensiuni mici:** prezintă modul de aplicare a acestei tehnici pentru probleme pe grafuri, cum ar fi determinarea sortărilor topologice sau numărarea grafurilor aciclice.

- **Capitolul 4 — SOS DP:** explică o metodă de optimizare ce presupune parcurgerea tuturor submăștilor unei măști date, cu aplicații în probleme de optimizare pe mulțimi mici.
- **Capitolul 5 — Numere prime și aplicații în programarea dinamică cu măști de biți:** prezintă modul în care numerele prime pot fi codificate sub formă de măști de biți și utilizate în diverse aplicații algoritmice.
- **Capitolul 6 — Plug DP:** detaliază tehnica utilizată în rezolvarea problemelor pe matrici de dimensiuni mici, cu exemple vizuale care facilitează înțelegerea transferului de stare între celule.
- **Capitolul 7 — Concluzii:** sintetizează rezultatele lucrării, aprecierile personale și eventualele direcții de extindere a temei.

În final, **Anexa A** conține grafice comparative bazate pe complexitățile teoretice ale algoritmilor, evidențiind diferențele dintre soluțiile brute și cele optimizate cu programare dinamică cu măști de biți. **Anexa B** include implementări auxiliare cu aritmetică modulară și extensii ale capitolului SOS DP.

1.6 Scurt istoric și aplicații practice

Conceptul de programare dinamică a fost introdus de Richard Bellman în anii 1950[1], având ca scop optimizarea deciziilor succesive într-o succesiune de etape. De-a lungul timpului, metoda a fost extinsă în multiple direcții, una dintre cele mai aplicabile variante pentru probleme discrete fiind programarea dinamică cu măști de biți.

Această tehnică s-a impus în mediul competițional, dar și în aplicații practice, datorită capacității sale de a gestiona eficient submulțimi, de a optimiza configurații de grafuri și de a modela stări binare într-un mod compact. În afara concursurilor de algoritmică, metodele cu măști de biți sunt utilizate în:

- Prelucrarea imaginilor digitale.
- Sisteme de securitate informatică, pentru configurarea drepturilor de acces.
- Compresia datelor și algoritmi de criptare.
- Simularea circuitelor logice.
- Teoria jocurilor, pentru reprezentarea stărilor posibile într-un spațiu discret.

Prin urmare, programarea dinamică cu măști de biți nu este doar o tehnică teoretică sau specifică mediului competițional, ci are aplicabilitate reală în numeroase domenii practice din informatică.

[1] Bellman [1]

Capitolul 2

Preliminarii

2.1 Prelucrarea măștilor de biți

În programele informatice, toate datele sunt stocate intern sub formă de biți, adică secvențe de 0 și 1. Acest capitol explică modul în care numerele naturale sunt reprezentate folosind biți și oferă exemple practice de utilizare a operațiilor pe biți.

Toate informațiile din acestă secțiune au la bază Capitolul 10 din Laaksonen [2].

2.1.1 Reprezentarea unui număr natural sub formă de biți

Un număr natural X poate fi reprezentat în binar pe B biți astfel:

$$X = 2^{B-1} \cdot b_{B-1} + 2^{B-2} \cdot b_{B-2} + \cdots + 2^1 \cdot b_1 + 2^0 \cdot b_0, \quad \text{unde } b_i \in \{0, 1\}.$$

Masca de biți asociată lui X este: $b_{B-1}b_{B-2}b_{B-3} \ldots b_1b_0$.

2.1.2 Operații pe măști de biți

În această secțiune, vom folosi numere reprezentate pe **8 biți** (de la bitul cel mai semnificativ la cel mai puțin semnificativ).

Operatorul & (ȘI)

Operația $x \,\&\, y$ returnează un număr ale cărui biți de 1 apar doar în pozițiile unde ambii operanzi x și y au biți de 1.

De exemplu, pentru $29 \,\&\, 16$:

$$29 = 0001\,1101$$
$$16 = 0001\,0000$$

Aplicând operația bit cu bit:

$$\begin{array}{cccccccccc} & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & (29) \\ \& & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & (16) \\ \hline = & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & (16) \end{array}$$

Operatorul | (SAU)

Operația $x \,|\, y$ returnează un număr ale cărui biți de 1 apar în pozițiile unde cel puțin unul dintre operanzii x și y are biți de 1.

De exemplu, pentru $18 \,|\, 6$:

$$18 = 0001\,0010$$
$$6 = 0000\,0110$$

Aplicând operația bit cu bit:

$$\begin{array}{cccccccccc} & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & (18) \\ | & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & (6) \\ \hline = & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & (22) \end{array}$$

Operatorul ^ (XOR)

Operația $x \hat{} y$ (XOR) returnează un număr ale cărui biți de 1 apar în pozițiile unde exact unul dintre operanzii x și y are biți de 1. Aceasta mai poate fi găsită sub notația $\oplus$.

De exemplu, pentru 21^3:

$$21 = 0001\,0101$$
$$3 = 0000\,0011$$

Aplicând operația bit cu bit:

$$\begin{array}{cccccccccc} & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & (21) \\ \hat{} & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & (3) \\ \hline = & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & (22) \end{array}$$

Deplasări pe biți

Deplasarea la stânga $x << k$ adaugă k biți de zero la sfârșitul numărului x, iar deplasarea la dreapta $x >> k$ elimină ultimii k biți din numărul x.

Exemple:

$$20 << 2 = 80$$

$$21 >> 3 = 2$$

Notă:

$x << k$ corespunde înmulțirii lui x cu 2^k

$x >> k$ corespunde împărțirii lui x la 2^k, rotunjit în jos.

Operatorul Tilda ($\sim$) Complement pe Biți

Operatorul $\sim$ este un operator unar care inversează fiecare bit al unui număr.
Exemplu:

$$29 = 0001\,1101$$
$$\sim 29 = 1110\,0010$$

2.1.3 Aplicații

Verificarea valorii bitului de pe poziția *k* din masca de biți *x*

Un număr de forma `1 << k` are doar bitul de pe poziția k setat, toți ceilalți biți fiind zero. Putem folosi astfel de numere pentru a accesa biți individuali dintr-un număr. În particular, al k-lea bit al lui x este 1 exact atunci când expresia `x & (1 << k)` este diferită de zero.

Activarea bitului de pe poziția *k* dintr-o mască de biți *x*

Activarea unui bit specific într-o mască de biți presupune setarea acelui bit la valoarea 1, fără a modifica ceilalți biți ai măștii.

Pentru a activa un bit specific, se folosește operația sau (`|`) bit cu bit între numărul original și o mască în care doar bitul dorit este setat la 1. Astfel, `x = x | (1 << k)` va activa bitul de pe poziția k în masca de biți x.

Dezactivarea bitului de pe poziția *k* dintr-o mască de biți *x*

Dezactivarea unui bit specific într-o mască de biți presupune setarea acelui bit la valoarea 0, fără a modifica ceilalți biți ai măștii.

Pentru a dezactiva un bit specific, se folosește operația și (`&`) bit cu bit între numărul original și complementul măștii în care doar bitul dorit este setat la 1. Astfel, `x = x & ~(1 << k)` va dezactiva bitul de pe poziția k în masca de biți x.

Inversarea bitului de pe poziția k dintr-o mască de biți x

Inversarea unui bit specific într-o mască de biți presupune schimbarea stării acelui bit de la 0 la 1 sau de la 1 la 0, fără a afecta ceilalți biți ai măștii.

Pentru a inversa un bit specific, se folosește operația XOR (`^`) bit cu bit între numărul original și o mască în care doar bitul dorit este setat la 1. Astfel, `x = x ^ (1 << k)` va inversa bitul de pe poziția k din masca de biți x.

Determinarea celui mai puțin semnificativ bit k setat dintr-o mască de biți x

Determinarea celui mai puțin semnificativ bit (LSB - Least Significant Bit) setat la 1 într-o mască de biți este o operație esențială în diverse aplicații de programare, cum ar fi algoritmi de optimizare, manipularea datelor binare și gestionarea resurselor. Această operație identifică poziția primului bit setat la 1, începând de la bitul cel mai puțin semnificativ (bitul de la poziția 0).

Pentru a determina cel mai puțin semnificativ bit setat într-o mască de biți x, se pot utiliza diverse tehnici bitwise. Una dintre cele mai eficiente metode implică utilizarea operației de AND (`&`) între x și complementul lui `x - 1`. Astfel, `LSB = x & ~(x - 1)` determină cel mai mic bit, iar k se poate calcula prin `log2(LSB)` din masca de biți x.

Pentru a elimina cel mai puțin semnificativ bit dintr-o mască de biți x, se poate folosi o metodă care implică operația AND (`&`) între x și `x - 1`. Astfel, `x - (1 << k) = x & (x - 1)` determină masca de biți x fără bitul k.

Determinarea celui mai semnificativ bit k setat dintr-o mască de biți x

Determinarea celui mai semnificativ bit (MSB - Most Significant Bit) setat la 1 într-o mască de biți x este o operație esențială în diverse domenii ale informaticii și ingineriei software. Această operație identifică poziția celui mai înalt bit care este setat la 1.

Într-o reprezentare binară a unui număr, cel mai semnificativ bit este bitul situat la cea mai înaltă poziție, adică cel mai din stânga. Astfel, `MSB = (1 << log2(x))` determină cel mai mare bit, k fiind `log2(x)`, din masca de biți x.

2.1.4 Funcții din C++ care lucrează pe măști de biți

Compilatorul `g++` oferă următoarele funcții:

- `__builtin_clz(x)`: numărul de zero de la începutul măștii x
- `__builtin_ctz(x)`: numărul de zero de la sfârșitul măștii x
- `__builtin_popcount(x)`: numărul de biți de 1 din masca x
- `__builtin_parity(x)`: paritatea numărului de biți de 1 din masca x

Capitolul 3

Programare dinamică cu măști de biți pe grafuri mici

3.1 Prezentare aplicații

Programarea dinamică cu măști de biți conține numeroase aplicații pe grafuri de dimensiuni mici. În continuare vor fi prezentate următoarele probleme cu rezolvări eficiente bazate pe această tehnică:

- *Numărul de sortări topologice dintr-un graf orientat aciclic.*
- *Numărul total de grafuri direcționate aciclice generate dintr-un graf neorientat.*

Motivația acestui capitol este rezolvarea problemei *Amusement Park* dată la Olimpiada Central Europeană (CEOI) din 2019.

3.1.1 Determinarea unei sortări topologice arbitrare dintr-un graf orientat aciclic

Pentru a putea aborda eficient problemele discutate anterior, este util să clarificăm noțiunile de graf orientat aciclic și sortare topologică. În final, vom prezenta și un algoritm clasic pentru obținerea unei sortări topologice arbitrare: Algoritmul lui Kahn.

Un graf orientat aciclic (DAG) este un graf orientat care nu conține cicluri, adică nu există un nod de la care, urmând arcele orientate, se poate ajunge înapoi la același nod.

Sortarea topologică a unui DAG este o permutare liniară a nodurilor astfel încât, pentru fiecare arc orientat $u \to v$, nodul u apare înaintea nodului v în permutare.

Dat fiind un graf orientat aciclic $G = (V, E)$, unde V reprezintă setul de noduri și E setul de arce, se dorește găsirea unei sortări topologice arbitrare ale grafului G.

Algoritmul lui Kahn Algoritmul se bazează pe o observație cheie: dacă pentru un nod u nu există un alt nod v astfel încât să existe muchia $v \to u$, atunci nodul u poate fi unul dintre următoarele noduri în sortarea topologică.

Astfel, algoritmul funcționează prin procesul repetat de a găsi noduri u cu proprietatea de mai sus (adică gradul interior al lui u la momentul respectiv este 0). Gestionarea acestor noduri se poate realiza cu ajutorul unei cozi, inserând de fiecare dată un nod cu grad interior 0 și ulterior scăzând gradul interior al nodurilor adiacente care ies din u.

3.1.2 Numărul de sortări topologice dintr-un graf orientat aciclic

Având graful orientat aciclic $G = (V, E)$, unde V reprezintă setul de noduri și E setul de arce, se dorește calcularea numărului total de sortări topologice distincte ale grafului G.

Pornind de la ideea centrală a Algoritmului lui Kahn, putem aborda problema calculului numărului de sortări topologice folosind programare dinamică, monitorizând starea curentă a grafului. În esență, pentru a face tranziția de la o stare x la o stare diferită $y \neq x$ este necesar să adăugăm în lista de sortare topologică un nod nou cu grad interior zero. Această abordare ne permite definirea următoarelor tablouri unidimensionale:

- `dependență[i]` = mască de biți care conține toate nodurile care au muchii către nodul i.
- `dp[mask]` = numărul de sortări topologice pentru masca de biți `mask`. Masca de biți reprezintă nodurile care aparțin grafului construit până în acel moment.

Observație: Masca de biți `01001` conține nodurile 0 și 3.

Pentru a verifica dacă un nou nod i se poate adăuga în lista cu sortarea topologică, trebuie ca toate dependențele acestuia să se afle deja în masca de biți. Astfel, expresia `(mask & dependentă[i]) == dependentă[i]` trebuie să fie adevărată.

Numărul de modalități va fi actualizat cu `dp[mask]` pentru noua mască de biți, astfel:

```
dp[mask | (1 << i)] += dp[mask]
```

Rezultatul final se află în tabloul `dp` la indexul măștii de biți care cuprinde toate nodurile:

```
dp[(1 << n) - 1]
```

Datorită numărului foarte mare de sortări topologice, rezultatul final va fi calculat modulo o constantă (`MOD`) folosind aritmetica modulară.

Implementare C++ 3.1: Funcție care calculează numărul de sortări topologice

```cpp
int countTopologicalSorts(int n, const vector<pair<int, int>>& edges) {
    vector<int> dependenta(n, 0), dp(1 << n, 0);
    for (auto [x, y] : edges) dependenta[y] |= (1 << x);
    dp[0] = 1;
    for (int mask = 0; mask < (1 << n); mask++) {
        for (int i = 0; i < n; i++) {
            if ((mask & (1 << i)) == 0 && (mask & dependenta[i]) ==
                dependenta[i]) {
                dp[mask | (1 << i)] = (dp[mask | (1 << i)] + dp[mask]) %
                    MOD;
            }
        }
    }
    return dp[(1 << n) - 1];
}
```

Analiză complexitate Complexitatea de timp este dată de secțiunea de cod în care se construiește rezultatul pentru fiecare mască de biți. Astfel, pentru fiecare mască de biți `mask` ($0 \leq$ `mask` $< 2^n$), încercăm adăugarea unui nou nod i ($0 \leq i < n$). Acest lucru duce la un ordin de complexitate $O(2^n \cdot n)$.

Complexitatea de spațiu este dată de tabloul unidimensional `dp` de dimensiune 2^n, ceea ce duce la un ordin $O(2^n)$.

Comparativ cu o abordare brute force care ar încerca fiecare permutare a lui $[1, 2, \ldots, n]$ (adică $n!$ permutări), rezultând într-o complexitate de timp de $O(n!)$, varianta cu programarea dinamică bazată pe măști de biți este mult mai eficientă pentru $n > 5$ (a se vedea Figura A.1 din Anexa A).

3.1.3 Numărul total de grafuri direcționate aciclice generate dintr-un graf neorientat

Dat fiind un graf neorientat $G = (V, E)$, unde V reprezintă setul de noduri și E setul de muchii, se dorește calcularea numărului total de grafuri direcționate aciclice generate în urma orientării muchiilor acestuia.

Pentru a construi un graf direcționat aciclic dintr-un graf neorientat G, muchiile E vor trebui să fie direcționate astfel încât să nu se creeze niciun ciclu. Pentru ca un graf să fie un DAG, cel puțin un nod trebuie să aibă gradul exterior 0. Aceste noduri pot fi gândite ca fiind noduri de tip „sink”, asemenea celor dintr-o rețea de transport în flux.

Un graf direcționat aciclic poate fi privit ca o rețea de astfel de noduri organizate pe niveluri, unde pe ultimul nivel se află nodurile cu gradul exterior 0, iar pe fiecare strat nodurile nu au muchii între ele.

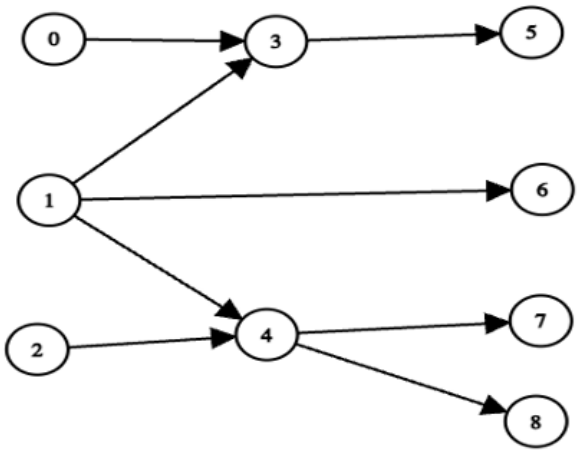

Figura 3.1: Graf orientat aciclic construit pe niveluri

Astfel, un graf direcționat aciclic se poate construi dintr-un graf neorientat în mod recursiv, pornind de la ultimul nivel și adăugând niveluri succesive cu muchii către nodurile de pe niveluri superioare.

Pe baza acestor definiții, se pot introduce următoarele tablouri unidimensionale:

- `dependență[i]`: mască de biți cu toate nodurile care ies din nodul i. **Exemplu:** masca de biți `00101` conține nodurile 0 și 2, pentru muchiile de tipul $(i, 0)$ și $(i, 2)$.
- `semn[mask]`: 1 dacă masca de biți `mask` conține un număr impar de noduri, și -1 dacă masca de biți conține un număr par de noduri.
- `independent[mask]`: flag pentru masca de biți `mask`, care este egal cu `true` dacă toate nodurile din `mask` sunt independente unul față de celălalt (nu există muchii între ele).
- `dp[mask]`: numărul de grafuri orientate aciclice care conțin nodurile din masca de biți `mask`.

Dependențele unui nod se pot calcula odată cu citirea muchiilor. Astfel, pentru o muchie (x, y) avem:

$$\texttt{dependență[x]} \mid= (1 \ll y) \quad \text{și} \quad \texttt{dependență[y]} \mid= (1 \ll x)$$

Semnul unei măști de biți poate fi calculat ușor folosind LSB, astfel:

$$\texttt{semn[mask]} = \texttt{semn[mask ^ LSB]} \times (-1)$$

O mască de biți este independentă dacă:

$$(\texttt{mask} \,\&\, \texttt{dependență[i]}) == 0 \quad \forall i \in \texttt{mask}$$

O modalitate intuitivă și eficientă de a descrie recursiv (pe niveluri, în spiritul reprezentării din Figura 3.1) procesul de calcul al numărului total de orientări aciclice ale unui graf neorientat este prin definirea unei relații de recurență pentru tabloul unidimensional `dp[mask]`.

$$\texttt{dp[mask]} = \sum_{S \subseteq \texttt{mask},\ S \neq \emptyset,\ \texttt{independent}[S]} \texttt{dp[mask} \setminus S] \times \texttt{semn[mask]}$$

Interpretare în straturi (niveluri) Alegem un subset S de noduri din `mask` care nu au muchii între ele (`independent[S] = true`) — acestea vor fi „noduri sink" (grad de ieșire 0) în nivelul curent. Orientăm toate muchiile din `mask` $\setminus S$ spre aceste noduri S, aplicând principiul includerii și excluderii pentru a evita numărătoarea dublă. Scoatem nodurile S (nivelul curent) din graf și reluăm procesul pentru `mask` $\setminus S$.

Astfel, în fiecare pas, „stratul" (setul de sink-uri) poate fi ales oricum, atâta timp cât nodurile din el nu sunt conectate între ele. Prin parcurgerea tuturor alegerilor posibile de astfel de straturi, suma rezultată oferă numărul total de orientări aciclice.

Detalii implementare Datorită numărului foarte mare de grafuri aciclice orientate, rezultatul final va fi calculat modulo o constantă (`MOD`) folosind aritmetica modulară.

Implementare C++ 3.2: Funcție pentru numărarea grafurilor orientate aciclice (DAG-uri) generate dintr-un graf neorientat

```
int countDAGs(int n, const vector<pair<int, int>>& edges) {
    vector<int> dependenta(n, 0), dp(1 << n, 0), independent(1 << n, 0),
        semn(1 << n, 0);
    for (auto [x, y] : edges) {
        dependenta[x] |= (1 << y);
        dependenta[y] |= (1 << x);
    }
    independent[0] = 1;
    for (int i = 0; i < (1 << n); i++) {
        int ultimul_bit = i & ~(i - 1);
        int ultima_masca = i ^ ultimul_bit;
        if (independent[ultima_masca] && ((ultima_masca & dependenta[
            __builtin_ctz(ultimul_bit)]) == 0))
            independent[i] = 1;
    }
    semn[0] = -1;
    for (int i = 1; i < (1 << n); i++) {
        int ultimul_bit = i & ~(i - 1);
        semn[i] = semn[i ^ ultimul_bit] * -1;
    }
    dp[0] = 1;
```

```
    for (int i = 1; i < (1 << n); i++) {
        for (int j = i; j; j = (j - 1) & i)
            if (independent[j]) {
                dp[i] = (1LL * dp[i] + 1LL * dp[i ^ j] * semn[j] % MOD +
                    MOD) % MOD;
            }
    }
    return dp[(1 << n) - 1];
}
```

Analiză complexitate Complexitatea de timp este dată de secțiunea de cod în care se construiește rezultatul pentru fiecare mască de biți. Astfel, pentru fiecare mască de biți i ($0 \leq i < 2^n$), încercăm fiecare submască j ($0 \leq j < 2^n$). Acest lucru duce la o complexitate de timp $O(3^n)$, deoarece pentru fiecare mască de biți i parcurgem toate submăștile sale j. Numărul de submăști ale unei măști cu k biți setați este 2^k. Pentru fiecare k, există $\binom{n}{k}$ astfel de măști. Prin urmare, complexitatea totală este dată de suma:

$$\sum_{k=0}^{n} \binom{n}{k} 2^k$$

care se simplifică la:

$$(1+2)^n = 3^n$$

Complexitatea de spațiu este dată de tabloul unidimensional `dp` de dimensiune 2^n, ceea ce duce la un ordin $O(2^n)$.

Comparativ cu o abordare brute force care încearcă să atribuie fiecărei muchii o direcție (ceea ce duce la o complexitate de timp de $O(2^m)$), varianta cu programarea dinamică bazată pe măști de biți este mult mai eficientă pentru grafuri cu multe muchii. Numărul maxim de muchii al unui graf neorientat cu n noduri este:

$$m = \frac{n \cdot (n-1)}{2}$$

(a se vedea Figura A.2 din Anexa A).

3.1.4 Amusement Park – CEOI 2019[1]

Ai fost angajat să supraveghezi proiectul unui parc de distracții. Proprietarul parcului a propus un proiect care include:

- n atracții.
- m tobogane direcționate care transportă vizitatorii între atracții.

[1] CEOI [3]

Toboganele sunt planificate astfel încât fiecare să aibă o direcție specifică (de la o atracție la alta). Totuși, unele configurații propuse încalcă legile fizicii, deoarece pot apărea cicluri imposibile (de exemplu, un tobogan de la atracția A la B, unul de la B la C, și altul de la C la A).

Trebuie să determini toate configurațiile legale ale toboganelor (prin păstrarea sau inversarea direcției lor), astfel încât:

- Toboganele respectă o „direcție în jos” — fiecare atracție poate fi plasată pe o altitudine unică, astfel încât toate toboganele merg „în jos”.

Costul unei configurații este numărul de tobogane cărora li s-a inversat direcția față de proiectul original.

Problemă Calculează suma costurilor pentru toate configurațiile legale, modulo 998 244 353.

Notă: Am folosit rezolvarea propusă de Ren [4] ca sursă de inspirație și punct de plecare pentru algoritmul prezentat.

Rezolvare Relațiile dintre tobogane pot fi modelate sub formă unui graf neorientat G. Pentru a îndeplini condiția direcției în jos, graful inițial trebuie transformat într-un graf orientat aciclic (DAG). Astfel, rezultatul problemei este suma costului creării fiecărui graf orientat aciclic.

Observație Se pot forma un număr par de grafuri orientate aciclice. Prin întoarcerea muchiilor unui graf orientat aciclic se poate forma alt DAG. Costul formării unui graf orientat aciclic și al inversului acestuia este egal cu m, deoarece muchiile acestora sunt complementare. Astfel, putem împerechea câte două DAG-uri, având împreună costul m.

Rezolvarea este identică cu cea din secțiunea 3.1.3, la fel și complexitatea de timp și de spațiu.

Formulă finală

$$\frac{m \cdot \texttt{nrDAG}(G)}{2}$$

Implementare C++ 3.3: Calcularea rezultatului final

```
int rez = 1LL * countDAGs(n, edges, MOD) * m % MOD * inv(2, MOD) % MOD;
```

Notă: Funcția `inv(x, MOD)` pentru calcularea inversului numărului x modulo `MOD` se află în *Anexa B*.

Capitolul 4

Sum over Subsets DP

4.1 Explicarea tehnicii SOS DP

Sum over Subsets (SOS) DP este o tehnică folosită pentru a calcula eficient suma valorilor pentru toate submăștile unei măști de biți date. O *submască* a unei măști este orice mască obținută prin dezactivarea unora dintre biții activi ai acesteia.

Problema generală Considerăm un vector A cu 2^n elemente. Pentru fiecare mască posibilă (reprezentată ca un număr binar pe n biți), trebuie să determinăm valoarea funcției F corespunzătoare acelei măști.

Funcția $F(\texttt{mask})$ reprezintă suma valorilor din vectorul A pentru toate submăștile `submask` ale lui `mask`.

$$F(\texttt{mask}) = \sum_{\texttt{submask} \subseteq \texttt{mask}} A[\texttt{submask}],$$

unde `mask` ia valori de la 0 la $2^n - 1$.

Exemplu Pentru $\texttt{mask} = 22$, avem:

$$F(22) = A(22) + A(20) + A(18) + A(16) + A(6) + A(4) + A(2)$$

Variantă brute force Varianta brută presupune iterarea prin toate perechile de măști de biți, însumând valorile $A[i]$ doar atunci când unul dintre ele este un subset al celuilalt.

Implementare C++ 4.1: Implementarea brute force

```
vector<int> F(1 << n);
for (int mask = 0; mask < (1 << n); mask++)
    for (int submask = 0; submask < (1 << n); submask++)
        if ((submask & mask) == submask)
            F[mask] += A[submask];
```

Soluția de mai sus duce la o complexitate de timp $O(4^n)$, aceasta fiind ineficientă pentru valori mari ale lui n.

Optimizare Folosind tehnica de iterare prin submăști, prezentată în secțiunea 3.1.3, putem reduce complexitatea de timp la un ordin $O(3^n)$.

Implementare C++ 4.2: Implementarea cu iterare eficientă prin submăști

```
vector<int> F(1 << n);
for (int mask = 0; mask < (1 << n); mask++) {
    F[mask] = A[0];
    for (int submask = mask; submask > 0; submask = (submask - 1) & mask) {
        F[mask] += A[submask];
    }
}
```

Chiar dacă metoda aceasta este bună, tehnica poate fi îmbunătățită. De exemplu, dacă masca de biți `mask` are x biți nesetați, atunci $A[\texttt{mask}]$ este însumată de 2^x ori. Dacă precomputăm aceste sume, putem elimina adunările repetate.

Definim:

$$V(\texttt{mask}, \texttt{ind}) = \{\texttt{submask} \subseteq \texttt{mask} \mid \texttt{submask} \,\&\, ((1 << \texttt{ind}) - 1) = \texttt{mask} \,\&\, ((1 << \texttt{ind}) - 1)\}$$

În termeni mai simpli, $V(\texttt{mask}, \texttt{ind})$ conține toate submăștile `submask` ale lui `mask` ale căror biți mai mari decât `ind` sunt identici cu cei ai lui `mask`.

Se deduce următoarea recurență:

$$V(\texttt{mask}, \texttt{ind}) = \begin{cases} V(\texttt{mask}, \texttt{ind} - 1), & \text{dacă bitul } \texttt{ind} \text{ este } 0 \\ V(\texttt{mask}, \texttt{ind} - 1) \cup V(\texttt{mask} \oplus (1 \ll \texttt{ind}), \texttt{ind} - 1), & \text{dacă bitul } \texttt{ind} \text{ este } 1 \end{cases}$$

Astfel, putem defini `dp[mask][ind]` ca o sumă parțială a tuturor subseturilor, astfel încât, la final, `F[mask]` este echivalent cu `dp[mask][n]`:

$$\texttt{dp[mask][ind]} = \sum_{\texttt{submask} \in V(\texttt{mask}, \texttt{ind})} A[\texttt{submask}], \quad \texttt{F[mask]} = \texttt{dp[mask][n]}$$

Complexitatea de timp rezultată are un ordin $O(2^n \cdot n)$.

Implementare C++ 4.3: Implementarea recurenței SOS DP optimizată

```
vector<int> F(1 << n);
vector<vector<int>> dp(1 << n, vector<int>(n + 1));
for (int mask = 0; mask < (1 << n); mask++) {
    dp[mask][0] = A[mask];
    for (int ind = 0; ind < n; ind++) {
        dp[mask][ind + 1] = dp[mask][ind];
```

```
        if (mask & (1 << ind))
            dp[mask][ind + 1] += dp[mask ^ (1 << ind)][ind];
    }
    F[mask] = dp[mask][n];
}
```

Implementare eficientă din punct de vedere al memoriei (reducere la o singură dimensiune) Această implementare optimizează spațiul la un singur tablou unidimensional `F`:

Implementare C++ 4.4: Implementarea finală a SOS DP cu un singur tablou

```
vector<int> F(1 << n);
F = A;
for (int ind = 0; ind < n; ind++) {
    for (int mask = 0; mask < (1 << n); mask++) {
        if (mask & (1 << ind)) {
            F[mask] += F[mask ^ (1 << ind)];
        }
    }
}
```

4.2 Aplicații în probleme de optimizare

4.2.1 Vowels[1]

Cerință Iahubina și-a creat un mic dicționar format din n ($1 \leq n \leq 10^4$) „3 cuvinte", fiecare fiind o succesiune de exact trei litere mici din primele 24 de litere ale alfabetului englez (de la `a` până la `x`). În noua ei limbă, a desemnat unele dintre aceste 24 de „caractere" drept vocale, iar restul drept consoane. Întregul limbajul este bazat după regula simplă și anume că un cuvânt este „corect" dacă și numai dacă conține cel puțin o vocală.

Cum a uitat exact care litere le-a ales ca vocale, ea consideră toate cele 2^{24} posibile subseturi de litere drept ipoteze de seturi de vocale. Pentru fiecare astfel de subset, numără câte cuvinte din dicționar sunt corecte și calculează pătratul acestui număr. În final, pentru output, ia toate aceste valori și le combină prin operația `XOR` pe biți (valoarea finală: `X`). Să se determine numărul `X`.

Rezolvare Pentru un cuvânt w (format din 3 litere) putem crea o mască cu 24 de biți, marcând fiecare literă prezentă în acesta (bitul 0 pentru apariția lui `a`, bitul 1 pentru apariția lui `b`, ..., bitul 23 pentru apariția lui `x`).

[1] Codeforces [5]

Exemplu Pentru cuvântul `abx`:

$$\texttt{abx} : 10000000000000000000011$$

Problema se reduce la găsirea numărului de măști `mask2` pentru fiecare `mask` astfel încât:

$$\texttt{mask} \ \& \ \texttt{mask2} \neq 0,$$

unde `mask` și `mask2` sunt măști de vocale (construite în același mod ca și anterior).

Putem reprezenta condiția pentru `mask2` sub formă de mulțime:

$$\{\texttt{mask2} \ : \ \texttt{mask} \ \& \ \texttt{mask2} = 0\} = \{\texttt{mask2} \ : \ (\sim \texttt{mask}) \mid \texttt{mask2} = \sim \texttt{mask}\}$$

Astfel, aflăm cardinalitatea mulțimii:

$$|\{\texttt{mask2} \ : \ \texttt{mask} \ \& \ \texttt{mask2} \neq 0\}| = n - |\{\texttt{mask2} \ : \ (\sim \texttt{mask}) \mid \texttt{mask2} = \sim \texttt{mask}\}| \quad (1)$$

Notăm:

$$A[x] = \text{numărul de măști } x \text{ din input.}$$

$$F(\texttt{mask}) = \sum_{\texttt{ind} \subseteq \texttt{mask}} A[\texttt{ind}]$$

Reprezentăm $F(\texttt{mask})$ prin mulțimi:

$$F(\texttt{mask}) = |\{\texttt{mask2} \ : \ (\texttt{mask} \mid \texttt{mask2}) = \texttt{mask}\}| \quad (2)$$

Astfel, din (1) și (2), obținem rezultatul pentru `mask` într-un mod eficient calculabil:

$$|\{\texttt{mask2} \ : \ \texttt{mask} \ \& \ \texttt{mask2} \neq 0\}| = n - F(\sim \texttt{mask})$$

Implementare C++ 4.5: Implementare finală pentru calculul valorii X

```
for (int i = 0; i < n; i++) {
    string s; cin >> s;
    int now = accumulate(s.begin(), s.end(), 0, [](int acc, char c) {
        return acc | (1 << (c - 'a'));
    });
    A[now]++;
}
F = A;  SOS_DP_add(F);
int X = 0;
for (int mask = 0; mask < (1 << 24); mask++) {
    int au_vocale = n - F[(1 << 24) - 1 - mask];
    X ^= (au_vocale * au_vocale);
}
```

Notă: Funcția `SOS_DP_add(F)` poate fi găsită în *Anexa B*.

Analiză complexitate Complexitatea de timp rezultată are un ordin $O(\ell \cdot 2^\ell)$, iar complexitatea de spațiu are un ordin $O(2^\ell)$, unde ℓ este numărul de litere din limbaj (24).

Comparativ cu o abordare brute force care verifică pentru fiecare mască `mask` numărul de cuvinte corecte (ceea ce duce la o complexitate de timp $O(2^\ell \cdot n)$), rezolvarea cu programare dinamică cu măști de biți este mai eficientă pentru $n > \ell$.

4.2.2 XorTransform – Lot România 2018 Baraj 2[2]

Cerință Se dă o matrice A de dimensiune $N \times M$ cu elemente naturale. Asupra valorilor acestei matrici se poate aplica transformarea definită mai jos, rezultând matricea A'.

$$A'(i,j) = A(i,j) \oplus A(i+1,j) \oplus A(i,j+1) \oplus A(i+1,j+1)$$

Dacă indicii depășesc limitele matricei, se consideră valoarea de acolo ca fiind 0.

Se dau Q interogări. La fiecare interogare se primește o valoare X. Pe baza acesteia se determină numărul de transformări notat cu K, calculat astfel: $K = X \oplus Y$, unde Y este răspunsul obținut la interogarea anterioară (cu $Y = 0$ la prima interogare).

Pentru fiecare interogare, se aplică K transformări succesive asupra matricei inițiale A, iar apoi se cere valoarea elementului aflat în colțul stânga-sus al matricei rezultate, adică $A(0,0)$.

Constrângeri

- $1 \leq N \cdot M \leq 2.5 \times 10^6$
- $1 \leq$ elementele matricei $\leq 2^{30}$
- $1 \leq K \leq 10^9$
- $1 \leq Q \leq 10^6$

Rezolvare Exemplu de modificare pentru $K = 1, 2, 3, \ldots, 5$.

Matricea Inițială A

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

K = 1

4	0	4	5
4	0	12	9
12	0	12	13
15	1	3	5

K = 2

0	8	4	12
8	0	4	4
2	14	7	8
14	2	6	5

K = 3

0	8	8	8
4	13	15	12
0	13	12	13
12	4	3	5

K = 4

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

K = 5

4	0	4	5
4	0	12	9
12	0	12	13
15	1	3	5

Figura 4.1: Matricea A dupa K transformări

[2]Infoarena [6]

Observație Elementele $A(i,j)$ (cu $(i,j) \neq (0,0)$ și $i,j \geq 0$) influențează elementul $A(0,0)$ într-un mod stabilit.

Considerăm matricea extinsă cu zerouri la infinit în jos și spre dreapta. Pentru un K fixat, ne propunem să determinăm câte poziții $(i,j) \neq (0,0)$ din matricea inițială influențează valoarea $A(0,0)$ după aplicarea a K transformări.

Se va demonstra că un element $A(i,j)$ va contribui la valoarea finală $A(0,0)$ dacă și numai dacă atât i, cât și j sunt submăști ale lui K:

$$(i,j) \text{ influențează } (0,0) \text{ după } K \text{ pași} \iff i \subseteq K \text{ și } j \subseteq K \iff (i \mid j) \subseteq K.$$

La fiecare pas, fiecare element contribuie la valori din poziții apropiate (într-un pătrat de 2×2), iar influențele se propagă recursiv. Se poate arăta că, după K pași, valoarea $A(i,j)$ contribuie la $A(0,0)$ de exact:

$$\texttt{Contr}(i,j) = \binom{K}{i} \cdot \binom{K}{j} \text{ ori,}$$

unde $\binom{K}{n}$ este coeficientul de combinări.

Pentru că operațiile sunt făcute cu `XOR`, contează paritatea produsului:

$$\texttt{Contr}(i,j) \equiv 1 \pmod 2 \iff \binom{K}{i} \equiv 1 \pmod 2 \text{ și } \binom{K}{j} \equiv 1 \pmod 2$$

Teorema lui Lucas Pentru trei numere pozitive m, n și p, următoarea relație de congruență este adevărată:

$$\binom{m}{n} \equiv \prod_{i=0}^{k} \binom{m_i}{n_i} \pmod p$$

unde:

$$m = m_k \cdot p^k + m_{k-1} \cdot p^{k-1} + \cdots + m_1 \cdot p + m_0$$

$$n = n_k \cdot p^k + n_{k-1} \cdot p^{k-1} + \cdots + n_1 \cdot p + n_0$$

Aplicăm convenția: $\binom{m}{n} = 0$, dacă $m < n$.

Pentru a calcula $\binom{x}{k} \equiv 1 \pmod 2$ ne vom folosi de *Teorema lui Lucas*:

$$\binom{x}{k} \pmod 2 = \prod_{b=0}^{B} \binom{x_b}{k_b} \pmod 2$$

Produsul va fi egal cu 0 doar atunci când există b astfel încât $k_b > x_b$. Deci:

$$\binom{x}{k} \equiv 1 \pmod 2 \quad \iff \quad k_b \leq x_b \quad \forall b \in [0, B)$$

adică x este submască a lui k.

Am demonstrat că $A(i, j)$ contribuie la $A(0, 0)$ dacă $(i \mid j) \subseteq K$.

Creăm un nou tablou unidimensional B de dimensiune $\max(i \mid j)$, unde $i \in [0, N)$ și $j \in [0, M)$.

$$B[x] = A(i, j) \quad \text{pentru } (i \mid j) = x.$$

Pentru o interogare vom răspunde cu $F[K]$ (calculat cu SOS DP), unde F este construit pe baza vectorului B (pentru a satisface condiția $(i \mid j) \subseteq K$).

Observăm că K poate fi foarte mare ($1 \leq K \leq 10^9$), dar $\max(i \mid j)$, cu $i \in [0, N)$ și $j \in [0, M)$, este mult mai mic. Astfel, putem trunchia K la ultimii $\lceil \log_2(\max(i \mid j) + 1) \rceil$ biți și să incrementăm dimensiunea tabloului unidimensional B la K_{MAX}.

$$K_{\text{MAX}} = 2^{\lceil \log_2(\max(i|j)+1) \rceil} \quad \text{unde } i \in [0, N),\ j \in [0, M).$$

Implementare C++ 4.6: Implementarea finală cu SOS DP pentru transformarea XOR

```cpp
void xorTransform(int N, int M,
                  const vector<vector<int>>& mat,
                  int Q, const vector<int>& queries,
                  int K_MAX, const string& outputFile) {
    vector<int> B(K_MAX, 0);
    for (int i = 0; i < N; ++i)
        for (int j = 0; j < M; ++j) {
            int k = i | j;
            B[k] ^= mat[i][j];
        }
    vector<int> F(K_MAX, 0);
    F = B;
    SOS_DP_xor(F);
    ofstream g(outputFile);
    int prev = 0;
    for (int idx = 0; idx < Q; ++idx) {
        int K = queries[idx] ^ prev;
        K &= (K_MAX - 1);
        int ans = F[K]; g << ans << '\n';
        prev = ans;
    }
    g.close();
}
```

Notă: Funcția `SOS_DP_xor(F)` poate fi găsită în *Anexa B*.

Analiză complexitate Complexitatea de timp rezultată are un ordin $O(Q + N \cdot M + K_{\text{MAX}} \cdot \log_2 K_{\text{MAX}})$, iar complexitatea de spațiu are un ordin $O(Q + N \cdot M + K_{\text{MAX}})$. Variabilele păstrează notațiile din rezolvare, iar $K_{\text{MAX}} \approx 2 \cdot \max(N, M)$.

Comparativ cu o abordare brute force care ar calcula matricea rezultată după aplicarea transformărilor, rezultând într-o complexitate de ordin $O(Q \cdot N \cdot M \cdot K)$, algoritmul prezentat este mult mai eficient.

4.2.3 And Subset Count – CSES: Extinderea operațiilor pe SOS DP[3]

Cerință Se dă un șir de N numere naturale. Să se determine numărul de submulțimi care, în urma aplicării operației `AND`, au ca rezultat k, unde $k \in [0, N]$.

Rezultatul se va afișa modulo $10^9 + 7$.

Constrângeri

- $1 \leq N \leq 2 \cdot 10^5$
- $0 \leq$ elementele șirului $\leq$ N

Rezolvare Aflăm pentru început câte numere Y din datele de intrare au valoarea X aplicând operația $X \,\&\, Y$ (supramăștile lui X):

$$F'(X) = |\{Y : Y \,\&\, X = X\}|$$

În rezolvările precedente, $F(X)$ contoriza:

$$F(X) = |\{Y : Y \,\&\, X = Y\}|$$

(adică număra submăștile lui X).

Rescriem $F'(X)$ printr-o sumă peste Y:

$$F'(X) = \sum_{\texttt{supmask} \supseteq X} A[\texttt{supmask}],$$

unde $A[X]$ este numărul de apariții ale lui X în datele de intrare.

$F'(X)$ se poate calcula eficient într-un mod asemănător cu $F(X)$:

[3]CSES [7]

Implementare C++ 4.7: Calculul eficient al lui F'(X)

```
vector<int> F_prime(LEN);
F_prime = A;
for (int ind = 0; ind < log2(LEN); ind++)
    for (int X = LEN - 1; X >= 0; X--) {
        if (!(X & (1 << ind))) {
            F_prime[X] += F_prime[X ^ (1 << ind)];
        }
    }
```

Complexitatea de timp rezultată are un ordin:

$$O(\texttt{LEN} \cdot \log_2 \texttt{LEN}),$$

unde `LEN` este cea mai mică putere a lui 2 mai mare sau egală decât toate elementele șirului de numere.

Rescriem $F'(X)$ astfel încât să cuprindă numărul de submulțimi nenule ale supramăștilor lui X:

$$F'(X) = 2^{F'(X)} - 1$$

Totuși, $F'(X)$ nu este încă rezultatul dorit. Observăm că $F'(X)$ de sus conține și submulțimile ale supramăștilor lui X (adică $\{Y : Y \ \& \ X = X\}$).

Definim operația de dereferențierii a supramăștilor:

Implementare C++ 4.8: Calculul „dereferențierii" supramăștilor

```
vector<int> F_prime(LEN);
F_prime = A;
for (int ind = 0; ind < log2(LEN); ind++) {
    for (int X = LEN - 1; X >= 0; X--) {
        if (!(X & (1 << ind))) {
            F_prime[X] -= F_prime[X ^ (1 << ind)];
        }
    }
}
```

Astfel, schimbăm operatorul din `+` în `-`. Această operație de „dereferențiere" se poate aplica și pentru submăști.

Obținem $F'(X)$ răspunsul dorit.

Implementare C++ 4.9: Cod complet

```
vector<int> AndSubsetCount(const vector<int>& A, int LEN) {
    vector<int> F_prime(LEN, 0);
    for (int x : A) {
        F_prime[x]++;
    }
```

```
    SOS_DP_prime_add(F_prime);
    for (int X = 0; X < LEN; ++X) {
        F_prime[X] = (pow_mod(2, F_prime[X], MOD) - 1 + MOD) % MOD;
    }
    SOS_DP_prime_sub(F_prime);
    return F_prime;
}
```

Notă: Funcția `pow_mod(base, exp, MOD)` pentru exponențiere rapidă, precum și funcțiile `SOS_DP_prime_add(F)` și `SOS_DP_prime_sub(F)` pot fi găsite în *Anexa B*.

Analiză complexitate Complexitatea de timp rezultată are un ordin $O(\texttt{LEN} \cdot \log_2 \texttt{LEN})$, iar complexitatea de spațiu are un ordin $O(\texttt{LEN})$, unde `LEN` este cea mai mică putere a lui 2 mai mare sau egală decât toate elementele șirului de numere.

Ambele complexități sunt datorate construirii și stocării tabloului unidimensional `F_prime` (SOS DP).

Comparativ cu o abordare brute force care ar contoriza fiecare submulțime, rezultând într-o complexitate $O(\texttt{M} \cdot 2^{\texttt{M}})$, rezolvarea cu SOS DP este semnificativ mai eficientă pentru un N mare.

Nota: În realizarea acestei secțiuni, am folosit informații și explicații din următoarele surse:

- Parwez, Huang și Gokhale [8] pentru detalii despre tehnica SOS DP.
- Saxena [9] pentru clarificarea ideilor și implementarea tehnicii.
- Dăscălescu [10] pentru soluții și explicații referitoare la transformarea XOR (Kilonova XOR Transform).
- Corn, Goh și Kulkarni [11] pentru aplicații ale teoremei lui Lucas la combinatorica modulară şi calcule de binomiale modulo prim.

Capitolul 5

Numere prime și aplicații în dinamică cu măști de biți

5.1 Reprezentarea numerelor prime sub formă de măști de biți și operațiile specifice

Idee generală În anumite probleme de teoria numerelor, putem reprezenta fiecare număr folosind o *mască de biți* care indică factorii săi primi. De exemplu, mulțimea $\{6, 50, 30, 77\}$ poate fi reprezentată astfel: $\{00011, 00101, 00111, 11000\}$ (în binar), unde fiecare bit marchează divizibilitatea cu $\{2, 3, 5, 7, 11\}$.

Această reprezentare permite următoarele echivalențe utile:

- **AND bitwise** corespunde lui CMMDC (Cel Mai Mare Divizor Comun) pe factori primi.
- **OR bitwise** corespunde lui CMMMC (Cel Mai Mic Multiplu Comun) pe factori primi.
- Parcurgerea biților activați echivalează cu parcurgerea factorilor primi.
- Parcurgerea submăștilor echivalează cu parcurgerea factorilor primi compuși.

O mulțime de numere are $CMMDC = 1$ dacă măștile lor de biți, aplicând operatorul `AND`, dau 0.

Notă: Pentru consolidarea cunoștințelor despre *Programaare dinamică cu măști de biți pe numere prime* și pentru clarificarea unor concepte de bază, am folosit explicațiile din Cao, Huang și Bai [12].

5.2 Exemple de probleme

5.2.1 F - Coprime Present (Atcoder ABC Round 195)[1]

Cerință Pentru fiecare număr întreg de la A la B, ai o carte pe care este scris acel număr. Vei oferi unele dintre ele (posibil niciuna) animalului tău de companie, Snuke. Snuke va fi fericit dacă, pentru fiecare pereche de cărți diferite, numerele scrise pe ele sunt prime între ele (adică orice două numere distincte sunt coprime), altfel el va fi trist.

Se cere: câte seturi de cărți îl vor face pe Snuke fericit?

Constrângeri

$$1 \leq A \leq B \leq 10^{18}, \quad B - A \leq 72$$

Toate valorile de intrare sunt numere întregi.

Rezolvare Rezolvarea se bazează pe observația următoare: Fie x un număr natural din intervalul $[A, B]$. Dacă x are ca și cel mai mic divizor prim un număr > 72, atunci acesta nu poate fi coprim cu orice alt număr din intervalul $[A, B]$. Astfel, pentru fiecare set format, factorii primi comuni vor fi din intervalul $[1, 72]$.

În intervalul $[1, 72]$ sunt 20 de numere prime:

$$\{2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71\}$$

Pentru fiecare număr din intervalul $[A, B]$, menținem o mască de biți cu toți factorii primi până la 72. De exemplu, numărului 30 îi corespunde masca de biți:

$$000 \ldots 00111$$

Două numere sunt coprime între ele dacă aplicând operatorul `&` între măștile de biți asociate acestora rezultatul este masca de biți:

$$000 \ldots 00000$$

Astfel, problema se poate rezolva folosind programarea dinamică, starea fiind masca de biți asociată factorilor primi prezenți în setul curent.

Avem recurența următoare:

$$\texttt{dp[mask} \mid \texttt{mască(număr)]} \mathrel{+}= \texttt{dp[mask]}, \quad \text{dacă } \texttt{mască(număr)} \ \& \ \texttt{mask} = 0,$$

unde:

[1] AtCoder [13]

- `număr` $\in [A, B]$
- `mască(număr)` = masca de biți cu toți factorii primi până în 72.

Implementare C++ 5.1: Cod complet

```cpp
long long calculateDp(long long A, long long B, const vector<int>&
    primesList) {
    const int m = primesList.size();
    vector<long long> dp(1 << m, 0LL);
    dp[0] = 1;

    for (long long i = A; i <= B; i++) {
        int nowMask = 0;
        for (int j = 0; j < m; j++) {
            if (i % primesList[j] == 0)
                nowMask |= (1 << j);
        }
        for (int mask = (1 << m) - 1; mask >= 0; mask--) {
            if ((mask & nowMask) == 0)
                dp[mask | nowMask] += dp[mask];
        }
    }

    return accumulate(dp.begin(), dp.end(), 0LL);
}
```

Analiză complexitate Complexitatea de timp este dată de parcurgerea numerelor din intervalul $[A, B]$ și iterarea prin măștile de factori primi. Astfel, se ajunge la un ordin de complexitate $O(2^m \cdot (B - A + 1))$, unde $m = 20$ (numărul de factori primi ≤ 72).

Complexitatea de spațiu este dată de tabloul unidimensional `dp` de dimensiune 2^m, ceea ce duce la un ordin $O(2^m)$.

Comparativ cu o soluție care ar încerca fiecare set posibil format de numere din intervalul $[A, B]$, conducând la o complexitate de timp $O(2^{(B-A+1)})$, varianta cu programare dinamică este mult mai eficientă în general pentru $B - A + 1 > 24$. (*Figura A.3 din Anexa A*).

5.2.2 NGCD – NO GCD[2]

Cerință Se dă un șir de N numere naturale. Fiecare număr nu conține divizori pătrați perfecți (în afară de divizorul 1), iar factorii primi din descompunere sunt mai mici decât 50. Să se determine numărul de perechi care au cel mai mic divizor comun egal cu 1 sau un număr prim.

[2]SPOJ [14]

Notă Două perechi (i, j) și (j, i) sunt diferite dacă $i \neq j$.

Constrângeri

$$1 \leq N \leq 10^5$$

Rezolvare Fiecare număr este format din cel mult 15 factori primi:

$$\{2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47\}.$$

Astfel, putem construi o mască de biți `mask[X]` pentru fiecare număr X, reprezentând divizibilitatea cu fiecare factor prim:

`mask[X]` are pe poziția b valoarea 1 dacă X se divide cu prim(b).

Pentru ca două numere X și Y să aibă cel mai mic divizor comun egal cu 1, măștile asociate acestora trebuie să nu conțină biți de 1 pe aceeași poziție:

$$\texttt{mask[X]} \ \& \ \texttt{mask[Y]} = 0. \tag{1}$$

Pentru ca două numere X și Y să aibă cel mai mic divizor comun un număr prim, măștile asociate acestora trebuie să conțină biți de 1 pe aceeași poziție doar într-o singură poziție (adică `AND` să fie o putere a lui 2):

$$\texttt{mask[X]} \ \& \ \texttt{mask[Y]} = 2^p. \tag{2}$$

Astfel, pentru fiecare X din input, se contorizează numerele Y care satisfac proprietățile de mai sus (condițiile (1) și (2)).

Numărul de valori pentru condiția (1) este:

$$\texttt{CNT1(mask[X])} = |\{\texttt{mask[Y]} : (\texttt{mask[X]} \ \& \ \texttt{mask[Y]}) = 0\}|$$

$$= |\{\texttt{mask[Y]} : (\sim \texttt{mask[X]} \mid \texttt{mask[Y]}) = \sim \texttt{mask[X]}\}|$$

$$= F(\sim \texttt{mask[X]})$$

Unde:

- $F(\texttt{mask})$ este funcția de sumă pe submăști din Capitolul 4 (SOS DP).

Numărul de valori pentru condiția (2) este:

$$\texttt{CNT2(mask[X])} = \left|\{\texttt{mask[Y]} : (\texttt{mask[X]} \ \& \ \texttt{mask[Y]}) = 2^b\}\right|$$

$$= \left|\{\texttt{mask[Y]} : ((\sim \texttt{mask[X]}) \mid 2^b \mid \texttt{mask[Y]}) = (\sim \texttt{mask[X]}) \mid 2^b\}\right|$$
$$- \left|\{\texttt{mask[Y]} : ((\sim \texttt{mask[X]}) \mid \texttt{mask[Y]}) = (\sim \texttt{mask[X]})\}\right|$$
$$= F((\sim \texttt{mask[X]}) \mid 2^b) - F(\sim \texttt{mask[X]})$$

Unde:

- b este un bit activ din `mask[X]` (adică X se divide cu prim(b)).
- F(`mask`) este funcția de sumă pe submăști din Capitolul 4 (SOS DP).

Implementare C++ 5.2: Implementarea finală pentru NGCD - NO GCD

```cpp
long long NGCD(int n, const vector<long long>& a, const vector<int>& primes
    ) {
    const int P = 15;
    int N = 1 << P;
    vector<int> mask(n + 1, 0);
    vector<long long> A(N, 0);
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < P; ++j)
            if (a[i] % primes[j] == 0)
                mask[i] |= (1 << j);
        A[mask[i]]++;
    }
    vector<long long> F = A;
    SOS_DP_add(F);
    long long ans = 0, fullMask = N - 1;
    for (int i = 0; i < n; ++i) {
        int X = a[i], tilda_X = fullMask ^ mask[i];
        for (int b = 0; b < P; ++b)
            if (mask[i] & (1 << b))
                ans += F[tilda_X | (1 << b)] - F[tilda_X]; // CNT2(mask)
        ans += F[tilda_X]; // CNT1(mask)
    }
    return ans;
}
```

Notă: Funcția `SOS_DP_add(F)` poate fi găsită în *Anexa B* (se va modifica tipul de date al parametrului F în long long)

Analiză complexitate Complexitatea de timp este dată de două aspecte: calcularea lui F și contorizarea numerelor Y pentru un număr din input X, rezultând un ordin de complexitate $O(2^m \cdot m + N \cdot m)$, unde N este cel din input, iar $m = 15$ (numărul maxim de factori primi).

Complexitatea de spațiu este dată de tablourile unidimensionale `mask` și `F`, de dimensiuni N și 2^m, ceea ce duce la un ordin $O(N + 2^m)$.

Capitolul 6

Plug DP

6.1 Introducere în conceptul de Plug DP

Definiție *Plug DP* (sau *Broken Profile DP*) este o tehnică bazată pe programare dinamică cu măști de biți, care ne permite să rezolvăm probleme complicate cu stări și tranziții relativ simple. Tehnica este un subset al programării dinamice cu măști de biți.

Problemele care se încadrează în această categorie au, în general, următoarele proprietăți:

- Sunt despre completarea unei grile 2D.
- Una dintre dimensiuni este mult mai mică decât cealaltă.
- La completarea grilei, fiecare celulă depinde doar de celulele adiacente.
- Celulele nu au multe valori posibile (de obicei doar 2).

A treia proprietate este deosebit de importantă, deoarece înseamnă că putem procesa celulele rând cu rând (imaginează-ți un șarpe care se înfășoară pe grilă). Atunci trebuie să ne preocupe doar celulele cele mai din jos care au fost deja procesate de pe fiecare coloană (de aici și numele de *"broken profile"*).

Pentru a ilustra această tehnică vom vizita o problemă clasică: *În câte moduri putem acoperi complet o grilă de dimensiuni* $N \times M$ *cu dominouri de* 1×2 *(așezate orizontal sau vertical).*

Această problemă poate fi rezolvată cu o abordare standard de mascare de biți, rând cu rând, dar tranzițiile pentru dinamică nu sunt foarte clare. Investigăm o abordare care folosește stări puțin diferite.

Definim:

`dp[i][j][mask]` = numărul de moduri de a acoperi cu dominouri primele $i-1$ rânduri și primele j coloane din rândul i, având o mască `mask` asociată stării.

Primele două dimensiuni sunt ușor de urmărit, iar prin masca asociată stării se înțelege următorul lucru:

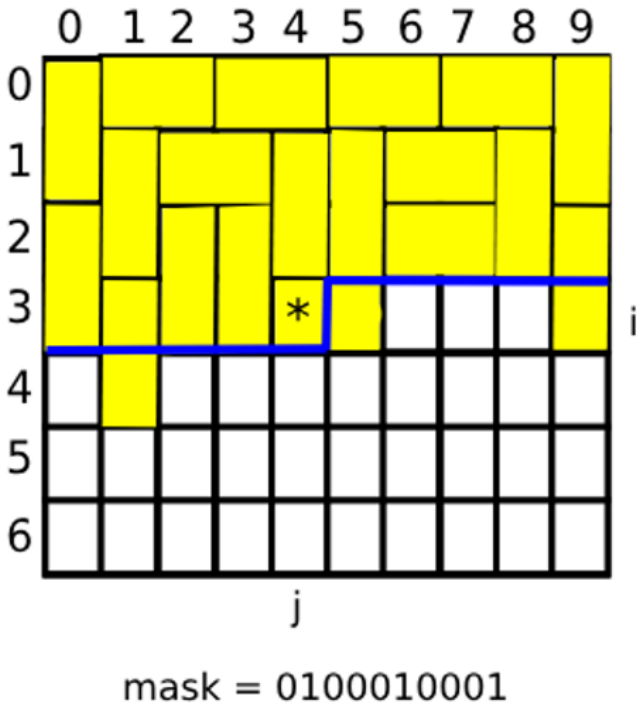

Figura 6.1: Plasare domino pentru fiecare celulă până în linia albastră

Figura 6.1.1 ilustrează o grilă cu 7 linii și 10 coloane. Parcurgând matricea în ordine crescătoare, pe linii și coloane, ajungem la poziția (i, j). Pentru această poziție, determinăm `mask` pe baza dominourilor deja plasate. Mai precis, `mask` este 1 pe pozițiile unde un domino iese în afara formei geometrice delimitate de linia albastră, și 0 pe pozițiile unde dominoul rămâne în interiorul acestei forme.

Tranziții În general, vrem să trecem de la celula $(i, j-1)$ la celula (i, j), adică să ne deplasăm pe rânduri. Observăm că doar 2 conectori se modifică atunci când ne deplasăm orizontal, acesta fiind principalul motiv pentru care metoda *Plug DP* este atât de puternică. Dacă numerotăm conectorii de la 0 la M, atunci doar conectorii $j-1$ și j își schimbă pozițiile.

O tranziție o notăm printr-un tuplu de forma:

$$(i, j-1, \texttt{mask}) \rightarrow (i, j, \texttt{mask\_new})$$

Notația este echivalentă, din punct de vedere al implementării, cu:

$$\texttt{dp}[i][j][\texttt{mask\_new}] \mathrel{+}= \texttt{dp}[i][j-1][\texttt{mask}]$$

Aflându-ne la poziția (i, j) cu masca de stare `mask`, putem face următoarele tranziții:

Cazul 1: Bitul j nu este activat în `mask`

- Plasăm un nou domino vertical:

$$(i, j-1, \texttt{mask}) \to (i, j, \texttt{mask} \oplus 2^j)$$

- Plasăm un nou domino orizontal, cu condiția ca bitul $j+1$ să nu fie activat în `mask`:

$$(i, j-1, \texttt{mask}) \to (i, j, \texttt{mask} \oplus 2^{(j+1)})$$

Cazul 2: Bitul j este activat în `mask` Plasăm restul de domino:

$$(i, j-1, \texttt{mask}) \to (i, j, \texttt{mask} \oplus 2^j)$$

Pentru a face tranziția din rândul $i-1$ la următorul rând i din matrice, vom înlocui prima stare în cele două cazuri de mai sus:

$$(i, j-1, \texttt{mask}) \text{ se transformă în } (i-1, M-1, \texttt{mask}), \quad \text{pentru } j = 0.$$

Răspunsul final se află la starea:

$$(N-1, M-1, 0).$$

Detalii pentru implementare Vom folosi următoarele convenții și pași pentru implementare:

- Folosim primul rând (0) pentru inițializare, iar matricea va avea valori începând cu rândul 1.
- Starea de început este:
$$\texttt{dp[0][M-1][0]} = 1.$$
- Tranziția comună din *Cazul 1* și *Cazul 2* va fi scrisă pe o singură linie, folosind operatorul `XOR` și verificări pe biți.

Implementare C++ 6.1: Implementarea completă Plug DP pentru acoperirea cu dominouri

```
dp[0][M - 1][0] = 1;
for (int i = 1; i <= N; i++) {
    for (int j = 0; j < M; j++) {
        int last_row, last_column;
        if (j == 0) last_row = i - 1, last_column = M - 1;
        else last_row = i, last_column = j - 1;
        for (int k = 0; k < (1 << M); k++) {
            dp[i][j][k ^ (1 << j)] = (dp[i][j][k ^ (1 << j)] + dp[last_row
                ][last_column][k]) % MOD;
            if (j != M - 1 && !(k & (1 << j)) && !(k & (1 << (j + 1)))) {
                dp[i][j][k ^ (1 << (j + 1))] =
                    (dp[i][j][k ^ (1 << (j + 1))] + dp[last_row][
                        last_column][k]) % MOD;
            }
        }
    }
}
int rezultat = dp[N][M - 1][0];
```

Optimizare memorie Putem observa faptul că starea actuală din dinamică depinde doar de starea calculată anterior, din celula `(last_row, last_column)`. Astfel, putem optimiza implementarea din punct de vedere al memoriei, memorând doar două tablouri unidimensionale:

- `dp[0][mask]` — starea anterioară
- `dp[1][mask]` — starea curentă

Implementare C++ 6.2: Implementarea optimizată Plug DP (2 rânduri de stări)

```
dp[0][0] = 1;
for (int i = 1; i <= N; i++) {
    for (int j = 0; j < M; j++) {
        for (int k = 0; k < (1 << M); k++) {
            dp[1][k ^ (1 << j)] = (dp[1][k ^ (1 << j)] + dp[0][k]) % MOD;
            if (j != M - 1 && !(k & (1 << j)) && !(k & (1 << (j + 1)))) {
                dp[1][k ^ (1 << (j + 1))] =
                    (dp[1][k ^ (1 << (j + 1))] + dp[0][k]) % MOD;
            }
        }
        for (int k = 0; k < (1 << M); k++)
            dp[0][k] = dp[1][k], dp[1][k] = 0;
    }
}
int rezultat = dp[0][0];
```

Analiză complexitate Complexitatea de timp este dată de secțiunea de cod în care se parcurg toate stările posibile ale matricei `dp`. Astfel, pentru fiecare stare a matricei `dp`, adică pentru fiecare combinație de linii N, coloane M și măști de biți de lungime 2^M, se încearcă generarea de configurații valide. Acest lucru duce la un ordin de complexitate $O(N \cdot M \cdot 2^M)$.

Complexitatea de spațiu este dată de cele două tablouri unidimensionale (curent și anterior) de dimensiune 2^M, ceea ce duce la un ordin $O(2^{(M+1)})$.

Comparativ cu o abordare *brute force* care plasează dominouri prin backtracking și are complexitatea de timp $O(2^{\frac{N \cdot M}{2}})$, varianta cu Plug DP este mult mai eficientă și scalabilă pentru matrici de dimensiuni mari.

6.2 Exemple de probleme care folosesc tehnica Plug DP

6.2.1 G. Grid Gradient

Cerință Nežmah a primit de ziua lui o grilă cu n rânduri și m coloane. Vrea să o umple cu numere de la 1 la 4, iar cum este obsedat de gradient, vrea ca diferența absolută între numerele scrise în celule adiacente (care au o latură comună) să fie exact unu. Cum este și mai obsedat de numărat, ajută-l să afle câte astfel de grile există, modulo 998.244.353!

Constrângeri

$$1 \leq n, m \leq 20$$

Rezolvare Observăm faptul că două celule adiacente au numere cu parități diferite. Astfel, formăm o tablă de șah în două moduri (prima celulă conține un număr par sau unul impar). Rezolvările pentru cele două cazuri sunt identice.

Presupunem pe tot parcursul rezolvării că prima celulă conține un număr par (2 sau 4).

Construim tabla doar cu numerele 2 și 3 (evident este un grid valid).

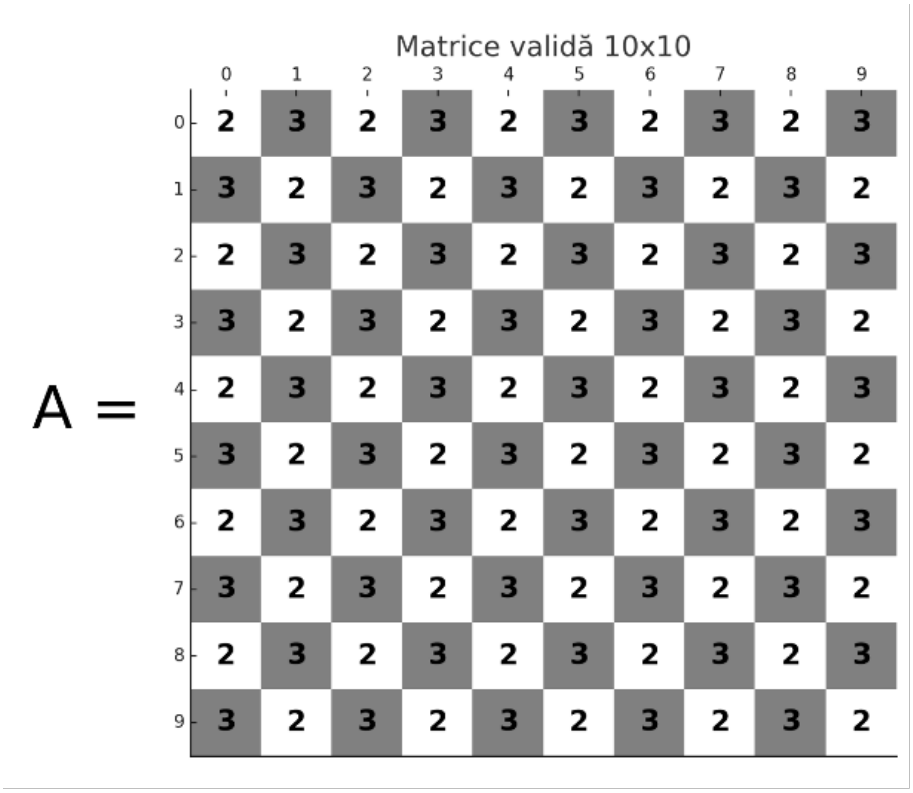

Figura 6.2: Matrice validă având doar valori de 2 și 3.

Pentru un grid valid M, notăm cu S mulțimea celulelor diferite față de matricea A din figura 6.1.2:

$$S = \{(i, j) : M_{(i,j)} \neq A_{(i,j)}\}.$$

Observăm că două celule din S nu pot fi adiacente pentru un grid valid.

Astfel, trebuie să contorizăm matricile M având condiția ca mulțimea S:

- să nu conțină două celule adiacente
- parităților matricii M și A să fie aceleași:

$$M_{(i,j)} \equiv A_{(i,j)} \pmod 2.$$

Pentru a rezolva problema de mai sus, ne folosim de tehnica *Plug DP*. Masca de stare va avea valoarea 1 pe poziția x dacă vom avea un element diferit față de $A_{(i,j)}$ (4 pentru 2 și 1 pentru 3) și 0 în rest.

Aflându-ne la poziția (i, j) cu masca de stare `mask`, putem face următoarele tranziții:

Tranziții

- **Cazul 1:** Bitul j este activat în `mask`. Plasăm un nou element egal cu $A_{(i,j)}$:

$$(i, j - 1, \texttt{mask}) \rightarrow (i, j, \texttt{mask} \oplus 2^j)$$

- **Cazul 2:** Bitul j nu este activat în `mask`.
  - Plasăm un nou element egal cu $A_{(i,j)}$:

$$(i, j-1, \texttt{mask}) \rightarrow (i, j, \texttt{mask})$$

  - Plasăm un nou element diferit de $A_{(i,j)}$ cu condiția ca bitul $j-1$ să nu fie activat în `mask`:

$$(i, j-1, \texttt{mask}) \rightarrow (i, j, \texttt{mask} \oplus 2^j)$$

Rezultatul final va fi înmulțit cu 2 (deoarece se consideră și matricea cu primul element impar).

Implementare C++ 6.3: Implementarea finală Plug DP pentru Grid Gradient

```
dp[0][0] = 1;
for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {
        for (int mask = 0; mask < (1 << m); mask++) {
            if (!(mask & (1 << j))) {
                dp[1][mask] = (dp[1][mask] + dp[0][mask]) % MOD;
            }
            if (!(mask & (1 << j)) && (j == 0 || !(mask & (1 << (j - 1))))) {
                int new_mask = mask ^ (1 << j);
                dp[1][new_mask] = (dp[1][new_mask] + dp[0][mask]) % MOD;
            }
            if (mask & (1 << j)) {
                int new_mask = mask ^ (1 << j);
                dp[1][new_mask] = (dp[1][new_mask] + dp[0][mask]) % MOD;
            }
        }
        for (int mask = 0; mask < (1 << m); mask++) {
            dp[0][mask] = dp[1][mask];
            dp[1][mask] = 0;
        }
    }
}
int result = 0;
for (int mask = 0; mask < (1 << m); mask++) {
    result = (result + dp[0][mask]) % MOD;
}
result = result * 2 % MOD;
```

Analiză complexitate Complexitatea de timp este dată de parcurgerea stărilor matricei `dp`, pentru fiecare combinație de linie n, coloană m și mască de biți de lungime 2^M. Astfel, se ajunge la un ordin de complexitate $O(n \cdot m \cdot 2^m)$.

Complexitatea de spațiu este dată de cele două tablouri unidimensionale (curent și anterior) de dimensiune 2^m, deci un ordin $O(2^{(m+1)})$.

Comparativ cu o abordare *brute force* care încearcă fiecare modalitate de a plasa cele 4 numere (cu o complexitate $O(4^{(n \cdot m)})$), varianta cu Plug DP este semnificativ mai eficientă pentru matrici de dimensiuni mari.

Notă: Pentru clarificarea și aprofundarea conceptelor de tip *Plug DP* sau *Broken Profile DP*, am consultat următoarele surse suplimentare:

- Qu [15] pentru detalii despre implementarea și optimizarea tehnicii *Plug DP*.
- Xiao [16] pentru idei și exemple suplimentare.
- Wiki [17] pentru mai multe probleme folosind tehnica Plug DP.

Capitolul 7

Concluzii

7.1 Modul de realizare a temei

Obiectivul acestei lucrări a fost de a explica modul în care măștile de biți sunt utilizate pentru reprezentarea și manipularea numerelor, precum și de a demonstra aplicarea tehnicii de programare dinamică cu măști de biți pentru rezolvarea diferitelor probleme algoritmice. Tema a fost concepută pentru a fi ușor de înțeles și de aplicat studenților pasionați de programare competitivă, oferind explicații detaliate pentru fiecare tip de problemă abordată.

7.2 Posibile dezvoltări ulterioare

Tehnica programării dinamice cu măști de biți oferă un cadru promițător pentru optimizări suplimentare, atât în ceea ce privește timpul de execuție, cât și consumul de memorie. În această lucrare, abordarea clasică `dp[mask]` a fost utilizată pentru a reprezenta și actualiza stările în mod eficient. Totuși, există oportunități clare de îmbunătățire a performanței prin utilizarea unor structuri de date alternative.

O direcție importantă constă în **utilizarea structurii `bitset`**, care poate oferi beneficii semnificative din punct de vedere al vitezei și al ocupării memoriei. Spre deosebire de vectorii tradiționali, `bitset` permite aplicarea directă a operațiilor logice la nivel de bit, fiind optimizat pentru execuție pe arhitecturi moderne. În anumite probleme, acest lucru se traduce printr-o reducere semnificativă a timpului de rulare.

În plus, pot fi explorate tehnici avansate precum:

- **Utilizarea bibliotecilor specializate** pentru lucrul cu seturi de biți, care pot integra optimizări hardware (precum instrucțiuni SIMD sau paralelism pe biți).
- **Reducerea spațiului de stări** prin identificarea stărilor redundante sau simetrice, reducând astfel complexitatea algoritmului.

Integrarea acestor îmbunătățiri poate conduce la soluții mai rapide, mai compacte și mai scalabile, permițând rezolvarea unor instanțe mai mari și mai complexe ale problemelor abordate.

7.3 Aprecieri personale privind relevanța rezultatelor

Consider că rezultatele și exemplele de aplicare ale programării dinamice cu măști de biți arată clar că este potrivită pentru programarea competitivă. Această tehnică nu numai că rezolvă probleme aparent dificile, dar ajută și la dezvoltarea unei gândiri algoritmice mai puternice, care este esențială pentru succesul în competiții.

Toate soluțiile implementate sunt disponibile pe GitHub[1] pentru consultare și studiu.

[1] https://github.com/dragosc1/Programare-dinamica-cu-masti-de-biti

Bibliografie

[1] Richard Bellman, Dynamic Programming, Princeton University Press, 1957.

[2] Antti Laaksonen, Competitive Programmer's Handbook, August 19, 2019, https://usaco.guide/CPH.pdf#page=105, Accesat: 15.01.2025.

[3] CEOI, Amusement Park – CEOI 2019, https://oj.uz/problem/view/CEOI19_amusementpark, Accesat: 11.06.2025.

[4] Shijie Ren, Amusement Park – CEOI 2019, https://usaco.guide/problems/ceoi-2019amusement-park/solution, Accesat: 11.03.2025.

[5] Codeforces, Vowels – Codeforces 383E, https://codeforces.com/problemset/problem/383/E, Accesat: 11.06.2025.

[6] Infoarena, XorTransform, https://www.infoarena.ro/problema/xortransform, Accesat: 11.06.2025.

[7] CSES, And Subset Count – Task 3141, https://cses.fi/problemset/task/3141, Accesat: 11.06.2025.

[8] Rameez Parwez, Siyong Huang și Aakash Gokhale, DP - Sum Over Subsets, https://usaco.guide/plat/dp-sos?lang=cpp, Accesat: 01.04.2025.

[9] Utkarsh Saxena, SOS DP, https://codeforces.com/blog/entry/45223, Accesat: 01.04.2025.

[10] Ștefan Dăscălescu, Kilonova XOR Transform, https://usaco.guide/problems/kilonova-xortransform/solution, Accesat: 10.05.2025.

[11] Patrick Corn, Pi Han Goh și Omkar Kulkarni, Lucas's theorem, https://brilliant.org/wiki/lucas-theorem/, Accesat: 08.06.2025.

[12] Michael Cao, Siyong Huang și Peng Bai, Bitmask DP - USACO Guide, https://usaco.guide/gold/dp-bitmasks?lang=cpp, Accesat: 04.06.2025.

[13] AtCoder, F - Coprime Present (Atcoder ABC Round 195), https://atcoder.jp/contests/abc195/tasks/abc195_f, Accesat: 10.06.2025.

[14] SPOJ, NGCD – NO GCD, https://www.spoj.com/problems/NGCD/, Accesat: 11.06.2025.

[15] Andi Qu, Broken Profile DP, https://usaco.guide/adv/dp-broken-profile?lang=cpp, Accesat: 25.05.2025.

[16] Edward Xiao, Codeforces Blog Entry 90841, https://codeforces.com/blog/entry/90841, Accesat: 26.05.2025.

[17] OI Wiki, Plug DP, https://oi-wiki.org/dp/plug/, Accesat: 04.06.2025.

Anexa A

Grafice comparații complexități

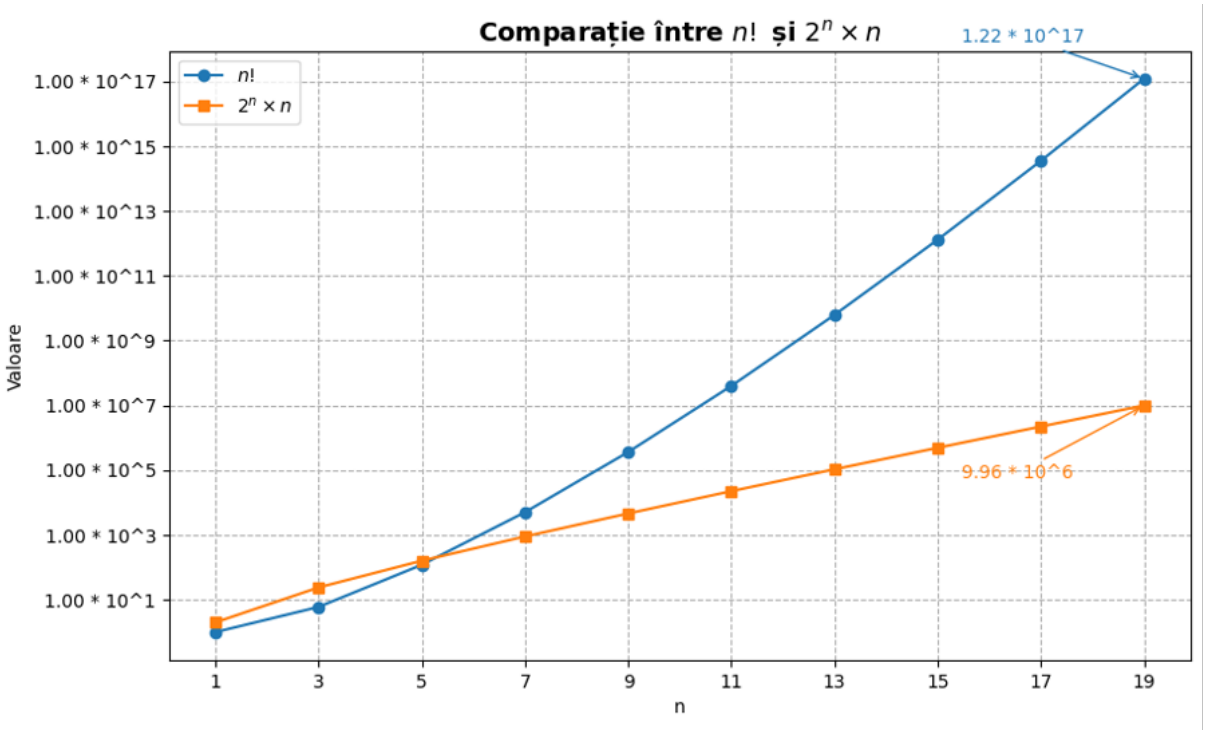

Figura A.1: Comparație între $n!$ și $2^n \cdot n$

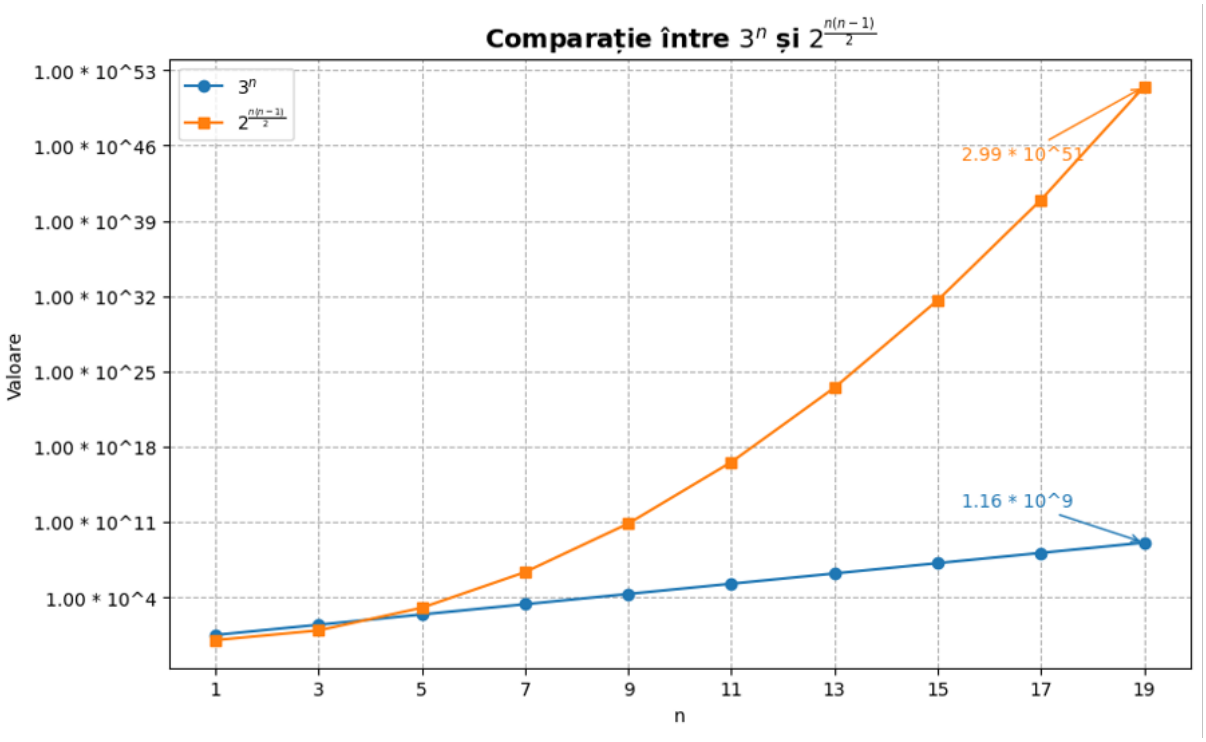

Figura A.2: Comparație între 3^n și $2^{\frac{n(n-1)}{2}}$

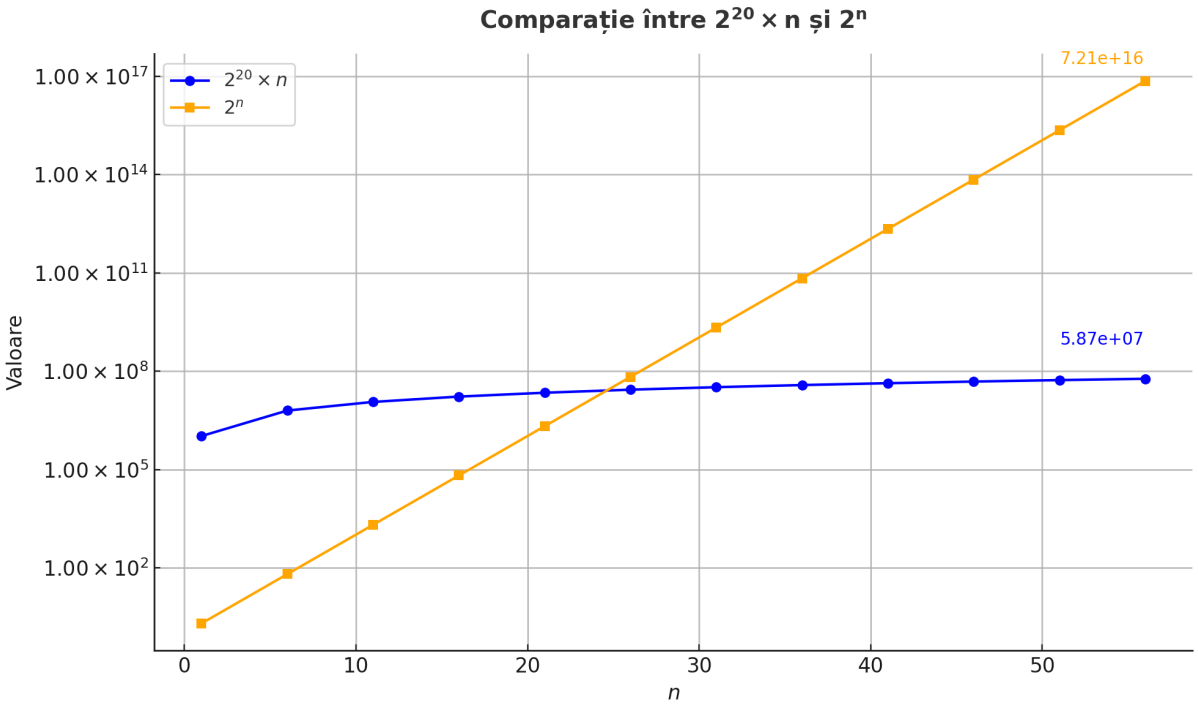

Figura A.3: Comparație între $2^{20} \cdot n$ și 2^n

Anexa B

Implementări funcții auxiliare

Aritmetică modulară

Implementare C++ B.1: Exponentiere rapidă modulo `MOD`

```cpp
int pow_mod(int base, int exp, int MOD) {
    int result = 1;
    base = base % MOD;
    while (exp > 0) {
        if (exp % 2 == 1) {
            result = (1LL * result * base) % MOD;
        }
        exp = exp >> 1;
        base = (1LL * base * base) % MOD;
    }
    return result;
}
```

Implementare C++ B.2: Calculul inversului modular folosind exponentiere rapidă

```cpp
int inv(int a, int MOD) {
    return pow_mod(a, MOD - 2, MOD);
}
```

SOS DP

Implementare C++ B.3: SOS DP pentru submăști cu operatorul XOR

```
void SOS_DP_xor(vector<int>& F) {
    int LEN = F.size();
    for (int ind = 0; ind < log2(LEN); ind++)
        for (int mask = 0; mask < LEN; mask++)
            if (mask & (1 << ind))
                F[mask] = F[mask] ^ F[mask ^ (1 << ind)];
}
```

Implementare C++ B.4: SOS DP pentru submăști cu operatorul +

```
void SOS_DP_add(vector<int>& F) {
    int LEN = F.size();
    for (int ind = 0; ind < log2(LEN); ind++)
        for (int mask = 0; mask < LEN; mask++)
            if (mask & (1 << ind))
                F[mask] = F[mask] + F[mask ^ (1 << ind)];
}
```

Implementare C++ B.5: SOS DP pentru supramăști cu operatorul +

```
void SOS_DP_prime_add(vector<int>& F) {
    int LEN = F.size();
    for (int ind = 0; ind < log2(LEN); ind++)
        for (int mask = LEN - 1; mask >= 0; mask--)
            if (!(mask & (1 << ind)))
                F[mask] = (F[mask] + F[mask ^ (1 << ind)]) % MOD;
}
```

Implementare C++ B.6: SOS DP pentru supramăști cu operatorul -

```
void SOS_DP_prime_sub(vector<int>& F) {
    int LEN = F.size();
    for (int ind = 0; ind < log2(LEN); ind++)
        for (int mask = LEN - 1; mask >= 0; mask--)
            if (!(mask & (1 << ind)))
                F[mask] = (1LL * F[mask] - F[mask ^ (1 << ind)] + MOD) %
                    MOD;
}
```