Université virtuelle du Burkina Faso (UV-BF)

Option : Master Fouille de données et Intelligence Artificielle

Mini Projet Complexité algorithmique

BADXOR - Bad XOR

BASSOLE Yipéné Harold Ezekiel
BAZIE Dureel Donaldson
COULIBALY Cheick Ahmed
OUATTARA Arnauld
KOALA Valentin

4 mai 2025

Plan

Introduction

Notre choix s'est porté sur le problème intitulé BADXOR - Bad XOR.

Description complète du problème

On dispose de deux tableaux : un tableau A de N éléments et un tableau B de M éléments.
Un **sous-ensemble de A est considéré comme mauvais** s'il existe un élément B_j dans B tel que la valeur **XOR** de tous les éléments du sous-ensemble soit égale à B_j, autrement dit si le **XOR du sous-ensemble** est dans B.

Objectif : déterminer le **nombre de sous-ensembles bons** (non mauvais). Ce nombre peut être très grand, donc il faut donner le résultat **modulo** 100000007.

Description complète du problème

Quelques précisions sont apportées concernant les entrées :

- $0 \leq N$, $M \leq 1000$
- Le sous-ensemble vide ($\emptyset$) est un cas spécial car son XOR vaut 0
- Jusqu'à T tests possibles, avec $1 \leq T \leq 20$

Description complète du problème

Il nous a paru utile d'apporter quelques définitions, pour une bonne compréhension :

- **Sous-ensemble** : ensemble contenant certains ou tous les éléments d'un ensemble original.
- **Opération binaire XOR ($\oplus$)** : retourne 1 si les bits sont différents, 0 s'ils sont identiques.

Sa table de vérité se présente comme suit :

$Bit1$	$Bit2$	$Bit1 \oplus Bit2$
0	0	0
0	1	1
1	0	1
1	1	0

- **Valeur XOR d'un sous-ensemble** : c'est le résultat de l'opération XOR appliquée à tous les éléments du sous-ensemble.

Description complète du problème

Considérons l'exemple traité dans le problème : on souhaite faire deux tests, ($T = 2$).

- Test 1 : $A = [1, 2]$ et $B = [0, 1, 2]$
- Test 2 : $A = [1]$ et $B = [0, 1, 3]$

Étude du premier cas : tous les sous-ensembles possibles de $A = [1, 2]$ sont : $\emptyset$, $\{1\}$, $\{2\}$, $\{1, 2\}$.

Tableau de résultat du test 1 :

Sous-ensembles de A	Binaire	XOR ($\oplus$)	Décimal	Bon / Mauvais
$\emptyset$	000	000	0	Mauvais ($\in B$)
$\{1\}$	001	001	1	Mauvais ($\in B$)
$\{2\}$	010	010	2	Mauvais ($\in B$)
$\{1, 2\}$	$001 \oplus 010$	011	3	Bon ($\notin B$)

Ainsi, 3 n'étant pas élément de B, il n'y a donc qu'un seul bon sous-ensemble dans A $= \{1, 2\}$.
D'où en sortie, on obtient le résultat 1 : case 1 : 1.

Cas pratique dans la vie réelle

Contexte sociétal, un problème de santé publique : le cas des produits dépigmentants

Dans de nombreux pays, particulièrement en Afrique, un phénomène préoccupant prend de l'ampleur : **la dépigmentation volontaire de la peau**. Influencées par des standards de beauté imposés à travers les médias, les réseaux sociaux, la publicité, la forte pression sociale, de nombreuses personnes — en particulier les femmes — cherchent à "éclaircir" leur peau.

Ce désir, souvent issu d'un **complexe d'infériorité** hérité de l'histoire coloniale ou des normes de beauté occidentales, pousse hommes et femmes à utiliser des **produits cosmétiques souvent dangereux**, voire destructeurs pour la peau.

Cas pratique dans la vie réelle

Problème scientifique

Des études menées par des *dermatologues, chimistes, médecins et pharmaciens* ont révélé que :

- Certains produits seuls peuvent être **toxiques**.
- Certains **mélanges** de produits cosmétiques peuvent produire des **réactions chimiques dangereuses**.
- D'autres produits, seuls ou combinés, peuvent être **bénéfiques pour la peau**.

Cas pratique dans la vie réelle

Solution numérique proposée : Application *"Cosmétiques & Moi"*

Afin de répondre concrètement au problème, le gouvernement fait appel aux informaticiens et autres experts pour développer une **application mobile éducative et préventive**.

Modélisation informatique

- Tous les produits cosmétiques sont représentés dans un tableau $A[]$.
- Les mélanges considérés comme dangereux sont codés dans un tableau $B[]$.
- Chaque produit peut être représenté sous forme de **valeur numérique** selon ses composants chimiques.
- L'opération **XOR (exclusif)** est utilisée pour simuler les effets de combinaisons chimiques entre plusieurs produits.

Cas pratique dans la vie réelle

Principe de fonctionnement

- L'utilisateur entre les produits qu'il souhaite utiliser.
- L'application génère **toutes les combinaisons possibles** (sous-ensembles).
- Pour chaque combinaison, elle calcule le **XOR** des valeurs.
- Si ce XOR est présent dans $B[]$, il s'agit d'un **mauvais mélange**.
- Sinon, le mélange est considéré comme **bon**.

Cas pratique dans la vie réelle

Exemple d'usage

Une utilisatrice sélectionne trois produits. L'application :

1. génère toutes les combinaisons possibles de ces produits ;
2. calcule le XOR de chaque combinaison ;
3. compare le résultat à la base de données $B[]$ des mélanges interdits ;
4. retourne un message de mise en garde ou d'approbation.

Cas pratique dans la vie réelle

Avantages

Étant donné que les campagnes de sensibilisation sont inefficaces, les publicités, les publicités convaincantes diffusées massivement, ne sont pas efficaces pour empêcher les gens d'utiliser des produits, on va plutôt les **accompagner dans des choix plus sains**.
L'objectif est de :

- Prévenir les dangers liés aux mélanges toxiques,
- Fournir des **conseils personnalisés** et accessibles,
- Promouvoir une beauté naturelle et respectueuse de la santé.

Résultat attendu

- Réduction des brûlures et effets secondaires liés aux produits cosmétiques.
- Prise de conscience progressive dans la population.
- Un outil intelligent d'éducation, de prévention et de soin de la peau.

Méthode de résolution du problème

Nous avons utilisé l'approche dynamique de résolution de problèmes, à savoir la programmation dynamique.

C'est quoi la programmation dynamique ?
La programmation dynamique est une méthode algorithmique pour résoudre les problèmes d'optimisation. Elle consiste à décomposer un problème en sous-problèmes, puis à les résoudre du plus petit au plus grand, en stockant les résultats intermédiaires (au lieu de les recalculer plusieurs fois), et à les réutiliser pour construire la solution finale.

Méthode de résolution du problème

Quand est-ce qu'on l'utilise ?

On utilise la programmation dynamique lorsque :

- Pour résoudre le problème initial, on est amené à résoudre plusieurs fois les mêmes sous-problèmes.
- Ou lorsque la solution optimale du grand problème dépend des solutions optimales de ses sous-problèmes.

Par exemple, la résolution par récursivité de la suite de Fibonacci :

```
fonction Fibonacci(n)
  si n = 0 ou n = 1
    retourner n
  sinon
    retourner Fibonacci(n-1) + Fibonacci(n-2)
```

Cette méthode est inefficace pour le calcul du n-ième terme car il y a beaucoup de répétitions inutiles (par exemple, `Fibonacci(2)` est recalculé plusieurs fois).

Méthode de résolution du problème

Méthodes de résolution en programmation dynamique

1. **Top-down (méthode descendante)** : On part du problème global, on résout récursivement les sous-problèmes tout en mémorisant les résultats intermédiaires (mémoïsation) dans un tableau ou un dictionnaire.
2. **Bottom-up (méthode ascendante)** : On commence par les plus petits sous-problèmes, puis on calcule progressivement les solutions des problèmes plus grands en utilisant le principe d'optimalité. Les résultats sont mémorisés dans un tableau.

Méthode de résolution du problème

Pourquoi la programmation dynamique est adaptée à notre problème `BADXOR` ?

Dans le problème `BADXOR`, il s'agit de compter le nombre de sous-ensembles de A dont le XOR n'appartient pas à l'ensemble B. Sachant que A est de taille N, il existe 2^N sous-ensembles possibles. Avec jusqu'à 20 cas de test, il n'est pas efficace de tous les générer naïvement.

La programmation dynamique nous permet de :

- Mémoriser le nombre de façons différentes d'obtenir un certain XOR à partir d'un sous-ensemble.
- Éviter de recalculer ou reconsidérer plusieurs fois les mêmes combinaisons.

Algorithme détaillé - en Pseudo code

Algorithm Algorithme 1/3

Input : Nombre de tests `nombre_test`, puis pour chaque test : un tableau `A` de taille `N`, un tableau `B` de taille `M`

Output : Le nombre total de sous-ensembles de `A` dont le XOR ne figure pas dans `B`

Algorithme détaillé - en Pseudo code

Algorithm Algorithm 2/3

```
MODULO ← 100000007
pour chaque test de 1 à nombre_test faire
    Lire N et M
    Lire le tableau A de N éléments
    Lire le tableau B de M éléments
    B_set ← ensemble des éléments de B

    compteur_xor[0...1023] ← 0
    compteur_xor[0] ← 1 ;        // Le XOR du sous-ensemble
    vide est 0
```

Algorithme détaillé - en Pseudo code

Algorithm Algorithm 3/3

```
pour chaque test de 1 à nombre_test faire
    pour chaque élément a dans A faire
        nouveau_compteur_xor ← copie de compteur_xor
        pour x de 0 à 1023 faire
            nouveau_xor ← x XOR a
            nouveau_compteur_xor[nouveau_xor]                ←
            (nouveau_compteur_xor[nouveau_xor]               +
            compteur_xor[x]) modulo MODULO
        compteur_xor ← nouveau_compteur_xor
    total ← 0
    pour x de 0 à 1023 faire
        si x ∉ B_set alors
            total ← (total + compteur_xor[x]) modulo MODULO
    Afficher "Case i: total"
```

Démonstration mathématique de l'algorithme

1. Définitions et Énoncé du Problème

Soient :

- $A = \{a_1, a_2, \ldots, a_N\}$ un ensemble de N entiers
- $B = \{b_1, b_2, \ldots, b_M\}$ un ensemble de M entiers

On souhaite compter le nombre de **sous-ensembles** $S \subseteq A$ tels que :

$$\bigoplus_{x \in S} x \notin B$$

où $\oplus$ désigne l'opération **XOR** (OU exclusif bit à bit).

2. Propriétés du XOR

- **Associativité et Commutativité** : l'ordre des opérations n'affecte pas le résultat
- **Élément neutre** : 0 est neutre ($x \oplus 0 = x$)
- **Auto-inverse** : $x \oplus x = 0$
  Donc, l'ensemble $\mathbb{N}$ muni de la loi de composition interne $\oplus$ est un groupe abélien.

Démonstration mathématique de l'algorithme

3. Approche par Programmation Dynamique

On définit :

- $dp[i][x]$ = nombre de sous-ensembles des i premiers éléments de A dont le XOR est x

Initialisation :

- Pour $i = 0$ (ensemble vide), seul $x = 0$ est possible :

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

(car le XOR d'un ensemble vide est 0)

Relation de récurrence : pour chaque élément a_i, deux choix existent pour chaque sous-ensemble existant :

1. **Ne pas inclure** a_i : les sous-ensembles restent inchangés
2. **Inclure** a_i : leur nouveau XOR devient $x \oplus a_i$

Ainsi, la mise à jour est :

$$dp[i][x] = dp[i-1][x] + dp[i-1][x \oplus a_i]$$

Optimisation spatiale : On peut utiliser un seul dictionnaire `dp` mis à jour itérativement.

Démonstration mathématique de l'algorithme

4. Preuve de Complétude

Théorème : Après avoir traité tous les éléments de A, le dictionnaire dp contient tous les XOR possibles des sous-ensembles de A, avec leur nombre d'occurrences.

Preuve par récurrence :

- **Pour** $i = 0$: Seul $dp[0][0] = 1$ (sous-ensemble vide)
- **Pour** $i - 1$: Supposons que $dp[i-1]$ contient tous les XOR des sous-ensembles de $\{a_1, \ldots, a_{i-1}\}$. Alors, pour a_i, chaque sous-ensemble peut :
  - Soit exclure a_i (conservant son XOR)
  - Soit l'inclure (modifiant son XOR en $x \oplus a_i$)
- **A l'ordre** i : Donc $dp[i]$ contient bien tous les XOR possibles

5. Calcul Final

Le résultat est la somme des comptes des XOR qui ne sont pas dans B :

$$\text{total} = \sum_{x \notin B} dp[M][x]$$

Analyse de Complexité - Ligne par Ligne

Instruction	Type d'opération	Complexité	Remarques
Lire nombre_test	Lecture	$O(1)$	Lecture d'un entier
MODULO $\leftarrow$ 100000007	Initialisation constante	$O(1)$	Affectation unique
Pour chaque test de 1 à nombre_test faire	Boucle principale	$O(T)$	$T \leq 20$
Lire N et M	Lecture	$O(1)$	Lecture des tailles
Lire tableau A	Lecture	$O(N)$	Lecture des N éléments
Lire tableau B	Lecture	$O(M)$	Lecture des M éléments
B_set $\leftarrow$ ensemble de B	Initialisation	$O(M)$	Conversion en set pour accès rapide
compteur_xor[0...1023] $\leftarrow$ 0	Initialisation tableau	$O(1)$	Taille constante (1024)
compteur_xor[0] $\leftarrow$ 1	Initialisation valeur	$O(1)$	Base : ensemble vide
Pour chaque a dans A	Boucle sur A	$O(N)$	Parcourt tous les éléments
copie compteur_xor $\rightarrow$ nouveau_compteur_xor	Copie tableau	$O(S)$	$S \leq 1024$ XORs possibles
Pour x de 0 à 1023	Boucle fixe	$O(S)$	$S = 1024$ max
nouveau_xor $\leftarrow$ x XOR a	Calcul	$O(1)$	Opération bit à bit
nouveau_compteur_xor[...] $\leftarrow$ ...	Écriture / mise à jour	$O(1)$	Ajout modulo MODULO
compteur_xor $\leftarrow$ nouveau_compteur_xor	Affectation tableau	$O(1)$	Référence ou remplacement
total $\leftarrow$ 0	Initialisation	$O(1)$	Préparation accumulation
Pour x de 0 à 1023	Boucle fixe	$O(S)$	Boucle sur tous les XOR
Si x B_set	Vérification d'appartenance	$O(1)$	Opération sur set
total $\leftarrow$ total + compteur_xor[x]	Accumulation	$O(1)$	Avec modulo
Afficher "Case i : total"	Sortie	$O(1)$	Affichage standard

Résumé de la complexité

- **Temps par cas** : $\mathcal{O}(N \times S)$ où $S \leq 1024$
- **Temps total (T cas)** : $\mathcal{O}(T \times N)$ en pratique
- **Espace** : $\mathcal{O}(S)$ soit constant (≤ 1024)

Choix du langage

Lors de la première implémentation, nous avons utilisé le langage **Python** pour sa simplicité et sa rapidité de développement. Cependant, cette version ne respectait pas les contraintes de temps imposées, notamment pour les cas de test volumineux, en raison de la relative lenteur de l'interpréteur Python.

Pour pallier ce problème et garantir une exécution plus efficace, nous avons opté pour une réécriture de l'algorithme en **C++**, un langage compilé offrant de meilleures performances. Ce changement nous a permis de répondre aux exigences de temps tout en conservant la logique initiale de l'algorithme.

Nous avons tenu à présenter tout de même les deux programmes

Programme de Résolution : Code Python - Partie 1

```
1
2  from collections import defaultdict
3
4  MODULO = 100000007
5  # lire le nombre de cas de test
6  nombre_de_cas = int(input())
7  if nombre_de_cas < 1 or nombre_de_cas > 20:
8      raise ValueError("Le nombre de cas de test doit être compris
           entre 1 et 20.")
```

Programme de Résolution : Code Python - Partie 2

```
1       # pour chaque cas de test, car tout ce bloc de code est un
            cas de test
2
3   for numero_cas in range(1, nombre_de_cas + 1):
4
5       # Lecture des tableaux A et B dont les éléments de chacun
            sont séparés par des espaces, ensuite, chaque élément
            de chaque tableau est converti en entier par (map(int,)
             et le tableau est stoqué sous forme de liste
6
7       taille_A, taille_B = map(int, input().split())
8       tableau_A = list(map(int, input().split())) if taille_A > 0
            else []
9       tableau_B = list(map(int, input().split())) if taille_B > 0
            else []
```

Programme de Résolution : Code Python - Partie 3

```
    # Vérification que les éléments de A et B sont compris entre
        0 et 1000

    for a in tableau_A:
        if a < 0 or a > 1000:
            raise ValueError("Élément invalide dans A.")
    for b in tableau_B:
        if b < 0 or b > 1000:
            raise ValueError("Élément invalide dans B.")

    dp = defaultdict(int)
    # Pour stocquer le nombre de sous-ensemble ayant un xor
        donné, Utilisation de defaultdict pour éviter les
        KeyError

    dp[0] = 1
    # Le sous-ensemble vide a un XOR de 0

    B = set(tableau_B)
```

Programme de Résolution : Code Python - Partie 4

```
    for a in tableau_A:
        nouveau_dp = dp.copy()
        for xor_actuel, count in dp.items():
            if count:  # nombre de sous-ens ayant un XOR actuel existant
                nouveau_xor = xor_actuel ^ a
                nouveau_dp[nouveau_xor] = (nouveau_dp.get(nouveau_xor, 0) + count) % MODULO
        dp = nouveau_dp

    total = 0
    for xor_value, count in dp.items():
        if xor_value not in B:
            total = (total + count) % MODULO

    # Affichage du résultat
    print(f"Case {numero_cas}: {total}")
```

Problème BADXOR.cpp (1/3)

```cpp
// Problème BADXOR.cpp de SPOJ résolu avec C++14

#include <iostream>
#include <vector>
#include <unordered_set>
#include <bitset>
using namespace std;

const int MODULO = 100000007;

int main() {
    ios_base::sync_with_stdio(false);
    cin.tie(nullptr);
    int test_cases;
    cin >> test_cases;
```

Problème BADXOR.cpp (2/3)

```
1     if (test_cases < 1 || test_cases > 20) {
2         cerr << "Le nombre de cas de test doit être compris
              entre 1 et 20.\n";
3         return 1;
4     }
5
6     for (int case_num = 1; case_num <= test_cases; ++case_num) {
7         int size_A, size_B;
8         cin >> size_A >> size_B;
9         if (size_A < 0 || size_B < 0) {
10            cerr << "Invalid array sizes.\n";
11            return 1;
12        }
```

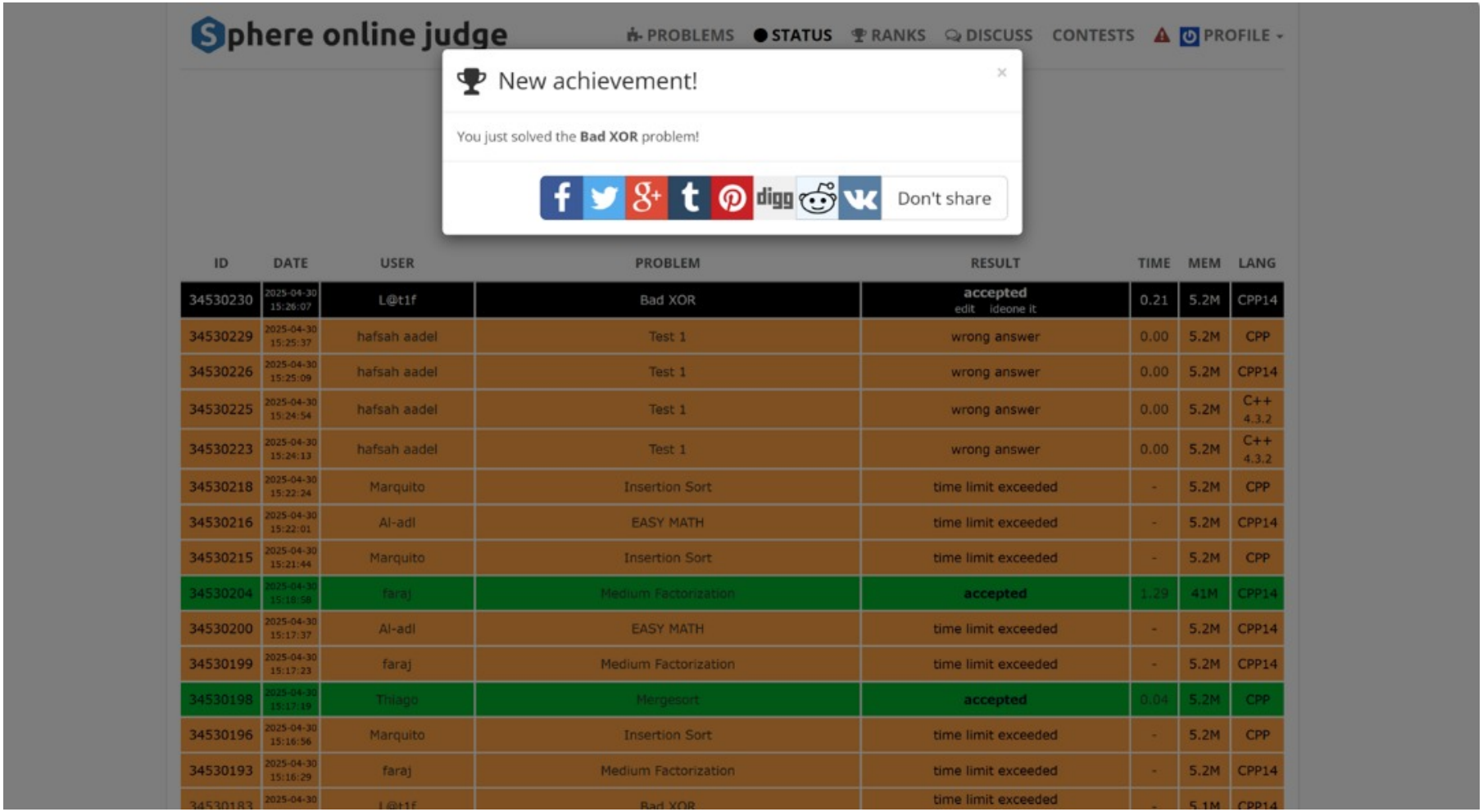
Problème BADXOR.cpp (3/3)

```
1         vector<int> A(size_A);
2         for (int i = 0; i < size_A; ++i) {
3             cin >> A[i];
4             if (A[i] < 0 || A[i] > 1000) return 1;
5         }
6         unordered_set<int> B_set;
7         for (int i = 0; i < size_B; ++i) {
8             int b; cin >> b;
9             if (b < 0 || b > 1000) return 1;
10            B_set.insert(b);
11        }
12
13        vector<int> dp(1024, 0);
14        dp[0] = 1;
15        for (int a : A) {
16            vector<int> new_dp = dp;
17            for (int x = 0; x < 1024; ++x) {
18                int new_xor = x ^ a;
19                new_dp[new_xor] += dp[x];
20                if (new_dp[new_xor] >= MODULO)
21                    new_dp[new_xor] -= MODULO;
22            }
23            dp = move(new_dp);
24        }
```

Problème BADXOR.cpp (3/3)

```
1
2           int total = 0;
3                   for (int x = 0; x < 1024; ++x) {
4                       if (!B_set.count(x)) {
5                           total = (total + dp[x]) % MODULO;
6                       }
7                   }
8
9                   cout << "Case " << case_num << ": " << total <<
                        '\n';
10              }
11              return 0;
12          }
```

Résolution du problème sur SPOJ - Soumission



ID	DATE	USER	PROBLEM	RESULT	TIME	MEM	LANG
34530230	2025-04-30 15:26:07	L@t1f	Bad XOR	accepted edit ideone it	0.21	5.2M	CPP14
34530229	2025-04-30 15:25:37	hafsah aadel	Test 1	wrong answer	0.00	5.2M	CPP
34530226	2025-04-30 15:25:09	hafsah aadel	Test 1	wrong answer	0.00	5.2M	CPP14
34530225	2025-04-30 15:24:54	hafsah aadel	Test 1	wrong answer	0.00	5.2M	C++ 4.3.2
34530223	2025-04-30 15:24:13	hafsah aadel	Test 1	wrong answer	0.00	5.2M	C++ 4.3.2
34530218	2025-04-30 15:22:24	Marquito	Insertion Sort	time limit exceeded	-	5.2M	CPP
34530216	2025-04-30 15:22:01	Al-adl	EASY MATH	time limit exceeded	-	5.2M	CPP14
34530215	2025-04-30 15:21:44	Marquito	Insertion Sort	time limit exceeded	-	5.2M	CPP
34530204	2025-04-30 15:18:58	faraj	Medium Factorization	accepted	1.29	41M	CPP14
34530200	2025-04-30 15:17:37	Al-adl	EASY MATH	time limit exceeded	-	5.2M	CPP14
34530199	2025-04-30 15:17:23	faraj	Medium Factorization	time limit exceeded	-	5.2M	CPP14
34530198	2025-04-30 15:17:19	Thiago	Mergesort	accepted	0.04	5.2M	CPP
34530196	2025-04-30 15:16:56	Marquito	Insertion Sort	time limit exceeded	-	5.2M	CPP
34530193	2025-04-30 15:16:29	faraj	Medium Factorization	time limit exceeded	-	5.2M	CPP14
34530183	2025-04-30	L@t1f	Bad XOR	time limit exceeded	-	5.1M	CPP14

Cas de test et résultats attendus

Cas #	Tableau A	Tableau B	Output (Nombre de bons sous-
1	[]	[0]	0
2	[5]	[]	2
3	[5]	[5]	1
4	[2, 3]	[1]	3
5	[2, 2]	[0]	2
6	[1, 2, 3]	[0, 1]	4
7	[1, 1, 1, 1]	[2, 3]	16
8	[1, 2, 3, 4, 5]	[1, 3, 5]	20
9	[1, 2, 3, 4, 5, 6]	[7, 0]	48
10	[10, 20, 30, 40, 50, 60, 70]	[0, 10, 110]	116
11	[0, 0, 0, 0, 0, 0, 0, 0]	[0, 1, 2, 3]	0
12	[1, 2, 3, 4, 5]	[]	32
13	[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]	[15]	960
14	[1000, 1000]	[0]	2
15	[500, 250, 250]	[0]	6
16	[0, 1, 2]	[3, 0]	4
17	[2, 4, 8, 16, 32]	[6, 24, 40]	29
18	[100, 200, 300, 400, 500, 600]	[0, 100, 200, 300]	56
19	[5, 5, 5, 5, 5]	[0, 5]	0
20	[0, 1, 1]	[0, 1]	0

Conclusion

- Le problème BADXOR nous a permis d'explorer des concepts clés de la programmation dynamique.
- Face à la complexité exponentielle du nombre de sous-ensembles possibles, une approche naïve serait inefficace.
- Grâce à une solution optimisée, nous avons réduit la complexité à $\mathcal{O}(N \times 1024)$ tout en limitant l'espace mémoire utilisé.
- Ce projet illustre l'intérêt de stocker et réutiliser des résultats intermédiaires pour améliorer les performances.
- Enfin, l'analyse nous a également permis de pratiquer la modélisation algorithmique et l'évaluation rigoureuse de la complexité.