République algérienne démocratique et Populaire
Ministère de l'enseignement supérieure et de la recherche scientifique
Université des sciences et de la technologie Houarie-Boumediene
Faculté d'informatique
Département d'informatique

Rapport de TP

Arithmétique modulaire

Exponentiation Modulaire

Calcul des puissances et leur utilité en cryptographie

Réalisé par :

- HADDADI Hadil G2 212131091285

Enseignante :

- Mme Dahmani

Spécialité : Sécurité des Systèmes Informatiques

2024/2025

Table des matières

Introduction

L'exponentiation modulaire est une opération fondamentale en cryptographie moderne. Elle est utilisée dans de nombreux protocoles cryptographiques tels que **RSA**, **Diffie-Hellman**, et les signatures numériques. Cette opération permet de calculer des puissances élevées tout en réduisant efficacement le résultat modulo un entier donné (elle consiste à calculer une puissance a^b puis à réduire le résultat modulo n, soit $a^b \bmod n$), ce qui est essentiel pour garantir la sécurité et l'efficacité des systèmes cryptographiques.

Cette opération joue un rôle central dans les systèmes de sécurité modernes, notamment pour assurer la confidentialité, l'intégrité, et l'authentification dans les communications numériques.

L'objectif de ce TP est d'explorer différentes méthodes permettant de calculer des puissances, d'implémenter quatre d'entre elles, et d'analyser leur efficacité, notamment en termes de complexité temporelle. Ce travail mettra en lumière les avantages et limites des différentes approches, en illustrant leur pertinence pour des applications cryptographiques.

Théorie de l'exponentiation modulaire

L'exponentiation modulaire calcule $c \equiv a^b \bmod n$ où :

- a est la base,
- b est l'exposant,
- n est le modulo.

Elle repose sur la réduction successive des puissances intermédiaires modulo n pour éviter des dépassements de capacité et assurer une exécution efficace.

Propriétés fondamentales :

1. $(a \cdot b) \bmod n = [(a \bmod n) \cdot (b \bmod n)] \bmod n$
2. $a^b \bmod n = (a^{b-1} \cdot a) \bmod n$.

Ces propriétés rendent l'exponentiation modulaire particulièrement adaptée aux environnements où les ressources mémoire et processeur sont limitées.

1. L'Importance de l'Exponentiation Modulaire en Cryptographie

1.1. Applications en cryptographie

- **RSA** : Utilise des exponentiations modulaires avec des exposants et des modules très grands pour le chiffrement et le déchiffrement (calcul des clés publiques et privées).
- **Diffie-Hellman** : Repose sur le calcul modulaire pour sécuriser l'échange de clés.
- **Algorithmes de signature numérique** : Permet de signer des messages de manière vérifiable en garantissant l'intégrité et l'authenticité.
- **Hachage et vérification :** Certaines fonctions de hachage intègrent des opérations modulaires pour garantir la distribution uniforme des valeurs.

1.2. Problématiques associées

- La grande taille des clés en cryptographie moderne (2048 bits ou plus).
- Besoin d'algorithmes optimisés pour des calculs rapides et sécurisés.

1.3 Problématiques de performance

Les calculs impliquant des bases et des exposants très grands sont gourmands en ressources. Cela peut poser des défis en termes de :

- **Temps d'exécution** : Les calculs naïfs deviennent rapidement impraticables.
- **Mémoire** : Les puissances intermédiaires peuvent dépasser les limites physiques.

Pour résoudre ces problèmes, plusieurs algorithmes d'exponentiation ont été développés, permettant une exécution rapide et sécurisée.

2. Présentation des Méthodes

Nous avons étudié et implémenté six méthodes pour le calcul de puissances, dont quatre principales et deux supplémentaires pour une exploration approfondie.

2.1 Méthode naïve :

- Principe : Multiplier *a* par lui-même *b* fois.
- Complexité : *O(b)*.
- Limites : Inefficace pour de grandes valeurs de *b*.

2.2 Méthode naïve avec modulo :

- Principe : Introduire une réduction modulo *n* après chaque multiplication pour éviter les dépassements de capacité.
- Complexité : *O(b)*.
- Avantage : Adaptée aux environnements avec des contraintes de mémoire.

2.3 Exponentiation rapide :

- Principe : Utilise la décomposition binaire de *b* pour réduire le nombre de multiplications.
- Complexité : *O(log(b))*.
- Avantage : Significativement plus rapide pour de grandes valeurs de b.

2.4 Exponentiation modulaire rapide :

- Principe : Combine l'exponentiation rapide avec des réductions modulo pour éviter les valeurs intermédiaires trop grandes.
- Complexité : *O(log(b))*.
- Pertinence : Utilisée dans des protocoles tels que **RSA** et **Diffie-Hellman**.

2.5 Multiplication successive réduite :

- Principe : Une variante optimisée de la méthode naïve avec réduction modulo après chaque multiplication.
- Complexité : *O(b)*.

2.6 Exponentiation modulaire récursive :

- Principe : Implémente l'exponentiation modulaire rapide sous forme récursive.
- Complexité : *O(log(b))*.

3. Implémentation

Les méthodes ont été implémentées en Python avec un script unifié permettant leur comparaison.

3.1 Méthode naïve :

```python
def puissance_naive(a, b):
    result = 1
    for _ in range(b):
        result *= a
    return result
```

3.2 Méthode naïve avec modulo :

```python
def puissance_naive_modulaire(a, b, n):
    result = 1
    for _ in range(b):
        result = (result * a) % n
    return result
```

3.3 Exponentiation rapide :

```python
def exponentiation_rapide(a, b):
    result = 1
    base = a
    while b > 0:
        if b % 2 == 1:  # Si l'exposant est impair
            result *= base
        base *= base
        b //= 2
    return result
```

3.4 Exponentiation modulaire rapide :

```python
def exponentiation_modulaire(a, b, n):
    result = 1
    base = a % n
    while b > 0:
        if b % 2 == 1:  # Si l'exposant est impair
            result = (result * base) % n
        base = (base * base) % n
        b //= 2
    return result
```

3.5 Multiplication successive réduite :

```python
def multiplication_successive_reduite(a, b, n):
    result = 1
    for _ in range(b):
        result = (result * a) % n
    return result

```

3.6 Exponentiation modulaire récursive :

```python
def exponentiation_modulaire_recursive(a, b, n):
    if b == 0:
        return 1
    half = exponentiation_modulaire_recursive(a, b // 2, n)
    half = (half * half) % n
    if b % 2 == 1:
        half = (half * a) % n
    return half

```

4. Test

Pour tester les méthodes, j'ai créé une fonction qui fait appel à tous ces derniers :

```python
def test_all_methods():
    a, b, n = 5, 13, 23
    print("Méthode naïve :", puissance_naive(a, b))
    print("Méthode naïve avec modulo :", puissance_naive_modulaire(a, b, n))
    print("Exponentiation rapide :", exponentiation_rapide(a, b))
    print("Exponentiation modulaire rapide :", exponentiation_modulaire(a, b, n))
    print("Multiplication successive réduite :", multiplication_successive_reduite(a, b, n))
    print("Exponentiation modulaire récursive :", exponentiation_modulaire_recursive(a, b, n))
```

Les six méthodes ont été testées avec les paramètres suivants :

- Base ***a=5***, Exposant ***b=13***, Modulo ***n=23***.

Résultats :

Méthode	Résultat calculé	Temps d'exécution estimé	Complexité
Méthode naïve	1220703125	Long	*O(b)*
Méthode naïve avec modulo	21	Long	*O(b)*
Exponentiation rapide	1220703125	Court	*O(log(b))*
Exponentiation modulaire rapide	21	Très court	*O(log(b))*
Multiplication successive réduite	21	Moyen	*O(b)*
Exponentiation modulaire récursive	21	Très court	*O(log(b))*

4. Analyse et Complexités

Analyse des résultats

1. Les méthodes naïves sont simples à comprendre et à implémenter, mais leur complexité linéaire *O(b)* les rend inadaptées pour des exposants très grands.
2. L'exponentiation rapide et ses variantes modulaires offrent des performances optimales grâce à leur complexité logarithmique *O(log(b))*.
3. Les méthodes modulaires sont particulièrement pertinentes pour les applications cryptographiques, car elles évitent les débordements.

Recommandations

- Utiliser l'exponentiation modulaire rapide dans des environnements où la sécurité et la performance sont essentielles.
- Favoriser la méthode récursive pour une implémentation élégante et compacte dans certains langages (ex. Python).

Conclusion

Ce travail a permis d’explorer diverses approches pour l’exponentiation, avec un focus particulier sur les applications cryptographiques. Les méthodes rapides et modulaires démontrent une nette supériorité en termes de performances, ce qui justifie leur utilisation dans les algorithmes modernes.

Ce TP illustre également l’importance de l’optimisation algorithmique dans le contexte de la sécurité des systèmes informatiques, où l'efficacité est souvent aussi cruciale que la sécurité elle-même.