Concours national d'informatique

Correction de la phase de sélection

Table des matières

Questionnaire 3
Question 1 : Voisins ASCII . . . 3
Question 2 : Un parcours semé d'embûches . . . 3
Question 3 : Qui est là ? . . . 4
Question 4 : 22 ! V'la le LaTeX . . . 4
Question 5 : Le secret du succès . . . 4
Question 6 : Beep Boop . . . 4
Question 7 : La Vie peut coder ! . . . 5
Question 8 : La boucle est bouclée . . . 5
Question 9 : Protocole à café . . . 5
Question 10 : Compiler l'avenir . . . 6
Question 11 : Git Kraken . . . 6

Capture The Flags 7
1.1 CTF1 - RSA101 . . . 7
1.2 CTF2 - ASR . . . 9
1.3 CTF3 - Gaussian . . . 12
1.4 CTF4 - Crackme . . . 15

Exercices 20
2.1 L'escalade de l'Yggdrasil . . . 20
2.2 Un peu d'ordre . . . 23
2.3 Ouroboros . . . 26
2.4 Bâtiments . . . 28
2.5 Coursier d'Asgard . . . 32
2.6 Coursier de Midgard . . . 37
2.7 Répartition Divine . . . 42

Questionnaire

Auteurs *Auguste Charpentier, Dorian Péron, Quentin Rataud*

Question 1: Voisins ASCII

> En encodage ASCII, quelle proposition est vraie ?
>
> — { + 1 = }
>
> — < + 1 = >
>
> — **<u>(+ 1 =)</u>**
>
> — [+ 1 =]

Dec	Hex		Dec	Hex		Dec	Hex		Dec	Hex		Dec	Hex		Dec	Hex		Dec	Hex		Dec	Hex	
0	00	NUL	16	10	DLE	32	20		48	30	0	64	40	@	80	50	P	96	60	`	112	70	p
1	01	SOH	17	11	DC1	33	21	!	49	31	1	65	41	A	81	51	Q	97	61	a	113	71	q
2	02	STX	18	12	DC2	34	22	"	50	32	2	66	42	B	82	52	R	98	62	b	114	72	r
3	03	ETX	19	13	DC3	35	23	#	51	33	3	67	43	C	83	53	S	99	63	c	115	73	s
4	04	EOT	20	14	DC4	36	24	$	52	34	4	68	44	D	84	54	T	100	64	d	116	74	t
5	05	ENQ	21	15	NAK	37	25	%	53	35	5	69	45	E	85	55	U	101	65	e	117	75	u
6	06	ACK	22	16	SYN	38	26	&	54	36	6	70	46	F	86	56	V	102	66	f	118	76	v
7	07	BEL	23	17	ETB	39	27	'	55	37	7	71	47	G	87	57	W	103	67	g	119	77	w
8	08	BS	24	18	CAN	40	28	(	56	38	8	72	48	H	88	58	X	104	68	h	120	78	x
9	09	HT	25	19	EM	41	29	)	57	39	9	73	49	I	89	59	Y	105	69	i	121	79	y
10	0A	LF	26	1A	SUB	42	2A	*	58	3A	:	74	4A	J	90	5A	Z	106	6A	j	122	7A	z
11	0B	VT	27	1B	ESC	43	2B	+	59	3B	;	75	4B	K	91	5B	[	107	6B	k	123	7B	{
12	0C	FF	28	1C	FS	44	2C	,	60	3C	<	76	4C	L	92	5C	\	108	6C	l	124	7C	|
13	0D	CR	29	1D	GS	45	2D	-	61	3D	=	77	4D	M	93	5D	]	109	6D	m	125	7D	}
14	0E	SO	30	1E	RS	46	2E	.	62	3E	>	78	4E	N	94	5E	^	110	6E	n	126	7E	~
15	0F	SI	31	1F	US	47	2F	/	63	3F	?	79	4F	O	95	5F	_	111	6F	o	127	7F	DEL

TABLE 1 – Tableau d'encodage ASCII

En jetant un œil au tableau d'encodage ASCII, on remarque que :

- le code ASCII de la parenthèse gauche est `40`, et le code ASCII de la parenthèse droite est `41` ;
- le code ASCII du chevron gauche est `60`, et le code ASCII du chevron droit est `62` ;
- le code ASCII du crochet gauche est `91`, et le code ASCII du crochet droit est `93` ;
- le code ASCII de l'accolade gauche est `123`, et le code ASCII de l'accolade droite est `125`.

La proposition correcte est donc celle concernant les parenthèses.

Question 2: Un parcours semé d'embûches

> En Python 2.7, quel est le résultat de :
>
> ```
> dico = {1: None}
> for i in dico:
>     print i
>     del dico[i]
>     dico[i + 1] = None
> ```
>
> — **<u>Le code affiche les entiers de 1 à 7 et s'arrête sans erreur</u>** ;
>
> — Le code affiche 1 et s'arrête sans erreur ;

> — Une boucle infinie, qui n'affiche rien ;
> — Une boucle infinie, qui affiche les entiers d'à partir de 1 ;
> — Une erreur, car les clés du dictionnaire changent pendant son parcours.

Exécuter le code en Python 2.7 affichera

```
1
2
3
4
5
6
7
```

et s'arrêtera. La bonne réponse est donc la première. Selon la documentation de Python [1] :

> *Iterating views while adding or deleting entries in the dictionary may raise a RuntimeError or fail to iterate over all entries.*

Cela explique le comportement inattendu du programme. Le même code exécuté en Python 3.6 s'arrêtera après avoir affiché 4, et depuis Python 3.7, le code renverra une erreur.

Question 3: Qui est là ?

> Considérez `let liste = [4, 3, 5];` en JavaScript. Laquelle de ces propositions retourne `true` ?
> — `-1 in liste`;
> — **`0 in liste`** ;
> — `"4" in liste`;
> — `3 in liste`.

En JavaScript, le mot clé « `in` » vérifie si l'élément est un indice valide du tableau. `liste` ayant trois éléments, alors les indices valides sont `0`, `1` et `2`.

Question 4: 22 ! V'la le LaTeX

> Laquelle de ces classes permet d'avoir la plus grande police d'écriture dans un document LaTeX ?
> — `\documentclass[11pt]{article}`;
> — **`\documentclass[12pt]{extarticle}`** ;
> — `\documentclass[13pt]{article}`;
> — `\documentclass[16pt]{extarticle}`.

La classe `article` ne supporte que les tailles de polices 10pt, 11pt et 12pt. La classe `extarticle` supporte plus de tailles, dont les tailles 8pt, 9pt, 14pt, 17pt et 20pt.

Lorsqu'une taille de police non supportée est indiquée, la taille obtenue est celle par défaut, 10pt.

Question 5: Le secret du succès

> Selon Tom Lehrer, quel est le secret du succès en mathématiques ?
> — Faire des exercices régulièrement ;
> — Effectuer des recherches par soi-même ;
> — Apprendre toutes les formules par cœur ;
> — **Plagier** .

Selon sa chanson "Lobachevsky" :

> *I am never forget the day I first meet the great Lobachevsky. In one word he told me secret of success in mathematics : Plagiarize !*

La référence est d'ailleurs citée par Wolfram Alpha [2].

1. `https://docs.python.org/3/library/stdtypes.html#dictionary-view-objects`
2. `https://www.wolframalpha.com/input?i=what+is+the+secret+of+success+in+mathematics%3F`

Question 6: Beep Boop

Selon un test de Turing inversé, la suite suivante a-t-elle été générée par un humain ?
`10101011010110101010110101101011101010101011101101010101110101101010101101011101`
`10101101011010101101`

- **Oui** ;
- Non.

Lors d'un test de Turing, une machine doit imiter le comportement humain aux yeux d'un juge humain. Ici, il s'agit d'un test de Turing inversé, c'est à dire qu'un humain essaye de se faire passer pour une machine en tentant de générer une suite “aléatoire” de zéros et de uns. Cependant, la plupart des tests révéleront que la suite a très probablement été écrite par un homme et pas générée aléatoirement. Il est d'ailleurs facile de vérifier que la suite ne contient jamais deux zéros d'affilés, ce qui est censé arriver une fois sur quatre en moyenne.

Voici d'ailleurs une plateforme permettant d'effectuer par vous-même un test de Turing inversé [3].

Question 7: La Vie peut coder !

Que se passe-t-il lorsque le logo de Girls Can Code ! est simulé dans le jeu de la vie ? (Le logo doit être représenté avec 21 cellules vivantes)

- Le logo créé une infinité de feux clignotants ;
- **Le logo génère trois planeurs** ;
- Le logo atteint une position stable après 85 itérations ;
- Toutes les cellules meurent au bout de 85 itérations.

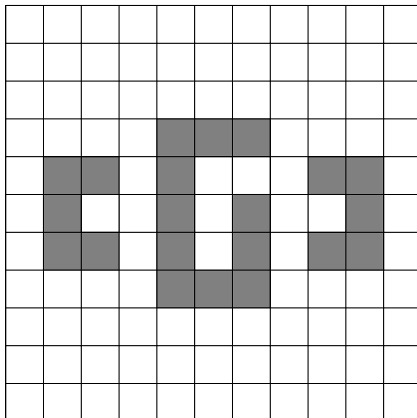

Figure 1 – Situation initiale du plateau

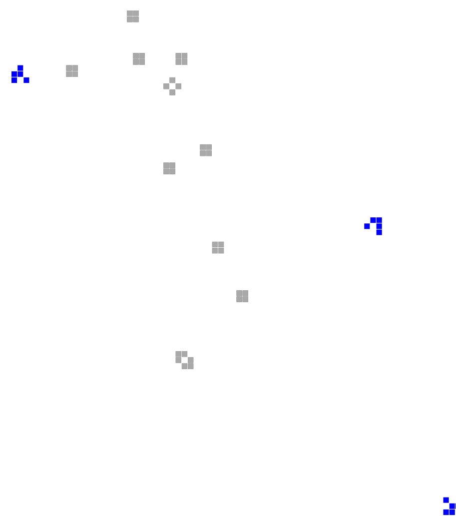

Figure 2 – Plateau après 301 itérations

Au bout de 301 itérations, on retrouve trois planeurs, en bleu dans la figure 2. Les autres cellules encore vivantes sont des structures stables, en gris.

Question 8: La boucle est bouclée

Quel symbole de la mythologie nordique est le nom d'une quine ?

- **Ouroboros** ;
- Yggdrasil ;
- Wyrd ;
- Vegvisir.

L'Uroboros Quine Relay [4] est un code Ruby qui affiche un code en Scala, ce code en Scala affiche un code en Scheme, puis qui, au bout de 128 langages triés par ordre alphabétique, revient sur le code Ruby initial.

3. `https://calmcode.io/blog/inverse-turing-test.html`
4. `https://github.com/mame/quine-relay`

Question 9: Protocole à café

> Quel code de retour HTTP devrait retourner une cafetière/théière combinée qui serait temporairement à court de café ?
>
> — 405 ;
>
> — 418 ;
>
> — **503** ;
>
> — 504.

Selon la RFC du 1er avril 1998, le code de retour 418 doit être retourné lorsque l'on essaye de faire du café avec une théière.

Cependant, selon la documentation développeur Mozilla [5] :

A combined coffee/tea pot that is temporarily out of coffee should instead return 503

Une cafetière/théière combinée temporairement à court de café devrait donc retourner le code de retour 503, *Service temporairement indisponible.*

Question 10: Compiler l'avenir

> Quel est le comportement de `make` lorsque l'on essaye de compiler un fichier modifié dans le futur ?
>
> — `make` affichera un message d'erreur et refusera de compiler le fichier ;
>
> — `make` ne compilera pas le fichier car il pense que le fichier est déjà à jour ;
>
> — **`make` affichera un avertissement pour prévenir que le fichier a été modifié dans le futur** ;
>
> — `make` affiche une référence à Retour vers le Futur.

On peut utiliser la commande `touch` pour éditer la date de dernière modification d'un fichier, et vérifier ce comportement :

```
  ~ touch -d "now +30 minutes" test.c
  ~ make test
make: Warning: File 'test.c' has modification time 1797 s in the future
cc     test.c   -o test
make: warning:  Clock skew detected.  Your build may be incomplete.
```

La compilation est bien effectuée et s'effectue sans erreur. Un simple avertissement s'affiche.

Question 11: Git Kraken

> Quel est le hash complet du commit décrit par Linus Torvalds comme étant un « Cthulhu merge » :
>
> — `2cde51fbd0f310c8a2c5f977e665c0ac3945b46b` ;
>
> — `2cde51fbd0f310c8a2c5f977e665c0ac3945b46c` ;
>
> — **`2cde51fbd0f310c8a2c5f977e665c0ac3945b46d`** ;
>
> — `2cde51fbd0f310c8a2c5f977e665c0ac3945b46e`.

Le message dont il est question est celui-ci [6]. On peut retrouver tous les détails du commit sur git.kernel.org [7].

5. https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/418
6. https://marc.info/?l=linux-kernel&m=139033182525831
7. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=2cde51fbd0f310c8a2c5f977e665c0ac3945b46d

Capture The Flags

Auteurs *Quentin Rataud, Odric Roux-Paris*

Testeurs *Matteo Ahouanto, Ronan Toullec-streicher*

1.1 CTF1 - RSA101

1.1.1 Introduction

Jøsëf Marchand a tenté de créer sa propre implémentation de RSA en Python, que vous retrouverez dans le fichier « rsa.py ». Seul le contenu de la chaîne `message` a été modifié et remplacé par des points d'interrogations.

Il a alors lancé son code, et le résultat de l'exécution de son code est donné dans le fichier « sortie.txt ». Cependant, Jøsëf a fait une erreur dans son code ! Trouvez l'erreur, et exploitez l'erreur pour retrouver le contenu original de la chaîne `message`.

1.1.2 Le code de Jøsëf Marchand

```python
from sympy import randprime

p, q = [randprime(0, 1 << 1024)] * 2
n = p * q

message = b"PROLOGIN{??????????????????????????????????????????????????????}"
m = int.from_bytes(message, 'little')
e = 0x10001
c = pow(m, e, n)

print(f"{n = }")
print(f"{e = }")
print(f"{c = }")
```

Le code génère deux nombres premiers secrets p et q, dont le produit est appelé N. Jøsëf encode ensuite le message secret en un entier m. Enfin, Jøsëf effectue un chiffrement RSA, en calculant $c = m^e \mod N$, avec $e = 65\,537$.

Les paramètres publics N, e, ainsi que le message chiffré c, sont affichés et disponibles dans le fichier `sortie.txt` :

```
n = 320685881677373631205650725707972737682824603785663885198446773206933391712829421688088
↪   233545407878834574212788697402700088844415191793915202426690049291793313975615275090075
↪   509389301227036817798496516808507908791027515939758546552650304297739563045221874084308
↪   508579571913554685565621837555847156150454977884333542262290916348663257254120786373195
↪   808201535164368899960156169300815437904821664406241503676361776224148091314282501998408
↪   766827352388903757454361045814074502708611629364074962711647634294705143426629113462244
↪   178207956860381053200086479481468428105648252751379628511809451514353293866315922224962
↪   19315041
e = 65537
```

```
c = 2393252881162915216011949508882236599538477008479334420941073633167173424635993064157 79
↪   5321062524786282776606047442208050654598416010364607737200883579481559302972032070904 63
```

1.2 CTF2 - ASR

1.2.1 Introduction

Maintenant que vous avez compris le fonctionnement de RSA, vous trouverez dans le fichier « asr.py » un algorithme que Jøsëf Marchand a conçu pour chiffrer un message en utilisant des permutations. Encore une fois, la seule modification effectuée est le contenu de la chaîne message, qui a été remplacé par des points d'interrogations.

Étant donné le résultat de l'exécution du code, dans le fichier « sortie.txt », parviendrez-vous à retrouver le contenu original de la chaîne message ?

1.2.2 Le code de Jøsëf Marchand

```python
from more_itertools import nth_permutation

n = 69
e = 71

message = b"PROLOGIN{?????????????????????????????}"
m = int.from_bytes(message, 'little')
permutation = nth_permutation(range(n), n, m)

c = list(range(n))
for _ in range(e):
    for i in range(n):
        c[i] = permutation[c[i]]

print(f"{n = }")
print(f"{e = }")
print(f"{c = }")
```

Le code encode le message secret en une permutation de 69 éléments, à l'aide de la fonction `nth_permutation`. Le message est ensuite chiffré en une nouvelle permutation, c, qui est obtenue en appliquant $e = 71$ fois la permutation encodée sur la permutation identité.

Enfin, les paramètres publics et la permutation chiffrée sont affichés dans la sortie.

```
n = 69
e = 71
c = [0, 45, 40, 59, 32, 55, 28, 39, 47, 41, 33, 57, 65, 56, 3, 43, 22, 26, 46, 51, 66, 8, 42,
↪ 31, 4, 18, 58, 6, 63, 62, 17, 37, 54, 67, 29, 60, 50, 13, 9, 12, 24, 61, 68, 19, 27, 25,
↪ 7, 20, 2, 44, 52, 21, 5, 38, 64, 11, 34, 14, 30, 23, 53, 1, 35, 16, 49, 10, 48, 15, 36]
```

1.2.3 Vulnérabilité

On peut faire un rapprochement entre cet algorithme et le chiffrement RSA. En effet, considérez c comme étant un élément de S_{69}, le groupe des permutations de 69 éléments.

En clair, reformulons légèrement l'algorithme en utilisant une classe pour représenter les permutations, afin de mieux montrer en quoi il s'agit ici d'un chiffrement RSA :

```python
from typing import List
from more_itertools import nth_permutation

n = 69
e = 71

# On considère identite, la permutation qui représente "1".
# La permutation identité est simplement [ 0, 1, 2, ..., 68 ]
identite: "Permutation"

class Permutation:
    def __init__(self, tableau: List[int]):
        self.tableau = tableau

    def __mul__(self, other: "Permutation") -> "Permutation":
        """
        On considère que "multiplier" deux permutations revient à appliquer
        la permutation
        """
        tableau = self.tableau.copy()
        for i in range(n):
            tableau[i] = other.tableau[self.tableau[i]]
        return Permutation(tableau)

    def __pow__(self, n: int) -> "Permutation":
        """
        Pour calculer self puissance n,
        on multiplie "1" (identite) par self, n fois.
        """
        permutation = identite
        for _ in range(n):
            permutation *= self
        return permutation

    def __repr__(self):
        return str(self.tableau)

identite = Permutation(list(range(n)))

message = b"PROLOGIN{??????????????????????????????}"
m_entier = int.from_bytes(message, 'little')
m = Permutation(nth_permutation(range(n), n, m_entier))

# Chiffrement RSA ?
c = pow(m, e)

print(f"{n = }")
print(f"{e = }")
print(f"{c = }")
```

Alors, on peut se demander s'il est possible d'effectuer un déchiffrement RSA pour retrouver m depuis c. L'algorithme de déchiffrement RSA nécessite simplement de connaître $|S_{69}|$, le nombre d'éléments qui existe dans le groupe. Il existe 69! permutations de 69 éléments, alors nous pouvons utiliser $x = 69!$, et calculer $d = e^{-1} \mod x$. Enfin, calculer c^d devrait redonner la permutation initiale, m.

1.2.4 Solution

```python
from typing import List
from more_itertools import permutation_index

n = int(input()[4:])
e = int(input()[4:])
c_tableau = eval(input()[4:])

# inclure la classe Permutation
c = Permutation(c_tableau)

# x = 69!
x = 1
for i in range(1, n+1):
    x *= i
d = pow(e, -1, x)

# déchiffrement RSA
m = pow(c, d)

message = permutation_index(m.tableau, range(n))
print(int.to_bytes(message, n, 'little').replace(b'\x00', b''))
```

Ce script devrait théoriquement fonctionner, cependant il ne terminera pas son exécution. Cela est dû à la valeur de d, qui est ici égal à `1687122770418294619165180805871704047388582836978369839092304628820846420111245819 67323943661971831`. Dans cet état, vous ne verrez probablement pas le bout de la boucle `for` dans la méthode `pow`.

Heureusement, il existe un algorithme qui permet de calculer une puissance beaucoup plus rapidement pour n'importe quelle opération associative. Il s'agit de l'algorithme d'exponentiation rapide, qui trouve le résultat de la puissance en effectuant $O(\log(b))$ multiplications :

$$a^b = \begin{cases} 1 & \text{si } b = 0, \\ \left(a^{b/2}\right)^2 & \text{si } b \text{ est pair}, \\ a \cdot a^{b-1} & \text{sinon} \end{cases}$$

On peut donc améliorer la méthode `pow` comme ceci :

```python
def __pow__(self, n: int) -> "Permutation":
    """
    Pour calculer self puissance n, on utilise l'algorithme d'exponentiation
    rapide
    """
    if n == 0:
        return identite
    elif n % 2 == 0:
        racine = pow(self, n // 2)
        return racine * racine
    else:
        return self * pow(self, n - 1)
```

Ainsi modifié, le script va afficher le flag sur la sortie standard : `PROLOGIN{Exponentiation Rapide, Et X=N!}`

Ici, il se trouve que la permutation originale appartenait à un sous-groupe de S_{69} d'ordre 64. Il était donc possible de retrouver m en utilisant n'importe quel multiple de 64 à la place de de 69!, et donc il était aussi possible de trouver la solution par force brute en appliquant la permutation à elle même, jusqu'à voir `PROLOGIN` apparaître au début du message.

1.3 CTF3 - Gaussian

1.3.1 Introduction

Jøsëf Marchand, fier d'avoir compris le chiffrement RSA, essaye alors d'implémenter RSA sur les nombres complexes. Il code alors un script Python, « gaussian.py », qui vous est donné, à la seule différence de la chaîne message dont le contenu a été remplacé par des points d'interrogations.

Cependant, si vous avez bien compris le fonctionnement de RSA, alors vous trouverez peut-être qu'il est possible de retrouver le contenu de chaîne depuis le résultat de son code, donné dans le fichier « sortie.txt ».

1.3.2 Le code de Jøsëf Marchand

```python
from sympy import isprime
from random import randint

KEY_SIZE = 512
MESSAGE_SIZE = KEY_SIZE // 8

class GaussianInteger:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(A, B):
        return GaussianInteger(A.x + B.x, A.y + B.y)

    def __sub__(A, B):
        return GaussianInteger(A.x - B.x, A.y - B.y)

    def __mul__(A, B):
        x = A.x * B.x - A.y * B.y
        y = A.x * B.y + A.y * B.x
        return GaussianInteger(x, y)

    def __floordiv__(A, B):
        denominator = B.x * B.x + B.y * B.y
        round_nearest = denominator // 2
        x = (A.x * B.x + A.y * B.y + round_nearest) // denominator
        y = (A.y * B.x - A.x * B.y + round_nearest) // denominator
        return GaussianInteger(x, y)

    def __mod__(A, B):
        candidate = A - (A // B) * B
        return candidate

    def __pow__(base, power, mod):
        result = GaussianInteger(1, 0)
        incr = base
        while power > 0:
            if power % 2 == 1:
                result *= incr
                result %= mod
            power //= 2
            incr *= incr
            incr %= mod
        return result

    @classmethod
    def from_bytes(cls, message):
        length = len(message) // 2
        x = int.from_bytes(message[:length], 'little', signed=True)
```

```python
        y = int.from_bytes(message[length:], 'little', signed=True)
        return cls(x, y)

    def to_bytes(self):
        left = self.x.to_bytes(MESSAGE_SIZE, 'little', signed=True)
        right = self.y.to_bytes(MESSAGE_SIZE, 'little', signed=True)
        return left + right


def generate_prime(bits):
    bits -= 2
    while True:
        x = randint(1, 1 << bits)
        y = randint(1, 1 << bits)
        if isprime(x*x + y*y):
            return GaussianInteger(x, y)


if __name__ == "__main__":
    P = generate_prime(KEY_SIZE)
    print("P =", P.to_bytes().hex())

    E = 0x10001
    print("E =", E)

    message = b"PROLOGIN{??????????????????????????????????????????????????????????????????}"
    assert len(message) < 2 * MESSAGE_SIZE
    M = GaussianInteger.from_bytes(message)

    C = pow(M, E, P)
    print("C =", C.to_bytes().hex())
```

Le code utilise l'algorithme de chiffrement RSA en utilisant des entiers Gaussiens, c'est à dire des nombres complexes ayant des entiers en partie réelle et imaginaire.

Plus en détails, le code commence par générer un premier Gaussien[8], c'est à dire un nombre complexe qui n'est pas le produit de deux autres entiers Gaussiens (sans compter les unités, $1, -1, i, -i$). Ce nombre premier Gaussien est appelé P, et nous travaillons par la suite sur le groupe des entiers Gaussiens modulo P, $\mathbb{Z}[i]/P\mathbb{Z}[i]$.

Le message est ensuite encodé en un entier Gaussien M, en utilisant les bits de poids faibles en partie réelle, et les bits de poids forts en partie complexe. Enfin, M est chiffré dans C en utilisant l'algorithme RSA dans $\mathbb{Z}[i]/P\mathbb{Z}[i]$, par $C = M^E \mod P$.

Enfin, les paramètres publics, l'exposant E, le modulo P et le message chiffré C, sont affichés sur la sortie :

```
P = f8ab7e8e5bf480f5bdbca13b6d71cede8d7f1f97964a71700080b9789f980983af142dc9f7ed36e13b3
↪   cf94f54f38d1c2652f252c343ba23c0545a9e7063281783ebb8557251ea0e4c55c9d98c2e63186204b
↪   6d2e24dec8c845acf6e7e0e0778265f05bf1b2db063769aab236dd87a1488d314f746ecc73b75ab442
↪   f98338b13
E = 65537
C = eb16ccce3978839b6143e601d8c7ef13ab737f0f013536585791c93fc9d463979822286b361d4d2280
↪   90f83d92e193f780663807f997bc03307589003d6b49fdbb4c67a83b22b1cf59a0c460c7b0ea321a27
↪   acdea9d40d5bd4e8fd6578542cb4e50c0f5b1eb3fa334c7379a064b95b5b3f51a0fbd921d8617d6d2e
↪   4d745f84f3
```

8. https://en.wikipedia.org/wiki/Gaussian_integer#Gaussian_primes

1.3.3 Vulnérabilité

Un chiffrement RSA peut être inversé si l'ordre du groupe est connu. Dans ce contexte, on cherche simplement à calculer l'ordre du groupe multiplicatif $G = \mathbb{Z}[i]/P\mathbb{Z}[i]$. Cet ordre peut en fait facilement être calculé, car l'ordre de G correspond à un de moins que la norme de P, qui est la somme des carrés de sa partie réelle et imaginaire. [9]

1.3.4 Solution

```python
from gaussian import GaussianInteger

P = bytes.fromhex(input()[4:])
E = int(input()[4:])
C = bytes.fromhex(input()[4:])

P = GaussianInteger.from_bytes(P)
C = GaussianInteger.from_bytes(C)

x = P.x * P.x + P.y * P.y - 1
d = pow(E, -1, x)
M2 = pow(C, d, P)
print(M2.to_bytes().replace(b'\x00', b''))
# b"PROLOGIN{Les Entiers Gaussiens Modulo P Forment Un Corps Fini D'Ordre |P|-1}"
```

9. https://en.wikipedia.org/wiki/Gaussian_integer#Describing_residue_classes

1.4 CTF4 - Crackme

1.4.1 Introduction

> *Jøsëf Marchand a trouvé par terre un CD d'installation pour NixOS. À l'intérieur se trouve un mystérieux programme lui demandant un mot de passe. Il a l'intime conviction que le mot de passe est caché à l'intérieur et qu'il lui permettra d'avoir gloire, bonheur et fortune pendant mille ans.*

Le défi posé par ce crackme consiste à découvrir la clé de série dissimulée à l'intérieur du binaire `crackme`. Le participant est chargé d'entrer une clé de série, et le binaire informe si la clé fournie est la bonne ou non.

Pour relever ce défi, l'utilisation d'un terminal sous Linux est requise (une machine virtuelle peut également être utilisée). Le challenge s'engage en lançant la commande suivante :

```
[odric@prologin]$ ./crackme “Mon super sérial”
```

En cas de clé incorrecte, le binaire affiche le message : `Mauvais sérial`.

Le binaire en question est un exécutable Linux 64 bits.

1.4.2 Analyse du binaire

Pour démarrer l'analyse, nous allons ouvrir le binaire avec Ghidra et décompiler la fonction principale `main`. Voici un extrait du code résultant :

```c
int main(int argc,char **argv)
{
  if (argc == 2) {
    puts(&DAT_00102018);
    return 0;
  }
  __printf_chk(1,"usage: %s <serial>\n",*argv);
                    /* WARNING: Subroutine does not return */
  exit(1);
}
```

Cette fonction est relativement simple :

- si deux arguments ne sont pas présents, le message `Mauvais Sérial` est affiché ;
- sinon, le mode d'utilisation est indiqué.

Cependant, une observation révèle une particularité intrigante : quelle que soit la clé entrée, le sérial est toujours considéré comme incorrect. Cela suggère la présence d'un élément supplémentaire.

Une inspection plus approfondie révèle que le binaire a été compilé avec une version récente du compilateur `GCC`. Parmi les fonctions importées, on retrouve plusieurs fonctions de la bibliothèque C standard (Glibc), notamment `exit`, `mprotect`, `putc`, `puts`, et `strlen`. Étant donné la taille du binaire et le linking dynamique, ces fonctions sont les seules à être utilisées.

Notre intérêt se porte alors sur les endroits où elles sont appelées. La fonction `strlen` est référencée plusieurs fois dans le code, notamment dans une fonction nommé `_INIT_1`. Cette dernière est particulière car son addresse est référencée dans la section `.init_array`. Cette section permet de déclarer les fonctions qui s'exécuteront avant le `main`.

Ces fonctions sont connues sous le nom de constructeurs. La syntaxe pour définir une fonction en tant que constructeur est la suivante :

```c
__attribute__((constructor)) void ma_super_fonction(int argc, char **argv);
```

Une particularité des constructeurs est qu'ils reçoivent également les arguments du main. Ainsi, dans notre contexte, le binaire peut extraire le sérial du deuxième argument et déterminer sa validité.

Analysons donc cette fonction :

```c
void _INIT_1(int param_1,long param_2)
{
  char *__s;
  bool bVar1;
  size_t sVar2;
  long lVar3;
  byte *pbVar4;
  byte *pbVar5;
  long in_FS_OFFSET;
  undefined4 local_3c;
  byte local_38 [40];
  long local_10;

  local_10 = *(long *)(in_FS_OFFSET + 0x28);
  if (param_1 == 2) {
    local_3c = 0xbebafeca;
    pbVar4 = &DAT_0010202b;
    pbVar5 = local_38;
    for (lVar3 = 0x1a; lVar3 != 0; lVar3 = lVar3 + -1) {
      *pbVar5 = *pbVar4;
      pbVar4 = pbVar4 + 1;
      pbVar5 = pbVar5 + 1;
    }
    __s = *(char **)(param_2 + 8);
    sVar2 = strlen(__s);
    if (sVar2 == 0x1a) {
      lVar3 = 0;
      bVar1 = true;
      do {
        if ((byte)(local_38[(ulong)((uint)lVar3 & 3) - 4] ^ __s[lVar3]) != local_38[lVar3]) {
          bVar1 = false;
        }
        lVar3 = lVar3 + 1;
      } while (lVar3 != 0x1a);
      if (bVar1) {
        mprotect(_DT_INIT,0x4000,7);
                    /* WARNING: Read-only address (ram,0x0010127a) is written */
        uRam000000000010127a = 0xe8;
                    /* WARNING: Read-only address (ram,0x0010127b) is written */
        uRam000000000010127b = 0xfffffffffffff8a;
      }
    }
  }
  if (local_10 == *(long *)(in_FS_OFFSET + 0x28)) {
    return;
  }
                    /* WARNING: Subroutine does not return */
  __stack_chk_fail();
}
```

- La variable `__s` correspond au sérial, étant le deuxième argument du binaire.
- La variable `sVar2` représente le nombre de caractères dans le sérial, et il doit être égal à `0x19` (25).

Lorsque la condition de vérification du nombre de caractères est satisfaite, une boucle parcourt chaque caractère.

- La variable `bVar1` est initialement vraie et mis à faux quand la condition dans la boucle est fausse. De plus, quand elle est vraie, des choses bizarres se passent.
- La variable `lVar3` sert d'index dans la boucle.

Nous concluons que pour que le sérial soit correct, chaque caractère du sérial entré par le candidat doit être chiffré avec l'opération `XOR` par une clé (représentée par la variable `local_3c`). Ensuite, caractère par caractère, il est comparé au bon sérial chiffré, situé à l'adresse `0x0010202b`.

Après avoir effectué un retypage et un renommage du code avec Ghidra, nous pouvons obtenir :

```c
void _INIT_1(int argc,char **argv)
{
  size_t len_serial;
  long i;
  byte *encrypted_serial;
  byte *encrypted_serial2;
  long in_FS_OFFSET;
  char key [4];
  byte local_38 [40];
  long local_10;
  bool result;
  char *serial;

  local_10 = *(long *)(in_FS_OFFSET + 0x28);
  if (argc == 2) {
    key[0] = -0x36;
    key[1] = -2;
    key[2] = -0x46;
    key[3] = -0x42;
    encrypted_serial = &DAT_0010202b;
    encrypted_serial2 = local_38;
    for (i = 0x1a; i != 0; i = i + -1) {
      *encrypted_serial2 = *encrypted_serial;
      encrypted_serial = encrypted_serial + 1;
      encrypted_serial2 = encrypted_serial2 + 1;
    }
    serial = argv[1];
    len_serial = strlen(serial);
    if (len_serial == 0x1a) {
      i = 0;
      result = true;
      do {
        if ((byte)(key[(uint)i & 3] ^ serial[i]) != local_38[i]) {
          result = false;
        }
        i = i + 1;
      } while (i != 0x1a);
      if (result) {
        mprotect(_DT_INIT,0x4000,7);
                    /* WARNING: Read-only address (ram,0x0010127a) is written */
        uRam000000000010127a = 0xe8;
                    /* WARNING: Read-only address (ram,0x0010127b) is written */
        uRam000000000010127b = 0xfffffffffffff8a;
      }
    }
  }
  if (local_10 == *(long *)(in_FS_OFFSET + 0x28)) {
    return;
  }
                    /* WARNING: Subroutine does not return */
  __stack_chk_fail();
}
```

Il est intéressant de noter que la clé possède 4 caractères. Mais dans la boucle, l'index de la clé est calculé comme ceci : `index_clé & 3`. Cette opération est une optimisation du compilateur de l'opérateur modulo. Ainsi quand l'index dépasse 3 (car on part de 0), il est remis à 0.

1.4.3 Déchiffrement du Sérial

Voici un code Python déchiffrant le sérial :

```
key = [0xca, 0xfe, 0xba, 0xbe]
encrypted_serial = [0x9a, 0xac, 0xf5, 0xf2, 0x85,
                    0xb9, 0xf3, 0xf0, 0xb1, 0xce,
                    0xd2, 0xf0, 0xfa, 0xa7, 0x8a,
                    0xcb, 0x8c, 0x97, 0xf4, 0xda,
                    0x87, 0xbb, 0x9b, 0x9f, 0xeb,
                    0x83, 0x00]
serial = []
for i in range(len(encrypted_serial)):
     serial.append(chr(encrypted_serial[i] ^ key[i % 4]))

print("".join(serial))
# PROLOGIN{0hN0Y0uFiNdME!!!}
```

Félicitations à ceux qui ont trouvé le flag !

1.4.4 Pour aller plus loin

Observons maintenant ce qui se déroule après la validation réussie du sérial. Le syscall `mprotect` entre en action, permettant la modification des permissions de la mémoire. Dans le binaire, `mprotect` ajuste les droits à l'adresse `_DT_INIT` (`0x00101000`) sur une plage de `0x4000` octets (soit 16384 octets) avec la permission 7, correspondant à `PROT_READ | PROT_WRITE | PROT_EXEC`, ce qui équivaut à enlever toutes les protections. Cette modification permet à la mémoire du binaire d'être lue, écrite et exécutée.

Les lignes suivantes insèrent le byte `0xe8` à l'adresse `0x10127a` et `0xffffffffffffff8a` à l'adresse `0x10127b`, toutes deux localisées dans la fonction `main`. Le byte `0xe8` correspond à une instruction near call [10], et `0xffffffffffffff8a` représente le décalage entre l'adresse où se trouve l'instruction `call` et celle de la fonction qui sera appelée. Ce décalage, négatif, est équivalent à -118.

Le programme insère dans la fonction `main` un appel qui redirige l'exécution vers une fonction située 118 octets en dessous. Pour calculer l'adresse où se situe la fonction, il faut juste faire :

$$\text{l'adresse où est inséré l'instruction} + 5 - \text{le décalage}$$

Le $+5$ est très important car il correspond aux nombres d'octets de l'instruction du `call` (`E8 00 00 00 00`). On a donc :

$$0x10127b + 5 - 118 = 0x10120a.$$

A cette adresse se trouve cette fonction :

```
void FUN_00101209(void)
{
  undefined8 *puVar1;
  long in_FS_OFFSET;
  undefined8 local_2c;
  undefined4 local_24;
  undefined8 local_20 [2];

  local_20[0] = *(undefined8 *)(in_FS_OFFSET + 0x28);
  local_2c = 0x3a213600733d3c11;
  local_24 = 0x59723f32;
  puVar1 = &local_2c;
  do {
    putc((int)(char)(*(byte *)puVar1 ^ 0x53),stdout);
    puVar1 = (undefined8 *)((long)puVar1 + 1);
  } while (puVar1 != local_20);
                    /* WARNING: Subroutine does not return */
  exit(0);
}
```

10. https://c9x.me/x86/html/file_module_x86_id_26.html)

On se rend vite compte qu'il déchiffre un message et l'affiche. C'est la fonction qui affiche le succès !
J'espère que ce petit challenge vous a plu. À l'année prochaine pour de nouveaux `Crackme` !
- *Cirdo*

Exercices

Auteurs *Oscar Chevalier, Augustin Glorian, Julie Fiadino, Célian Raimbault, Quentin Rataud*

Testeurs *Amélie Bertin, Raphaël Gonon, Arthur Léonard*

2.1 L'escalade de l'Yggdrasil

2.1.1 Énoncé

Jøsëf Marchand, désireux d'aider les dieux, croise au pied de l'Yggdrasil, Höder. Le dieu étant aveugle, celui-ci lui demande de l'aide pour rejoindre le haut de l'arbre afin de retrouver sa famille et de participer à la photo familiale.

Höder se déplace en effectuant des sauts de branche en branche.

Jøsëf connait les N différences de hauteur entre les branches consécutives de l'Yggdrasil.

Aidez Jøsëf à calculer le plus grand saut qu'Höder devra faire entre la première branche et la branche la plus haute de l'arbre.

On ne considère dans ce calcul que les sauts qui font prendre de la hauteur, c'est à dire ceux qui décrivent une différence de hauteur positive. Si la première branche, sur laquelle Höder se situe, est déjà la branche la plus haute, alors affichez 0.

Si plusieurs branches se situent à la hauteur maximale, alors la photo de famille se trouvera sur la première d'entre elles.

2.1.2 Solution

La réponse à la question ne dépend pas de la hauteur de la première branche. En effet, considérons simplement pour chaque branche i sa hauteur h_i par rapport à la première branche. h_i correspond donc à la différence de hauteur entre la branche i et la première branche.

On commence donc avec $h_i = 0$, et, connaissant la différence Δ_i entre la branche i et $i + 1$ dans l'entrée, on peut calculer $h_{i+1} = h_i + \Delta_i$ en parcourant simplement le tableau de l'entrée dans l'ordre.

Ensuite, l'énoncé demande de considérer la branche la plus haute, c'est à dire, l'indice j pour lequel la valeur de h_j est maximale. On peut trouver cette valeur en enregistrant toutes les valeurs de h_i puis en cherchant la plus grande valeur, ou directement en une itération en enregistrant simplement la plus grande valeur de h_i obtenue jusqu'à présent (ainsi que la valeur de i associée) lors de la première itération.

Enfin, l'objectif est de trouver la différence de hauteur la plus grande se situant avant cette branche. En clair, il faut reparcourir la liste des Δ donnée en entrée pour trouver la plus grande valeur de Δ_i pour $i < j$.

2.1.3 Algorithme

Algorithme 1 : L'escalade de l'Yggdrasil

Entrées : Le nombre N de branches de l'arbre, moins 1,
le tableau Δ des différences de hauteur entre les branches consécutives
Résultat : Le plus grand saut que devra effectuer Höder pour atteindre la branche la plus haute de l'Yggdrasil

$(i, h_i) \longleftarrow (0, 0)$;
$(j, h_j) \longleftarrow (0, 0)$;
tant que $i < N$ **faire**
    $(i, h_i) \longleftarrow (i + 1, h_i + \Delta_i)$;
    si $h_i > h_j$ **alors**
        $(j, h_j) \longleftarrow (i, h_i)$;
    fin
fin
`plus_grand_saut` $\longleftarrow 0$;
pour $i \leftarrow 0$ **à** j **faire**
    si $\Delta_i >$ `plus_grand_saut` **alors**
        `plus_grand_saut` $\longleftarrow \Delta_i$;
    fin
fin
retourner `plus_grand_saut`;

2.1.4 Réalisation

Python

```python
def le_plus_grand_saut(n: int, differences: List[int]) -> None:
    hauteur_max = 0
    hauteur_max_index = 0
    hauteur_actuelle = 0
    for i, delta in enumerate(differences):
        hauteur_actuelle += delta
        if hauteur_actuelle > hauteur_max:
            hauteur_max = hauteur_actuelle
            hauteur_max_index = i

    plus_grand_saut = 0
    for i in range(hauteur_max_index + 1):
        delta = differences[i]
        plus_grand_saut = max(plus_grand_saut, delta)

    print(plus_grand_saut)
```

C++

```
void le_plus_grand_saut(int n, const std::vector<int>& hauteurs) {
    int max_hauteur = 0;
    int max_hauteur_index = 0;
    int hauteur = 0;
    for (int i = 1; i <= n; i++) {
        int delta = hauteurs[i-1];
        hauteur += delta;
        if (hauteur > max_hauteur) {
            max_hauteur = hauteur;
            max_hauteur_index = i;
        }
    }

    int plus_grand_saut = 0;
    for (int i = 0; i < max_hauteur_index; i++) {
        plus_grand_saut = std::max(plus_grand_saut, hauteurs[i]);
    }

    std::cout << plus_grand_saut << std::endl;
}
```

2.2 Un peu d'ordre

2.2.1 Énoncé

Höder est arrivé à temps pour réaliser la photo familiale. Frigg, la divinité de la famille et du mariage, porte une attention particulière à cette photo et envisage que chaque membre soit ordonné. Frigg se sent débordée vu le nombre de proches faisant partie de la photo, c'est pourquoi Jøsëf Marchand propose son aide pour ordonner rapidement toute la famille.

Cette famille est composée de N personnes.

Le but est de ranger chaque personne en fonction de sa taille. Le premier sur la photo doit être le plus petit et le dernier doit être le plus grand. Pour ordonner tout ce monde, nous pouvons faire cette opération autant de fois que voulu :

- Choisir i.
- Inverser la i^e et la $(i+K)^e$ personne.

K est le **nombre magique**, il vous est donné.

Afficher s'il est possible d'ordonner les personnes de la famille.

2.2.2 Stratégie - Validation

Le but de cet exercice est de savoir s'il est possible de trier une liste sachant qu'il est seulement possible de permuter deux éléments à une distance K.

Une stratégie possible est d'utiliser une variante du tri à bulles [11] :

Algorithme 2 : Un peu d'ordre, version tri à bulles

Entrées : Le nombre magique K,
Le nombre de personnes N,
La liste des tailles de chaque personne, T
Résultat : `Vrai` s'il est possible de trier la liste, `Faux` sinon.
répéter
  pour $i \leftarrow 0$ **à** $N-K-1$ **faire**
    si $T_i > T_{i+K}$ **alors**
      `Echanger`(T_i, T_{i+K});
    fin
  fin
jusqu'à N *fois*;
retourner `est_triee`(T);

S'il est possible de trier le tableau, alors le tableau sera trié au bout de maximum N itérations.

Cependant, cette stratégie fonctionne en une complexité temporelle de $O(N^2)$, et a donc peu de chances de passer les tests de performance.

2.2.3 Réalisation

Python

```
def ordre(k: int, n: int, tailles: List[int]) -> None:
    # Tri à bulles
    for _ in range(n):
        for i in range(n - k):
            if tailles[i] > tailles[i + k]:
                tailles[i], tailles[i + k] = tailles[i + k], tailles[i]

    est_triee = (tailles == sorted(tailles))
    print('OUI' if est_triee else 'NON')
```

11. `https://fr.wikipedia.org/wiki/Tri_%C3%A0_bulles`

2.2.4 Stratégie - Performance

Une analyse un peu plus approfondie révèle qu'il peut être utile de partitionner le tableau donné selon les classes résiduelles des positions, modulo K. Étant donné que l'on ne peut qu'échanger les positions de deux éléments étant séparé de K éléments, on ne peut pas changer les éléments appartenant à une même famille. Cependant, grâce au tri à bulles, on sait également qu'il est possible de réordonner comme on le souhaite les éléments d'une même famille.

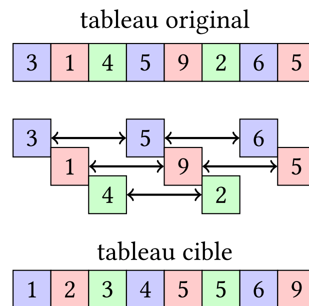

FIGURE 2.3 – Partition d'un tableau de 8 éléments pour $K = 3$

La question est alors, est-il suffisant de réordonner les familles afin de trier l'intégralité du tableau ?

Notre nouvelle stratégie va donc être de comparer les familles du tableau original et du tableau trié. Si la population d'une famille est différente dans le tableau trié, alors il sera impossible de trier le tableau en réordonnant les familles.

Algorithme 3 : Un peu d'ordre

Entrées : Le nombre magique K,
Le nombre de personnes N,
La liste des tailles de chaque personne, T
Résultat : `Vrai` s'il est possible de trier la liste, `Faux` sinon.
$T' \longleftarrow$ `copie_triee`(T) ;
pour $i \leftarrow 0$ **à** $K - 1$ **faire**
    $F \longleftarrow$ `multiensemble vide`;
    $F' \longleftarrow$ `multiensemble vide`;
    `// On construit les familles`
    $j \longleftarrow i$;
    tant que $j < N$ **faire**
        `ajouter` T_j à F;
        `ajouter` T'_j à F';
        $j \longleftarrow j + K$
    fin
    `// On compare les familles`
    si $F \neq F'$ **alors**
        retourner *Faux*
    fin
fin
retourner *Vrai*;

On peut comparer les familles à l'aide d'un multiensemble (`multiset` en C++, `Counter` en Python), ou simplement en triant la famille.

La complexité temporelle de cette approche est réduite. Selon l'implémentation, le tri du tableau original peut s'effectuer en $O(N)$ avec un tri comptage [12], ou en $O(N \log N)$ avec la plupart des tris populaires. Derrière, tous les éléments du tableau sont ajoutés deux fois dans un multiensemble. Selon l'implémentation, l'ajout d'un élément dans un multiensemble peut se faire en $O(1)$ ou $O(\log N)$, ce qui donne un total de $O(N)$ ou $O(N \log N)$ pour comparer les familles.

La complexité temporelle finale peut donc aller de $O(N)$ à $O(N \log N)$, ce qui doit être suffisant pour passer les tests de performance.

12. `https://fr.wikipedia.org/wiki/Tri_comptage`

2.2.5 Réalisation

Python

```python
from collections import Counter

def ordre(k: int, n: int, tailles: List[int]) -> str:
    objectif = list(sorted(tailles))
    for start in range(k):
        famille = Counter(tailles[start::k])
        attendue = Counter(objectif[start::k])
        if famille != attendue:
            return "NON"
    else:
        return "OUI"
```

2.3 Ouroboros

2.3.1 Énoncé

Jörmungandr est un serpent légendaire qui entoure tout Midgard, en passant par certaines villes, jusqu'à se mordre sa queue. Les villes rencontrées en partant de la queue de Jörmungandr jusqu'à sa tête sont données en entrée dans une liste de chaînes de caractères. Sa tête — et donc sa queue, qui se situent au même endroit car il se mord la queue — se trouvent toujours **entre** deux villes. Initialement, elles se situent entre la dernière ville listée et la première (souvenez-vous que la liste décrit un cycle), et Jörmungandr avance de gauche à droite dans la liste.

Dans M années aura lieu le Ragnarök. Chaque année jusqu'au Ragnarök, Jörmungandr prend l'une des actions suivantes :

- `A` : Avancer d'une ville de manière circulaire dans la liste ;
- `M` : Manger la ville la plus proche devant sa tête ;
- `R` : Se retourner, ce qui signifie qu'il avancera alors dans la direction opposée par la suite. S'il avançait de gauche à droite, il avance alors de droite à gauche, et inversement ;
- `C` : Recracher la dernière ville qu'il a mangé et qu'il n'a pas encore recraché, pour la placer devant sa tête.

Notez donc que si, après s'être retourné, Jörmungandr se dirige de droite à gauche dans la liste, il mangerait alors la ville située immédiatement à gauche de sa tête.

Skuld, Norne du futur, vous indique les actions prochaines de Jörmungandr pour les M prochaines années. Aidez Jøsëf Marchand à prédire l'état final des villes lors du Ragnarök, en les listant dans l'ordre en partant de sa queue jusqu'à sa tête.

2.3.2 Stratégie - Validation

On peut simplement essayer de maintenir l'état des villes de sorte à garder comme invariant : « La liste `villes` représente les villes restantes, dans l'ordre, en partant de la queue de Jörmungandr ». Aussi, les villes ayant été mangées par Jörmungandr seront enregistrées dans une pile, afin de pouvoir accéder à « la dernière ville qu'il n'a pas encore recraché » avec un dépilage.

En gardant ceci en tête :

- Avancer revient à déplacer le premier élément de la liste `villes` à la fin de la liste ;
- Manger une ville revient à empiler le premier élément de la liste `villes` dans l'estomac, et de l'enlever de la liste `villes` ;
- Retourner Jörmungandr revient à inverser la liste `villes` ;
- Recracher une ville revient à la dépiler de l'estomac et de l'insérer au début de la liste `villes`.

L'insertion/suppression au début d'une liste, ainsi que l'inversion d'une liste se fait en $O(N)$, avec N la taille de la liste. Dans le pire des cas, les M actions sont des requêtes ayant un coût de $O(N)$. La complexité temporelle de cet algorithme est donc $O(MN)$, ce qui a peu de chance de passer les tests de performance.

2.3.3 Réalisation

Python

```python
def situation_finale(n: int, m: int, villes: List[str], actions: List[str]) -> None:
    estomac = []
    for action in actions:
        match action:
            case 'A':
                villes.append(villes.pop(0))
            case 'M':
                estomac.append(villes.pop(0))
            case 'R':
                villes.reverse()
            case 'C':
                villes.insert(0, estomac.pop())

    print(*villes, sep = '\n')
```

2.3.4 Stratégie - Performance

Il est possible de réduire le coût du retournement en gardant en mémoire la direction de Jörmungandr plutôt que de retourner la liste des villes, et changer le comportement des autres actions en fonction de la direction de Jörmungandr. Par exemple, on peut imaginer un booléen `direction` :

- Lorsque le booléen est vrai, Jörmungandr avance dans le sens original, et la liste `villes` représente les villes en partant de la queue de Jörmungandr jusqu'à la tête.
- Lorsque le booléen est faux, Jörmungandr avance dans le sens inverse, et la liste `villes` représente les villes en partant de la tête de Jörmungandr jusqu'à la queue.

Ainsi, on peut retourner Jörmungandr en inversant simplement le booléen en $O(1)$.

Les listes ne sont pas cependant pas adaptées pour supporter l'insertion/la suppression au début de la liste. Pour ce faire, on peut utiliser une structure de données comme les files d'attente à double extrémité [13], qui permettent une insertion et suppression au début à la fin de la file en temps constant.

Ces deux modifications permettent d'effectuer toutes les actions en $O(1)$. La complexité finale est donc $O(M)$.

2.3.5 Réalisation

Python

```python
def situation_finale(n: int, m: int, villes: List[str], actions: List[str]) -> None:
    villes = deque(villes)
    estomac = []
    direction = True
    for act in actions:
        match act, direction:
            case 'A', True:
                villes.append(villes.popleft())
            case 'A', False:
                villes.appendleft(villes.pop())

            case 'R', True:
                direction = False
            case 'R', False:
                direction = True

            case 'M', True:
                estomac.append(villes.popleft())
            case 'M', False:
                estomac.append(villes.pop())

            case 'C', True:
                villes.appendleft(estomac.pop())
            case 'C', False:
                villes.append(estomac.pop())

    if not direction:
        villes.reverse()

    print(*villes, end = '\n')
```

13. https://fr.wikipedia.org/wiki/File_d%27attente_%C3%A0_double_extr%C3%A9mit%C3%A9

2.4 Bâtiments

2.4.1 Énoncé

Jörmungandr, le serpent se mordant la queue, n'est pas très adroit. Étant géant et passant dans plusieurs villes, il peut casser des bâtiments en faisant un faux mouvement. Verdandi est une norne mais ne peut voir que dans le présent, elle aimerait alors connaître à l'avance l'état des bâtiments.

Le royaume de Midgard est composé de N villes. Jörmungandr se déplace de ville en ville. Comme il se mord la queue, le chemin des villes qu'il emprunte forme une boucle.

Quand Jörmungandr se déplace dans une ville, le nombre de bâtiments cassés dans cette ville devient alors le plus grand nombre de bâtiments cassés entre cette ville et les $K - 1$ suivantes.

Le premier mouvement impacte la première ville ($i = 1$), puis le second mouvement impacte la seconde ville ($i = 2$) et ainsi de suite.

Afin d'aider Verdandi, votre objectif est le suivant : à partir du nombre initial de bâtiments cassés dans chaque ville, déterminer le nombre de bâtiments cassés dans chaque ville après R mouvements du serpent.

2.4.2 Stratégie - Validation

Une mise en œuvre naïve de l'algorithme consiste à simplement simuler les effets des R mouvements du serpent, un à un, dans l'ordre.

Algorithme 4 : Algorithme naïf pour bâtiments

Entrées : Le nombre N de villes,
le nombre R de mouvements du serpent,
Le nombre K de villes impliquées dans chaque mouvement,
Le nombre de bâtiments V_i initialement cassés dans chaque ville i

Résultat : Le nombre de bâtiments cassés dans chaque ville après les mouvements du serpent. Le tableau V est modifié en place.

```
pour i ← 0 à R − 1 faire
    max ⟵ 0;
    pour j ← i à i + K faire
        si V_{j mod n} > max alors
            max ⟵ V_{j mod n} ;
        fin
    fin
    V_{i mod n} ⟵ max;
fin
```

La complexité résultante est donc $O(RK)$, qui a peu de chance de passer les tests de performance.

2.4.3 Réalisation

Python

```python
def batiments(n: int, r: int, k: int, villes: List[int]) -> None:
    for i in range(r):
        nombre_batiments = 0
        for j in range(i, i + k):
            nombre_batiments = max(nombre_batiments, villes[j % n])
        villes[i % n] = nombre_batiments

    print(*villes)
```

2.4.4 Stratégie - Performance

Maximisation Rapide

Une première piste d'optimisation consiste à utiliser une structure de données capable de répondre à des requêtes sur le maximum d'un intervalle rapidement.

Comme nous modifions également le tableau, une structure comme un arbre de segment permet de réduire la complexité de la boucle interne en $O(\log N)$. La complexité finale est ainsi réduite en $O(R \log N)$. Cette complexité est meilleure que $O(RN)$ mais peut ne pas suffire pour passer les tests de performance.

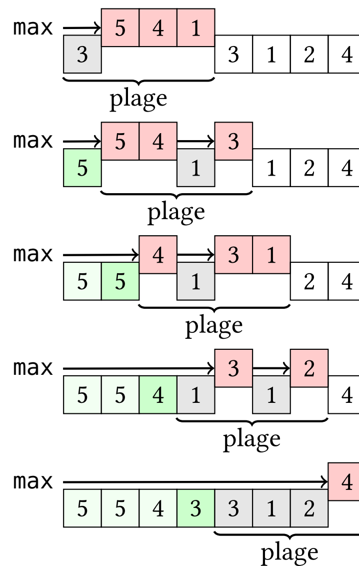

FIGURE 2.4 – Fenêtre glissante pour trouver efficacement les maximum de plage

Une meilleure approche est d'utiliser un algorithme de fenêtre glissante. L'idée est de maintenir, à chaque étape, une liste des éléments encore susceptibles d'être le maximum d'une plage à venir dans la plage actuellement traitée. Un élément devient *inactif* lorsqu'un élément supérieur à lui se situe à sa droite dans la plage considérée, car il ne sera donc jamais le plus grand élément d'une plage. Ces éléments inactifs sont représentés en gris dans la Figure 2.4. On garde alors en mémoire uniquement les éléments actifs, en rouge, dans une file à double extrémité. Par définition, ces éléments sont toujours par ordre décroissant dans la file, et donc, on peut facilement obtenir le maximum de la plage en regardant le premier élément de la file.

Pour faire avancer la plage :

— Si le premier élément de la file sort de la plage, on le défile.
— On se prépare à enfiler le nouvel élément à droite de la plage. Pour ce faire, on défile à droite tous les éléments de la file qui sont inférieurs au nouvel éléments, car ces éléments deviennent inactifs.
— On enfile le nouvel élément à droite de la plage.

En utilisant une file à double extrémité, l'enfilage et le défilage à gauche et à droite se font en $O(1)$. Étant donné que chaque élément impliqué dans une plage se fait enfiler et défiler une seule fois, la complexité temporelle de cette approche est alors de $O(R + K)$, ce qui est suffisant pour passer les tests de performance.

2.4.5 Réalisation

C++

```
void batiments(int n, int r, int k, vector<int>& villes) {
    deque<int> actifs;

    int fin = 0;
    for (int i = 0; i < r; i++) {
        while (fin < i + k) {
            while (!actifs.empty() && villes[actifs.back() % n] < villes[fin % n])
                actifs.pop_back();

            actifs.push_back(fin);
            fin++;
        }

        villes[i % n] = villes[actifs.front() % n];

        if (actifs.front() == i)
            actifs.pop_front();
    }

    for (int i = 0; i < n; i++) {
        cout << villes[i];
        if (i != n - 1)
            cout << " ";
    }
    cout << endl;
}
```

Simulation groupée

Nous verrons ici une stratégie différente. L'objectif va ici être de se débarrasser du facteur R, en évitant de simuler les actions une à une. En fait, les éléments dans la liste finale correspondent au maximum d'une certaine plage d'éléments dans la liste originale, qui dépend de K, de R et de leur position. Le but est de trouver, pour chaque élément de la liste `villes`, quels éléments vont être inclus dans cette plage.

La Figure 2.5 montre les tailles de plages à considérer pour $N = 8$, $K = 3$, et R allant jusqu'à 18. Par exemple, pour $R = 1$, le premier élément du tableau est le maximum d'une plage de trois éléments, et les autres éléments du tableau restent inchangés. Appelons ce tableau, contenant pour chaque position la taille de la plage à considérer, $T(R)$. On a donc ici $T(1) = [3, 1, 1, 1, 1, 1, 1, 1]$.

L'objectif est de prédire le contenu de $T(R)$ le plus rapidement possible. En suivant la logique de la Figure 2.5, il est facile de calculer $T(R)$ en $O(R)$, ce qui permet déjà de passer les tests de performances. Mais en faisant quelques observations, il est possible d'accélérer ce calcul :

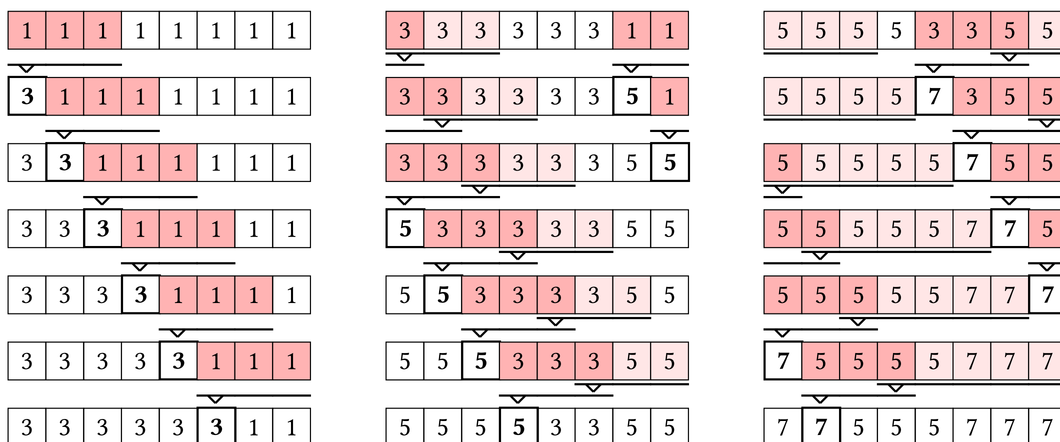

FIGURE 2.5 – Taille des plages à considérer pour des valeurs consécutives de R

- À l'étape R, la valeur en position $R - 1 \mod N$ est modifiée.
- Pour R allant de 1 à $N - K + 1$, la valeur K est ajoutée au tableau.
- Pour R allant de $N - K + 2$ à $2(N - K + 1)$, la valeur $2K - 1$ est ajoutée au tableau.
- En règle générale, à l'étape R, la valeur $K + \left\lfloor \frac{R-1}{N-K+1} \right\rfloor \cdot (K - 1)$ est ajoutée au tableau.

On peut alors simplement simuler uniquement les étapes $R - N - 1$ à R afin d'obtenir une à une chacune des cases de $T(R)$ en $O(N)$.

En reprenant la stratégie de maximisation rapide depuis la plus petite valeur du tableau, on obtient un algorithme qui trouve la réponse en $O(N)$.

2.4.6 Implémentation

Python

```
def batiments(n: int, r: int, k: int, villes: List[int]) -> None:
    # Calcul de T(R)
    taille_fenetre = [1] * n
    for etape in range(r - n, r):
        v = k + etape // (n - k + 1) * (k - 1)
        taille_fenetre[etape % n] = max(1, v)

    # Fenêtre glissante
    resultat = [0] * n
    actifs = deque()
    fin = r
    for i in range(r, r + n):
        while fin - i < taille_fenetre[i % n]:
            while len(actifs) > 0 and villes[actifs[-1]] < villes[fin % n]:
                actifs.pop()
            actifs.append(fin % n)
            fin += 1

        resultat[i % n] = villes[actifs[0]]
        if actifs[0] == i % n:
            actifs.popleft()

    print(*resultat)
```

2.5 Coursier d'Asgard

2.5.1 Énoncé

Les dieux nordiques doivent appliquer un protocole de communication très spécial pour prévenir les dieux des villes qui se font détruire par le serpent Jörmungandr. Ce protocole, nommé HTTP (pour Hel's Transcendent Transfer Protocol), a pour objectif de s'assurer que tous les dieux reçoivent le message.

Jøsëf est recruté par les dieux pour mettre en place le protocole de communication.
Le protocole est le suivant :

- Le message est d'abord placé dans une enveloppe, puis remis dans les mains d'un dieu quelconque, appelé le *dieu initial.*
- Cette enveloppe ne peut ensuite transiter qu'entre deux dieux qui partagent le même prénom ou le même nom.
- Cette enveloppe doit passer entre les mains de tous les dieux.
- **Un dieu doit toujours transmettre l'enveloppe à un autre dieu qui ne l'a pas encore reçu, si possible.**
- **Si un dieu ne peut plus transmettre l'enveloppe à un autre dieu qui ne l'a pas encore reçu, il doit redonner l'enveloppe au dieu qui lui a initialement donné.**

Étant donné la liste (non ordonnée) des paires de dieux s'étant transmis au moins une fois l'enveloppe, déterminez s'il est possible que les transmissions aient respecté le protocole HTTP, et si oui, déterminez qui a pu être le dieu initial.

2.5.2 Stratégie - Validation

Reformulation du problème Essayons de traduire l'énoncé avec de la théorie des graphes. Appelons $G = (S, A)$ le graphe non orienté représentant la situation, où :

- S est l'ensemble des dieux,
- A est l'ensemble des couples de dieux (u, v), où u et v partagent le même prénom ou le même nom.

L'entrée du problème nous indique un ensemble E de couples de dieux (u, v). L'objectif est de déterminer pour chaque potentiel dieu initial R si cet ensemble de couples est compatible avec le protocole HTTP.

- « Cette enveloppe ne peut transiter qu'entre deux dieux qui partagent le même prénom ou le même nom ». Cela veut déjà dire que les couples de dieux indiqués doivent faire partie des arêtes du graphe G, c'est à dire : $E \subseteq A$. Aussi, comme l'enveloppe passe nécessairement de dieu en dieu, alors le sous-graphe de G induit par E doit être connexe. Appelons ce sous-graphe T.
- « Cette enveloppe doit passer entre les mains de tous les dieux ». Cela signifie que le sous-graphe T doit être couvrant.
- « Un dieu doit toujours transmettre l'enveloppe à un autre dieu qui ne l'a pas encore reçu ». Cela interdit la présence de cycle dans T. T doit donc être un arbre couvrant de G.
- « Si un dieu ne peut plus transmettre l'enveloppe à un autre dieu qui ne l'a pas encore reçu, il doit redonner l'enveloppe au dieu qui lui a initialement donné ». Il s'agit ici de la description du retour sur trace d'un parcours en profondeur. T doit donc être un arbre de recherche couvrant de G.

Le problème est donc, étant donné un graphe G non orienté et un sous-graphe T, de déterminer si T peut être un arbre de recherche de G, et si oui, quelles peuvent en être les racines.

Analyse Premièrement, vérifier que T est un arbre couvrant de G est assez direct :

```
def chemin_valide(n: int, dieux: List[str], m: int, passations: List[str]) -> None:
    index_prenom: Dict[str, int] = {} # On attribue un index à chaque prénom
    index_nom: Dict[str, int] = {}    # On attribue un index à chaque nom
    dieux_compresse: List[Tuple[int, int]] = [(-1, -1)] * n
                  # On represente les dieux par un tuple (index prenom, index nom)
    index_dieu: Dict[Tuple[int, int], int] = {}
                  # On attribue un index à chaque dieu

    for i, dieu in enumerate(dieux):
        # On représente sépare les prénoms/noms des dieux
        prenom, nom = dieu.split()

        if prenom not in index_prenom:
            index_prenom[prenom] = len(index_prenom)
```

```
    if nom not in index_nom:
        index_nom[nom] = len(index_nom)

    dieux_compresse[i] = (index_prenom[prenom], index_nom[nom])
    index_dieu[dieux_compresse[i]] = i

T_adj = [ [] for _ in range(n) ] # Listes d'adjacence de T

# Construction de T
for arete in passations:
    # Vérification que T est un sous-graphe de G
    prenom1, nom1, prenom2, nom2 = arete.split()
    prenom1 = index_prenom[prenom1]
    prenom2 = index_prenom[prenom2]
    nom1 = index_nom[nom1]
    nom2 = index_nom[nom2]

    if prenom1 != prenom2 and nom1 != nom2:
        print("NON")
        return

    i = index_dieu[(prenom1, nom1)]
    j = index_dieu[(prenom2, nom2)]

    T_adj[i].append(j)
    T_adj[j].append(i)

# Vérification que T est un arbre couvrant de G
visite = [ False ] * n

def verification_arbre(index = 0, parent = None):
    # On marque le dieu comme étant visité
    visite[index] = True
    for voisin in T_adj[index]:
        if voisin == parent:
            # C'est l'arête que l'on vient de prendre qui est en double.
            # On peut l'ignorer
            continue

        if visite[voisin]:
            # T contient un cycle, ce n'est donc pas un arbre
            return False

        if not verification_arbre(voisin, index):
            # Rétro-propagation de la détection de cycle
            return False

    return True

if not verification_arbre():
    # T n'est pas un arbre
    print("NON")
    return

if not all(visite):
    # T n'est pas couvrant
    print("NON")
    return
```

Maintenant, la question est de vérifier, pour chaque sommet racine R, si l'arbre enraciné en R peut être un arbre de recherche de G. Étant donné T enraciné en R et G, on peut classifier toutes les arêtes de G en trois catégories :

- Les arêtes appartenant à T forment les *arcs couvrants*;
- Les arêtes reliant deux sommets de la même branche forment les *arêtes arrières*. Ces arêtes relient des sommets à leurs descendants. Il existe un chemin dans T qui relie les deux extrémités de l'arête;
- Enfin, la dernière catégorie d'arête, reliant deux sommets situés sur des branches différentes, forment les *arêtes croisées*. Il n'existe pas de chemin dans l'arbre reliant les deux extrémités d'une arête croisée.

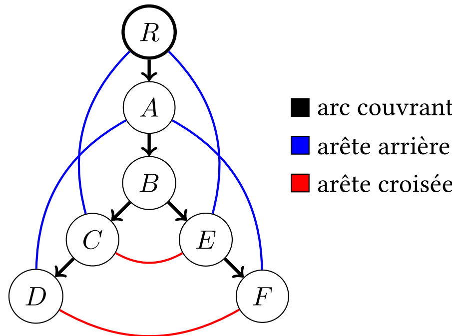

Figure 2.6 – Classification des arêtes d'un graphe

Si T est un arbre de recherche, alors il n'y aura pas d'arête croisée, puisque deux sommets ne peuvent plus être adjacents sans être descendants l'un de l'autre. S'il existait une arête croisée, alors le parcours en profondeur aurait dû emprunter cette arête avant d'effectuer un retour sur trace, vu que l'arête mène vers une branche différente, qui n'a donc pas été visitée au moment du parcours.

La manière traditionnelle de classifier les arêtes est d'utiliser un compteur pour noter l'ordre dans lequel on entre et on sort de chaque sommet dans le parcours en profondeur :

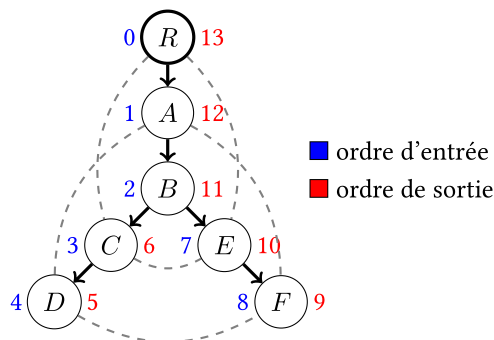

Figure 2.7 – Utilisation d'un compteur pour classifier les arêtes

Considérons qu'une arête relie deux sommets u et v :

- Si u est un descendant de v, alors v est découvert en premier, puis u est découvert. Ensuite, u est quitté, puis v est quitté. On a alors $\texttt{entree}_v < \texttt{entree}_u < \texttt{sortie}_u < \texttt{sortie}_v$
- Si cette condition est fausse, il s'agit alors d'une arête croisée.

Pour chaque racine possible, après avoir calculé le compteur d'entrée/sortie pour chaque sommet, on peut vérifier l'existence ou non d'une arête croisée dans le graphe. Si aucune arête croisée n'existe, alors la racine a pu être un dieu initial tout en respectant le protocole HTTP.

```python
dieux_initiaux = []
for R in range(n):
    entree = [-1] * n
    sortie = [-1] * n
    compteur = 0

    def parcours_compteur(index, parent = None):
        nonlocal compteur
        entree[index] = compteur
        compteur += 1

        for voisin in T_adj[index]:
            if voisin == parent:
                continue
            parcours_compteur(voisin, index)

        sortie[index] = compteur
        compteur += 1
    parcours_compteur(R)

    candidat = True
    for u in range(n):
        for v in G_adj[u]:
            if entree[u] < entree[v] < sortie[v] < sortie[u]:
                # v est un descendant de u
                continue
            if entree[v] < entree[u] < sortie[u] < sortie[v]:
                # u est un descendant de v
                continue

            # u - v est une arête croisée
            candidat = False
            break

        if not candidat:
            break

    if candidat:
        dieux_initiaux.append(R)

if len(dieux_initiaux) == 0:
    print("NON")
else:
    print("OUI")
    for index in dieux_initiaux:
        print(*dieux[index])
```

D'autres méthodes existent pour classifier les arêtes. Par exemple, on peut maintenir pendant le parcours en profondeur l'ensemble des sommets sur le chemin entre la racine et le sommet visité. Une arête arrière reliera un sommet à un de ses parents, qui se situera donc dans cet ensemble.

Étant donné que $|A| = O(N^2)$, cet algorithme fonctionne en $O(N^3)$. Cela n'est en revanche insuffisant pour passer les tests de performance.

2.5.3 Stratégie - Performance

Le principal défaut dans le précédent algorithme est que nous classifions la totalité des $O(N^2)$ arêtes pour chaque potentiel dieu initial. On peut grandement accélérer cette recherche d'arête croisée en utilisant directement la définition de G sans même le précalculer.

Au moment du parcours de l'arbre, une arête croisée relie un sommet à un sommet qui n'est pas encore visité. Comme ces deux sommets sont reliés, cela implique que les deux dieux concernés partagent le même prénom ou le même nom. Nous cherchons donc à savoir, au moment du parcours du sommet, s'il existe un autre dieu qui n'a pas encore été visité et qui possède le même prénom ou le même nom.

Nous pouvons donc simplement enregistrer le nombre de dieux possédant chaque prénom et chaque nom et qui n'ont pas encore été visité dans un tableau. Maintenir et inspecter ce tableau se fait en $O(1)$ pour chaque sommet, nous effectuerons donc un unique parcours de l'arbre en $O(N)$ pour chaque potentiel dieu initial, ce qui résulte en un algorithme en $O(N^2)$.

```
dieux_initiaux = []
for R in range(n):
    compteur_prenom = [0] * len(index_prenom)
    compteur_nom = [0] * len(index_nom)
    for prenom, nom in dieux_compresse:
        compteur_prenom[prenom] += 1
        compteur_nom[nom] += 1

    def potentiel_dieu_initial(index, parent = None):
        prenom, nom = dieux_compresse[index]
        compteur_prenom[prenom] -= 1
        compteur_nom[nom] -= 1

        for voisin in T_adj[index]:
            if voisin == parent:
                continue

            if not potentiel_dieu_initial(voisin, index):
                return False

        if compteur_prenom[prenom] > 0:
            return False
        if compteur_nom[nom] > 0:
            return False
        return True

    if potentiel_dieu_initial(R):
        dieux_initiaux.append(R)
```

2.6 Coursier de Midgard

2.6.1 Énoncé

Les dieux nordiques doivent appliquer un protocole de communication très spécial pour prévenir les dieux des villes qui se font détruire par le serpent Jörmungandr. Ce protocole, nommé HTTP (pour Hel's Transcendent Transfer Protocol), a pour objectif de s'assurer que tous les dieux reçoivent le message une seule fois.

Jøsëf est recruté par les dieux pour mettre en place le protocole de communication.
Le protocole est le suivant :

- Le message d'abord placé dans une enveloppe, puis remise dans les mains d'un dieu quelconque.
- Cette enveloppe ne peut ensuite transiter qu'entre deux dieux qui partagent le même prénom ou le même nom.
- Cette enveloppe doit passer dans les mains de tous les dieux.
- **Chaque dieu ne peut recevoir le message qu'une unique fois**
- **Si au transfert précédent le message est passé entre deux dieux qui ont le même prénom, alors au transfert suivant le message doit passer entre deux dieux qui ont le même nom, et inversement.**

Étant donné la liste des noms des dieux, proposez si possible un chemin de dieu en dieu qui respecte le protocole HTTP.

2.6.2 Stratégie - Validation

Une stratégie simple est simplement de construire toutes les permutations des dieux donnés en entrée, jusqu'à trouver une permutation qui respecte le protocole HTTP.

On peut commencer par effectuer le même prétraitement que pour Coursier d'Asgard :

```
def afficher_chemin(n: int, dieux: List[str]) -> None:
    ## Pré-traitement
    index_prenom: Dict[str, int] = {} # On attribue un index à chaque prénom
    index_nom: Dict[str, int] = {}    # On attribue un index à chaque nom
    dieux_compresse: List[Tuple[int, int]] = [(-1, -1)] * n
                  # On represente les dieux par un tuple (index prenom, index nom)
    index_dieu: Dict[Tuple[int, int], int] = {}
                  # On attribue un index à chaque dieu

    for i, dieu in enumerate(dieux):
        # On représente sépare les prénoms/noms des dieux
        prenom, nom = dieu.split()

        if prenom not in index_prenom:
            index_prenom[prenom] = len(index_prenom)
        if nom not in index_nom:
            index_nom[nom] = len(index_nom)

        dieux_compresse[i] = (index_prenom[prenom], index_nom[nom])
        index_dieu[dieux_compresse[i]] = i
```

Ensuite, on propose ici une fonction `trouve_chemin`, qui tente de trouver une permutation de dieux respectant le protocole HTTP. Si une telle permutation existe, cette permutation est enregistrée dans le premier paramètre, `chemin`, et la fonction retourne vrai. Si aucune permutation n'existe, la fonction ne modifie pas ses paramètres et retourne faux.

Pour ce faire, la fonction itère sur la liste des dieux restants à la manière d'une file. Le dieu ainsi défilé est vérifié contre le dernier dieu de la permutation en train d'être construite. Ce dieu est dit *compatible* s'il respecte les contraintes de l'énoncé, c'est à dire :

- Il partage le même prénom ou le même nom que le dernier dieu du chemin,
- On n'utilise pas deux fois une transmission utilisant les prénoms, idem pour les noms.

La première condition est vérifiée en comparant simplement les prénoms et les noms des dieux. La seconde condition est vérifiée grâce à un paramètre supplémentaire `coup`, qui indique la nature de la dernière transition empruntée. Une valeur de 0 indique que la dernière transition utilise deux prénoms identiques, et une valeur de 1 indique que la dernière transition utilise deux noms identiques. Une valeur de -1 indique qu'aucune transmission n'a encore eu lieu.

Si un dieu parmi les dieux restants est compatible avec le chemin en train d'être construit, alors on tente d'empiler ce dieu au bout du chemin et on effectue un appel récursif pour essayer de trouver une permutation. Si l'appel récursif renvoie vrai, alors une permutation valide a été trouvée.

```python
# On itère sur toutes les permutations
def trouve_chemin(chemin, index_restants, dernier_coup):
    dieux_restants = len(index_restants)
    if dieux_restants == 0:
        return True

    for _ in range(dieux_restants):
        index = index_restants.pop()
        # Vérification compatibilité
        if len(chemin) == 0:
            coup = -1
        else:
            for coup_tente in range(2):
                if coup_tente == dernier_coup:
                    # On ne peut pas passer deux fois par le même prénom/nom
                    continue
                dieu = dieux_compresse[index]
                dernier_dieu = dieux_compresse[chemin[-1]]
                if dernier_dieu[coup_tente] == dieu[coup_tente]:
                    coup = coup_tente
                    break
            else:
                # Dieu incompatible
                index_restants.insert(0, index)
                continue

        chemin.append(index)
        if trouve_chemin(chemin, index_restants, coup):
            return True

        index_restants.insert(0, index)
        chemin.pop()
    return False

chemin = []
index_restants = list(range(n))

if not trouve_chemin(chemin, index_restants, -1):
    print("IMPOSSIBLE")
else:
    for index in chemin:
        print(dieux[index])
```

La complexité temporelle de cette approche est de $O(N \cdot N!)$, à cause de l'insertion en début de tableau en $O(N)$ à l'intérieur de la fonction. En utilisant une file, on peut effectuer cette opération en $O(1)$ et réduire la complexité temporelle à $O(N!)$. Cette complexité est beaucoup trop élevée pour espérer passer les tests de performance.

2.6.3 Stratégie - Performance

En fait, la stratégie précédente étudie le déplacement de dieu en dieu. En modélisant un tel graphe où les dieux sont les sommets, le problème demande de trouver un chemin Hamiltonien dans un graphe, avec des contraintes supplémentaires. Cependant, le problème du chemin Hamiltonien est un problème de décision NP-complet. Il faut donc trouver une approche différente pour espérer passer les tests de performance.

Au contraire, si nous représentons les prénoms et les noms comme les sommets d'un graphe biparti, et les dieux comme les arêtes de ce graphe, alors l'énoncé demande de trouver un chemin Eulérien [14] dans ce graphe. Voici une représentation du problème pour les dieux suivants :

```
0dric Apik
0dric Brik
1dric Apik
1dric Brik
Tonb Apik
```

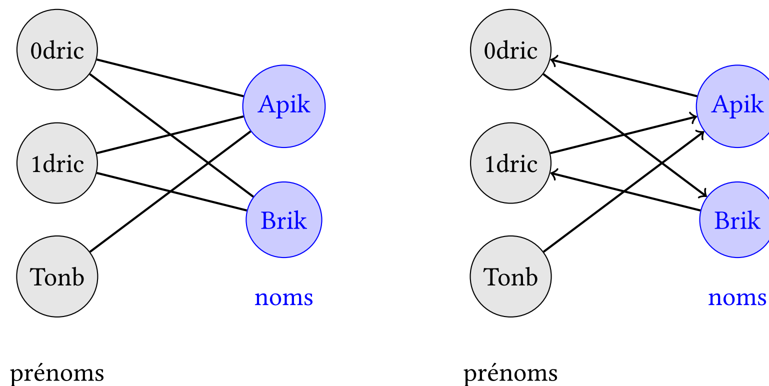

FIGURE 2.8 – Représentation du problème sous la forme d'une recherche de chemin Eulérien

Le chemin Eulérien indiqué, Tonb - Apik - 0dric - Brik - 1dric - Apik, représente une solution du problème :

```
Tonb Apik
0dric Apik
0dric Brik
1dric Brik
1dric Apik
```

Un chemin Eulérien ne peut exister que si :

- Le graphe est connexe ;
- Le nombre de sommets ayant un degré impair est soit 0, soit 2.

En clair, pour qu'une solution au problème existe, il doit exister soit 0, soit 2 noms ou prénoms qui sont partagés par un nombre impair de dieux. Si tous les noms et prénoms sont partagés par un nombre pair de dieux, alors nous pouvons commencer notre chemin par n'importe quel dieu. Sinon, nous devons commencer notre chemin par un sommet de degré impair.

14. https://en.wikipedia.org/wiki/Eulerian_path

Commençons par construire ce graphe :

```
def chemin(n: int, dieux: List[str]) -> None:
    # On attribue un index à chaque prénom
    prenoms: List[str] = []
    prenom_index: Dict[str, int] = {}
    # On attribue un index à chaque nom
    noms: List[str] = []
    nom_index: Dict[str, int] = {}

    adj = [
        [set() for _ in range(n)], # liste d'adjacence des prénoms
        [set() for _ in range(n)], # liste d'adjacence des noms
    ]

    for dieu in dieux:
        prenom, nom = dieu.split()
        if prenom not in prenom_index:
            prenoms.append(prenom)
            prenom_index[prenom] = len(prenom_index)
        if nom not in nom_index:
            noms.append(nom)
            nom_index[nom] = len(nom_index)

        adj[0][prenom_index[prenom]].add(nom_index[nom])
        adj[1][nom_index[nom]].add(prenom_index[prenom])
```

On utilise ici des ensembles dans la liste d'adjacence afin d'avoir une suppression efficace, qui sera nécessaire par la suite.

Ensuite, on peut vérifier si un chemin Eulérien existe en cherchant tous les sommets de degré impair :

```
    # On sélectionne tous les sommets de degré impair
    bouts = [
        (nature, index)
        for nature in range(2)
        for index in range(n)
        if len(adj[nature][index]) % 2 == 1
    ]

    if len(bouts) == 0:
        # Le graphe est Eulérien, on sélectionne un sommet arbitraire
        bouts = [(0, 0)] * 2

    if len(bouts) != 2:
        # Le graphe n'a pas de chemin Eulérien
        print("IMPOSSIBLE")
        return
```

Si un tel chemin existe, on peut utiliser l'algorithme de Hierholzer [15] afin de trouver un chemin Eulérien :

```
# Algorithme de Hierholzer
pile = [bouts.pop()]
chemin_eulerien = []
while len(pile) > 0:
    nature, index = pile[-1]
    if len(adj[nature][index]) == 0:
        chemin_eulerien.append(pile.pop())
        continue

    index_suivant = adj[nature][index].pop()
    adj[1 - nature][index_suivant].remove(index)
    pile.append((1 - nature, index_suivant))

# On vérifie que toutes les arêtes ont été consommées
for nature in range(2):
    for index in range(n):
        if len(adj[nature][index]) > 0:
            # Le graphe n'est pas connexe
            print("IMPOSSIBLE")
            return

for i in range(len(chemin_eulerien) - 1):
    nature1, index1 = chemin_eulerien[i]
    nature2, index2 = chemin_eulerien[i+1]
    if nature1 == 0:
        prenom = prenoms[index1]
        nom = noms[index2]
    else:
        prenom = prenoms[index2]
        nom = noms[index1]

    print(prenom, nom)
```

Grâce à la suppression dans les ensembles en $O(1)$, l'algorithme résultant a une complexité de $O(N)$, ce qui permet de passer les tests de performances.

15. `https://fr.wikipedia.org/wiki/Graphe_eul%C3%A9rien#Algorithme_de_Hierholzer`

2.7 Répartition Divine

2.7.1 Énoncé

N dieux veulent se répartir le contrôle des villes parmi V villes existantes. Les dieux sont numérotés de 0 à $N-1$, et les villes de 0 à $V-1$. Chaque dieu possède une certaine affection pour chaque ville, indiquée dans un tableau `villes`. Pour chaque dieu, il vous est donné une chaîne binaire de taille V : le caractère en position i vaut `1` si le dieu veut contrôler la ville i, et `0` sinon.

Lorsque certaines villes parmi les V villes sont abandonnées, alors deux dieux vont tenter de se répartir équitablement leur contrôle. Cependant, il peut y avoir des villes dites *contestées* dont les deux dieux veulent le contrôle. Si le nombre de villes contestées est pair, alors pas de problème, les dieux peuvent se les répartir équitablement. Mais si ce nombre est impair, ils se battront inévitablement.

Alors, pour un certain sous-ensemble de villes abandonnées, on dit que deux dieux sont compatibles si le nombre de villes contestées par les deux dieux dans ce sous-ensemble est pair.

Étant donné un sous-ensemble S de villes dites abandonnées, et un entier Δ, on appelle alors $R(S, \Delta)$ le nombre de X tels que les dieux X et $X \oplus \Delta$ sont compatibles, où $\oplus$ représente l'opération « ou exclusif » au niveau du bit.

Afin d'évaluer l'animosité générale chez les dieux, Jøsëf Marchand doit alors répondre à R requêtes. Chaque requête prend la forme d'un sous-ensemble de villes S, auquel Jøsëf Marchand devra répondre en appelant la fonction `Hashe` avec, consécutivement, toutes les valeurs de $R(S, \Delta)$ pour toutes les valeurs possibles de Δ entre 1 et $N-1$.

La fonction `Hashe` est donnée en pseudo-code ci-dessous :

```
Hashe(valeur) {
    X = (X * valeur + Z + 37) % 1000 000 007;
    Y = (X * 13 + 36 * Y + 257) % 1000 000 009;
    Z = (Y * valeur + 4 * valeur + X * X + 7) % 998 244 353;
}
```

Faites attention aux dépassements entier ! En C++ par exemple, il faut utiliser un type d'entier 64 bits, tel que `long long`.

Les valeurs de `X`, `Y` et `Z` sont initialisées à 0 au début de l'exercice et ne sont **jamais** réinitialisées. Après chaque requête, Jøsëf Marchand doit afficher les nouvelles valeurs de `X`, `Y` et `Z`.

Aidez Jøsëf Marchand à répondre à toutes ces requêtes.

2.7.2 Stratégie - Validation

La mise en œuvre de l'exercice pour passer les tests de validation est assez directe :

```python
X = Y = Z = 0

def Hashe(V):
    global X, Y, Z
    X = (X * V + Z + 37) % 1000000007
    Y = (X * 13 + 36 * Y + 257) % 1000000009
    Z = (Y * V + 4 * V + X * X + 7) % 998244353

def somme_compatible(
    v: int, n: int, villes: List[List[str]], r: int, requetes: List[List[str]]
) -> None:
    for requete in requetes:
        for delta in range(1, n):
            compatibles = 0
            for x in range(n):
                a = x
                b = x ^ delta
                conflits = 0
                for i in range(v):
                    if requete[i] == villes[a][i] == villes[b][i] == '1':
                        conflits += 1
                if conflits % 2 == 0:
                    compatibles += 1
            Hashe(compatibles)
        print(X, Y, Z)
```

La complexité temporelle de cette approche est de $O(RN^2V)$, ce qui est insuffisant pour passer les tests de performances.

2.7.3 Stratégie - Performance

Transformée de Fourier ? La fonction $R(S,\Delta)$ est une fonction qui effectue une opération sur toutes les paires de dieux i,j tel que $i \oplus j = \Delta$. Dénotons « $\cdot$ » l'opération qui, pour deux sous-ensembles de villes A et B désirés par deux dieux, indique si les deux dieux sont en conflit, c'est à dire, si le nombre de villes appartenant à S, A et B est impair.

$$A \cdot B = \begin{cases} 1 & \text{si les dieux sont en conflit} \\ 0 & \text{sinon} \end{cases}$$

Appelons R_Δ un résultat intermédiaire très proche de la valeur recherchée, qui est le résultat d'une $\oplus$-convolution :

$$R_\Delta = \sum_{i \oplus j = \Delta} \texttt{villes}_i \cdot \texttt{villes}_j$$
$$R = \texttt{villes} * \texttt{villes},$$

où $*$ représente une $\oplus$-convolution.

Pour un ensemble S de villes considérées, R_Δ donne le compte de dieux en conflits, c'est à dire, $N - R(S,\Delta)$. Calculer la valeur de R_Δ nous donne directement la valeur de $R(S,\Delta)$. En utilisant le théorème des convolutions [16], on obtient

$$\mathcal{F}(R) = \mathcal{F}(\texttt{villes}) \cdot \mathcal{F}(\texttt{villes})$$

où $\mathcal{F}$ est liée à une transformée de Fourier — ici, une transformée de Walsh-Hadamard. [17]

Tableau d'Autocorrélation En cryptographie, on appelle *Tableau d'autocorrélation* (ACT) [18] d'une liste L de N nombres binaires sur V bits la matrice $N \times 2^V$ telle que

$$\texttt{ACT}(\Delta, S) = \sum_{i \oplus j = \Delta} (-1)^{S \cdot L_i \oplus S \cdot L_j},$$

où « $\cdot$ » est cette fois-ci un produit vectoriel sur $GF(2)$. En considérant deux nombres binaires de V bits A et B, où A_i indique la valeur du bit en position i dans A, on a ici :

$$A \cdot B = \bigoplus_{i=0}^{V-1} A_i B_i$$

En clair, $A \cdot B$ vaut 0 si le nombre de bits en commun entre A_i et B_i est pair, et 1 sinon. Cela est très proche de notre précédente définition de « $\cdot$ », à l'exception que `ACT` effectue un $\oplus$ entre $S \cdot L_i$ et $S \cdot L_j$, alors que nous voudrions un « ET » binaire à la place, afin de "sélectionner" un sous-ensemble de villes S à considérer.

Le tableau d'autocorrélation se calcule traditionnellement en effectuant la transformée d'Hadamard du tableau de distribution des delta (DDT) [19], un tableau de taille $N \times 2^V$ défini comme suit :

$$\texttt{DDT}(\Delta, S) = \#\{i, j \mid i \oplus j = \Delta, L_i \oplus L_j = S\}$$

Pour notre problème, nous allons donc utiliser une définition légèrement différente du tableau de distribution, en remplaçant le $\oplus$ sur le masque S par un « ET » binaire :

$$\texttt{DDT}(\Delta, S) = \#\{i, j \mid i \oplus j = \Delta, L_i L_j = S\}$$

16. https://en.wikipedia.org/wiki/Convolution_theorem
17. https://fr.wikipedia.org/wiki/Transform%C3%A9e_de_Hadamard
18. https://doc.sagemath.org/html/en/reference/cryptography/sage/crypto/sbox.html#sage.crypto.sbox.SBox.autocorrelation_table
19. https://doc.sagemath.org/html/en/reference/cryptography/sage/crypto/sbox.html#sage.crypto.sbox.SBox.difference_distribution_table

Ce tableau se calcule facilement en $O(N2^V + N^2)$:

```
using namespace std;

#define MAXVILLES 13
#define MAXMASK 1 << MAXVILLES
#define MAXDIEUX 8192

int V;
int villes[MAXDIEUX];
int DDT[MAXDIEUX][MAXMASK];

int lire_bitmask() {
    string s;
    cin >> s;

    int bitmask = 0;
    for (int i = 0; i < V; i++) {
        bitmask <<= 1;
        if (s[i] == '1')
            bitmask += 1;
    }
    return bitmask;
}

int main() {
    // Lecture de l'entrée
    int N;
    cin >> V >> N;

    for(int i = 0; i < N; i++) {
        int mask = lire_bitmask();
        villes[i] = mask;
    }

    // Initialisation du tableau DDT
    for (int delta = 0; delta < N; delta++) {
        for (int s = 0; s < 1 << V; s++) {
            DDT[delta][s] = 0;
        }
    }

    // Calcul du tableau DDT
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            int delta = i ^ j;
            int s = villes[i] & villes[j];
            DDT[delta][s]++;
        }
    }
}
```

La transformée d'Hadamard de ce tableau ainsi modifié nous renverra un tableau légèrement différent du tableau d'autocorrélation :

$$\mathcal{F}(\texttt{DDT})(\Delta, S) = \sum_{i \oplus j = \Delta} (-1)^{SL_i \cdot SL_j}$$

En ne considérant qu'un sous-ensemble de villes S, $(-1)^{SL_i \cdot SL_j}$ renvoie 1 si le nombre de villes contestées par les dieux i et j est pair, et -1 sinon. C'est quasiment la valeur de $R(S, \Delta)$! On retrouve d'ailleurs :

$$R(S, \Delta) = \frac{1}{2} \left(\mathcal{F}(\texttt{DDT})(\Delta, S) + N \right)$$

En utilisant la transformée de Walsh-Hadamard rapide [20], on parvient à calculer notre variante du tableau d'autocorrélation en $O(NV2^V)$:

```
void walsh_spectrum(int *array) {
    for (int h = 1; h < 1 << V; h <<= 1) {
        for (int i = 0; i < 1 << V; i += 2*h) {
            for (int j = i; j < i + h; j++) {
                int x = array[j];
                int y = array[j + h];
                array[j] = x + y;
                array[j + h] = x - y;
            }
        }
    }
}

int main() {
    // (Coupé) Calcul de DDT
    // ...

    for (int delta = 0; delta < N; delta++)
        walsh_spectrum(DDT[delta]);
}
```

On peut alors répondre à toutes nos requêtes grâce à ce tableau :

```
long long X = 0;
long long Y = 0;
long long Z = 0;

void Hashe(long long value) {
    X = (X * value + Z + 37) % 1000000007;
    Y = (X * 13 + 36 * Y + 257) % 1000000009;
    Z = (Y * value + 4 * value + X * X + 7) % 998244353;
}

int main() {
    // (Coupé) Calcul de ACT
    // ...

    int R;
    cin >> R;

    for (int i = 0; i < R; i++) {
        int s = lire_bitmask();

        for (int delta = 1; delta < N; delta++) {
            long long V = (ACT[delta][s] + N) / 2;
            Hashe(V);
        }
        cout << X << ' ' << Y << ' ' << Z << '\n';
    }
}
```

La complexité finale de notre algorithme est de $O(N^2 + NV2^V + RN)$, ce qui est suffisant pour passer les tests de performances si implémenté judicieusement.

20. `https://en.wikipedia.org/wiki/Fast_Walsh%E2%80%93Hadamard_transform`