RIBAUT Thomas, Dereze Alexandra – FISA3 Informatique

Programmation linéaire et complexité

Semestre 6

Professeur : Monsieur MANDIAU René

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES HAUTS-DE-FRANCE

Université Polytechnique HAUTS-DE-FRANCE

Introduction :

Ce TP va nous permettre d'approfondir les complexités des algorithmes de Hanoi, Fibonacci, du crible d'Eratosthène, des algorithmes de tri et du compte est bon. Pour se faire, nous allons réaliser notre code en langage Python sous l'environnement de développement PyCharm.

I) TP 1

A) Exercice 1

a) L'énoncé

Hanoi Écrivez un programme permettant de résoudre le problème des tours de Hanoï pour un entier n passé en paramètre.

— Effectuez plusieurs mesures du temps d'exécution ($n = 1, 2, 5, 10, 15, \ldots$) et tracez la courbe résultante ainsi qu'une courbe de tendance.

— Comparez vos résultats avec la complexité théorique calculée en TD.

b) Le compte-rendu

La tour de Hanoi est un puzzle mathématique où nous avons trois tiges et n disques. L'objectif du puzzle est de déplacer la pile entière vers une autre tige, en obéissant aux règles simples suivantes :

- Un seul disque peut être déplacé à la fois.
- Chaque déplacement consiste à prendre le disque supérieur de l'une des piles et à le placer au-dessus d'une autre pile, c'est-à-dire qu'un disque ne peut être déplacé que s'il s'agit du disque le plus haut d'une pile.
- Aucun disque ne peut être placé sur un disque plus petit.

PSEUDO-LANGAGE :

Entrées :
- n : entier
- a, b, c : caractère – Représente chaque tour

```
Fonction Hanoi(n, a, b, c)
Début :
    Si n = 1 alors :
        Ecrire(''Transfert de A vers C pour terminer'');
    Sinon :
        Hanoi(n-1, a, c, b);
        Ecrire(''Transfert de A vers C'');
        Hanoi(n-1, b, a, c);
    Fin du sinon
Fin
```

```python
def hanoi(n, A, B, C):
    #A : départ, B: intermédiaire, C : Arrivée;
    if n != 1:
        hanoi(n-1, A, C, B)
        hanoi(n-1, B, A, C)
```

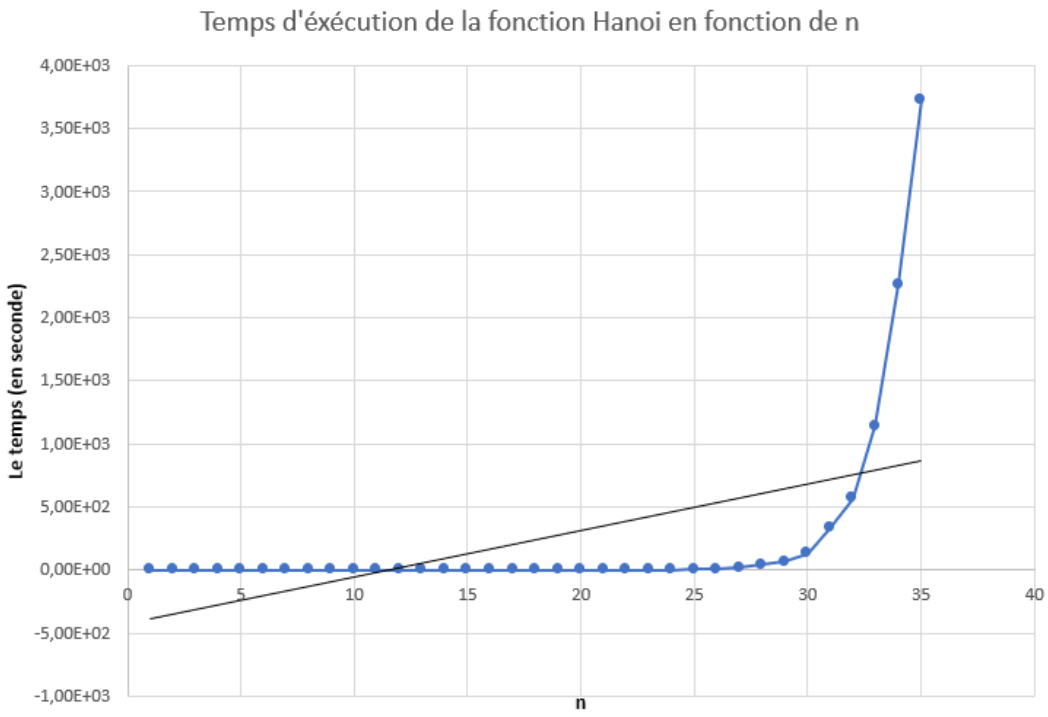
FIGURE 1 : SCRIPT PERMETTANT LA REALISATION DES TOURS D'HANOI

Les résultats :

n	1	5	10	15	20
Temps (en seconde)	1,60E-12	4,60E-12	0,000125	0,0050152	0,1699627

n	25	26	27	28	29
Temps (en seconde)	4,548550899	9,0602215	18,0916788	38,5596422	62,6238053

n	30	31	32	33	34	35
Temps (en seconde)	130,60384	329,53868	565,73076	1141,8044	2261,3234	3727,1641



La courbe résultante

La courbe de tendance

Rappelons la complexité théorique vu en TD :

Nous avons déterminé une complexité temporelle pour l'algorithme de Hanoi : T(n) = Θ(2n).

Ainsi, pour n=1 $T_{théorique}(1)= 2^1$ = 2 secondes

n= 10 $T_{théorique}(10)= 2^{10}$ = 1 024 secondes

n= 20 $T_{théorique}(20)= 2^{20}$ = 1 048 576 secondes

⋮

n=35 $T_{théorique}(35)= 2^{35}$ = 3.44 E+10 secondes

On remarque que les valeurs théoriques et nos résultats varient énormément. Nous l'expliquons car les résultats théoriques se base sur sur le fait qu'un ordinateur effectue 106 opérations à la seconde. Or, lors de la pratique, nous utilisons un ordinateur qui possède des performances plus grandes c'est-à-dire que nos tests s'effectuent avec un ordinateur qui effectue 1.8 milliards d'opérations à la seconde.

On pourrait trouver des valeurs approchées en prenant la valeur théorique divisé par la vitesse de base de notre ordinateur divisés par la vitesse de base de l'ordinateur théorique.

Ainsi, on pourrait trouver des valeurs approchées avec la formule suivante :

$$T_{pratique} = \frac{T_{theorique}}{1.8\ x\ 10^9}\ x\ \frac{1}{106}$$

Ainsi, nos estimations ressemblent à :

n=1 $T_{pratique}$ (1)= $\frac{2}{1.8\ x\ 10^9}\ x\ \frac{1}{106}$ = 1.05E-11 secondes

n= 10 $T_{théorique}(10)=$ $\frac{1024}{1.8\ x\ 10^9}\ x\ \frac{1}{106}$ = 5.37E-09 secondes

n= 20 $T_{théorique}(20)= \frac{1048576}{1.8\ x\ 10^9}\ x\ \frac{1}{106}$ = 5.49E-06 secondes

⋮

n=35 $T_{théorique}(35)= \frac{3.44 \text{ x } 10\text{^}10}{1.8\ x\ 10^9}\ x\ \frac{1}{106}$ = 0.18 secondes

Cela reste des estimations, cela se rapproche plus de nos valeurs que les valeurs théoriques.

B) Exercice 2

a) L'énoncé

Complexité et récursivité Écrivez le programme permettant de calculer la suite récurrente suivante :

$$u_n = \begin{cases} 1 & si \quad n = 0 \quad ou \quad n = 1 \\ u_{n-1} + u_{n-2} & si \quad n \geq 2 \end{cases}$$

— Proposer un algorithme itératif et Mesurer les temps d'exécution pour différentes valeurs de n.

— Proposer une version récursive et effectuer la même opération. Constatez-vous une amélioration des temps d'exécution ?

— Il existe un algorithme de complexité logarithmique pour calculer cette suite, expliquez le, implantez le et expérimentez.

b) Le compte-rendu

La série de Fibonacci génère le numéro suivant en ajoutant deux numéros précédents. La série de Fibonacci commence à partir de deux nombres – F_0 et F_1. La série de Fibonacci satisfait aux conditions suivantes :

$F_n = F_{n-1} + F_{n-2}$

Avec comme valeurs de départ $F_0 = 1$ et $F_1 = 1$.

PSEUDO-LANGAGE, VERSION ITERATIVE :

Entrées :

- n ∈ N

```
Fonction Fibonacci_iteratif(n)
Début :
        Variables : u, v, t, i
        Si n = 0 ou n = 1 alors :
                Renvoyer n
        Fin du si
        u = 0, v = 1
        Pour i allant de 2 à n , faire :
                t = u + v
                u = v
                v = t
        Fin du pour
        Renvoyer v
Fin
```

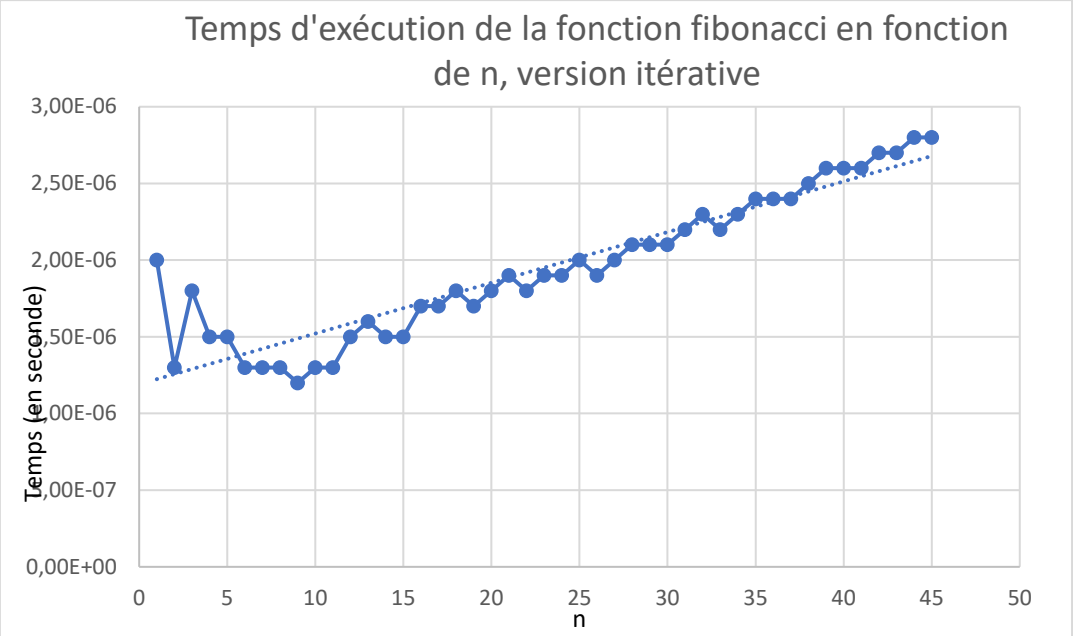
```python
def calc_fibonacci_iteratif(n: int) -> int:
    u: int = 0
    v: int = 1

    for i in range(2, n):
        t = u + v
        u = v
        v = t

    return v
```

FIGURE 2 : SCRIPT PERMETTANT LA REALISATION DE FIBONACCI, VERSION ITERATIVE

Les résultats :

n	1	5	10	15	20
Temps (en seconde)	2,30E-06	1,50E-06	1,30E-06	1,50E-06	1,80E-06

n	25	30	35	40	45
Temps (en seconde)	2,00E-06	2,10E-06	2,40E-06	2,60E-06	2,80E-06



La courbe résultante

La courbe de tendance

PSEUDO-LANGAGE, VERSION RECURSIVE :

Entrées :

- n : entier

Fonction Fibonacci_recursif(n)
Début :
    Si n = 0 ou n = 1 alors :
        Renvoyer 1
    Sinon
        Renvoyer Fibonacci_recursif(n-1) + Fibonacci_recursif(n-2)
    Fin du sinon
Fin

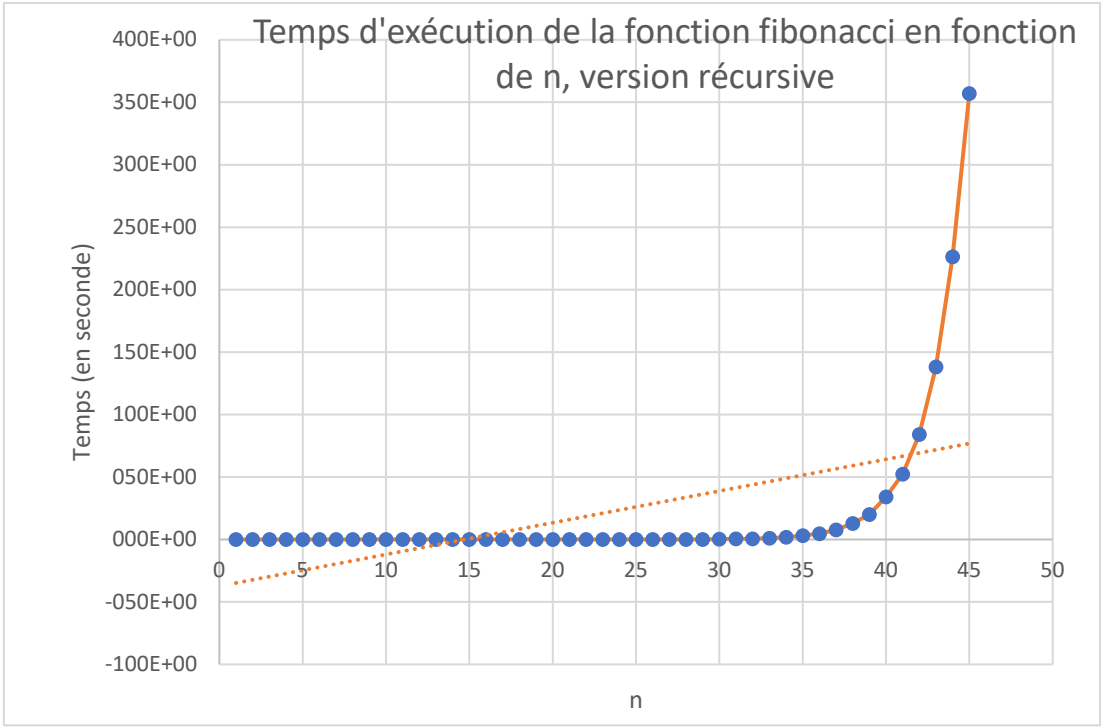
```python
def calc_fibonacci_recurrence(n: int) -> int:
    if n == 0 or n == 1:
        return 1
    else:
        return calc_fibonacci_recurrence(n - 1) + calc_fibonacci_recurrence(n - 2)
```

FIGURE 3 : SCRIPT PERMETTANT LA REALISATION DE FIBONACCI, VERSION RECURSIVE

Les résultats :

n	1	5	10	15	20
Temps (en seconde)	1,30E-06	2,40E-06	1,87E-05	0,0002242	0,0037541

n	25	30	35	40	45
Temps (en seconde)	0,0239679	0,2687983	3,1333328	34,1331011	357,029967



La courbe résultante

La courbe de tendance

On observe clairement que les temps d'exécution de la version récursive sont supérieurs aux temps d'exécution de la version itérative. Pour la version itérative, on aura un temps d'exécution qui se situe dans une fourchette entre 1.30E-06 et 2.80E-06 secondes, au contraire pour la version récursive, on se situera entre 1.30E-06 et 357 secondes.

La suite de fibonacci créé un arbre qui part du haut vers le bas, le résultat sera les feuilles. Fibonacci part du haut des branches et dès qu'il descend il va créer 2 nouvelles branches pour chaque feuille tant que l'on est encore dans n.

Pour arriver aux résultats logarithmiques on va séparer deux cas où n est pair et ou n est impair. On va passer une valeur pour n mais l'algorithme va diviser le résultat en une liste de deux cases. Le résultat final sera toujours la case de droite. Dans ce cas quand n est impair alors on va prendre n-1/2 pour calculer fibonacci, quand il est pair, ce sera n/2. Il arrive donc à récupérer n en utilisant des moitiés de n. Pour n=10 par exemple, il va récupérer n_5 puis n_2. On parle de complexité logarithmique car il calcule la moitié de ce qu'il réalise dans le l'algorithme récursif de fibonacci.

PSEUDO-LANGAGE, VERSION LOGARITHMIQUE :

Entrées :
- n ∈ N

Fonction Fibonacci_log(n)
Début :
    Variables : u, v, t, i
    Si n = 0 alors :
        Resultat = [0,1]
    Sinon si n=1 :
        Resultat = [1,1]
    Sinon :
    Variable de 2 listes d'entiers u et v
    Si n MOD 2 =1 :
        V = Fibonacci_log(Le plus grand entier de (n-1)/2)
        Result = [v[0] * 2 + v[1]+2, (2*v[0] + v[1] * v[1]]
    Sinon :
        V = Fibonacci_log(Le plus grand entier de (n/2)
        Result = [2*v[1] – v[0]) * v[0], v[0]*2 + v[1] * 2]

    Renvoyer result
Fin

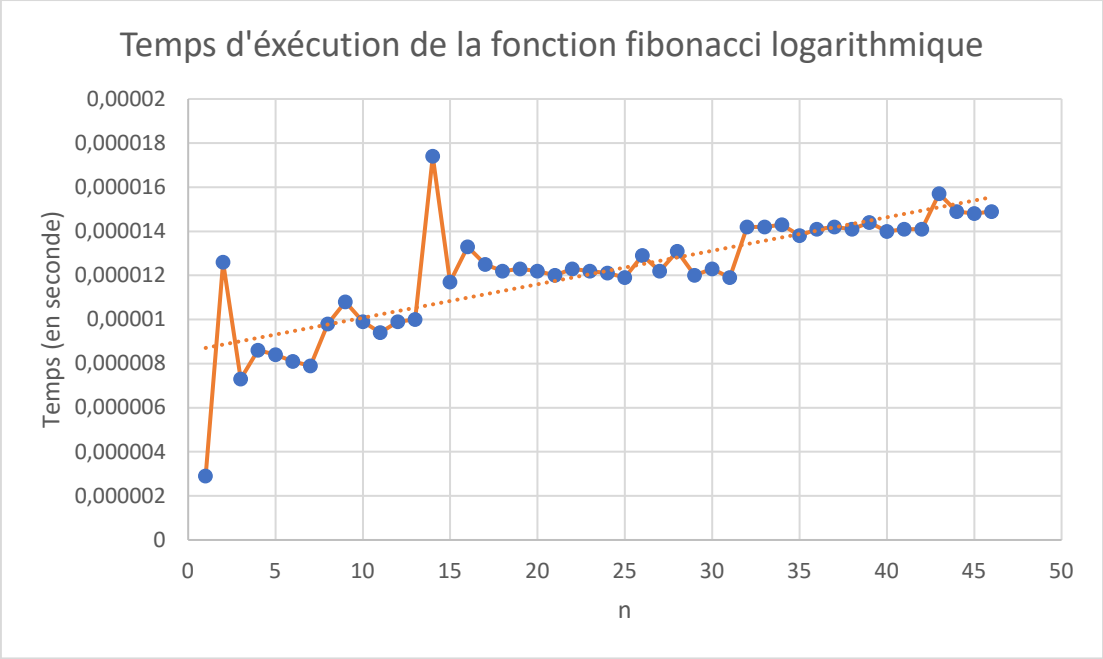
```python
def calc_fibonacci_log(n: int) -> List[int]:
    if n == 0:
        result = [0, 1]
    elif n == 1:
        result = [1, 1]
    else:
        v: List[int]
        u: List[int]

        if n % 2 == 1:
            v = calc_fibonacci_log(floor((n - 1) / 2))
            result = [v[0] ** 2 + v[1] ** 2, (2 * v[0] + v[1]) * v[1]]
        else:
            v = calc_fibonacci_log(floor(n / 2))
            result = [(2 * v[1] - v[0]) * v[0], v[0] ** 2 + v[1] ** 2]

    return result
```

FIGURE 4 : SCRIPT PERMETTANT LA REALISATION DU CRIBLE D'ERATOSTHENE

Les résultats :

n	1	5	10	15	20
Temps (en seconde)	2,90E-06	8,4E-06	9,90E-06	1,17E-05	1,22E-05

n	25	30	35	40	45
Temps (en seconde)	1,19E-05	1,23E-05	1,38E-05	1,40E-05	1,48E-05



La courbe résultante

La courbe de tendance

Suite à la question précédente, on sait que la version itérative est plus rapide que la version récursive. Nous allons donc comparer la version itérative et la version logarithmique. On observe clairement que les temps d'exécution de la version logarithmique se situe dans une fourchette entre 2.90E-06 et 1.45E-06 secondes, au contraire pour la version itérative, on se situera entre 1.30E-06 et 2.80E-06 secondes. Ainsi, on peut observer que la version itérative est plus rapide lorsque n est petit. Plus n grandit, plus la version logarithmique sera plus rapide que la version itérative.

I) Exercice 3

a) L'énoncé

Crible d'Eratosthène Eratosthène était un savant grec qui avait proposé une méthode de recherche des nombres premiers inférieurs à N. Le principe du crible est simple. On se donne un entier N arbitraire et on constitué initialement la liste des entiers compris entre 1 et N. L'algorithme consiste à *cribler* (cocher) tous les nombres qui ne sont pas premiers de la manière suivante :

1. On se place au début de la liste et on coche 1
2. On avance jusqu'au prochain nombre p qui n'a pas été coché, et on crible ses multiples
3. On recommence au point 2 tant qu'on n'a pas atteint la fin de la liste

Proposer un algorithme qui effectue cette tâche, et calculer sa complexité.

b) Le compte-rendu

Le principe du crible est simple. On se donne un entier N arbitraire et on constitue initialement la liste des entiers compris entre 1 et N. L'algorithme consiste à cocher (cribler) tous les nombres qui ne sont pas premiers de la manière suivante :

- On se place au début de la liste et on crible 1 ;
- On avance jusqu'au prochain nombre p qui n'a pas été criblé (donc 2 initialement) ;
- On crible ses multiples, on coche tous les nombres kp pour tout k∈[2,$\lfloor\frac{N}{p}\rfloor$] ;
- On recommence au point 2 tant qu'on n'a pas atteint la fin de la liste.

PSEUDO-LANGAGE :

Entrées :
- i : entier
- Un tableau

Fonction supprimer_multiples(i, tableau)

Début :

Pour chaque valeur (x) dans le tableau :

Si (x > 1) et (x modulo i =0) :

On supprime x du tableau

Fin si

Fin Pour

Fin

Entrées :
- n : entier

Fonction Crible(n):

Début :

Tableau d'entiers de 1 a n

Pour chaque i dans le tableau

supprimer_multiples(i, Tableau)

Fin Pour

Retourner Tableau

Fin

```python
def Supprimer_multiples(i, tableau):
    for x in tableau:
        if x > i and x % i == 0:
            tableau.remove(x)


def Crible(N):
    tableau = [i for i in range(1, N)]
    for i in tableau:
        Supprimer_multiples(i, tableau)
    return tableau
```

FIGURE 5 : SCRIPT PERMETTANT LA REALISATION DU CRIBLE D'ERATOSTHENE

Entrées : n : entier Fonction Crible(n): début : Tableau d'entiers de 1 a n Pour chaque i dans le tableau supprimer_multiples(i, Tableau) Fin Pour Retourner Tableau Fin	Cout	Favorable	Défavorable
	C1	n	n
	C2	n	n

Entrées : i : entier, un tableau Fonction supprimer_multiples(i, tableau) Début : Pour chaque valeur (x) dans le tableau : Si (x > 1) et (x modulo i =0) : On supprime x du tableau Fin si Fin Pour Fin	Cout	Favorable	Défavorable
	c1	n	n
	c2	0	n/2
	c3	0	n/2

Cas Favorable : $T_{Fav}(n) = c1 \cdot n + C1 \cdot n + C2 \cdot n$; d'où la complexité dans le cas favorable est en n.(c1+C1+C2).

Cas Défavorable : $T_{Déf}(n) = c1 \cdot n + c2.\ (n/2) + c3.(n/2) + C1 \cdot n + C2 \cdot n$; d'où la complexité dans le cas défavorable est en n.(c1+C1+C2) + (n/2).(c2 + c3).

II) TP 2

a) L'énoncé

L'objectif est d'implémenter les algorithmes de tri

- l'algorithme de tri par sélection
- l'algorithme de tri par insertion
- l'algorithme de tri à bulles
- l'algorithme de tri fusion

Évaluations temporelles

- Paramétrer votre fonction de manière à être capable de faire varier la taille du tableau. Nous pouvons admettre qu'une taille de 10^3 éléments est dite *petite*. La taille dite *moyenne* est définie telle que le tri par insertion soit 10 fois plus lent que pour une *petite* taille. La taille dite *grande* est quant à elle, définie telle que le tri par insertion soit 100 fois plus lent que pour une taille *moyenne*. Vous préciserez bien évidemment votre taille *moyenne* et votre taille *grande*. - Indiquer clairement le choix de ces tailles de tableau et fixer les pour chaque algorithme

Analyse des résultats

- Mesurez les temps d'exécution pour chaque algorithme précédemment introduits
- Tracez pour chaque algorithme les courbes des temps d'exécution.
- Commentez en quelques lignes vos résultats : comparez par exemple vos résultats avec les complexités théoriques vues en cours, indiquez s'il s'agit (i) du cas optimal, (ii) du pire des cas ou (iii) du cas moyen.

b) Le compte-rendu

L'ensemble des tests se baseront sur des nombres inférieurs à 500 et sur des longueurs de listes n qui vont varier lors des tests.

- Tri par sélection

```python
def permuter(tableau: List[int], index_case1: int, index_case2: int) -> List[int]:
    tmp = tableau[index_case1]
    tableau[index_case1] = tableau[index_case2]
    tableau[index_case2] = tmp
```

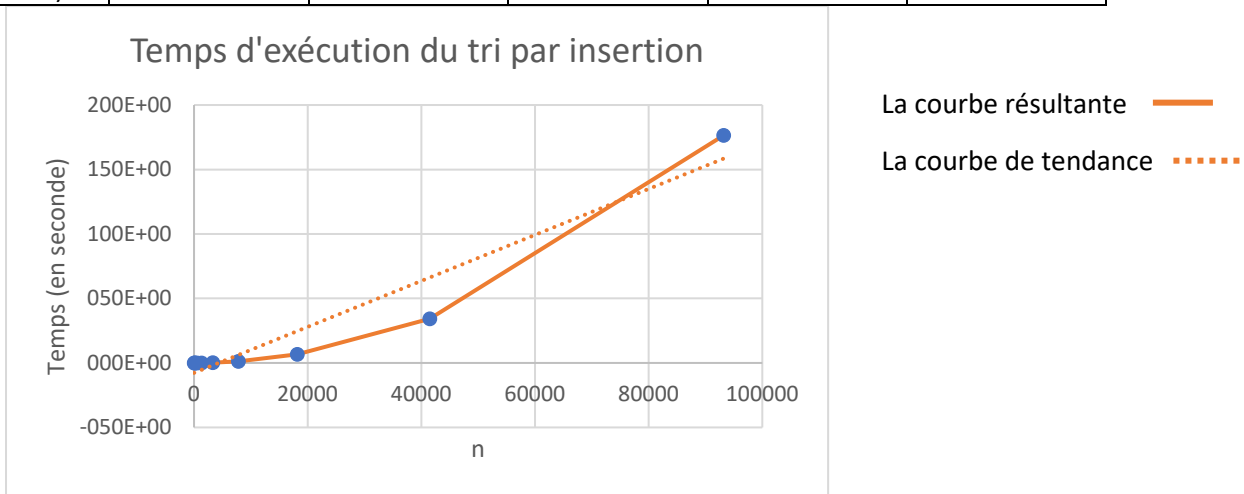
```python
def tri_selection(tableau: List[int]) -> List[int]:
    for i in range(len(tableau)):
        min: int = i

        for j in range(i + 1, len(tableau)):
            if tableau[j] < tableau[min]:
                min = j

        if min != j:
            permuter(tableau, i, min)

    return tableau
```

FIGURE 6 : SCRIPT PERMETTANT LA REALISATION TRI PAR SELECTION

Les résultats :

n	2	44	168	496	1312
Temps (en seconde)	1,33E-05	0,0001737	0,0009834	0,006213	0,0377084

n	3264	7808	18176	41472	93184
Temps (en seconde)	0,2298947	1,2334525	6,5949144	34,2720156	176,401854



- Tri par insertion

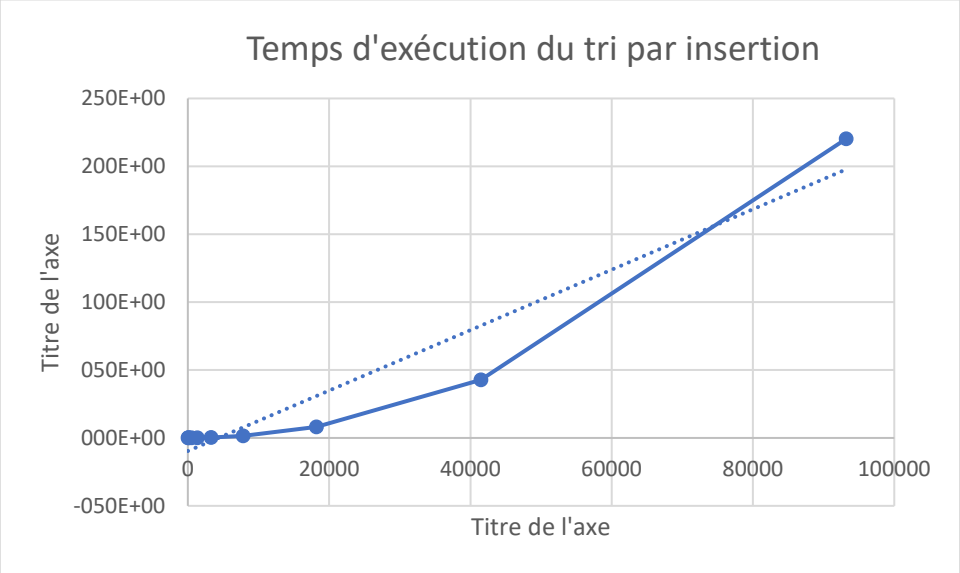
```python
def tri_insertion(tableau: List[int]) -> List[int]:
    for j in range(0, len(tableau)):
        cle: int = tableau[j]
        i: int = j - 1

        while i >= 0 and tableau[i] > cle:
            tableau[i + 1] = tableau[i]
            i -= 1

        tableau[i + 1] = cle

    return tableau
```

FIGURE 7 : SCRIPT PERMETTANT LA REALISATION DU TRI PAR INSERTION

Les résultats :

n	2	44	168	496	1312
Temps (en seconde)	1,17E-05	1,72E-04	1,07E-03	6,67E-03	4,40E-02

n	3264	7808	18176	41472	93184
Temps (en seconde)	2,68E-01	1,50E+00	8,15E+00	4,28E+01	2,20E+02



La courbe résultante

La courbe de tendance

- Tri à bulles

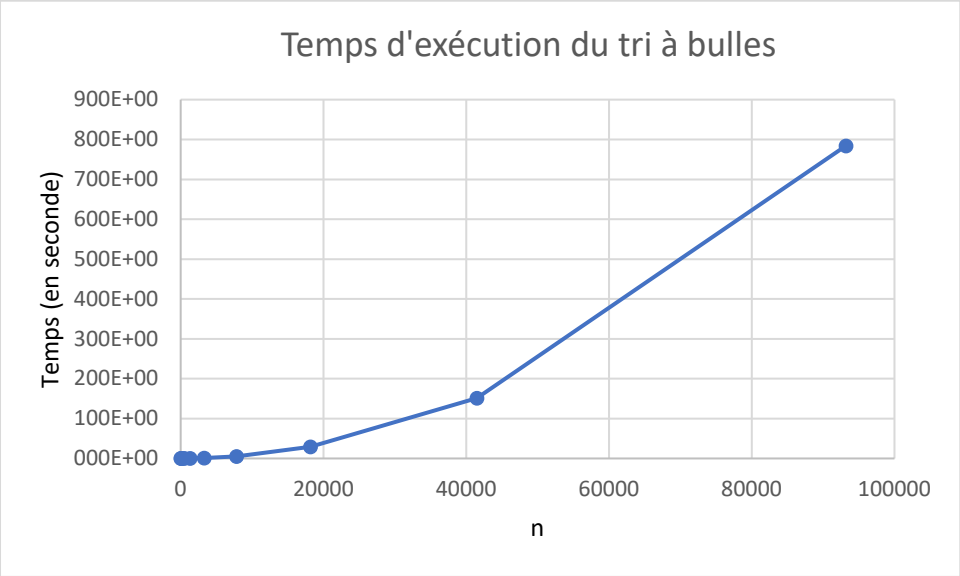
```python
def tri_bulles(tableau: List[int]) -> List[int]:
    echange: bool = True

    while echange:
        echange = False
        for i in range(1, len(tableau) - 1):
            if tableau[i] > tableau[i + 1]:
                permuter(tableau, i, i + 1)
                echange = True

    return tableau
```

FIGURE 8 : SCRIPT PERMETTANT LA REALISATION DU TRI A BULLES

Les résultats :

n	2	44	168	496	1312
Temps (en seconde)	1,24E-05	2,22E-04	2,48E-03	2,08E-02	1,54E-01

n	3264	7808	18176	41472	93184
Temps (en seconde)	9,63E-01	5,46E+00	2,90E+01	1,51E+02	7,84E+02



La courbe résultante

La courbe de tendance

- Tri fusion

```
def tri_fusion_recursif(L):
    n = len(L)
    if n > 1:
        p = int(n / 2)
        L1 = L[0:p]
        L2 = L[p:n]
        tri_fusion_recursif(L1)
        tri_fusion_recursif(L2)
        L[:] = fusion(L1, L2)
```

```
def tri_fusion(L):
    M = list(L)
    tri_fusion_recursif(M)
    return M
```

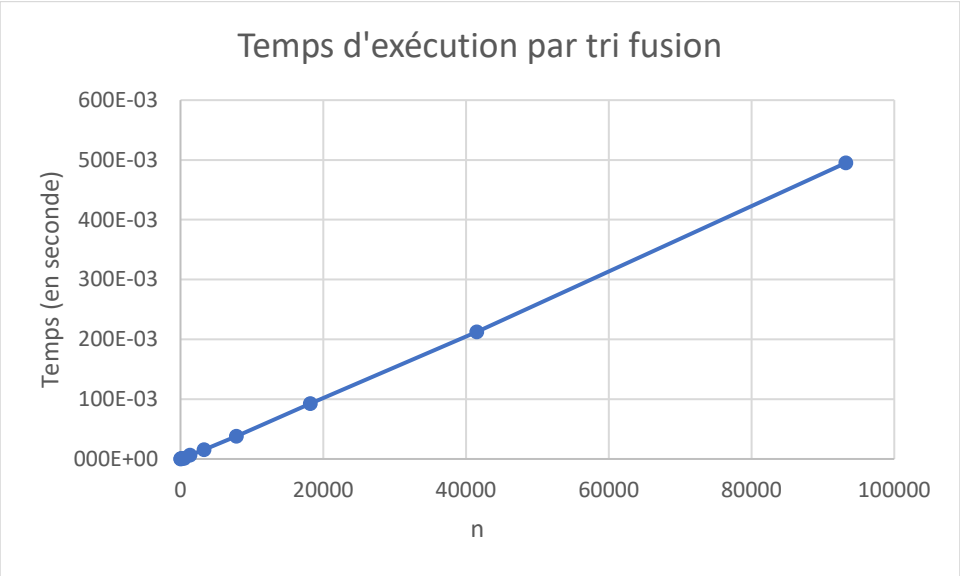
```
def fusion(L1, L2):
    n1 = len(L1)
    n2 = len(L2)
    L12 = [0] * (n1 + n2)
    i1 = 0
    i2 = 0
    i = 0
    while i1 < n1 and i2 < n2:
        if L1[i1] < L2[i2]:
            L12[i] = L1[i1]
            i1 += 1
        else:
            L12[i] = L2[i2]
            i2 += 1
        i += 1
    while i1 < n1:
        L12[i] = L1[i1]
        i1 += 1
        i += 1
    while i2 < n2:
        L12[i] = L2[i2]
        i2 += 1
        i += 1
    return L12
```

FIGURE 9 : SCRIPT PERMETTANT LA REALISATION DU TRI A BULLES

Les résultats :

n	2	44	168	496	1312
Temps (en seconde)	1,38E-05	0,0001871	0,0007579	0,0007579	0,0060962

n	3264	7808	18176	41472	93184
Temps (en seconde)	0,0156121	0,037775	0,0922861	0,2121882	0,4948566



La courbe résultante

La courbe de tendance

On remarque clairement que l'algorithme par tri fusion est le plus rapide de tous les algos de tri étudiés sur des valeurs grandes. Sur des petites valeurs, ça sera l'algorithme par insertion le plus rapide.

III) TP 3

a) L'énoncé

Objectif : Mesurer la complexité d'un algorithme d'exploration : "Le compte est bon"

Dans un célèbre jeu télévisé, les candidats doivent en 1 minute atteindre un résultat donné en utilisant 6 nombres et les quatre opérations arithmétiques : $+, -, \times, /$. Ce résultat est compris entre 100 et 999 et les nombres sont tirés au sort parmi un ensemble de 28 plaques composé de :

- 20 plaques numérotées de 1 à 10 (2 par nombre)
- 2 plaques de 25
- 2 plaques de 50
- 2 plaques de 75
- 2 plaques de 100

Les calculs doivent suivre les règles suivantes :

- Chaque nombre ne peut être utilisé qu'une seule fois.
- Les opérations sont restreintes aux entiers naturels positifs. Par exemple : les divisions ne sont autorisées que si le reste est nul. De même les soustractions renvoyant un résultat négatif sont interdites.

Ce jeu peut être résolu en utilisant la technique d'exploration suivante. Soit R le résultat à atteindre et P_n l'ensemble des n plaques tirées au sort. Considérons toutes les paires (p_i, p_j) de deux entiers pris dans P_n. Pour chaque (p_i, p_j) il est possible d'appliquer l'une des quatre opérations et construire ainsi 4 nouveaux ensembles :

$$P_{n-1} = (p_i + p_j) \cup (P_n - \{p_i, p_j\})$$
$$P_{n-1} = (p_i \times p_j) \cup (P_n - \{p_i, p_j\})$$
$$P_{n-1} = (p_i - p_j) \cup (P_n - \{p_i, p_j\})$$
$$P_{n-1} = (p_i / p_j) \cup (P_n - \{p_i, p_j\})$$

Le processus d'exploration consiste à tester successivement si R est atteignable avec l'un des 4 ensembles P_{n-1}. A chaque construction d'un nouvel ensemble P_{n-k}, on vérifie si R appartient à P_{n-k}. Si R n'appartient à aucun ensemble P_0 constructible à partir de P_n, alors il n'y a pas de solution.

Exemple simple avec 3 plaques :

Considérons le résultat $R = 120$ à atteindre à partir de l'ensemble $P_3 = \{2; 10; 100\}$.

A partir de cet ensemble P_3, il est possible de construire les couples suivants : $couples(P_3) = \{(2, 10); (2, 100); (10, 100)\}$.

Considérons le premier couple : $(2, 10)$.

En appliquant l'addition sur ce premier couple, on peut construire un premier ensemble P_2 égale à $\{12, 100\}$. Il faut donc tester si R peut être obtenu à partir de cet ensemble P_2. Un seul couple peut être obtenu à partir de P_2 : $(12, 100)$. En appliquant l'addition, on obtient un ensemble $P_3 = \{112\}$ dont le seul élément est différent de R. En appliquant la multiplication, on obtient un autre ensemble : $P_3 = \{1200\}$ dont le seul élément est aussi différent de R. En appliquant la soustraction $(120-12)$, on obtient un troisième ensemble $P_3 = \{88\}$ ne contenant pas R. La division n'est pas applicable puisque le reste de $100/12$ est non nul.

Cela signifie que le premier ensemble P_2 construit à partir de l'addition et du couple $(2, 10)$ ne permet pas d'obtenir R. Appliquons donc la multiplication pour construire un second ensemble $P_2 = \{20, 100\}$. Ici encore, un seul couple peut être obtenu à partir de P_2 : $(20, 100)$. En appliquant l'addition, on obtient un ensemble $P_3 = \{120\}$ contenant le résultat R recherché. L'algorithme s'arrête donc et renvoie la solution : $(2 \times 10) + 100$.

Sujet : Écrire un programme permettant de résoudre le jeu « Le compte est bon » à l'aide de l'algorithme précédemment décrit.

Analyse de l'algorithme : Nous vous proposons de structurer votre réflexion en fonction des réponses que vous allez fournir pour ces questions

1. Calculer le nombre d'ensembles examinés par l'algorithme dans le pire des cas en fonction de n. Combien y a-t-il d'ensembles lorsque $n = 6$?
2. Déterminer la complexité de l'algorithme d'exploration.
3. Déterminer le temps d'exécution moyen de l'algorithme utilisé pour résoudre le jeu "le compte est bon". (Faire une moyenne sur plusieurs mesures).

Question bonus : Ce problème est-il NP-complet ?

b) Le compte-rendu

Réalisation de l'algorithme permettant de résoudre le jeu « Le compte est non ».

```
Calculer = {'+': lambda a, b: a + b,
            '*': lambda a, b: a * b,
            '-': lambda a, b: a - b if a - b > 0 else 0,
            '/': lambda a, b: a / b}

Inverser = {'+': '-', '-': '+', '*': '/', '/': '*'}
```

FIGURE 10 : SCRIPT PERMETTANT DE CALCULER SELON CHAQUE OPERATEUR

```
def exploration(resultat_attendu: int, plaquettes: List[int]) -> List[Tuple[str, str, int]]:
    result: List[Tuple[str, str, int]] = []

    if len(plaquettes) == 1:
        result = [int(resultat_attendu)] if resultat_attendu == plaquettes[0] else []
    else:
        for nombre in plaquettes:
            for operateur in '+-*/':
                try:
                    pn_moins_un = Calculer[Inverser[operateur]](resultat_attendu, nombre)

                    for solution in exploration(pn_moins_un, [pn for pn in plaquettes if pn != nombre]):
                        result.append((operateur, solution, int(nombre)))
                except ZeroDivisionError:
                    pass

    return result
```

FIGURE 11 : SCRIPT PERMETTANT DE VERIFIER SI UN RESULTAT R EST ATTEIGNABLE OU NON A PARTIR DES PLAQUETTES

```python
def afficher_resultat(res) -> str:
    if isinstance(res, int):
        result = str(res)
    else:
        op, a, b = res

        if isinstance(a, int):
            result = "{} {} {}".format(a, op, b)
        else:
            result = "({}) {} {}".format(afficher_resultat(a), op, b)

    return result
```

FIGURE 12 : SCRIPT PERMETTANT DE RETOURNER LA LIGNE DE CALCULPERMETTANT D4ATTEINDRE LE RESULTAT R.

```python
def generer_debut_aleatoire(nb_plaquettes: int) -> Tuple[int, List[int]]:
    plaquettes: List[int] = []
    for i in range(nb_plaquettes):
        plaquettes.append(random.randint(100))

    return random.randint(899) + 100, plaquettes
```

FIGURE 13 : SCRIPT PERMETTANT DE TIRER AU HASARD 6 PLAQUES ET RENVOYANT DE MANIERE ALEATOIRE UN RESULTAT ENTRE 100 ET 999.

Analyse de l'algorithme :

1) Le pire des cas dans cet algorithme se traduit par le fait que l'on ne trouve aucune combinaison permettant d'atteindre le résultat R.

   L'exemple donné dans l'énoncé nous permet de comprendre le fonctionnement du compte est bon.

   Pour un ensemble P_3 = {a,b,c}, il est possible de construire 3 couples : {(a,b) ; (a,c) ; (b,c)}

   Prenons le premier couple (a,b), on va d'abord effectuer les opération suivantes sur celui-ci :

   - Addition : (a+b, c)
   - Soustraction (a-b, c)
   - Multiplication : (a*b, c)
   - Division : (a/b, c)

   On suppose ici que a/b renvoie un nombre entier et que a-b >0.

   Pour les couples de nouveau créés, nous pouvons effectuer pour chacun d'eux 4 nouvelles opérations pour obtenir un ensemble.

Pour les couples (a,c) et (b,c), on va effectuer sur chacun d'eux les 4 opérations pour obtenir un ensemble.

Ainsi, pour un couple de n=3, on va d'abord avoir 4 opérations suivi de 4 opérations sur ces opérations puis 4 opérations sur les deux derniers couples.

A l'aide de ces informations, nous en avons déduit la formule suivante :

$$= \frac{n\ x\ 4^{n-1}}{2}$$

Ainsi pour n=6, le nombre d'ensemble sera de $\frac{6\ x\ 4^5}{2} = 3\ 072\ ensembles$

Dans notre algorithme, nous prenons en compte les couples dans les 2 sens, c'est-à-dire que pour un ensemble {a,b,c}, on trouvera 6 couples qui sont : {(a,b) ;(a,c) ;(b,c) ;(b,a) ;(c,a) ;(c,b)}. Ainsi dans notre cas, la complexité dans un cas défavorable est de :

$$= n\ x\ 4^{n-1}$$

2)

Fonction exploration (R, plaquette) result: List[Tuple[str, str, int]] = [] S'il a que 1 plaquette alors : Si le resultat attendu = plaquettes[0] alors result = [resultat_attendu] sinon : result = [] sinon: Pour chaque plaquette Pour chaque opérateur : P_{n-1} = Calculer[Inverser[operateur]](resultat_attendu, nombre) Pour chaque solution dans exploration(P_{n-1}, [pn for pn in plaquettes if pn != nombre]): Ajouter dans result(operateur, solution, int(nombre)) return result	si	Favorable	Défavorable
	c1	1	0
	c2	0	0
	c3	0	0
	c4	0	0
	c5	0	1
	c6	0	n
	c7	0	4
	c8	0	1
	c9	0	4
	c10	0	1
	c11	1	1

Cas Favorable : $T_{Fav}(n)$ = c1 + c11 d'où la complexité dans le cas favorable est en c1+c11.

Cas Défavorable : $T_{Déf}(n)$ = c5 + c6. n + c7.4+ c8 + c9 · 4 + c10 + c11 ; d'où la complexité dans le cas défavorable est en n.c6 + 4.(c7+c9) + c5 + c8 + c10 +c11.

3)

Les résultats des temps d'exécution :

Test	1	2	3	4	5
Temps (en seconde)	0.8520786	0.78552899	0.763833	0.099262099	0.806129899

Test	6	7	8	9	10
Temps (en seconde)	0.1144536999	0.794209099	0.1035640999	0.8731285	0.799386600

Test	11	12	13	14	15
Temps (en seconde)	0.799386600000127	0.1027407400	0.7814318	0.8765038	0.87461609

Test	16	17	18	19	20
Temps (en seconde)	1.1128012	1.07880319	0.834681700	0.9330262000	0.89419079999

La moyenne obtenue du temps d'exécution du compte est bon est de 0,71486326 secondes.

Sources :

N°	Quoi ?	Sources
1	Hanoi	https://www.geeksforgeeks.org/c-program-for-tower-of-hanoi/
2	Fibonacci	https://www.tutorialspoint.com/data_structures_algorithms/fibonacci_series.htm
3	Crible d'Eratosthène	http://zanotti.univ-tln.fr/ALGO/I51/Crible.html
4	Tri fusion	https://www.f-legrand.fr/scidoc/docmml/algorithmes/general/tri/tri.html