Informatique & Cybersécurité
DEPREDURAND Dan
KERBELLEC Alphonse
15/11/2024

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES HAUTS-DE-FRANCE

Université Polytechnique HAUTS-DE-FRANCE

Compte-rendu TP Complexité

Sommaire:

I. Introduction

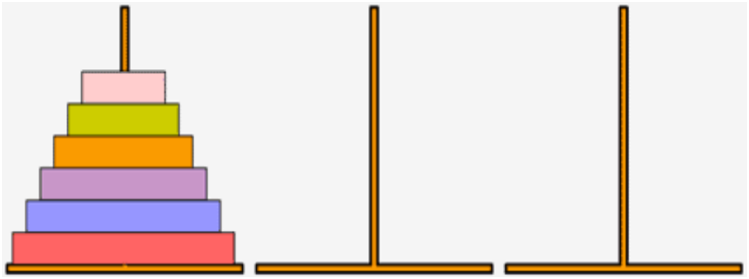
Le TP1 a pour objectif de nous familiariser avec les ordres de grandeur en complexité. Il comprend l'écriture et l'étude en complexité des programmes des tours de Hanoi, de la suite de Fibonacci et du crible d'Eratosthène.

Le TP3 a pour objectif d'écrire un algorithme capable de trouver une solution au jeu "Le compte est bon et d'étudier sa complexité.

Nos programmes seront écrits en python, afin de pouvoir utiliser la bibliothèque "matplotlib.pyplot" pour représenter graphiquement l'évolution du temps que met l'algorithme à s'effectuer en fonction de la taille n des données d'entrée.

II. TP1

A. Hanoi



TOUR A TOUR B TOUR C

Il y a 3 tours : A,B,C. Et n disques de différentes tailles sur la tour A. Un disque ne peut être placé que sur un disque de taille inférieure (ou le socle d'une tour). Un disque doit être positionné sur une tour. L'objectif est de déplacer tous les disques sur la tour C, en les déplaçant un par un.

1. Pseudo code de notre programme et son analyse:

```
Procédure Hanoi(n,A,B,C)
Entrées : n : entier;  A,B,C : caractères
-.1 début
-.2    // A : départ, B : intermédiaire, C : Arrivée ;
-.3    si (n = 1) alors
-.4           écrire(" Transférer de ", A, " vers ", C )
-.5    sinon
-.6           Hanoi (n - 1, A, C,B) ;
-.7           Ecrire (" Transférer de ", A, " vers ", C) ;
-.8           Hanoi (n - 1, B, A, C);
-.9 fin
```

Nous avons réutilisé la procédure vue en TD.

```
Procédure Tracer_graph(T)
Entrées : t : tableau de réels
Variables : N : liste d'entiers de 1 à 30
-.1 début
-.2    tracer le graph de T en fonction de N
-.3    mettre le titre à "Temps d'exécution des tours de Hanoï pour n allant de 1 à
15"
-.4    mettre le titre de l'axe X à "Nombre de disques (n)"
-.5    mettre le titre de l'axe Y à "Temps d'exécution (secondes)"
-.6    afficher le graph
-.7 fin
```

Nous avons créé une procédure permettant de tracer un graphique, qui utilise la bibliothèque python matplotlib.
Pour tracer ce graphique, nous avons besoin de deux tableaux ou listes de même nombre d'éléments, c'est pourquoi nous initialisons une liste N au début de la procédure.
Ensuite on utilise simplement les fonctions de la libraire qui nous permettent de changer les titres et d'afficher le graphique.

```
Programme Principal:
Variables : start, end : réels; Temps : liste de réels
-.1 début
-.2    pour i allant de 1 à 30, faire:
-.3           start ← démarrage du temps
-.4           Hanoi(i, 'A', 'B', 'C', D')
-.5           end ← fin du temps
-.6           ajouter à Temps ← end - start
-. 7          afficher("Temps d'exécution pour {i} disques : {Temps[i-1]:.10f}
secondes")
-.8    Tracer_graph(Temps)
-.9 end
```

Dans notre programme principal nous exécutons simplement la procédure de Hanoi le nombre de fois demandé par l'utilisateur, en mesurant à chaque fois le temps mis par la machine pour résoudre le problème avec i disques.
On ajoute ensuite le temps dans la liste Temps, créée au début du programme.
Enfin on appelle la procédure Tracer_graph pour tracer le graphique des exécutions du problème des tours de Hanoi à i disques, en fonction du temps.

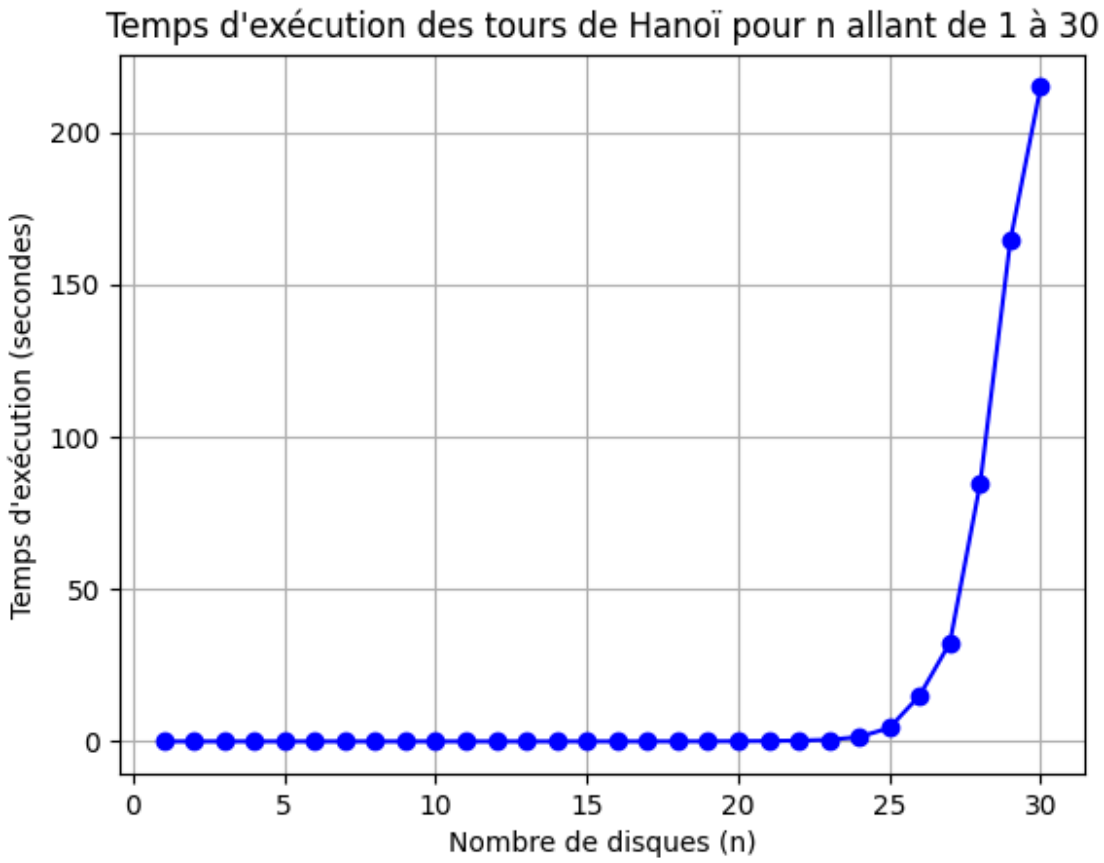
2. Résultats théoriques en terme de complexité

En TD nous avons pu déterminer que la complexité théorique du problème des tours de Hanoi était:

$$T(n) = \theta(2^n)$$

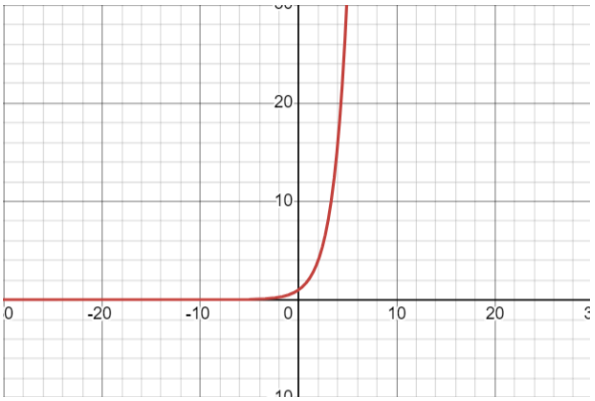
3. Résultats numériques obtenus

Voici le graphique obtenu pour les exécutions du problème des tours de Hanoi de 1 à 30 disques, en fonction du temps:



4. Comparaison entre les résultats théoriques et pratiques

On peut observer que la courbe obtenue après l'exécution du problème des tours de Hanoi à 30 disques, ressemble énormément à la courbe de $f(x) = 2^x$:

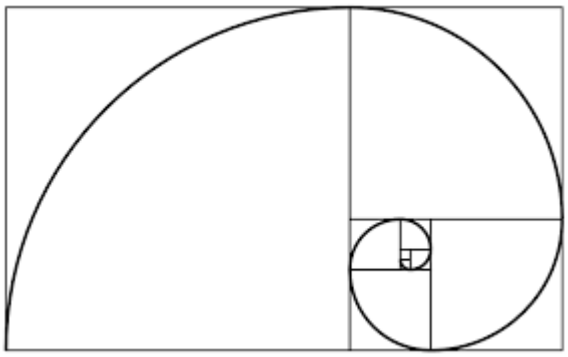


5. Conclusion

En conclusion, l'analyse des tours de Hanoï confirme la complexité exponentielle théorique de l'algorithme.

Cette complexité exponentielle montre clairement que l'algorithme devient impraticable pour des valeurs élevées de n, avec un temps d'exécution qui explose rapidement. Cette limitation souligne donc l'inefficacité de l'algorithme pour un grand nombre de disques, et l'inefficacité en général pour des problèmes de même nature, exponentielle.

B. Presque Fibonacci



On veut étudier la suite de Fibonacci décalée d' un terme. Notre suite est définie par

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

1. Pseudo code de notre programme et son analyse:

```
Fonction Fibonacci_itératif(n)
Variables:
-.1 début
-.2    si n<0, alors:
-.3           retourner “erreur”
-.4    si n = 0 OU N = 1, alors:
-.5           retourner 1
-.6    sinon:
-.7           u ← 0
-.8           u_1 ← 1
-.9           u_2 ← 1
-.10          pour i allant de 1 à n-2, faire:
-.11                 u ← u_1 + u_2
-.12          u_1 ← u
-.13          u_2 ← u_1
-.14   retourner u
-.15 fin
```

Pour la version itérative du problème, on traite d’abord les cas spéciaux, puis on utilise l'algorithme général qui permet de trouver le n_iéme terme de la suite de Fibonacci.

```
Fonction Fibonacci_récursif(n)
-.1 début
-.2    si n = 0 OU N = 1, alors:
-.3           retourner 1
-.4    sinon
-.5           retourner ( (Fibonacci_récursif(n-1) + (Fibonacci_récursif(n-2) )
-.6 end
```

La version récursive fonctionne sur la base du même algorithme qu’en itératif, sauf qu’on le fait en récursif.

```
Fonction Fibonacci_logarithmique(n)
Variables: M, K : tableaux d'entiers
-.1 début
-.2    M ← [1,1,1,0]
-.3    K ← power_2x2(M,n)
-.4    retourner(K[1])
-.5 end
```

Pour la fonction logarithmique, on utilise une procédure de multiplication de matrice ainsi qu'une procédure d'exponentiation rapide de matrice (power_2x2), dont les pseudo codes n'ont pas été détaillés ici, mais qui sont simplement les algorithme classiques que l'on peut trouver sur Wikipedia par exemple.

```
Programme Principal:
Variables : start, end : réels; Temps : liste de réels
-.1 début
-.2    pour i allant de 1 à 30, faire:
-.3          start ← démarrage du temps
-.4          Fibonacci((itératif, récursif ou logarithmique))
-.5          end ← fin du temps
-.6          ajouter à Temps ← end - start
-. 7         afficher("Temps d'exécution pour {i} disques : {Temps[i-1]:.10f}
secondes")
-.8    Tracer_graph(Temps)
-.9 end
```

Comme pour les tours de Hanoi, le programme principal est le même, sauf que l'on appelle une des trois fonctions de Fibonacci, en le demandant à l'utilisateur (ce n'est pas montré en pseudo-code ici car encombrant).

2. Résultats théoriques en terme de complexité

Itératif: On utilise une simple boucle pour calculer les termes de la suite de Fibonacci, et qui permet de calculer la valeur en n étapes

$$T(n) = \theta(n)$$

Récursif: En approche récursive, à chaque appel de la fonction, nous faisons deux appels supplémentaires (pour n-1 et n-2), menant à une explosion exponentielle d'appels.

$$T(n) = \theta(2^n)$$

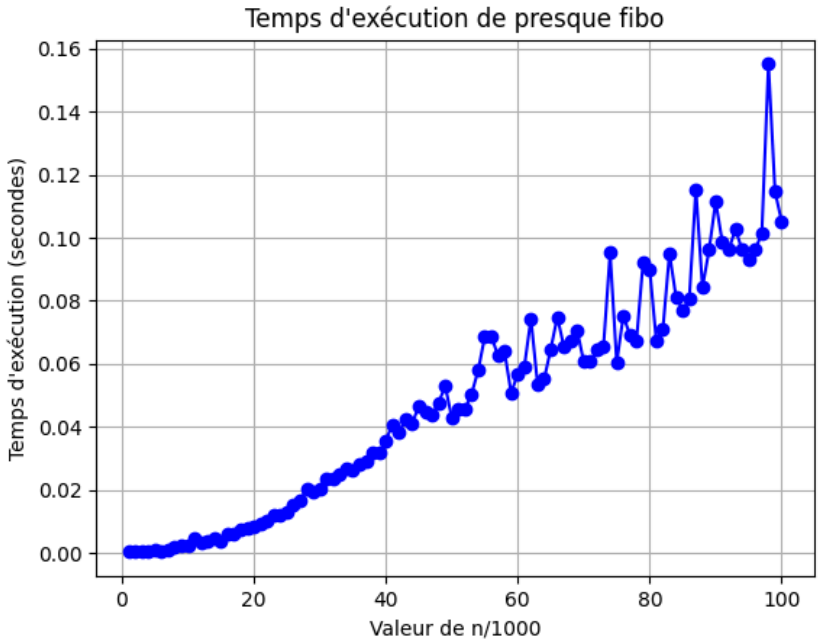
Logarithmique: En approche logarithmique on utilise l'exponentiation rapide des matrices, qui permet de calculer directement le n-ième terme d'une matrice, en effectuant seulement log(n) multiplications.

$$T(n) = \theta(log_2(n))$$

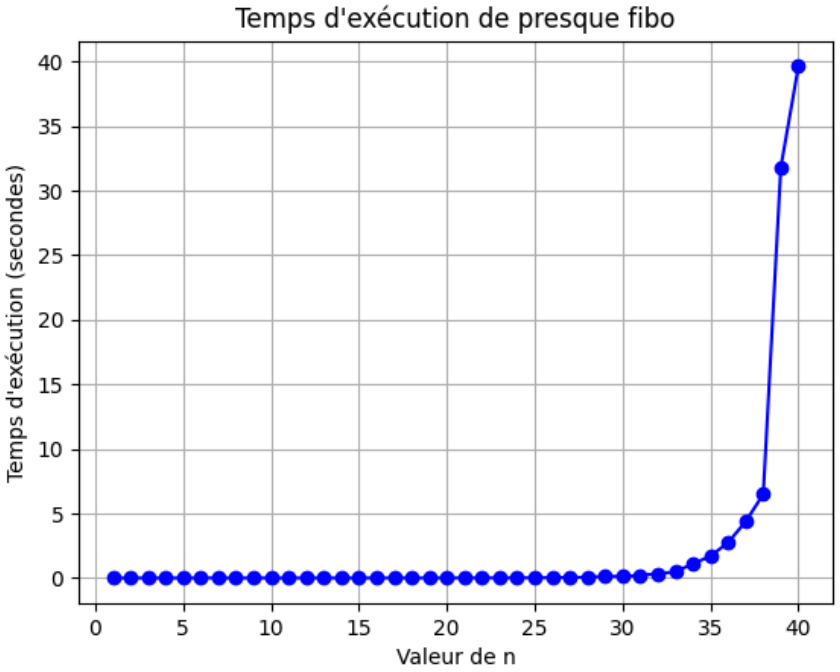
3. Résultats numériques obtenus

Voici les graphiques obtenus pour les exécutions du problème de fibonacci des trois différentes manières, en fonction du temps:

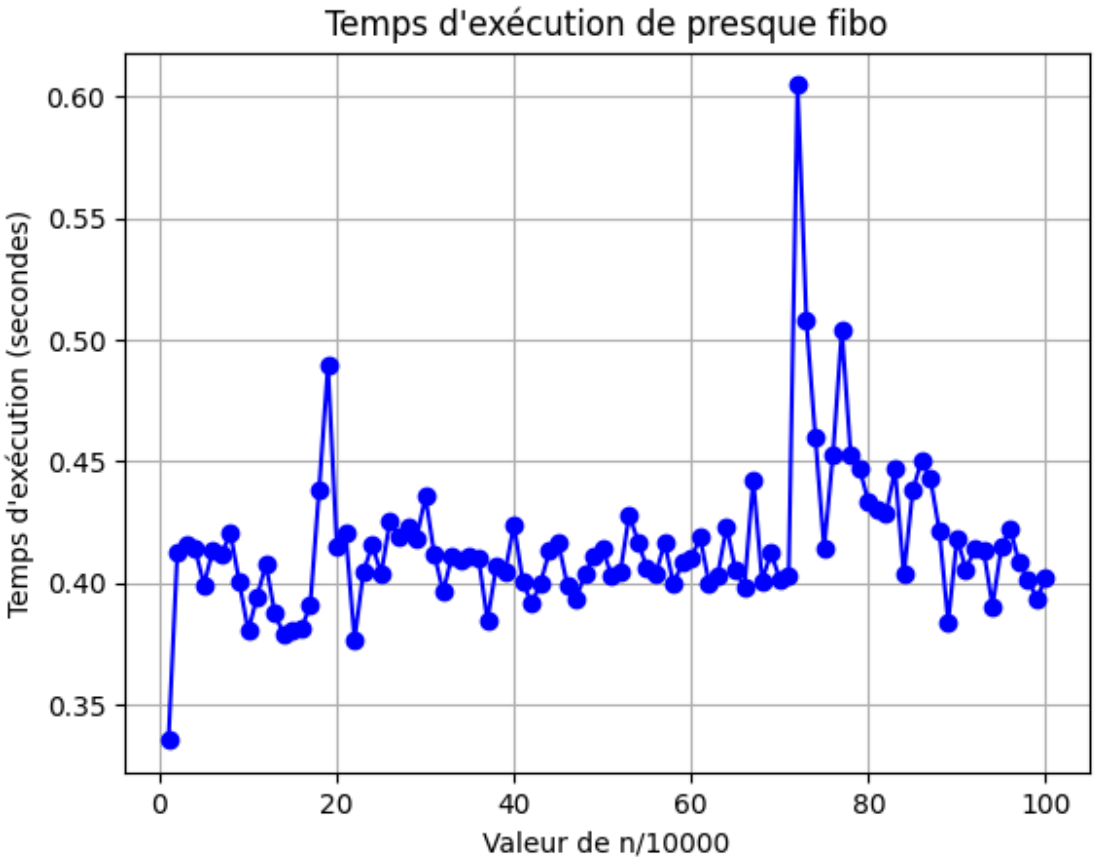
1. Exécution itérative pour n=100 000



2. Exécution récursive pour n=40



3. Exécution logarithmique pour n=1 000 000



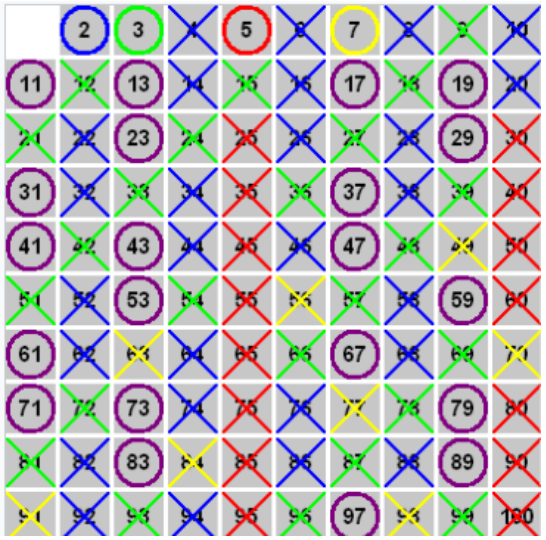
4. Comparaison entre les résultats théoriques et pratiques

Que ce soit pour la version itérative, récursive ou logarithmique, les courbes obtenues sont similaires aux résultats attendus d'après les calculs théoriques. Un courbe linéaire d'ordre n pour l'approche itérative, une courbe d'ordre 2^n pour l'approche récursive, et enfin une courbe logarithmique pour l'approche logarithmique.

5. Conclusion

Finalement, on aurait pu naïvement penser que l'approche récursive allait être la plus efficace, cependant dans notre problème, la récursivité implique de recalculer les mêmes valeurs à chaque nouveau terme de la suite tous les termes, ce qui est très laborieux.
On a pu donc voir que l'approche plus "simple" qu'est la manière itérative permettait de calculer les termes pouvant aller dans les centaines de milliers, mais nous avons ensuite pu constater que l'approche logarithmique était encore plus efficace et qu'elle nous permettait de monter jusqu'au million !

C. Crible d'Erathostène



Le crible d'Eratosthène est le suivant :

On parcourt les entiers de 2 à N et on crible tous les multiples de 2 (sauf 2)

Puis, on parcourt les entiers de 3 à N on crible tous les multiples de 3 (sauf 3)

On réitère l'opération jusqu'à atteindre N.

Les entiers non criblés sont des nombres premiers.

Complexité du Crible d'Eratosthène:

Fonction Eratosthène(n)

```
Variables: L : tableau d'entiers
-.1 début                                                    // (1)
-.2   L ← liste de taille (n + 1) remplie de 0               // O(n)
-.3   L[0] ← 1                                               // O(1)
-.4   L[1] ← 1                                               // O(1)
-.5
-.6   Pour i de 2 à ⌊√n⌋ faire                                      // O(√n)
-.7       Si L[i] = 0 alors                                  // O(1)
-.8           Pour j de i + 1 à n faire                 // O(n/i)
-.9               Si j mod i = 0 alors                  // O(1)
-.10                  L[j] ← 1                           // O(1)
-.11              Fin Si
-.12          Fin Pour
-.13      Fin Si
-.14  Fin Pour
-.15
-.16  Supprimer L[0]                                     // O(n)
-.17  retourner(L)                                       // O(1)
-.18 fin                                                 // O(1)
```

La somme des itérations de les deux boucles imbriquées pour tous les i est approximativement:

$\sum_{i=2}^{\sqrt{n}} \frac{n}{i}$ ce qui équivaut à une complexité $T(n) = \theta(n * log_2(log_2(n)))$

III. TP3

A. Introduction

Nous disposons de 28 plaques :
[1, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 6, 7, 7, 8, 8, 9, 9, 10, 10, 25, 25, 50, 50, 75, 75, 100, 100]
Ainsi que d'un nombre R aléatoire compris entre 100 et 999
L'objectif est d'atteindre R avec 6 plaques et en utilisant des opérations.
Les opérations disponibles sont l'addition, la multiplication, la soustraction (seulement si le résultat est positif) et la division (seulement si le résultat est entier). Chacune des 6 plaques ne peut être utilisée qu'une fois, mais on peut ne pas utiliser toutes les plaques.

Nous avons codé un programme qui nous renvoie la liste des opérations à effectuer pour trouver R à partir d'une liste de 6 entiers.

B. Les différentes fonctions

Principe
La fonction principale est la fonction Q3_ResultatAtteignable(P,R)
Les fonctions précédentes servent à calculer les résultats intermédiaires nécessaires.
Notons PN, la liste initiale, Pn la liste de taille n, avec 0< n <N+1.
Notre algorithme est itératif et exhaustif.
À partir de PN, il calcule tous les PN-1 et les stocke dans un document texte. C'est-à-dire toutes les listes de tailles N-1 que l'on peut former à partir des règles de calculs autorisées.
L'algorithme parcourt le document texte, récupère les PN-1, regarde si R est atteint, si ou il s'arrête, sinon il calcule les PN-2 et les stocke dans un deuxième document texte. Le processus est répété jusqu'à ce que R soit atteint où jusqu'à ce que les Pn soient de taille 1.
L'algorithme retourne toutes les opérations effectuées pour atteindre R si R est atteignable, et un message d'erreur sinon.
Seules les fonctions principales Q2_constructionPn et Q3_ResultatAtteignable(P,R) seront détaillées ici. L'entièreté des fonctions sera disponible en annexe, commentée.

REMARQUE
Le programme marche mais en fonction de la machine, il peut y avoir un souci sur le chemin d'accès aux fichiers textes. Il faut dans ce cas préciser le chemin.

Question 1

```
def CopieListe(L):
# PREND    : liste L
# RETOURNE : une copie de la liste L, indépendante de l'originale.
```

Utilisée pour créer une copie d'une liste, afin de pouvoir modifier la copie sans modifier l'originale.
Complexité Un parcours de la liste, complexité en len(L).

```
def Q1_ConstruireCouple(P):
# PREND    : liste P = Pn avec len(P)>=2
# RETOURNE : liste(liste(C[0],C[1],a,b..)) FINAL = liste de tous les
(couples + reste des éléments)
```

Nous permet d'obtenir tous les couples possibles dans P. Les couples prennent une fois chaque valeur de P (donc possiblement deux fois la même valeur, si elle est présente en double dans P). Et ce, sans notion d'ordre : ainsi pour les entiers 3 et 6, il n'y aura qu'un seul couple (3,6) et pas (3,6) et (6,3).
La fonction renvoie la liste, avec les deux premiers éléments permutés avec le couple.
Complexité Notons p=len(P). La création des couples se fait par double parcours avec des boucles. Soit une complexité en $O(p^2)$

1. Question 2

```
def f(a,b,C,s):
# PREND    : int a = indice 1 ; int b = indice 2, couple C ; char s =
symbole de l'opération
# RETOURNE : char, (C[a] + C[b])
```

C'est une fonction qui a pour but d'alléger l'écriture dans Q2_Calculs couples.
Si on rentre a=1, b=0, C=(7,3), s="+" On obtient la chaine de caractère "(3+7)"
Complexité On peut considérer que le coût est constant

```
def Q2_CalculsCouple(C):
# PREND    : couple C
# RETOURNE : liste(couple(entier,char)) FINAL =
[(somme,"a+b"),(produit,"a*b"),...]
```

Cette fonction prend un couple, et renvoie les 3 ou 4 entiers positifs résultant des opérations possibles avec les règles du jeu "le compte est bon".
Complexité Pour un couple, on regarde si les calculs sont possibles, si oui on les effectue. Le coût est constant

```python
def Q2_constructionPn(P):
# PREND    : liste P = Pn
# RETOURNE : liste(couple(liste,char)) FINAL = [(Pn-1,calcul pour obtenir
Pn-1),(Pn-1,calcul pour obtenir Pn-1)...]
    FINAL=[]
    L = Q1_ConstruireCouple(P)
    l = len(L)

    for i in range(l):
        C = L[i][0],L[i][1]
        M = Q2_CalculsCouple(C)
        m = len(M)

        for j in range(m):
            # M = [(somme,"a+b"),(produit,"a*b"),...]
            # L[i][:2] est Pn auquel on a enlevé le couple d'élément
            TEMP = ([M[j][0]] + L [i][2:] , M[j][1])
            FINAL.append(TEMP)

    return(FINAL)
```

A partir de Pn (n>1), cette fonction construit les Pn-1 en utilisant Q1_ConstruireCouples et Q2_CalculsCouples.

Complexité Soit p=len(P) la complexité est en $O(p^2) + O(p! / 2(p-2))*4$

Soit $O(p^2 + (p-1)!)$

2. Question 3

```
def Q3_ResultatAtteignable(P,R):
# PREND    : liste P = la liste d'origine ; entier R = résulat à atteindre
# RETOURNE : char calculs = tous les calculs effectués

 # Cas particulier
    if R in P :
        return("R est déjà dans la liste")

 # Initialisation
    # on crée nos fichier text de stockage
    Stockage_Pn = open("Complexite_TP3_Stockage1.txt", "w")
    Stockage_Pn_1 = open("Complexite_TP3_Stockage2.txt", "w")

    # on ajoute P au bon format dans Stockage1.txt
    Stockage_Pn.write(str((P,""))+"\n")
    Stockage_Pn.write("end")

    # on ferme nos fichier
    Stockage_Pn.close()
    Stockage_Pn_1.close()


 # Algorithme
    # on veut s'arrêter quand les Pn sont de tailles 1. on effectue donc N-1 fois
l'algorithme
    N = len (P)
    for i in range (1,N):
        # on ouvre les fichiers
        if (i%2==0) :
            Stockage_Pn = open("Complexite_TP3_Stockage2.txt", "r")
            Stockage_Pn_1 = open("Complexite_TP3_Stockage1.txt", "w")
        else :
            Stockage_Pn = open("Complexite_TP3_Stockage1.txt", "r")
            Stockage_Pn_1 = open("Complexite_TP3_Stockage2.txt", "w")

        # on lit Stockage_Pn ligne par ligne
        ligne_texte = Stockage_Pn.readline()
        while ligne_texte != "end" :
            ligne_eval = eval(ligne_texte) # = couple(Liste = Pn, char = calculs)
            ligne_texte = Stockage_Pn.readline()

            # On profite de la lecture pour tester si R est trouvé
            if R in ligne_eval[0]:
                return(ligne_eval[1])

            # On construit et stock les Pn-1 dans le deuxième fichier text
            Constr_Pn_ligne = Q2_constructionPn(ligne_eval[0]) # =
liste(couple(liste,char))
```

```
            a = len(Constr_Pn_ligne)
           # On met à jour la chaîne de caractère des différentes
opérations effectuées
            for j in range(a):
                couple_temp = (Constr_Pn_ligne[j][0], ligne_eval[1] +
Constr_Pn_ligne[j][1])
                Stockage_Pn_1.write(str(couple_temp)+"\n")

        Stockage_Pn.close()
        Stockage_Pn_1.write("end")
        Stockage_Pn_1.close()

 # Vérification finale pour savoir si R est dans les P1
    # on ouvre Stockage_Pn_1 en mode lecture
    if (i%2==0) :
        Stockage_Pn_1 = open("Complexite_TP3_Stockage1.txt", "r")
    else :
        Stockage_Pn_1 = open("Complexite_TP3_Stockage2.txt", "r")

    # on parcourt les P1 à la recherche de R
    ligne_texte = Stockage_Pn_1.readline()
    while ligne_texte != "end" :
        ligne_eval = eval(ligne_texte)
        ligne_texte = Stockage_Pn_1.readline()
        if ligne_eval[0][0] == R :
            return(ligne_eval[1])

    Stockage_Pn_1.close()

    # si R n'est pas trouvé à cette dernière étape alors
    return("R ne peut pas être atteint")
```

L'initialisation crée les fichiers textes, ou les réinitialise s' ils existent déjà, et rentre PN dans un format particulier afin de pouvoir démarrer l'algorithme.
Le format des fichiers textes est le suivant : chaque ligne comporte un couple contenant Pn, et toutes les opérations effectuées pour obtenir Pn.
L'algorithme lit le fichier Pn ligne par ligne regarde si R est atteint, calcule les Pn-1 correspondant à chaque ligne et les stocke dans le fichier Pn-1. On met à jour la chaîne de caractère correspondant aux opérations effectuées quand on écrit les Pn-1 dans le fichier Pn-1 (soit les opérations de Pn, plus l'opération pour atteindre le Pn-1 en question).
L'algorithme s'arrête si R est trouvé, où si Pn devient de taille 1.
Vérification finale : Comme la recherche de R se fait avant le calcul des Pn-1, cette recherche ne s'applique pas aux P1. Il est donc nécessaire de les vérifier à part, à la fin.
Si R n'est pas trouvé après cette étape, il n'est pas atteignable, la fonction s'arrête et renvoie "R ne peut pas être atteint".

Remarque : on utilise deux fichiers textes stockage1 et stockage2 qui contiennent tour à tour les Pn et les Pn-1. Cette méthode est utilisée pour éviter de devoir créer un fichier texte par Pn, pour ne pas utiliser des emplacements mémoire pour rien.

Complexité soit N=len(PN) et n=len(Pn)

Cas particulier O(N)

Initialisation coût constant

Algorithme La complexité est de la forme suivante

$$\sum_{i=0}^{N-2} (4)^{i} * \left(\prod_{k=0}^{i-2} \binom{N-k}{2} * [(N-i+1) + (N-i)^{2} + (N-i+1)!\ ] \right)$$

(1)(2) (3) (4) (5) (6)

(1) la complexité dépend du nombre de Pn, qui change à chaque itération. On somme.
(2) Il y a 3 ou 4 opérations possibles.
(2) et (3) Taille de Pn à l'étape i
(4) Test de présence de R
(5) et (6) Q2_constructionPn

Ce qui se simplifie :

- Entre les termes (4) (5) et (6), on garde le plus important, c'est à dire la factorielle.
- $$\binom{N-k}{2} = \frac{(N-k)!}{(N-k-2)! * 2!} = \frac{(N-k)(N-k-1)(N-k-2)!}{2 * (N-k-2)!} = \frac{(N-k)(N-k-1)}{2}$$
- $$\prod_{k=0}^{i-2} \frac{(N-k)(N-k-1)}{2} = \left(\frac{1}{2}\right)^{i} \frac{N!\,(N-1)!}{(N-i+2)!\,(N-i+1)!}$$
- $$\left(\frac{1}{2}\right)^{i} \frac{N!\,(N-1)!}{(N-i+2)!\,(N-i+1)!} * (N-i+1)! \leq \left(\frac{1}{2}\right)^{i} \frac{(N!)^{2}}{(N-i)!}$$
- $$\sum_{i=0}^{N-2} (4)^{i} \left(\frac{1}{2}\right)^{i} \frac{(N!)^{2}}{(N-i)!} \leq \sum_{i=0}^{N-2} (2)^{i} (N!)^{2} \leq (2)^{N} (N!)^{2}$$

  On peut considérer que l'algorithme est en complexité $O(2^{N}(N!)^{2})$

Vérification finale on parcourt tous les P1 (donc i=N-2 avec la formule précédente), il y en a au maximum

$$4^{N-2} * \prod_{k=0}^{N-2} \binom{N-k}{2}$$

soit pour N=6, il y en a 691 200

3. Question 4

Les fonctions ont été adaptées pour stocker et renvoyer une chaîne de caractère contenant tous les calculs effectués.

4. Question 5

```
def Q5_CreerPlaque():
# RETOURNE : liste L = les plaques à utiliser
    L=[]
    for i in range(1, 11):
        L.append(i)
        L.append(i)
    for i in range (1,5):
        L.append(i*25)
        L.append(i*25)
    return(L)

def Q5_Plaques_R():
    INDICES = []
    P = []
    L = Q5_CreerPlaque()
    l = len(L)
    while len(INDICES)<6 :
        a = rd.randint(0,l-1)
        if a not in INDICES :
            INDICES.append(a)
            P.append(L[a])
    R = rd.randint(100,999)
    return(P,R)
```

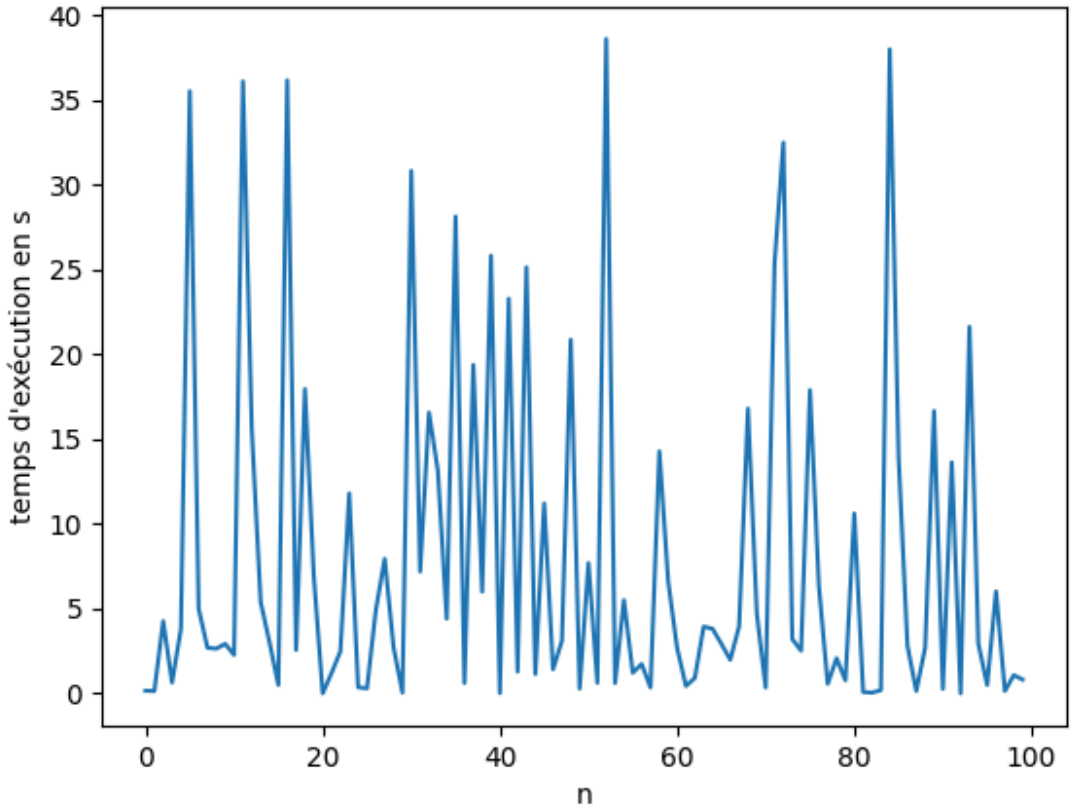
On utilise la fonction randint pour obtenir nos entiers aléatoires.
On accepte un nouvel entier aléatoire que s' il est différent des tirages précédents. Cette méthode n’est pas à privilégier pour un grand nombre d’entier (car la probabilité de tomber sur un entier déjà tiré serait grande) mais reste acceptable pour un tirage de 6 entiers parmi 28.

C. Temps moyen d’exécution du programme

On utilise le programme suivant pour appliquer l’algorithme Q3_ResultatAtteignable pour plusieurs PN et plusieurs valeurs de R

```
#Listes utilisées pour tracer le graphique
X=[]
Y=[]
n = 100

for i in range(n) :
    P,R = Q5_Plaques_R()

    timer = time.process_time()
    TEMP = Q3_ResultatAtteignable(P,R)
    print(i, TEMP)
    timer = (time.process_time() - timer)

    X.append(i)
    Y.append(timer)

# Moyenne
print(Y)

somme = 0
for i in range(n):
    somme += Y[i]
moyenne = (somme/n)
print ("\nLa moyenne de temps d'exécution est",moyenne)

# Affichage de la courbe
plt.plot(X,Y)
plt.xlabel("n")
plt.ylabel("temps d'exécution en s")
plt.show()
```

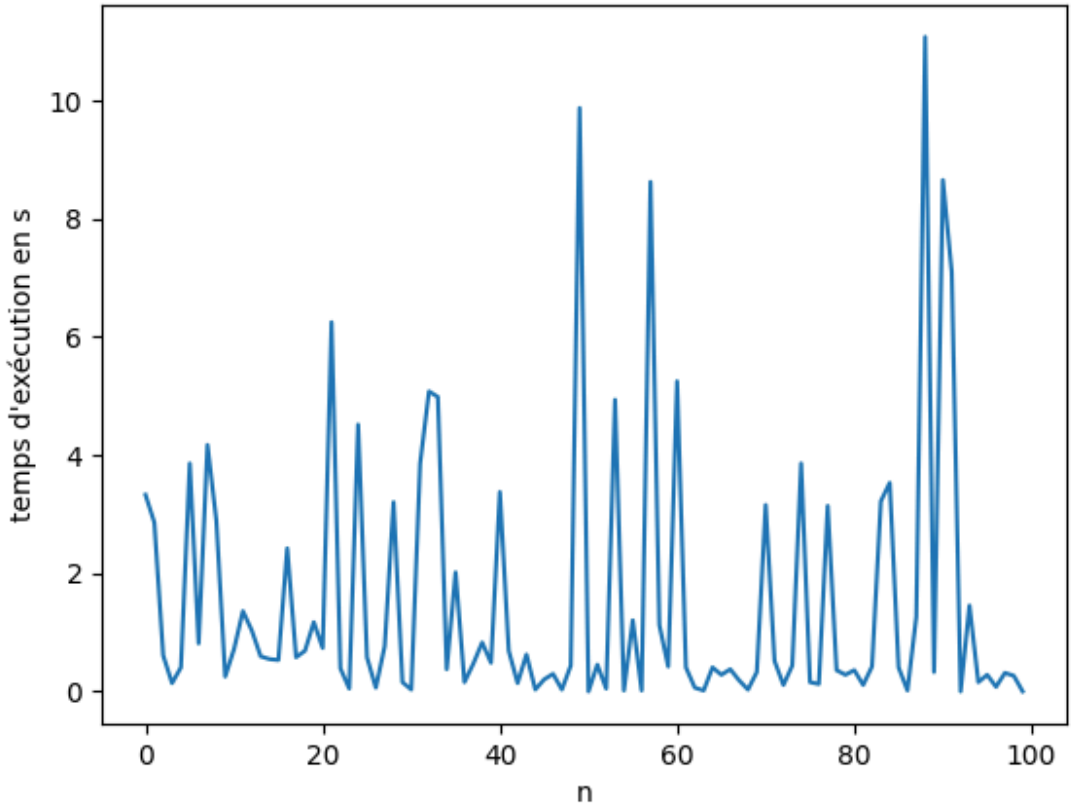
Sur PC portable, le résultat obtenu est :



0.15625	0.140625	4.265625	0.640625	3.8125	35.515625	5.0	2.6875	2.640625	2.921875
2.265625	36.09375	15.53125	5.390625	3.015625	0.5	36.15625	2.5625	17.953125	6.90625
0.0	1.203125	2.484375	11.796875	0.359375	0.296875	4.90625	7.9375	2.71875	0.046875
30.8125	7.1875	16.5625	13.140625	4.40625	28.125	0.609375	19.375	6.0	25.8125
0.015625	23.28125	1.28125	25.125	1.140625	11.203125	1.421875	3.09375	20.859375	0.28125
7.6875	0.625	38.59375	0.609375	5.5	1.21875	1.734375	0.34375	14.28125	6.59375
2.6875	0.4375	0.90625	3.9375	3.8125	2.921875	1.984375	3.953125	16.796875	4.6875
0.34375	25.359375	32.484375	3.203125	2.515625	17.890625	6.34375	0.5625	2.078125	0.765625
10.609375	0.078125	0.046875	0.1875	37.984375	13.90625	2.8125	0.140625	2.703125	16.65625
0.265625	13.625	0.0	21.625	2.9375	0.5	6.015625	0.140625	1.078125	0.8125

La moyenne de temps d'exécution est 7.98625

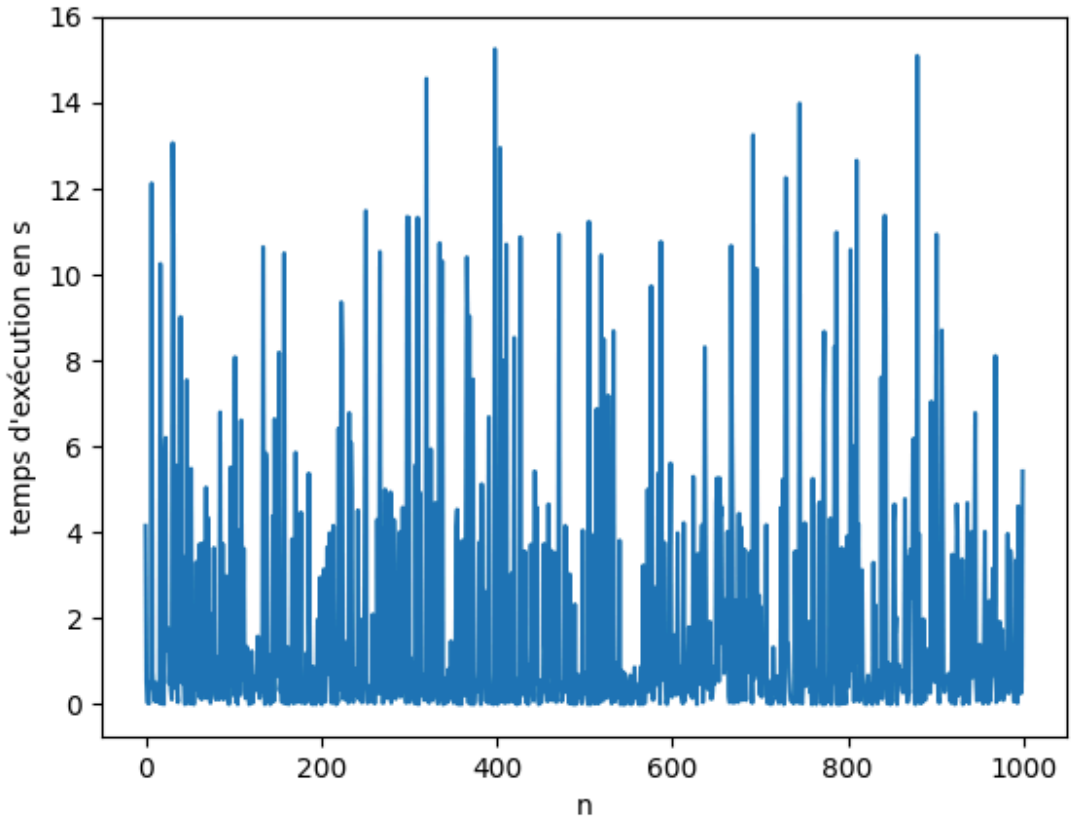
Sur PC tour, pour le résultat obtenu est :



3.328125	2.859375	0.609375	0.140625	0.40625	3.859375	0.8125	4.171875	2.890625	0.25
0.71875	1.359375	1.03125	0.59375	0.546875	0.53125	2.421875	0.578125	0.6875	1.171875
0.734375	6.25	0.390625	0.046875	4.515625	0.578125	0.0625	0.765625	3.203125	0.15625
0.03125	3.859375	5.078125	4.984375	0.375	2.015625	0.15625	0.46875	0.828125	0.484375
3.375	0.6875	0.140625	0.625	0.03125	0.203125	0.296875	0.03125	0.4375	9.875
0.0	0.453125	0.046875	4.9375	0.015625	1.203125	0.015625	8.625	1.125	0.421875
5.25	0.40625	0.0625	0.015625	0.40625	0.28125	0.375	0.1875	0.03125	0.328125
3.15625	0.515625	0.109375	0.4375	3.859375	0.15625	0.125	3.140625	0.359375	0.28125
0.359375	0.109375	0.421875	3.21875	3.53125	0.40625	0.015625	1.265625	11.078125	0.328125
8.65625	7.09375	0.0	1.453125	0.15625	0.28125	0.078125	0.3125	0.265625	0.0

La moyenne de temps d'exécution est 1.55015625 s

Sur PC tour, pour n=1000 on obtient :



La moyenne de temps d'exécution est 1.71428125 s

D. Autres pistes

Les principales difficultés rencontrées sont liées au grand volume de données à manipuler. La première version du programme stockait les données dans une liste, mais la liste devenait trop grande, et le programme cessait de fonctionner. Une deuxième version récursive du programme a été essayée, mais la gestion simultanée du stockage des opérations et de la récursivité s'est avérée complexe. Et de même, le nombre d'appels récrusifs devenait trop important pour le programme.
La solution adoptée a été de passer par l'intermédiaire de fichiers textes avec un programme itératif (pour plus de simplicité pour l'estimation de la complexité), mais il doit sans doute exister une façon plus optimisée de stocker les valeurs et les calculs, car il y a beaucoup de redondance dans le stockage des données.