GRELAUD Jérémy L1
DUMAS Guillaume

Projet N°01 – Python

Le nombre de SCHUR S(n)

1, 6, 10, 18, 21, 23, 26, 30, 34, 38, 43, 45, 50, 54, 65, 74...

2, 3, 8, 14, 19, 20, 24, 25, 36, 46, 47, 51, 62, 73...

4, 5, 15, 16, 22, 28, 29, 39, 40, 41, 42, 48, 49, 59...

7, 9, 11, 12, 13, 17, 27, 31, 32, 33, 35, 37, 53, 56, 57, 61, 79...

44, 52, 55, 58, 60, 63, 64, 66, 67, 68, 69, 70, 71, 72, 75, 76, 77, 78, 80...

SOMMAIRE

1 INTRODUCTION 3

2 Stratégie de réalisation du projet 4

3 Programmes de préparation 5

3.1 Description des variables 5

3.2 Manipulation des couleurs et colorations : solutions et algorithmes 6

3.2.1 Coloration des N premiers nombres entiers en n couleurs choisies aléatoirement 6

3.3 Programmes nécessaires au programme final : solutions et algorithmes 8

3.3.1 Affichage de la liste de tous les triplets 8

3.3.2 Vérification de la liste des triplets : VRAI s'ils sont tous non monochromatiques 9

3.3.3 Conversion d'un nombre décimal en base b 10

3.3.4 Affichage de toutes les colorations possibles à partir de n couleurs pour N nombres entiers 12

4 Calcul du nombre de Schur : réalisation finale 14

4.1 Programme de calcul de S(n) 14

4.1.1 Description des variables 14

4.1.2 Solution et algorithme 14

4.2 Résultats 16

4.3 Optimisation 16

4.4 Encadrement de S(n) pour n>=6 17

5 Conclusion 18

1 INTRODUCTION

Commençons par rappeler ce qu'est le nombre de Schur :

En 1916, le mathématicien juif allemand Issai Schur (1875-1941) a inventé une suite de nombres S(1),S(2),S(3),…. si intrigants qu'ils restent un mystère pour nous.

Ces nombres de Schur sont définis par le théorème suivant :

Théorème (Schur, 1916). Pour n donné, il existe un nombre $S(n)$ défini ainsi : c'est le plus grand entier N pour lequel il est possible de colorier en n couleurs les entiers de 1 à N de façon à exclure tout triplet monochromatique de la forme $(a, b, a+b)$ avec $a, b, a+b$ compris entre 1 et N.

Les connaissances actuelles sur les *nombres de Schur* restent limitées et il existe une conjecture à partir de S(6) :

$$\frac{3^n - 1}{2} \leq S(n) \leq 3 \times n! - 1$$

Nous ne connaissons que les quatre premiers, et à partir du sixième, on estime les valeurs à partir de l'intervalle obtenu sur l'image ci-dessus.

Nous allons donc tenter de calculer les quatre premiers nombres de Schur !

2 STRATEGIE DE REALISATION DU PROJET

Nous allons détailler la démarche adoptée lors de ce premier projet en Python. Il nous a d'abord fallu relire les premières pages du projet avec les exemples et les explications pour comprendre le but de ce projet.
Ensuite, nous avons traité tous les exercices d'échauffement et de préparation.
Lors de cette partie nous avons établi un cahier des charges pour définir l'utilisation et la correspondance de tous nos noms de variables lors de ce projet.
Nous détaillerons dans la suite de ce rapport les différents choix de structure de variables et les solutions algorithmiques mises en œuvre pour mener à bien ce projet.

Le travail préparatoire nous a permis d'identifier les étapes importantes menant au programme final :

- Convertir tous les nombres entiers entre 0 et le nombre de colorations possibles-1, en base n dans le cas de n couleurs choisies. On associera ensuite une coloration correspondant à la liste des bits générés.
- Générer toutes les colorations possibles en fonction d'un nombre de couleurs n et d'un nombre d'entiers N.
- Former les triplets en suivant les règles de Schur, à partir de N entiers.
- Vérifier si les triplets sont non monochromatiques.

Après ce travail préparatoire, nous avons pu passer aux applications et à la réalisation finale.

Pour ce projet, nous avons utilisé le logiciel Thonny et le langage Python.

3 PROGRAMMES DE PREPARATION

Les exercices de programmation proposés en 3.2 et 3.3 nous ont permis de nous familiariser avec la manipulation des couleurs et des colorations d'entiers, et de comprendre les principales difficultés du projet.

Dans ce rapport, nous détaillerons seulement les programmes les plus complexes et les plus importants pour le programme final.

3.1 DESCRIPTION DES VARIABLES

Nous avons choisi pour l'ensemble de nos programmes de normaliser le type de données et le nom des variables utilisés.
Pour une même donnée et une même utilisation, on utilise le même type et le même nom de variable.
Nous avons donc utilisé les structures de données et variables suivantes :

color	Liste des couleurs utilisables (souvent au nombre de 6) (liste de codes couleurs)
my_color	Liste des couleurs à utiliser pour la coloration (liste de codes couleurs) Cette liste est souvent définie aléatoirement à partir de **color**
alea_color	Index de la couleur choisie aléatoirement parmi la liste **color** (entier)
color_unsused	Liste des couleurs qui ne sont pas encore utilisées au cours de la génération d'une coloration. (liste de codes couleurs) Cette liste permet de s'assurer de l'utilisation de toutes les couleurs, elle doit être vide à la fin de la génération d'une coloration si toutes les couleurs ont été utilisées
coloration	Liste de taille N contenant les couleurs à associer aux entiers de 1 à N. (liste de codes couleurs)
triplet	Liste constituée des trois valeurs du triplet (a , b , a+b) avec a+b≤p pour un p donné. (liste d' entiers)
liste_triplets	Liste de tous les triplets (liste de listes d'entiers)
quotient	Quotient d'une division euclidienne (entier) Correspond aussi au dividende de la division suivante lors de divisions euclidiennes successives par b pour la conversion d'un nombre décimal en base b
reste	Reste de la division euclidienne du *quotient* par n (entier)

liste_reste	Liste stockant les restes des divisions euclidiennes successives par b, lors de la conversion d'un nombre décimal en base b. (liste d'entiers) Cette liste doit être lue dans l'ordre inverse pour obtenir le nombre en base b
Liste_bits_bonordre	Liste stockant dans le bon ordre les bits correspondant au nombre en base b (liste d'entiers) **liste_reste** inversée

Nous avons choisi d'utiliser des listes lorsqu'il était nécessaire de stocker des données de manière ordonnée pour pouvoir y accéder. Par exemple, « triplet [1] » pour afficher la deuxième valeur du triplet ou encore « coloration [2] » qui représente la couleur associée à l'entier 3.

3.2 MANIPULATION DES COULEURS ET COLORATIONS : SOLUTIONS ET ALGORITHMES

3.2.1 Coloration des N premiers nombres entiers en n couleurs choisies aléatoirement

Exercice 3.2.2 :
Le but est de colorer les N premiers nombres entiers à partir de n couleurs choisies aléatoirement. De plus, les n couleurs doivent toutes être présentes.

Stratégie :
1° étape :
On sélectionne un nombre aléatoire n de couleurs entre 1 et 6, puis ces couleurs sont choisies aléatoirement parmi la liste de couleurs utilisables, *color*. Afin de s'assurer que toutes les couleurs choisies seront différentes, une fois une couleur aléatoire ajoutée à la liste de couleurs à utiliser, *my_color*, on la retire de *color*.

2° étape :
Le nombre d'entiers N est saisi par l'utilisateur. Afin d'éviter une boucle infinie par la suite, on oblige l'utilisateur à saisir un N au moins égal à 6 (le nombre maximal de couleurs utilisables).

3° étape :
On génère une coloration aléatoire des N entiers à partir des couleurs de *my_color.* On recommence cette génération tant que toutes les couleurs de *my_color* n'ont pas été utilisées.
Pour ce faire on utilise la liste *color_unused.* Avant la génération de chaque coloration, on initialise *color_unused* à partir de my_color, puis lors de la génération de la coloration, si une couleur utilisée est encore dans *color_unused*, on la retire. A la fin de la génération, si *color_unused* n'est pas vide, on recommence une nouvelle coloration.

4°étape :
On affiche les N premiers nombres entiers à partir des couleurs de *coloration.*

// Algo 3.2.2 coloration de N entiers en n couleurs aléatoires

Variables locales:

color, tableau de 6 codes couleurs, contient les 6 couleurs utilisables
nb_color_used, entier, nombre de couleurs à utiliser pour la coloration
alea_color, entier, index dans le tableau *color* de la couleur choisie aléatoirement

my_color, tableau de *nb_color_used* codes couleurs, contient les couleurs à utiliser pour la coloration
color_unused, tableau de codes couleurs, contient les codes couleurs restant à utiliser pour la coloration
N, entier, nombre d'entiers à colorer
coloration, tableau de N codes couleurs, contient les codes couleurs des N entiers à colorer
choix_random, code couleur, code couleur choisi aléatoirement dans *my_color*
i,m,n,k, entiers compteurs de boucle

```
DEBUT:
    color <- tableau des 6 couleurs (W,R,G,O,B,P)
    my_color <- tableau vide
    nb_color_used <- nombre aléatoire entre 1 et 6

    Pour i allant de 0 à nb_color_used-1
        alea_color <- nombre aléatoire entre 0 et taille(color)-1
        ajouter color[alea_color] à my_color
        retirer color[alea_color] de color
    Fin pour

    Faire
        Afficher "Saisir le nbr d'entiers à colorer (au moins 6):"
        N <- saisir
    Tant que N<6

    Faire
        color_unused <- []
        Pour k allant de 0 à taille(my_color)-1
            ajouter my_color[k] à color_unused
        Fin pour
        coloration <- []
        Pour n allant de 0 à N-1
            choix_random <- choix aléatoire dans my_color
            ajouter choix_random à coloration
            Si choix_random est dans color_unused
                retirer choix_random de color_unused
            Fin si
        Fin pour
    Tant que taille(color_unused)≠0

    Afficher "nbr de couleurs à utiliser" + (nb_color_used)

    Pour m allant de 1 à N
        Afficher (coloration[m-1],m,end = "")
    Fin pour
Fin
```

3.3 PROGRAMMES NECESSAIRES AU PROGRAMME FINAL : SOLUTIONS ET ALGORITHMES

3.3.1 Affichage de la liste de tous les triplets

Exercice 3.3.2 :
Le but est de générer à partir d'un entier p (p ≤ 100), la liste de tous les triplets (a, b, a+b) avec a+b ≤ p.

Stratégie :
1° étape :
On oblige l'utilisateur à saisir un entier $0 \leq p \leq 100$.

2° étape :
Formation des triplets pour *a* de 1 à *p*. On ne prend en compte que les *b* de *a* à *p*. On ne prend pas en compte les triplets où *a<b*, car l'ordre entre *a* et *b* ne joue aucun rôle. On ne forme enfin un triplet que si $a+b \leq p$.

3° étape :
Affichage de la liste de tous les triplets (a, b, a+b) obtenus.

//Algo de formation des triplets

Variables locales:

liste_triplets, tableau de tableau d'entiers, contient la liste de tous les triplets
triplet, tableau d'entiers, contient les 3 valeurs du triplet (a,b,a+b)
p, entier, entier maximum saisi par l'utilisateur
a,b, entiers, compteurs de boucle

```
DEBUT:
    liste_triplets <- tableau vide
    triplet <- tableau vide

    Faire
        Afficher "Choisir un entier p positif inférieur ou égal à 100:"
        p <- saisir
    Tant que p>100 ou p<0

    Pour a allant de 1 à p
        Pour b allant de a à p
            Si (a+b)<=p
                réinitialisation du tableau triplet
                ajouter a à triplet
                ajouter b à triplet
                ajouter (a+b) à triplet
                ajouter triplet à liste_triplets
            Fin si
        Fin pour
    Fin pour

    Afficher "Pour p=" + (p) + "la liste de tous les triplets de type (a,b,a+b) est: " +
    (liste_triplets)
FIN
```

3.3.2 Vérification de la liste des triplets : VRAI s'ils sont tous non monochromatiques

Exercice 3.3.3 :
Le but est de générer une coloration aléatoire de N nombres entiers à partir de 2 couleurs, et d'afficher VRAI si tous les triplets formés à partir de cette coloration ne sont pas monochromatiques, FAUX sinon.

Stratégie :
1° étape :
On sélectionne aléatoirement un nombre N d'entiers entre 1 et 20 et on génère une coloration aléatoire de ces entiers à partir de 2 couleurs (R et B) en s'assurant que les 2 couleurs sont utilisées (même méthode que l'algorithme 3.2.2)

2° étape :
Formation des triplets (a, b, a+b) avec a+b ≤ N. On génère tous les triplets pour les afficher à la fin (même méthode que l'algorithme 3.3.2). Pour chaque triplet, on teste s'il est monochromatique si on n'a pas encore trouvé de triplet monochromatique. Si on en a déjà trouvé un, le programme affichera de toute façon FAUX.

3° étape :
Affichage de la liste de tous les triplets (a, b, a+b) obtenus.
Affichage de VRAI si tous les triplets ne sont pas monochromatiques, FAUX sinon.

//Algo Vérification de la liste des triplets : VRAI s'ils sont tous non monochromatiques

Variables locales:

choix_random, code couleur, choisi aléatoirement dans *my_color*
my_color, tableau des 2 codes couleurs à utiliser pour la coloration
color_unsused, tableau de codes couleurs, contient les codes couleurs restant à utiliser pour la coloration
coloration, tableau de codes couleurs de taille N , contient les couleurs à associer aux entiers de 1 à N.
entier_max, entier, le nombre N d'entiers à colorer
liste_triplets, tableau de tableau d'entiers, contient la liste de tous les triplets
triplet, tableau d'entiers, contient les 3 valeurs du triplet (a,b,a+b)
i,a,entiers, compteurs de boucle

DEBUT:
```
    my_color <- tableau des 2 couleurs à utiliser
    coloration <- tableau vide
    entier_max <- un entier aléatoire entre 2 et 20

    Faire
        color_unused <- tableau des 2 couleurs à utiliser
        coloration <- tableau vide
        Pour i allant de 0 à entier_max -1
            choix_random <- choix aléatoire dans my_color
            ajouter choix_random à coloration
            Si choix_random est dans color_unused
```

```
                    retirer choix_random de color_unused
                Fin si
            Fin pour
        Tant que taille(color_unused) ≠ 0

        Afficher "La coloration:"
        Pour i allant de 0 à entier_max-1
            Afficher(coloration[i],(i+1))
        Fin pour

        liste_triplets <- tableau vide
        triplet <- tableau vide
        non_monochr <- True

        Pour a allant de 1 à entier_max
            Pour b allant de a à entier_max
                Si (a+b)<=entier_max
                    triplet <- tableau vide
                    ajouter a à triplet
                    ajouter b à triplet
                    ajouter (a+b) à triplet
                    ajouter triplet à liste_triplets
                    Si non_monochr = True
                        Si (coloration[a-1]=coloration[b-1]) ET
                            (coloration[b-1]=coloration[a+b-1])
                            non_monochr<-False
                        Fin si
                     Fin si
                Fin si
            Fin pour
        Fin pour

        Afficher "La liste de tous les triplets est:"
        Pour i allant de 0 à taille(liste_triplets)-1
            triplet <- liste_triplets[i]
            Afficher"("
            Afficher (coloration[triplet[0]-1],triplet[0]) + ","
            Afficher (coloration[triplet[1]-1],triplet[1]) + ","
            Afficher (coloration[triplet[2]-1],triplet[2])
            Afficher ")"
        Fin pour

        Afficher (non_monochr) + ": tous les triplets ne sont pas monochromatiques."
FIN
```

3.3.3 Conversion d'un nombre décimal en base b

Exercice 3.3.8 :
Le but est de convertir un nombre décimal en base b (b>1).

Stratégie :

1° étape :

On réalise une série de divisions euclidiennes successives par l'entier b tant que le quotient obtenu est non nul. Le premier dividende est le nombre décimal, et le quotient d'une division devient le dividende de la suivante. On stocke la liste des restes dans la variable *nbr_baseb* .

Cette liste devra être lue dans le sens inverse pour obtenir les bits du nombre en base b dans le bon ordre.

2° étape :

Formation de la chaîne de caractères correspondant aux bits du nombre en base b dans le bon sens d'affichage. Pour les restes supérieurs à 9, le bit correspondant est une lettre, A pour 10, B pour 11 etc. On utilise donc le code ASCII du caractère désiré pour convertir le nombre en bit.

3° étape :

Affichage de la chaîne de bits correspondant au nombre en base b

//Algo conversion d'un nombre décimal en base b

Variables locales:

nbr_base10, entier, le nombre décimal saisi par l'utilisateur
nbr_baseb, tableau d'entiers, contient les bits en base b, stockés dans le sens inverse
sauvegarde, entier, conservation du nombre saisi par l'utilisateur pour l'affichage final
chaine, chaine de caractères, stocke les bits du nombre en base b dans le bon sens d'affichage
bit, entier, les différents éléments du nombre en base b
i, entier, compteur de boucle

```
DEBUT:
    Afficher "Saisir un nombre décimal positif"
    nbr_base10 <- saisir
    nbr_baseb <- tableau vide
    sauvegarde <- nbr_base10
    Faire
        Afficher "Choisir votre base b (au moins 2)"
        nbr_baseb <- saisir
    Tant que nbr_baseb<2

    Tant que nbr_base10 ≠ 0
        ajouter MOD(nbr_base10,b) à nbr_baseb
        nbr_base10 <- ENT(nbr_base10,b)
    Fin tant que

    chaine <- ""
    Si sauvegarde = 0
        chaine <- "0"
    Fin si

    Pour i allant de taille((nbr_baseb)-1) à 0 avec un pas de -1
        bit <- nbr_baseb[i]
        Si bit >= 10
            bit <- chr(bit+55)
```

```
            Fin si
            chaine += str(bit)
        Fin pour

        Afficher "Le nombre"+(sauvegarde)+"en base"+(b)+"donne"+(chaine)
FIN
```

3.3.4 Affichage de toutes les colorations possibles à partir de n couleurs pour N nombres entiers

Exercice 3.3.9 :
Le but est de stocker la liste de toutes les colorations possibles de N entiers à partir de n couleurs, et d'afficher cette liste à la fin.

QT 3.3.5 : A l'aide de 2 couleurs, le nombre de colorations possibles pour colorier N entiers est 2^N.
QT 3.3.6 : A l'aide de n couleurs, le nombre de colorations possibles pour colorier N entiers est n^N.

Stratégie :
On a vu que pour colorier N entiers avec n couleurs, il y a n^N colorations possibles. Afin d'obtenir toutes ces colorations, il suffit de convertir en base n chaque entier de 0 à n^N-1.
On génère ainsi une liste de N bits compris entre 0 et n-1, ce qui nous donne le rang de la couleur dans la liste *color*.
Par exemple pour n=2 et N=2, les entiers entre 0 et 3 (2^2-1) sont codés de la manière suivante en base 2 : 00, 01, 10, 11.
Si on considère que la couleur rouge a pour rang 0 et la couleur bleue a pour rang 1, on obtient bien les 4 colorations RR, RB, BR, BB

1° étape :
On génère la liste des colorations. On convertit tous les entiers de 0 à n^N-1 en base n selon la méthode utilisée pour l'exercice 3.3.8 . On prend soin de compléter *liste_reste* par des 0 pour obtenir N bits, avant de l'inverser pour obtenir la liste des bits dans le bon ordre. Cette liste correspond à une coloration (elle stocke en fait le rang de la couleur dans *color).*

2° étape :
On affiche chaque coloration de la liste en associant chaque entier de 1 à N à la bonne couleur dans la liste *color*.

//Algo affichage de toutes les colorations possibles pour n couleurs et N nombres entiers

Variables locales:

color, tableau de 6 codes couleurs, contenant les 6 couleurs utilisables
liste_reste, tableau d'entiers , contient les restes des divisions euclidiennes successives par n, lors de la conversion d'un nombre décimal en base n
liste_bits_bonordre, tableau d'entiers, stocke les bits correspondant au nombre en base n
quotient, entier, correspond au quotient d'une division euclidienne et au dividende de la suivante
reste, entier, reste de la division euclidienne du *quotient* par n
i,z,j , entiers, compteurs de boucle
liste_colorations, tableau de tableaux, contient la liste de toutes les colorations

```
DEBUT:
    Afficher "Saisir un nombre de couleur n:"
    n <- saisir
    Afficher "Nombre d'entiers à colorier : "
    N <- saisir
    color <- tableau de 6 codes couleurs
    liste_colorations <- tableau vide

    Pour i allant de 0 à (n**N)-1
        liste_reste <- tableau vide
        liste _bits _bonordre <- tableau vide

        Si i=0
            Pour z allant de 0 à N-1
                ajouter 0 à liste _bits _bonordre
            Fin Pour
        Sinon
            quotient <- i
            Tant que quotient ≠ 0
                reste <- MOD(quotient,n)
                quotient <- ENT(quotient,n)
                ajouter reste à liste_reste
            Fin tant que

            Pour z allant de 0 à N-taille(liste_reste)-1
                ajouter 0 à liste_reste
            Fin pour

            Pour z allant de taille(liste_reste) à 0 avec un pas de -1
                ajouter liste_reste[z] à liste _bits _bonordre
            Fin pour
        Fin si

        Ajouter liste_bits_bonordre à liste_colorations

    Fin pour

    Pour j allant de 0 à n^N-1
        Afficher "Coloration n°"+(j+1)+": "
        Pour k allant de 0 à N-1
            Afficher color[liste _colorations[j][k]] + (k+1)
        Fin pour
    Fin pour
FIN
```

4 CALCUL DU NOMBRE DE SCHUR : REALISATION FINALE

4.1 PROGRAMME DE CALCUL DE S(N)

4.1.1 Description des variables

Parmi les variables utilisées dans les programmes de préparation, on réutilise seulement pour le programme final les variables : *quotient*, *reste*, *liste_reste* et *coloration*. Par contre pour le calcul du nombre de Schur, *coloration* ne contient pas les codes couleurs mais leur rang dans un tableau de couleur quelconque. Cela nous suffit car nous n'avons pas besoin de réellement colorer les entiers (pas d'affichage). C'est pourquoi nous n'utilisons d'ailleurs pas les variables *color* et *my_color*. Nous n'utilisons pas non plus *triplet* et *liste_triplets* car nous générons et testons les triplets à la volée, on ne les stocke pas.

4.1.2 Solution et algorithme

Le but de ce programme est de calculer le nombre de Schur pour un entier n donné. Le nombre de Schur correspond au plus grand entier N pour lequel il est possible de colorier en n couleurs les entiers de 1 à N en excluant tout triplet monochromatique de la forme (a, b, a+b) avec a+b ≤ N.

Stratégie :

On teste la validité des N successifs. Pour chaque N on génère des colorations *(cf méthode de l'exercice 3.3.9)* jusqu'à en trouver une valide. Pour tester la validité d'une coloration on génère les triplets (a, b, a+b) et on teste à la volée s'ils sont monochromatiques *(cf exercice 3.3.3)*. On s'arrête dès qu'on trouve un triplet monochromatique, la coloration est alors non valide, et valide si aucun triplet n'est monochromatique.
Si on a trouvé une coloration valide on passe au N suivant sinon on arrête la boucle pour les N.
Pour le dernier N, aucune coloration n'est valide, donc le nombre de Schur correspond à N-1.

//Algo calcul de S(n)
Variables locales:

N, entier, compteur représentant le nombre d'entiers à colorier à chaque étape
coloration, tableau d'entiers, contient le rang (entre 0 et n-1) de la couleur à associer à chaque entier de 1 à N
quotient, entier, correspond au quotient d'une division euclidienne et au dividende de la suivante
reste, entier, reste de la division euclidienne du quotient par n
liste_reste, tableau d'entiers , contient les restes des divisions euclidiennes successives par n, lors de la conversion d'un nombre décimal en base n
a,b,i,z, entiers, compteurs de boucle
non_monochr, booléen, vrai tant qu'un triplet n'est pas monochromatique
une_coloration_valide, booléen, vrai tant qu'on a au moins une coloration valide pour un N donné
coloration_non_valide, booléen, vrai tant qu'on a pas trouvé de coloration valide lors de la génération des colorations

DEBUT:

```
coloration <- tableau vide

Faire
    Afficher "Saisir le nombre de couleurs (max 6, min 2)"
    n <- saisir
```

```
Tant que n<2 OU n>6

N <- 1
Faire
    i <- 0
    coloration_non_valide <- True
    Tant que coloration_non_valide = True ET i < (n^N)-1
        liste_reste <- tableau vide
        coloration <- tableau vide

        Si i = 0
            Pour z allant de 0 à N-1
                ajouter 0 à coloration
            Fin pour
        Sinon
            quotient <- i
            Tant que quotient ≠ 0
                reste <- MOD(quotient,n)
                quotient <- ENT(quotient,n)
                ajouter reste à liste_reste
            Fin tant que

            Pour z allant de 0 à N-taille(liste_reste)-1
                ajouter 0 à liste_reste
            Fin pour

            Pour z allant de taille(liste_reste)-1 à 0 avec un pas de -1
                ajouter liste_reste[z] à coloration
            Fin pour
        Fin si

        non_monochr <- True

        a <- 1
        Tant que non_monochr = True ET a<N
            b <- a
            Tant que non_monochr = True ET b<N
                Si (a+b)<=N
                    Si non_monochr = True
                        Si coloration[a-1]=coloration[b-1] ET
                          coloration[b-1]=coloration[a+b-1]
                            non_monochr <- False
                        Fin si
                    Fin si
                Fin si
                b <- b+1
            Fin tant que
            a <- a+1
        Fin tant que

        Si non_monochr = True
            coloration_non_valide <- False
```

```
                    Fin si

                    i <- i+1
                Fin tant que

                Si coloration_non_valide = True
                    une_coloration_valide <- False
                Sinon
                    N <- N+1
                Fin si
            Tant que une_coloration_valide = True

            Afficher "Le nombre de Schur pour le n choisit vaut : " + (N-1)
FIN
```

4.2 RESULTATS

QT 3.4.2 : S (2) = 4. En effet, pour 2 couleurs, au N=5 il n'y a aucune coloration valide alors qu'il y en a pour N=4 (cf exemple 3.1), donc le nombre de Schur vaut 4.

QT 3.3.4 : Nous avons réussi à calculer 2 nombres de Schur : S(2) et S(3). Au-delà, les calculs deviennent trop longs à effectuer. Il nous faut déjà environ 69 secondes pour calculer S(3) sur un PC avec un processeur intel core i5 8ème génération (2.30 GHz). Pour S(3), environ 65 secondes sont dédiées à la dernière étape soit N = 14, donc pour traiter 3^{14} colorations. Cela nous donne un temps moyen de 14 µs par coloration. Pour le calcul de S(4), le résultat connu étant 44, la dernière étape pour N = 45 et donc 4^{45} colorations prendrait au moins $5.3*10^{14}$ années.

4.3 OPTIMISATION

Nous avons cherché à optimiser notre programme. Nous avons essayé d'éviter de générer pour chaque N toutes les colorations jusqu'à en trouver une valide, car ceci implique de générer toutes les colorations possibles au dernier N. Nous avons donc stocké pour chaque N uniquement la liste des colorations valides, dont on se sert pour générer les colorations à tester au N suivant. En effet, on a vu dans l'exemple donné en 3.1 que lorsqu'une coloration n'est pas valide à un N donné, toutes les colorations issues de celle-ci à un rang N supérieur sont invalides.

Effectivement, avec notre second programme(v2), pour S(3), les performances sont bien meilleures, le temps de calcul est inférieur à la seconde. Par contre, nous n'arrivons toujours pas à calculer S(4) car pour N = 17 nous rencontrons un problème mémoire dû au dépassement de la taille maximale d'une liste (2147483647).

```
15
16
Traceback (most recent call last):
  File "E:\DUMAS_GRELAUD\3.4.3 -Calcul de S(n) v2.py", line 31, in <module>
    liste_coloration_utile.append(equiv+l)
MemoryError
>>> |
```

Finalement, notre programme initial nous permettait d'éviter la gestion de listes trop longues grâce à la conversion du numéro de chaque coloration en base n, ce qui nous a évité les problèmes liés à la mémoire mais pas ceux de performances. Ce second programme, performant jusqu'à S(3), pose des

problèmes mémoire au-delà. La gestion des listes ne semble pas être une solution, on n'a pas trouvé de compromis.

4.4 Encadrement de S(n) pour n>=6

Nous avons appliqué les formules données en 4.1. La seule difficulté était le calcul de la factorielle dans la borne supérieure. Nous avons appliqué l'algorithme suivant pour le calcul de la factorielle n ! :

```
// Algo factorielle
Variables locales :
fact,i, entiers

DEBUT :
    fact <- 1
    Pour i allant de 1 à n
        fact <- fact * i
    Fin pour
FIN
```

On trouve par exemple 364 ≤ S(6) ≤ 2159

5 CONCLUSION

Ce projet fut intéressant à mener pour plusieurs raisons

Tout d'abord, ce fut l'occasion d'expérimenter une fois de plus l'esprit d'équipe. Après les TPE de première, nous n'avions plus l'occasion de mener un projet à plusieurs avec une date limite. Ce fut ainsi une nouvelle occasion de s'entrainer à travailler à plusieurs, capacité essentielle de l'ingénieur.

Ce fut aussi, bien malgré nous, l'occasion de découvrir une difficulté majeure du travail en groupe : celle des « trop » nombreuses versions de chaque code. Par exemple : 3.3.2 ; 3.3.2v2 ; 3.3.2 v_final ; 3.3.2 v_Final. Cette difficulté supplémentaire nous servira d'enseignement pour la prochaine fois. En effet, c'est durant ce projet que nous avons découvert un outil du nom de GitHub, un logiciel de gestion de versions de codes. Lorsqu'on travaille sur un tel projet, il est important de savoir ce que notre équipe à modifié ou de pouvoir travailler ensemble sur le même programme tout en étant distant. C'est ce que permet GitHub.

Ce projet fut également formateur en termes de technique de programmation en Python. Nous avons pu approfondir et améliorer notre utilisation des listes tout en s'entraînant à une logique algorithmique. Nous avons également découvert l'affichage des couleurs en Python, de quoi déjà imaginer de nouveaux projets personnels de jeu en 2D dans la console. Nous avons enfin pris conscience des problèmes de performances et de gestion de la mémoire.

Enfin, ce projet nous a permis de nous former sur la rédaction d'un rapport détaillé. C'est aussi une compétence clé de l'ingénieur, souvent amené à rédiger des rapports sur les projets qu'il mène.

Nous restons donc dans l'attente d'un nouveau projet tout aussi formateur