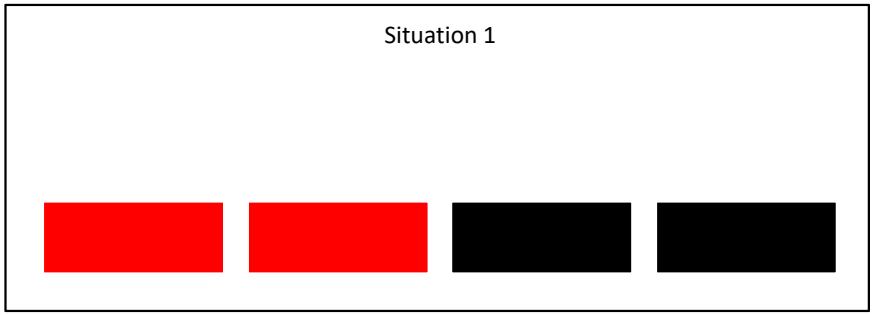
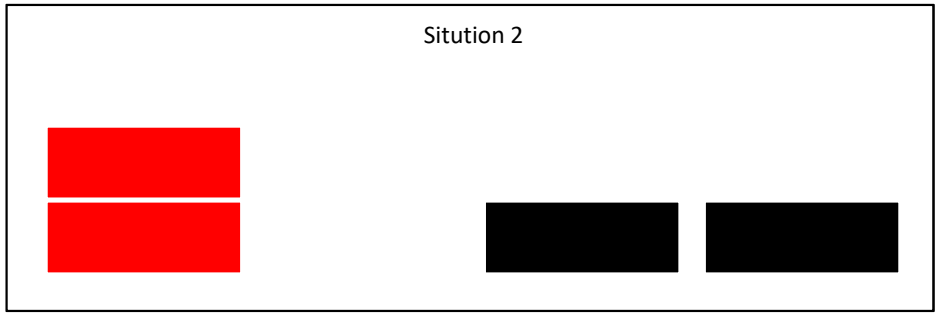
Q1-Etat de la liste x **dans les 2 situations**





- Nombre de tablettes /couleur : N=2
- Couleurs : ci=0 pour le rouge, ci=1 pour le noir

- Situation 1: x = [[0,1] , [0,1] , [1,1] , [1,1]]
- Situation 2: x = [[0,2],[1,1],[1,1]]

Q2-Fonction empile(x, i, j) prenant en paramètres la liste de situations x et les indices i,j de 2 piles pi et pj tels que i≠j.

La fonction retourne la liste de listes LXij des situations atteignables par empilement.

```
def empile( x,i,j ):
    """ empile ( x : list, i : int, j : int )-> list
        entrees : x, liste de listes(ci,hi), represente la situation initiale.
                : i, j, entiers, indices des piles que l'on veut empiler
        sortie; LXij , liste de situations possibles après empilement           """
    LXij = []
    ci , hi = x[i]
    cj , hj = x[j]
    if ci==cj or hi==hj : #empilement possible
        reste = x[:i]+x[i+1:j]+x[j+1:]
        pij = [ci,hi+hj] # pile ci sur cj (arbitrairement)
        X = reste + [pij]
        X.sort()
        LXij.append(X)
        if ci!=cj: # si couleur différente => ajouter la situation avec cj au dessus
            pji = [cj,hi+hj] # pile cj sur ci
            X = reste + [pji]
            X.sort()
            LXij.append(X)
    return LXij
```

- Q3 **Fonction coups(x)** qui prend en paramètre une situation x (liste) et retourne une liste

```
def coups(x):
    """ coups(x : list) -> list
      entree: x, liste de liste(ci,hi), situation initiale
      sortie: ensemble des situations possibles avec ttes les combinaisons d'empilement possibles
    """
    LX = []
    for i in range( len(x) ):
        for j in range( i+1 , len(x) ):
            LXij = empile( x,i,j )
            for X in LXij:
                if X not in LX: # On n'ajoute la situation que si elle n'a pas déjà été stockée
                    LX.append(X)
    return LX
x = [ [0,2],[0,3],[1,1],[1,2] ]
lesCoups1 = coups(x)
print( lesCoups1 )
# affiche [[[0, 5], [1, 1], [1, 2]], [[0, 3], [0, 4], [1, 1]], [[0, 3], [1, 1], [1, 4]],
[[0, 2], [0, 3], [1, 3]]]
x = [[0,1],[0,1],[1,1],[1,1]]
lesCoups2 = coups(x)
print( lesCoups2 )
# affiche [[[0, 2], [1, 1], [1, 1]], [[0, 1], [0, 2], [1, 1]], [[0, 1], [1, 1], [1, 2]],
[[0, 1], [0, 1], [1, 2]]]
```

- **Partie 2 : Création du graphe du jeu**
- Q4-**Fonction init(C, N)** prenant en argument le nombre de couleurs C et le nombre de tablettes par couleur N et renvoyant la liste x0 situation initiale du jeu triée

```
def init(C,N):
    """ init( C : int, N: int ) -> list
        entrees : C, entier positif, nombre de couleurs differentes
                : N , entier positif, nombre de tablettes par couleur
        sortie : x0, liste de listes (ci,hi) correspondant à la situation initiale
    """
    x0 = []
    for c in range(C):
        for _ in range(N):
            x0.append([c,1])
    x0.sort() # Inutile compte tenu de ma programmation avec c croissant
    return x0

test = init( 3, 4 )
print(test)
# affiche [[0, 1], [0, 1], [0, 1], [0, 1], [1, 1], [1, 1], [1, 1], [1, 1], [2, 1], [2, 1]
, [2, 1], [2, 1]]
```

- Q5-F**onction récursive Tuple(L)** réalisant cette transformation de liste en Tuple.

```
def Tuple(L):
    if L==[]:
        return ()
    elif type(L[0])!=list:
        return tuple(L)
    else:
        Res = []
        for l in L:
            Res.append(Tuple(l))
        return tuple(Res)
```

- Q6- **Fonction graphe(C , N)** réalisant le parcours en largeur des possibilités.

```
from collections import deque
def graphe(C,N):
    """ graphe(C : int ,N:int )-> dict
        entrees:C, entier positif, nombre de couleurs differentes
                :N , entier positif, nombre de tablettes par couleur
        sortie:G, dictionnaire,le:tuple de la situation,valeur:liste de liste de situation atteignables  """
    # constitution de la situation initiale
    x0 = init(C,N)
    G = {} # initialisation du dictionnaire G qui représente le graphe
    file = deque()
    file.append( x0 ) # initialisation de la file avec la situation initiale x0
    while len( file ) != 0:
        x = file.popleft()  # recuperation du 1er element de la file
        Cle = Tuple(x)      # génération de la cle sous forme de Tuple
        LCoupsSuivants = coups(x) #Recuperation de la liste des situations atteignables depuis x
        G[Cle] = LCoupsSuivants # ajout dans le graphe du couple clé, valeur
        # ajout dans la file des differentes situations (si elles n'ont pas déjà été stockées) p
our traitement ultérieur
        for c in LCoupsSuivants:
            if Tuple(c) not in G:
                file.append(c)
    return G
```

Q7- Que représentent N1 et N2 ?

```
N1 = len(Graphe)
N2 = sum([len(Graphe[x]) for x in Graphe])
```

N1 : nombre de sommets (12 sur l'exemple)
N2 : nombre d'arêtes (16 sur l'exemple)

```
{((0, 1), (0, 1), (1, 1), (1, 1)): [[[0, 2], [1, 1], [1, 1]],
                                    [[0, 1], [0, 2], [1, 1]],
                                    [[0, 1], [1, 1], [1, 2]],
                                    [[0, 1], [0, 1], [1, 2]]],
 ((0, 2), (1, 1), (1, 1))         : [[[0, 2], [1, 2]]],
 ((0, 1), (0, 2), (1, 1))         : [[[0, 3], [1, 1]],
                                    [[0, 2], [0, 2]],
                                    [[0, 2], [1, 2]]],
 ((0, 1), (1, 1), (1, 2))         : [[[0, 2], [1, 2]],
                                    [[1, 2], [1, 2]],
                                    [[0, 1], [1, 3]]],
 ((0, 1), (0, 1), (1, 2))         : [[[0, 2],
                                      [1, 2]]],
 ((0, 2), (1, 2))                 : [[[0, 4]],
                                    [[1, 4]]],
 ((0, 3), (1, 1))                 : [],
 ((0, 2), (0, 2))                 : [[[0, 4]]],
 ((1, 2), (1, 2))                 : [[[1, 4]]],
 ((0, 1), (1, 3))                 : [],
 ((0, 4),): [],
 ((1, 4),): []
 }
```

Q7- Que représentent N1 et N2 ?

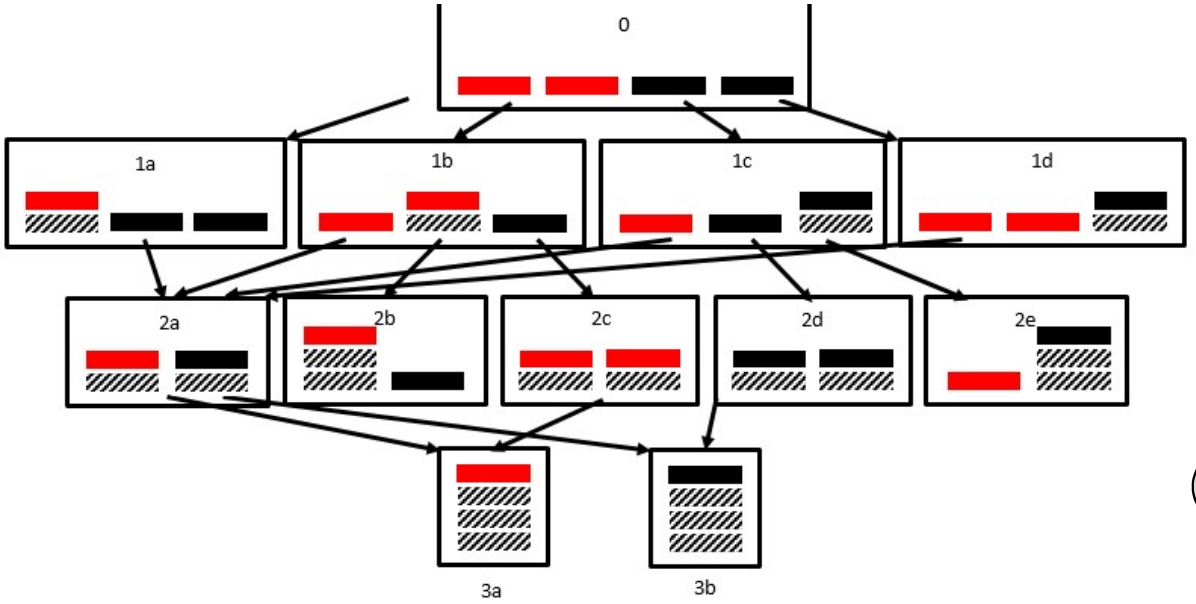
```
N1 = len(Graphe)
N2 = sum([len(Graphe[x]) for x in Graphe])
```

N1 : nombre de sommets (12 sur l'exemple)
N2 : nombre d'arêtes (16 sur l'exemple)

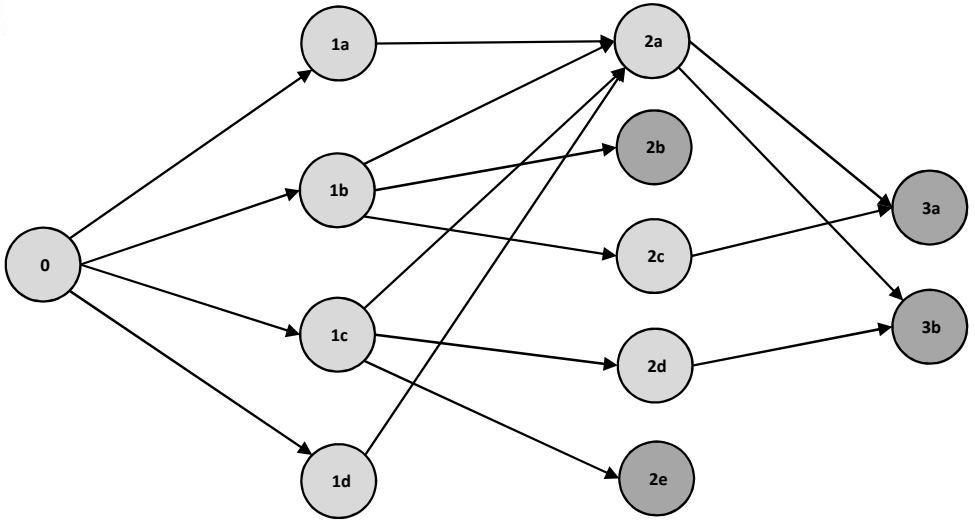
Q8- Remplissage tableau :

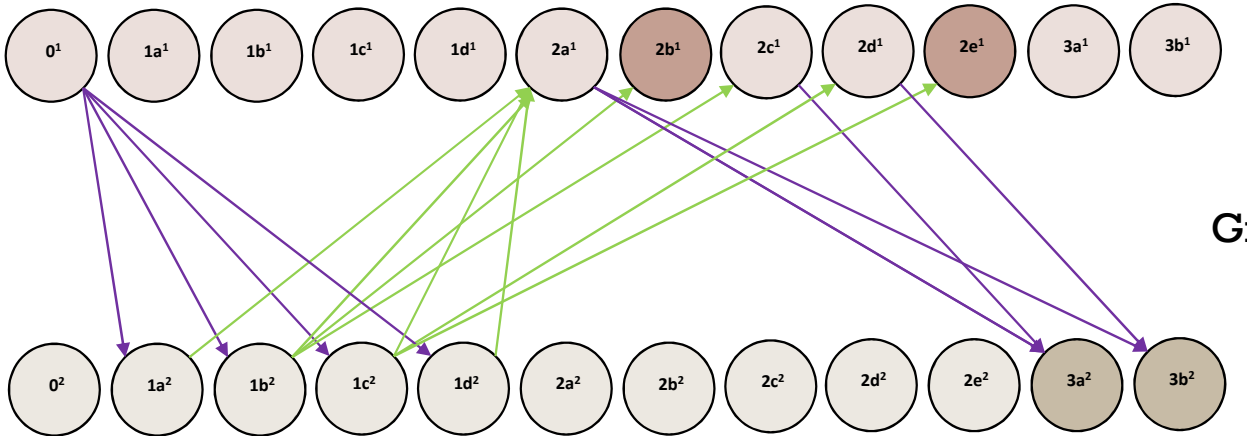
```
from time import perf_counter
C,N = 2,2 # A modifier pour chaque cas
tic = perf_counter()
Graphe = graphe(C,N)
toc = perf_counter()
T = toc - tic

print("Pour C = "+str(C)+" et N = "+str(N))
Nb_Sommets = len(Graphe) # N1
print("Sommets:",Nb_Sommets)
#N2
Nb_Aretes =sum([len(Graphe[x])for x in Graphe])
print("Arêtes:",Nb_Aretes)
print("Temps (s):",T)
```

	N =	1	2	3	4
C = 1	Sommets	1	2	3	5
	Arêtes	0	1	2	5
	Temps (s)	$2{,}1.10^{-5}$	$8{,}0.10^{-6}$	$1{,}6.10^{-5}$	$2{,}9.10^{-5}$
C = 2	Sommets	3	12	43	133
	Arêtes	2	16	90	386
	Temps (s)	$1{,}1.10^{-5}$	$8{,}8.10^{-5}$	$1{,}0.10^{-3}$	$2{,}8.10^{-2}$
C = 3	Sommets	7	92	696	4220
	Arêtes	6	234	2832	23 487
	Temps (s)	$2{,}6.10^{-5}$	$3{,}9.10^{-3}$	1,2	6560
C = 4	Sommets	23	728		
	Arêtes	48	3040		
	Temps (s)	$9{,}1.10^{-4}$	1,6		



Graphe orienté fini





Graphe biparti

- **Partie 2 : Graphe biparti**

Q9- F**onction sommets_12(G,C,N)** prenant en argument le graphe G, N et C, et retournant les 2 Tuples des sommets S1 et S2 des joueurs J1 et J2

```
def sommets_12( G, C, N ):
    """ sommets_12( G, C, N )
        entrees : G, dictionnaire, represente le graphe
                : C, N, entiers representant le nombre de couleurs et le nombre de tablettes
        sorties: 2 tuples pour les sommets de J1 et de J2
    """
    S1 = []
    S2 = []
    n = N*C    # nombre total de tablettes = nb de piles initial
    for e in G:
        if len(e)%2 == n%2:
            S1.append(e)
        else:
            S2.append(e)
    return tuple(S1),tuple(S2) # S1 et S2 déjà Tuples, donc Tuples inutile
```

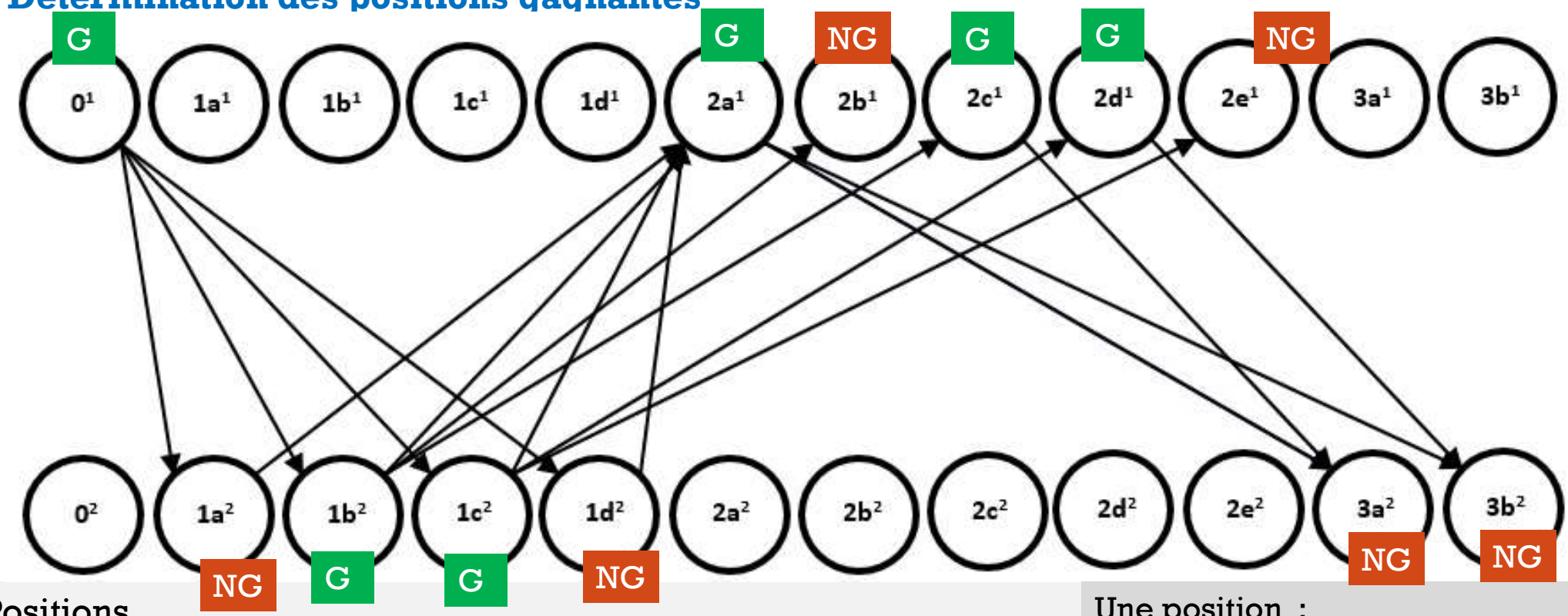
Q10- Créer les Tuples S1 et S2 dans le cas C=N=2.

```
C,N = 2,2
Graphe = graphe( C , N )
S1,S2 = sommets_12( Graphe , C , N )
```

Q11- Pour C=N=2, et en prenant à chaque fois le 1er successeur identifié dans le graphe, **afficher une partie et préciser le joueur gagnant**

```
C,N = 2,2
Graphe = graphe(C,N) #generation du graphe du jeu
x0 = init(C,N) #situation initiale
Joueur = 1
print( "Joueur:" , Joueur )
print( "Jeu:", x0 )
lesSuccesseurs = Graphe[Tuple(x0)] #successeurs de la situation initiale

while len( lesSuccesseurs )>0:
        succ = lesSuccesseurs[0] #1er successeur
        Joueur = 3 - Joueur
        print( "Joueur:", Joueur )
        print( "Jeu:",succ )
        lesSuccesseurs = Graphe[Tuple(succ)] #successeurs du 1er successeur
```

Détermination des positions gagnantes



Positions

2b 2e 3a 3b pas gagnantes car pas de successeurs

2a 2c 2d gagnantes car au moins un successeur n'est pas gagnant

1a et 1d pas gagnants car tous les successeurs sont gagnants (2a)

1b gagnant car au moins un successeur pas gagnant (2b)

1c gagnant car au moins un successeur pas gagnant (2e)

0 gagnant car au moins un successeur pas gagnant (1a et 1d)

Finalement, les positions gagnantes sont: **0 1b 1c 2a 2c 2d**

Une position :

- est gagnante s'il existe au moins un successeur qui n'est pas gagnant
- **N'est pas gagnante** si :
- Elle n'a pas de successeurs
- Aucun de ses successeurs n'est pas gagnant = Tous ses successeurs sont gagnants