Correction du DS4

Partie I

1. Le nombre de sommets vide diminue de 1 à chaque coup donc la partie se termine en $\boxed{\text{au plus } n \text{ coups}}$

2. Un joueur gagne lorsque l'autre ne peut plus jouer, ce qui finira par arriver d'après la question précédente, donc $\boxed{\text{il y aura toujours un vainqueur}}$

3. a) $\boxed{\texttt{valides = [4,5,7,10]}}$

b) $\boxed{\texttt{gains = []}}$ car quel que soit le coup joué par J_2, le joueur J_1 pourra toujours placer un jeton sur le sommet 6 donc le joueur J_2 ne peut pas gagner (en un seul coup).

c) Avec la position actuelle des jetons 1, J_1 ne peut plus placer ses jetons que sur le sommet 6 ; mais J_2 ne peut pas jouer sur ce sommet pour empêcher J_1 de s'y mettre. On peut donc examiner les coups possibles de J_2 :
- ⋆ si J_2 joue ailleurs qu'en 10, J_1 jouera forcément en 6 et J_2 pourra toujours jouer en 10 pour gagner la partie
- ⋆ si J_2 joue en 10, J_1 jouera en 6 et J_2 gagnera la partie en jouant sur un des sommets 4, 5 ou 7

On a donc $\boxed{\texttt{posG = [4,5,7,10]}}$

4. Le nombre de sommets étant pairs, il suffit de justifier que J_2 ne peut pas être bloqué (avant J_1) ; dans le pire des cas, il placera un dernier jeton sur le dernier sommet du graphe et c'est J_1 qui sera bloqué. Si J_1 peut placer une jeton sur le sommet s, il s'agit de justifier que le sommet $s+1$ (ou 0 si $s = n-1$) est accessible pour J_2 : le sommet $s+1$ est libre car s'il était occupé par J_1, J_1 ne pourrait pas jouer en s ; les deux voisins de $s+1$ sont s, qui est occupé par J_1, et $s+2$ qui ne peut pas être occupé par J_2 car pour que J_2 ait déjà placé un jeton en $s+2$, vue la stratégie, il aurait fallu que J_1 ait déjà placé un jeton en $s+1$, ce qui n'est pas possible. Ainsi, J_2 ne sera jamais bloqué avant J_1 donc $\boxed{J_2 \text{ gagne la partie}}$

5. Si un joueur commence par jouer en 0 alors il perd la partie au coup suivant car il ne pourra pas jouer un coup supplémentaire. Si un joueur a déjà joué un coup, ailleurs qu'en 0, il ne peut plus joueur en 0 et si le sommet 0 est libre alors tous les sommets périphériques restent accessibles aux deux joueur. Le joueur qui possède une stratégie gagnante est donc celui qui complètera les sommets périphériques. Ainsi $\boxed{J_1 \text{ gagne si } n \text{ est pair, sinon c'est } J_2}$

Partie II

1. Toutes les listes d'adjacences sont vides :

```
def creerGraphe(n) :
    return {i:[] for i in range(n)}
```

]

2. Le graphe doit être non orienté donc i est joignable depuis j et j est joignable depuis i. La fonction doit modifier `G` par effet de bord et ne doit rien renvoyer :

```
def ajouterArete(G,i,j) :
    G[i].append(j)
    G[j].append(i)
```

3. 0 peut être lié aux $n-1$ autres sommets, 1 à tous les autres mais il ne faut pas compter la liaison avec 0, pour 2 il reste $n-2$ nouvelles liaison possibles donc $p \leqslant \sum_{k=0}^{n}(k-1)$ et $\boxed{p \leqslant \dfrac{n(n-1)}{2}}$

4. Après avoir tiré au sort 2 entiers, il faut contrôler qu'ils sont distincts et que cela correspond bien à une nouvelle arête ; `randint(a,b)` renvoie un entier de $[\![a,b]\!]$ (bornes comprises)

```
def plateau(n,p) :
    G = creerGraphe(n)
    for _ in range(p) :
        i,j = randint(0,n-1),randint(0,n-1)
        while i == j or (i in G[j]) :
            i,j = randint(0,n-1),randint(0,n-1)
        ajouterArete(G,i,j)
    return G
```

Cette fonction n'est pas très efficace car quand p est grand, la boucle `while` peut mettre du temps à finir car on risque de tirer au sort une arête déjà existante un grand nombre de fois consécutivement.

5. Chaque joueur joue à tour de rôle et `etat` ne comporte pas les sommets libres donc la longueur de `etat` (sa parité) suffit pour savoir quel joueur doit jouer ensuite :

```
def joueur(etat) :
    return 1+len(etat)%2
```

6. On teste tous les voisins de s pour voir s'il sont occupés (sinon il ne figure pas dans `etat`) et si le jeton qui y est placé porte le numéro j :

```
def test(G,s,etat,j) :
    for voisin in G[s] :
        if voisin in etat and etat[voisin] == j :
            return False
    return True
```

7. On teste tous les sommets de G, ceux qui conviennent sont ceux qui sont libres et pour lesquels aucun voisin n'est occupé par un jeton du joueur courant (qu'il faut commencer pas déterminer)

```
def possibles(G,etat) :
    j = joueur(etat)
    L = []
    for s in G :
        if s not in etat and test(G,s,etat,j) :
            L.append(s)
    return L
```

8. La fonction `possibles` utilise n fois la fonction `test` (le coût de `joueur` et du test `s not in etat` sont constants avec la structure de dictionnaire). Dans le pire des cas, le sommet s possède $n-1$ voisins donc la complexité de `test` est $O(n)$ et celle de `possibles` est $\boxed{O(n^2)}$

9. Un état est terminal si le joueur courant n'a plus de coup possible donc si $\boxed{\texttt{len(possibles(G,etat)) == 0}}$

Partie III

1. a) Le parcours en largeur est un parcours du graphe par distance croissante donc comme on cherche un sommet vérifiant une certaine propriété le plus proche possible de s, un tel parcours est plus adapté.

b) On fait la recherche jusqu'à ce que J_2 puisse jouer en s et il faut « enfiler » les sommets non vus dans Q pour faire le parcours :

```
def suivant(G,etat,s) :
    dicoPossibles = {x : False for x in G}
    for x in possibles(G,etat) :
        dicoPossibles[x] = True
    Q = [s]
    vus = {x : False for x in G}
    while not dicoPossibles[s] :
        s = Q[0]
        Q = Q[1:]
        vus[s] = True
        for voisin in G[s] :
            if not vus[voisin] :
                Q.append(voisin)
    return s
```

c) Si on faisait le test `while not x in L` après avoir posé `L = possibles(G,etat)`, le coût du test serait en $O(L)$ alors qu'il est en $O(1$ avec un dictionnaire.

d) Dans le cas d'un graphe étoilé, une telle stratégie amènerait J_2 à jouer au centre (sommet 0 avec les représentations précédentes) donc à perdre.

2. Comme on modifie `etat` dans la fonction pour tester les positions suivantes, il faut annuler cette modification pour les tests suivants.

```
def gagnantJ2(G,etat) :
    P = possibles(G,etat)
    j = joueur(etat)
    if len(P) == 0 and j == 1 :      # J1 est bloqué
        return True
    elif len(P) == 0 :               # J2 est bloqué
        return False
    elif j == 2 :        # on vérifie si un coup de J2 amène à le faire
        gagner
        test = False
        for s in P :
```

```
            etat[s] = 2
            test = test or gagnantJ2(G,etat)
            del(etat[s])
        return test
    else :              # on vérifie si tous les coups de J1 amène à
        faire gagner J2
        test = True
        for s in P :
            etat[s] = 1
            test = test and gagnantJ2(G,etat)
            del(etat[s])
        return test
```

3. Pour déterminer si un état est une position gagnante, on utilise la fonction `possible` de complexité $O(n^2)$ puis appelle récursivement la fonction `gagnantJ2` sur toutes les positions utilisables, qui sont au pire $n-1$. La complexité de cette fonction dépend donc du nombre de positions possibles sur le graphe : pour un graphe totalement libre ayant n sommets, il y a n façons de compléter l'état initial avec le premier jeton puis $n-1$ avec le deuxième, ...On a donc $C(n) = O(n^2) + \sum_{k=1}^{n} C(n-1)$ ce qui donne $\boxed{C(n) = O(n!)}$

4. Un dictionnaire est une structure mutable donc non hachable et ne peut donc pas être utilisé comme clef d'un autre dictionnaire.

5. a)
```
def encode(etat) :
    s = 0
    for c in etat :
        s += etat[c]*3**c
    return s
```

b) Il s'agit de retrouver l'écriture en base 3 d'un entier ; il faut faire attention de ne pas stocker les chiffres nuls dans le dictionnaire : l'entier `i` dans la fonction suivante correspond au numéro du chiffre calculé, donc à la clef dans le dictionnaire

```
def decode(k) :
    d = {}
    i = 0
    while k > 0 :
        q,r = k//3,k%3
        if r > 0 :
            d[i] = r
        i += 1
        k = q
    return d
```

6. On met à jour le dictionnaire `PG2` si la valeur n'a pas déjà été calculée, et on la renvoie si elle y est déjà :

```
def GagnantJ2(G,etat) :
    cle = encode(etat)
    if cle not in PG2 :
        P = possibles(G,etat)
        j = joueur(etat)
        if len(P) == 0 and j == 1 :
            r = True
        elif len(P) == 0 :
            r = False
        elif j == 2 :
            test = False
            for s in P :
                etat[s] = 2
                test = GagnantJ2(G,etat) or test
                del(etat[s])
            r = test
        else :
            test = True
            for s in P :
                etat[s] = 1
```

```
                test = GagnantJ2(G,etat) and test
                del(etat[s])
            r = test
        PG2[cle] = r
    return PG2[cle]
```

Dans la fonction précédente, il est important de faire les affectations dans la variable `test` *dans cet ordre : le* `or` *n'est pas vraiment commutatif donc si on met* `test = test or GagnantJ2(G,etat)` *et si la variable* `test` *contient* `True`*, le résultat sera toujours* `True` *et le nouvel appel à la fonction* `GagnantJ2` *ne sera pas fait et donc le dictionnaire* `PG2` *ne sera pas rempli pour tous les sommets de la liste* `P`*. Ceci le poserait pas de problème pour cette fonction mais pourrait en poser pour la fonction suivante, qui a besoin que* `PG2` *soit rempli.*

7. Il s'agit de trouver l'état suivant dans `PG2` qui est encore gagnant pour J_2, parmi les coups possibles pour J_2 (il peut y en avoir plusieurs donc il suffit d'en renvoyer un) ; à nouveau, après chaque test sur `etat`, il faut annuler la modification pour le test suivant

```
def jouerJ2(G,etat) :
    P = possibles(G,etat)
    trouve = False
    while not trouve :
        s = P.pop()
        etat[s] = 2
        if PG2[encode(etat)] :
            r = s
            trouve = True
        del(etat[s])
    return r
```

8. Même avec la mémoïsation, lors du premier appel le dictionnaire `PG2` est vide et il est possible que le premier appel nécessite de le remplir intégralement. Comme il y a a potentiellement 3^n état (une clef à plus ou moins 3 valeurs possibles : soit elle n'est pas présente, soit elle est associée à la valeur 1, soit à la valeur 2), la fonction précédente peut avoir une complexité exponentielle. Une heuristique permettrait de ne pas faire la recherche jusqu'au bout en limitant la profondeur de recherche à un certain nombre de coups à suivre.

9. Il s'agit d'attribuer une valeur statistique à une état : plus la situation est favorable pour J_2, plus la valeur doit être grande. En premier lieu, il vaut mieux modifier les valeurs pour lesquelles la conclusion est certaine (afin de faire des comparaisons plus facilement), on remplace donc les booléens `True` et `False` respectivement par 0 et 1 : une position gagnante est notée 1, une perdante est notée 0. Reste à attribuer aux autres positions des valeurs dans $]0,1[$. On peut par exemple considérer que plus un état laisse de possibilités, plus il est favorable au joueur courant. Si on note N_1 et N_2 le nombre de coups possibles pour J_1 et J_2 (on compte dans un état donné les coups possibles pour les deux joueurs, indépendamment du joueur qui doit effectivement jouer) ; on peut alors poser $h(\texttt{etat}) = \dfrac{N_2}{N_1+N_2}$: plus la valeur de h est grande, plus le nombre de coups jouables par J_2 est grand par rapport au nombre de coups jouables par J_1.