TD : PROGRAMMATION DYNAMIQUE – JEU DE RÖCKSE

I) MISE EN SITUATION : RAPPELS SUR LES TROIS PREMIÈRES PARTIES

I.1. Partie I : Sauts et chemins

1. Tester la fonction `poids(T,chemin)`, étant donné un plateau de jeu T et un chemin `chemin`, renvoie le poids de ce chemin :

```
# Le poids total du chemin optimal est bien de -7
>>> poids(T, chemin_optimal)
-7
```

2. La fonction est donnée ci-dessous : quelle est sa complexité ?

   La complexité est en O(L) où L est la longueur du chemin (nombre de cases). On parcourt chaque case une fois et l'accès à T[i][j] se fait en O(1).

3. Tester la fonction `appliquer_sauts(i,j,sauts)` qui, étant donné une case (i, j) et une liste de sauts `sauts`, applique les sauts dans l'ordre donné par la liste sauts, en partant de la case (i, j) et renvoie la case atteinte. On suppose que le chemin indiqué par sauts reste dans la grille.

```
# Trois déplacements vers la droite puis un déplacement en bas à
gauche, en partant de la case (0,0) mènent bien à la case (1,2)
>>> liste_sauts = [sauts[0], sauts[0], sauts[0], sauts[1]]
>>> appliquer_sauts(0, 0, liste_sauts)
(1, 2)
```

4. Tester la fonction `sauts_corrects(sauts,bonus,chemin)` qui, étant donné l'ensemble des sauts par défaut représenté par la liste `sauts`, les sauts associés aux cases bonus `bonus` et un chemin `chemin`, renvoie `True` si les sauts utilisés dans le chemin sont corrects et `False` sinon. On suppose que le chemin reste dans la grille.

```
# Les sauts par défaut, associés au bonus en case (1,2) sont en
# accord avec le chemin optimal
>>> sauts_corrects(sauts,bonus,chemin_optimal)
True
# Par contre sans le bonus, ces sauts par défaut ne sont pas en
# accord avec le chemin optimal car le saut (1,0) est interdit
>>> sauts_corrects(sauts,{},chemin_optimal)
False
```

5. Tester la fonction `sauts_bien_formes(sauts,bonus)` qui, étant donné la liste des sauts par défaut sauts et les sauts associés aux cases bonus `bonus`, vérifie que chaque saut de ces listes satisfait la condition (*). La fonction renvoie `True` si c'est le cas et `False` sinon.

```
# Les sauts par défaut sont bien formés avec ou sans le bonus
>>> sauts_bien_formes(sauts,bonus)
True
>>> sauts_bien_formes(sauts,{})
True
# Mais avec le bonus (0,-1) (déplacement à gauche) ils ne sont plus
# bien formés
>>> sauts_bien_formes(sauts,{(1,2):[(0,-1)]})
False
```

I.2. Partie II : Recherche exhaustive

1. Quelle est la longueur maximale L d'un chemin de (0, 0) à (N-1, N-1) pour n'importe quel ensemble de sauts satisfaisant (*) ? Donner un exemple d'ensemble de sauts par défaut pour lequel se réalise ce chemin de longueur L.

   Pour aller de (0,0) à (N−1, N−1), on doit augmenter i de N−1 et j de N−1. Avec des sauts satisfaisant la condition (*), chaque saut fait progresser d'au moins 1 dans le sens lexicographique. Le chemin le plus long est obtenu quand chaque saut ne fait progresser que d'exactement 1 : soit i augmente de 1 (et j peut diminuer ou rester constant), soit i reste constant et j augmente de 1.
   Le chemin le plus long alterne donc entre des déplacements « vers le bas » (i+1) et « vers la droite » (j+1), passant par 2N−1 cases au total.
   Exemple de sauts : sauts = [(0, 1), (1, 0)] (→ et ↓)
   Avec ces sauts, un chemin de longueur maximale serait :
   (0, 0) → (0, 1) → (0, 2) → ... → (0, N−1) → (1, N−1) → ... → (N−1, N−1)
   … soit N cases sur la première ligne puis N−1 cases vers le bas, pour un total de 2N−1 cases.

2. Tester la fonction `trouve_complet(T,sauts,bonus,sauts_max)` qui utilise la fonction récursive décrite précédemment et renvoie le couple `(poids,sauts)` où poids est le poids du chemin trouvé et `sauts` est la liste des sauts du chemin trouvé.

```
# Avec sauts_max=100, le chemin optimal est trouvé par recherche
# exhaustive avec un poids de -7. Ce chemin passe par la case bonus
# (1,2) et utilise le saut (1,0) débloqué :
>>> trouve_complet(T,sauts,bonus,100)
(-7, [(0, 1), (0, 1), (1, -1), (0, 1), (1, 0), (1, 0), (0, 1)])

# Avec sauts_max = 5, on ne trouve pas un chemin jusqu'à l'arrivée
>>> trouve_complet(T,sauts,bonus,5)
(-11, [(0, 1), (0, 1), (1, -1), (0, 1), (1, 0)])

# Sans bonus et avec sauts_max=100, on trouve le chemin optimal avec
# un poids de -6. Cela montre l'importance des cases bonus.
>>> trouve_complet(T,sauts,{},100)
(-6, [(0, 1), (0, 1), (1, -1), (1, -1), (0, 1), (0, 1), (1, 1)])
```

3. La fonction est donnée ci-dessous. Quelle est sa complexité ?

   À chaque niveau de récursion, on explore jusqu'à `S = len(sauts)` sauts (où S peut inclure les sauts bonus activés) et chaque saut valide mène à un appel récursif. Il y a en tout (`saut_max + 1`) niveaux :
   - **Niveau 0 : 1 appel max**
   - **Niveau 1 : S appels max**
   - **Niveau 2 : S^2 appels max**
   - **…**
   - **Niveau horizon : S^{sauts_max} appels max**

   La complexité est donc en $O(S^{sauts\_max})$, soit $O(S^{2N-1})$ car max(sauts_max) = 2N-1.

4. La fonction trouve_complet_rec() perd beaucoup de temps à refaire les mêmes calculs. On peut l'améliorer en enregistrant les résultats déjà calculés pour une valeur fixée de sauts_max. Expliquer en quelques lignes comment procéder. Quelle serait alors la complexité ?

C'est le principe de la mémoïsation : on utilise un dictionnaire pour stocker les résultats déjà calculés.
La clé est de la forme (i, j, tuple(sauts_disponibles)). Avant chaque calcul récursif, on vérifie si le résultat est déjà présent dans le dictionnaire. Si oui, on le retourne directement. Sinon, on effectue le calcul et on stocke le résultat avant de le retourner.
Remarque : utiliser une clé de la forme (i, j, sauts_disponibles) n'est pas possible car sauts_disponibles n'est pas hashable ! Il faut le transformer en tuple.

```
# Exemple de structure :
memo = {}  # dictionnaire de mémoïsation

def trouve_complet_rec_memo(...):
    cle = (i, j, tuple(sauts_disponibles))
    if cle in memo:
        return memo[cle]
    # ... calcul ...
    memo[cle] = resultat
    return resultat
```

Ici, chaque configuration `(i, j, tuple(sauts_disponibles))` n'est calculée qu'une seule fois et l'accès avec le dictionnaire est en O(1). Or, pour traiter une clé on itère S = len(sauts) fois au maximum et chaque itération fait des opérations en O(1). Le coût total pour traiter une clé est donc de O(S). Le coût global est donc O(K·S), où K est le nombre de clés distinctes effectivement rencontrées.

On peut majorer (i, j) par N^2, et donc majorer le nombre de clés distinctes par $K \leq M \cdot N^2$, où M est le nombre de tuples de sauts disponibles. Sans bonus, l'ensemble des sauts disponibles est constant et donc M = 1. Avec n bonus, le nombre de configurations possibles est au plus de 2^n, donc on a $M \leq 2^n$ où n est le nombre de bonus et $K \leq 2^n \cdot N^2$.

On obtient donc un coût global de $O(2^n \cdot N^2 \cdot S)$, où n est le nombre de bonus, N est la taille de la grille (N x N), et S est le nombre de sauts disponibles par défaut.

II) PARTIE III : RECHERCHE GLOUTONNE

1. Utiliser la fonction `trouve_complet()` (voir page 5) sur l'exemple donné en introduction pour trouver le chemin lorsque k = 2 ? Est-ce un chemin optimal ?

```
# Recherche récursive depuis (0,0) avec k=2 : (0,0) -> (0,2)
>>> trouve_complet(T,sauts,bonus,2,0,0)
(-8, [(0, 1), (0, 1)])
# Recherche récursive depuis (0,2) avec k=2 : (0,2) -> (2,2)
>>> trouve_complet(T,sauts,bonus,2,2,2)
(-11, [(1, -1), (1, 1)])
# Recherche récursive depuis (2,2) avec k=2 : (2,2) -> (3,2)
>>> trouve_complet(T,sauts,bonus,2,3,2)
(-2, [(0, 1), (1, -1)])
# Recherche récursive depuis (3,2) avec k=2 : (3,2) -> (3,3)
(4, [(0, 1)])
# On trouve le chemin :
# [(0,0),(0,1),(0,2),(1,1),(2,2),(2,3),(3,2),(3,3)]
# poids du chemin : non optimal
>>> poids(T,[(0,0),(0,1),(0,2),(1,1),(2,2),(2,3),(3,2),(3,3)])
-5
```

2. Écrire la fonction `trouve_glouton(T,sauts,bonus,k)` qui, étant donnés la grille de jeu T, la liste des sauts par défaut sauts, les sauts associés aux cases bonus `bonus` et l'horizon k, effectue cette recherche gloutonne et renvoie le couple `(poids,sauts)` où poids est le poids du chemin trouvé et sauts est la liste des sauts du chemin trouvé. En augmentant k, obtient-on forcément un chemin de poids plus petit ?

```python
def trouve_glouton(T,sauts,bonus,k):
    case_finale = (len(T)-1,len(T)-1)
    case = (0,0)
    chemin = [(0,0)]          # Pour enregistrer le chemin

    while case != case_finale:
        # Cherche les meilleures actions à l'horizon k
        _, actions = trouve_complet(T,sauts,bonus,k,case[0],case[1])

        # Applique les actions et enregistre les poids
        for a in actions:
            # position initiale
            i = case[0]
            j = case[1]

            # Applique l'action sur la case courante
            case = appliquer_sauts(i,j,[a])

            # Sauvegarde le chemin
            chemin.append(case)
    return chemin, poids(T,chemin)
```

 # Recherche gloutonne depuis (0,0) avec k=2:
>>> trouve_glouton(T,sauts,bonus,2)
([(0, 0), (0, 1), (0, 2), (1, 1), (2, 2), (2, 3), (3, 2), (3, 3)], -5)
 # Recherche gloutonne depuis (0,0) avec k=3:
>>> trouve_glouton(T,sauts,bonus,3)
([(0, 0), (0, 1), (0, 2), (1, 1), (1, 2), (2, 2), (3, 2), (3, 3)], -7)
 # Recherche gloutonne depuis (0,0) avec k=4:
>>> trouve_glouton(T,sauts,bonus,4)
([(0, 0), (0, 1), (0, 2), (1, 1), (2, 2), (3, 3)], -6)

Augmenter k améliore l'optimalité locale (on optimise sur un horizon plus long), mais ne rend pas l'algorithme optimal et ne garantit pas une amélioration monotone, car l'engagement sur des blocs de k coups peut mener à des décisions globalement moins bonnes.

III) PARTIE IV : RECHERCHE PAR PROGRAMMATION DYNAMIQUE

III.1. Encodage des cases bonus activées

1. Écrire la fonction `code_bonus(masque_bonus)` qui renvoie le code associé au masque binaire `masque_bonus` représentant l'ensemble des cases bonus activées.

```
def code_bonus(masque_bonus):
    code = 0
    for i in range(len(masque_bonus)):
        code = code + masque_bonus[i]*2**i
    return code
```

III.2. Récurrence

1. Écrire la définition de poids `opt[i][j][<b₀ … bₙ₋₁>]`. On notera Γ l'ensemble des sauts par défaut et Δ_k l'ensemble des sauts associé à la k-ième case bonus.

On suppose que tous les sauts de Γ et Δ_k sont bien formés et satisfont (*).
On note, pour un état `code_bonus` = <b₀ … bₙ₋₁> l'ensemble des sauts bien formés autorisés par `code_bonus` :

$$S(\text{code\_bonus}) = \Gamma \cup \left(\bigcup_{k:b_k=1} \Delta_k \right)$$

Alors l'équation de récurrence peut s'écrire :

$$\text{poids\_opt}[i][j][< b_0 \dots b_{n-1} >]] = T[i][j] + \min_{\substack{(\delta_i,\delta_j)\in S(code\_bonus') \\ 0\le i+\delta_i\le N, 0\le i+\delta_j\le N}} \left(poids\_opt[i+\delta_i][j+\delta_j][code\_bonus'] \right)$$

Conditions usuelles :

- **∀ code_bonus, poids_opt[N-1][N-1][code_bonus] = T[N-1][N-1]**
- **Si l'ensemble sur lequel on prend le minimum est vide et (i, j) ≠ (N-1, N-1), on pose poids_[i][j][code_bonus] = + ∞**

III.3. Algorithme

L'itération « à l'envers » sur les (i, j) en partant de N - 1 jusqu'à 0 est justifiée par la récurrence : pour calculer les poids_opt[i][j][·] , on utilise les termes poids_opt[i+δ_i][j+δ_j][·]. Or, par la condition (*) (i + δ_i, j + δ_j) > (i, j) au sens lexicographique et qu'on ne prend en compte que les sauts qui sont dans la grille, cela garantit qu'en parcourant i et j à rebours, les valeurs des successeurs ont déjà été calculées.

Concernant l'itération sur les masques des bonus, quand on est en (i, j), on peut activer (au plus) le bonus de cette case, donc on passe d'un ensemble B de bonus activés à un ensemble $\mathbf{B' = B \cup \{k\}}$ si (i, j) est la case bonus k. On ne désactive donc jamais un bonus et l'ensemble des bonus actifs ne fait qu'augmenter au cours d'un chemin. En termes de masque binaire, le nouveau masque qui active éventuellement le nouveau bonus contient tous les 1 du masque précédent (et éventuellement un 1 de plus), donc B $\subseteq$ B'. Un état avec peu de bonus actifs dépend donc (potentiellement) d'un état avec plus de bonus actifs et pour que les valeurs soient disponibles en remplissage « bottom-up », il faut calculer d'abord les masques les plus « grands » puis descendre (ordre décroissant au sens de l'inclusion).
Par exemple, avec n=3 bonus, un ordre décroissant commence par ceux qui ont le plus de 1 :

- **Couche 3 : 111**
- **Couche 2 : 110, 101, 011**
- **Couche 1 : 100, 010, 001**
- **Couche 0 : 000**

Ainsi, quand on calcule un état sur 010 (seul le bonus 1 actif), un successeur peut activer un bonus supplémentaire et passer à 011 ou 110, qui ont été calculés auparavant.

Avec une approche top-down (récursive + mémoïsation), il n'y aurait pas besoin d'imposer cet ordre explicitement : la récursion irait « chercher » ce dont elle a besoin.

1. Écrire une fonction `combinaisons_bonus(nb_bonus)` qui, étant donné le nombre total de cases bonus `nb_bonus`, renvoie la liste de masques bonus ordonnée de manière décroissante dans le sens de l'inclusion, en codant l'algorithme ci-dessus.

```python
def combinaisons_bonus(nb_bonus):
    file=[[True for i in range(nb_bonus)]]
    masques = [file[0]]

    # On défile tant que la file n'est pas vide
    while len(file) !=0:
        choix = file.pop(0)

        # Insère une coordonnée True -> False
        for i,val in enumerate(choix):
            modif = choix[:]
            if val == True:
                modif[i] = False
                if modif not in file:
                    file.append(modif)
                    masques.append(modif)
    return masques
```

2. Tester la fonction `ranger_bonus(bonus)` qui renvoie un couple `(bonus_au_rang, rang_du_bonus)` tel que :
   - `bonus_au_rang[k]` est la liste des sauts activés par la case bonus dont le numéro est k ;
   - `rang_du_bonus[(i,j)]` est le numéro de la case bonus (i, j) dans un masque de bonus.

```
>>> ranger_bonus(bonus)
([[(1, 0)]], {(1, 2): 0})
# Liste des bonus au rang 0 : [(1, 0)]
# Rang du bonus en case (1,2) : 0
#  => Sur la case (1,2), il y a un bonus dont le rang est 0 dans le
# masque de bonus et ce bonus vaut (1,0)
```

```
>>> ranger_bonus({(1, 2): [(1, 0),(-1,0)], (1,3):[(-1,-1)]})
([[(1, 0), (-1, 0)], [(-1, -1)]], {(1, 2): 0, (1, 3): 1})
# Liste des bonus au rang 0 : [(1, 0),(-1,0)]
# Liste des bonus au rang 1 : [(-1, -1)]
# Rang du bonus en case (1,2) : 0
# Rang du bonus en case (1,3) : 1
# => Sur la case (1,2), il y a deux bonus dont le rang est 0 dans le
# masque de bonus et ces bonus valent [(1,0),(-1,0)]
# => Sur la case (1,3), il y a un bonus dont le rang est 1 dans le
# masque de bonus et ce bonus vaut (-1,-1)
```

3. Écrire une fonction :

   `trouver_sauts_possibles(sauts,bonus_au_rang,masque_bonus)`

   … qui, étant donnés les sauts par défaut `sauts`, la liste `bonus_au_rang` renvoyée par `ranger_bonus(bonus)` et le masque des bonus activés `masque_bonus`, renvoie l'ensemble des sauts possibles.

```python
def trouver_sauts_possibles(sauts,bonus_au_rang,masque_bonus):
    # Exemples de paramètres:
    # sauts = [(0, 1), (1, -1), (1, 1)]
    # bonus_au_rang = [[(1, 0), (0, 1)], [(-1, -1)], [(-1,1)]]
    # masque_bonus = [False, True, False]

    # Les sauts par défaut sont des sauts possibles
    sauts_possibles = sauts[:]

    # Pour chaque k=True, récupère les sauts associés
    # et les ajoute aux sauts possibles
    for k in range(len(masque_bonus)):
        if masque_bonus[k] == True:
            # ajoute les sauts associés au rang k
            for saut in bonus_au_rang[k]:
                sauts_possibles.append(saut)
    return sauts_possibles
```

4. Tester la fonction :

   `ajouter_bonus(bonus,rang_du_bonus,i,j,bonus_actifs,code_bonus_actifs)`

   … qui active la case bonus (i, j) dans le code des cases bonus activées `code_bonus_actifs`.

   Si (i, j) n'est pas une case bonus, la fonction renvoie `code_bonus_actifs`. L'argument `rang_du_bonus` est la structure renvoyée par `ranger_bonus(bonus)` et l'argument `bonus_actifs` est le masque des bonus actifs dont le code est `code_bonus_actifs`.

   # Liste des bonus en case (1,2) et (2,1) :
   >>> bonus = {(1, 2): [(1, 0)], (2, 1): [(0, 1)]}
   # Tente d'ajouter un bonus sur une case qui n'en contient pas
   >>> ajouter_bonus(bonus, rang_du_bonus, 0, 0, [False, False], 0)
   0 # <= Retourne le code_bonus initial

   # Ajoute le bonus de la case (1,2) (rang 0) à un masque [False,False] = 0
   ajouter_bonus(bonus, rang_du_bonus, 1, 2, [False, False], 0)
   1 # <= [False,False] est passé à [True,False] = 1

   # Ajoute le bonus de la case (2,1) (rang 1) à un masque [False,False] = 0
   ajouter_bonus(bonus, rang_du_bonus, 2, 1, [False, False], 0)
   2 # <= [False,False] est passé à [False,True] = 2

   # Ajoute le bonus de la case (2,1) (rang 1) à un masque [True,False] = 1
   ajouter_bonus(bonus, rang_du_bonus, 2, 1, [True, False], 1)
   3 # <= [True,False] est passé à [True,True] = 3

5. Compléter les sept parties manquantes indiquées par << ... >> dans le code.

   (Voir fichier « 8.2. TD7 – JeuDeRockse – Correction.py » ou page suivante.

   # poids_opt, saut_opt = trouve_dynamique(T,sauts,bonus)
   # poids_opt2, saut_opt2 = trouve_dynamique(T,sauts,{})

   # Poids optimal depuis la case (0,0) avec bonus en case (1,2) non activé :
   # poids_opt[0][0][0] => -7 : Valeur cohérente

   # Poids optimal sans bonus en case (1,2) dans le jeu :
   # poids_opt2[0][0][0] => -6 : Valeur cohérente

   # Sauts optimaux depuis la case (1,1) avec bonus en case (1,2) disponible
   # saut_opt[1][1] => [(0, 1, 0), (1, -1, 1)] : Va bien à droite (0, 1) car le bonus
   # n'a pas encore pu être activé

   # Sauts optimaux depuis la case (1,2) avec bonus en case (1,2) disponible :
   # si on est en (1,2) avec masque 0, la transition doit activer le bonus (masque 1) et
   # rendre le saut (1,0) utilisable.
   # saut_opt[1][1] => [(1, 0, 1), (1, 0, 1)] : Le bonus est activé et on va en bas (1,0)

```python
def trouve_dynamique(T,sauts,bonus):
    N = len(T)
    nb_bonus = len(bonus)
    nb_code_bonus = 2**nb_bonus
    # Format poids_opt : (i,j,nb_code_bonus)
    poids_opt = [[[INFINI for bonus_code in range(nb_code_bonus)]
                          for j in range(N)]
                          for i in range(N)]
    # Format saut_opt : (i,j,nb_code_bonus)
    saut_opt = [[[(0,0,0) for bonus_code in range(nb_code_bonus)]
                          for j in range(N)]
                          for i in range(N)]
    # Récupère les bonus et les rangs associés
    (bonus_au_rang ,rang_du_bonus) = ranger_bonus(bonus)
    # Itère sur toutes les combinaisons des masques bonus possibles
    # par ordre décroissant au sens de l'inclusion
    for bonus_actifs in combinaison_bonus(nb_bonus):
        # Récupère le code_bonus associé au masque courant
        code_bonus_actifs = code_bonus(bonus_actifs)
        # Cas de base sur la case (N-1, N-1)
        poids_opt[N-1][N-1][code_bonus_actifs] = T[N-1][N-1]
        # Récupère les sauts possibles depuis la case finale
        # pour les différentes possibilités des masques bonus
        sauts_possibles = trouver_sauts_possibles(sauts,bonus_au_rang,bonus_actifs)
        # Itération à l'envers sur les (i,j) en partant de (N-1)->0
        for i in range(N-1,-1,-1):
            for j in range(N-1,-1,-1):
                if i == N-1 and j == N-1:
                    continue
                # Mise à jour des bonus disponibles sur la case (i,j)
                # pour le masque bonus_actif courant
                code_bonus_dest = ajouter_bonus(bonus,rang_du_bonus,i,j,
                                                bonus_actifs,code_bonus_actifs)
                # Si (i,j) est dans les bonus -> ajoute le bonus aux sauts possibles
                # depuis cette case
                if (i,j) in bonus:
                    sauts_possibles_final = sauts_possibles + bonus[(i,j)]
                else:
                    sauts_possibles_final = sauts_possibles
                # Recherche du poids_opt[i_s][j_s][code_bonus] minimum
                # parmi tous les successeurs
                for (delta_i ,delta_j) in sauts_possibles_final:
                    # Calcul de la case du successeur
                    i_dest = i + delta_i
                    j_dest = j + delta_j
                    # Si le successeur est bien dans la grille
                    if (i_dest in range(N) and j_dest in range(N)):
                        # Applique la récurrence
                        poids_opt_dest = poids_opt[i_dest][j_dest][code_bonus_dest]
                        # Compare le poids et enregistre le poids minimum
                        # dans poids_opt[i][j][code_bonus]
                        if (poids_opt[i][j][code_bonus_actifs] > poids_opt_dest):
                            poids_opt[i][j][code_bonus_actifs] = poids_opt_dest
                            saut_opt[i][j][code_bonus_actifs] =
                                                (delta_i,delta_j,code_bonus_dest)
                # Ajoute T[i][j] au poids du successeur optimum trouvé
                poids_opt[i][j][code_bonus_actifs] += T[i][j]
    return (poids_opt, saut_opt)
```

6. Quelle est la complexité de cette fonction ?

N : la taille de la grille (NxN)
n : le nombre de bonus disponibles
S_{max} : un majorant du nombre de sauts testés : $S_{max} = O(N^2)$ (tous les (δ_i, δ_j) possibles)

La structure du programme est :
- **boucle principale sur tous les masques de bonus : 2^n**
- **pour chaque masque, double boucle sur (i, j) : N^2**
- **pour chaque (i, j), boucle sur tous les sauts possibles : $\leq N^2$**
- **opérations internes en O(1) (comparaison, additions, accès aux éléments)**

Donc la complexité en temps est en $O(2^n \cdot N^2 \cdot Smax) = O(2^n \cdot N^4)$

Il y a aussi l'appel de la fonction `trouver_sauts_possibles()` appelé une fois par masque. Cette fonction construit la liste des sauts disponibles en parcourant les bonus actifs, son coût est de $O(S_{max}) = O(N^2)$.

La complexité totale en temps est donc de $O(2^n \cdot (N^2 + N^2 \cdot Smax)) = O(2^n \cdot N^4)$

En mémoire, les listes `poids_opt` et `saut_opt` contiennent chacun $N^2 \cdot 2^n$ entrées et donc en $O(N^2 \cdot 2^n)$.

7. Écrire la fonction `solution_dynamique(saut_opt,N)` qui, étant donnés la structure `saut_opt` calculée dans la question précédente et la dimension N de la grille, renvoie le chemin optimal correspondant. La fonction renvoie le chemin vide [] s'il n'existe pas de chemin entre la case de départ et celle d'arrivée.

```python
def solution_dynamique(saut_opt,N):
    # case de départ
    case = (0,0)
    code_bonus = 0

    # Sauvegarde du chemin
    chemin = [case]

    # Tant qu'on n'est pas arrivé sur la case finale
    while case[0] != N-1 or case[1] != N-1:
        # Récupère le saut optimum
        saut = saut_opt[case[0]][case[1]][code_bonus]

        if (saut[0], saut[1], saut[2]) == (0, 0, 0):
            return []  # pas de chemin

        # Applique le saut et le code
        case = (case[0] + saut[0], case[1] + saut[1])

        # Met à jour le code_bonus
        code_bonus = saut[2]
        # Met à jour le chemin
        chemin.append(case)

    return chemin
```

```
# Grille de jeu du sujet
T = [[2, -4, -6, 0],
   [1, -2, 2, 3],
   [-2, 2, -3, 4],
   [-1, 4, -3, 7]]

sauts = [(0, 1), (1, -1), (1, 1)]   # Sauts par défaut
bonus = {(1, 2): [(1, 0)]}         # Case bonus en (1,2)
poids_opt, saut_opt = trouve_dynamique(T,sauts,bonus)
```

⇨ [(0, 0), (0, 1), (0, 2), (1, 1), (1, 2), (2, 2), (3, 2), (3, 3)]

```
# Chemin impossible
T = [[0, 0, 0, 0],
    [0, 0, 0, 0],
    [0, 0, 0, 0],
    [0, 0, 0, 0]]

sauts = [(1, 0)]         # on descend uniquement
bonus = {}               # pas de bonus

poids_opt, saut_opt = trouve_dynamique(T,sauts,bonus)
```

⇨ []