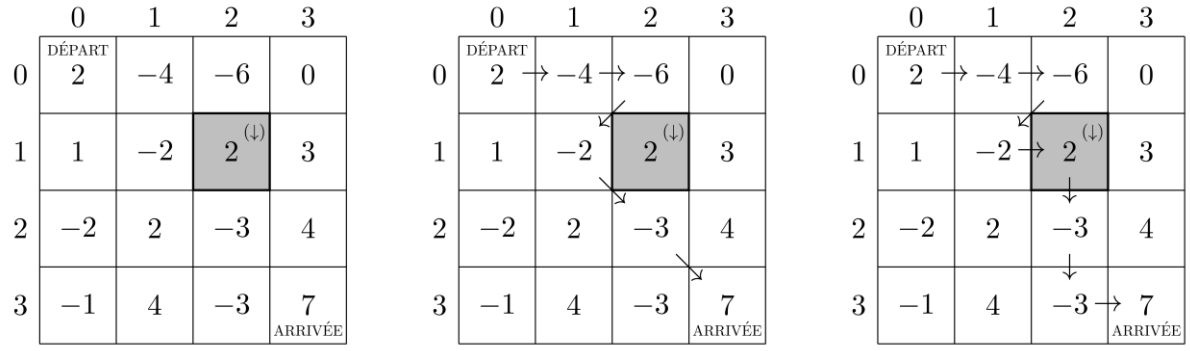
TD : PROGRAMMATION DYNAMIQUE – JEU DE RÖCKSE

(TD inspiré de XENS-MP-PC-PSI Info 2025)

On dispose sur les cases d'une grille N × N des pénalités et des gains comptés comme des pénalités négatives. Le jeu de Röckse débute à la case (0, 0) et cherche un chemin vers la case (N - 1,N - 1) qui *minimise* les *pénalités*. À chaque étape du chemin, un nombre fini de déplacements (*sauts*) est autorisé. Des *cases bonus* ajoutent, une fois atteintes, des sauts possibles pour la suite du chemin. La figure ci-dessous donne un exemple de grille et deux chemins possibles, en supposant que les sauts autorisés sur la grille sont (i, j) → (i, j + 1) (→), (i, j) → (i + 1, j - 1) (↙) et (i, j) → (i + 1, j + 1) (↘). On suppose par ailleurs une case bonus en (1, 2) (grisée) qui, une fois atteinte, ajoute aux sauts autorisés (i, j) → (i + 1, j) (↓) :



(a) Grille de jeu

(b) Chemin non-optimal
poids : −6

(c) Chemin optimal
poids : −7

Considérons le chemin décrit en (c). On part de la case de départ (0, 0) et, en appliquant les sauts successifs →, →, ↙, → on arrive sur la case bonus. À partir de cette case, le saut ↓ est autorisé. On applique ensuite les sauts ↓, ↓, → pour atteindre la case d'arrivée (3, 3). Le chemin obtenu est décrit par la liste des cases :

```
Chemin = [(0,0), (0,1), (0,2), (1,1), (1,2), (2,2), (3,2), (3,3)]
```

Son *poids* est la somme des pénalités contenues dans ces cases, soit -7. On peut montrer que c'est un chemin de poids minimal — on dit qu'il est *optimal*. Le chemin décrit en (b) est correct, mais son poids est de -6. Il n'est donc pas optimal.

Représentation des données. La grille de jeu N × N est représentée par une liste de listes d'entiers T telle que T[i][j] est la pénalité à la case (i, j). On suppose que cette représentation est bien formée, c'est-à-dire que toutes les listes ont la même taille N. Sur notre exemple :

```
T = [[2,-4,-6,0], [1,-2,2,3], [-2,2,-3,4], [-1,4,-3,7]]
```

Un saut (i, j) → (i + δ_i, j + δ_j) est représenté par le couple d'entiers (δ_i, δ_j). L'ensemble des sauts possibles est ainsi une liste de couples. Sur notre exemple, l'ensemble des sauts possibles par défaut (avant d'avoir atteint une case bonus) est :

```
sauts = [(0 ,1), (1 , -1), (1 ,1)]
```

Les sauts activés par les cases bonus sont enregistrés dans un dictionnaire bonus tel que `bonus[(i, j)]` est la liste des sauts activés par la case bonus (i, j). Sur notre exemple :

```
bonus = {(1 ,2): [(1 ,0)]}
```

Complexité. La complexité d'une fonction F est le nombre d'opérations élémentaires (addition, multiplication, affectation, test, etc.) nécessaires à l'exécution de F. Lorsque cette complexité dépend de plusieurs paramètres n et m, on dira que F a une complexité en $O(\phi(n,m))$ lorsqu'il existe trois constantes A, n_0 et m_0 telles que la complexité de F est inférieure ou égale à $A \cdot \phi(n,m)$, pour tout $n \geq n_0$ et $m \geq m_0$. Lorsqu'il est demandé de donner la complexité d'un programme, le candidat devra justifier sa réponse en utilisant le code du programme.

Rappels sur Python. L'utilisation de toute fonction Python sur les listes ou sur les dictionnaires autre que celles mentionnées dans ce paragraphe est interdite. Sur les listes, on autorise les opérations suivantes, *dont la complexité* est en O(1) :

- `len(L)` renvoie la longueur de la liste `L`.
- `L[i]` désigne l'élément d'indice `i` de la liste `L`, pour $0 \leq$ `i` $<$ `len(L)`.
- `L.append(e)` ajoute en place l'élément `e` à la fin de la liste `L`.

Les opérations suivantes sont également autorisées et leur complexité est en O(n) :

- `L1 + L2` renvoie une nouvelle liste (de longueur n) qui est la concaténation des listes L1 et L2.
- `range(n)` renvoie la liste [0, 1, ..., n-1].
- `range(n-1,-1,-1)` renvoie la liste [n-1, ..., 0].
- `L.pop(0)` retire le premier élément `e` de la liste `L` (de longueur n) et renvoie `e`.
- `(e in L)` renvoie `True` si l'élément `e` est dans la liste `L` (de longueur n), et `False` sinon.
- `L[:]` renvoie une copie de la liste `L` (de longueur n).

On autorise également les constructions suivantes :

- La construction `for e in L` parcourt (itère sur) les éléments de la liste `L` du premier élément (d'indice 0), au dernier élément (d'indice `len(L)-1`) avec la complexité O(`len(L)`).
- La construction `[f(e) for e in L]` produit la même liste que le code suivant, et avec la complexité `len(L)` fois la complexité de `f` :

```
result = []
for e in L:
    result.append(f(e))
return result
```

Sur les dictionnaires, on autorise uniquement les opérations suivantes, dont la complexité est en O(1) :

- Le test `(e in d)` renvoie `True` si `e` est une clé du dictionnaire `d`, et `False` sinon.
- L'accès `d[e]` à l'élément associé à la clé `e` dans le dictionnaire `d`.

On autorise également les constructions suivantes sur les dictionnaires :

- La construction `for (k,v) in d` parcourt les éléments du dictionnaire d.
- La construction `d[k] = v` affecte la valeur `v` à la clé `k`.

Enfin, on supposera donnée une variable globale `INFINI` qui contient un entier supérieur ou égal à tout entier utilisé dans les programmes de ce sujet.

Organisation : Dans ce TD, on commence par récapituler les deux premières parties du sujet original :

- Partie I : porte sur les fonctions de base sur les chemins et les sauts ;
- Partie II : propose de trouver un chemin optimal avec une recherche exhaustive ;

Ensuite, dans la partie III du sujet original on utilise les résultats de la partie II pour construire une méthode de recherche gloutonne.

Enfin, la partie IV du sujet original étudie une résolution du problème par programmation dynamique.

Vous utiliserez les fichiers Python :

- « 8.2. TD7 - JeuDeRockse.py » : Fichier source à utiliser pour faire vos essais et à compléter ;
- « LibJeuDeRockse.py » : Fichier de bibliothèque à placer dans le même répertoire que le fichier source.

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

Cette mise en situation reprend l'ensemble des questions des trois premières parties du sujet et demande de tester les fonctions qui étaient demandées à être écrites (ainsi que de répondre à quelques questions théoriques associées), afin de se familiariser avec le problème du sujet.

Les fonctions sont déjà écrites et vous devez trouver par vous-même des exemples pour les tester.

I.1. Partie I du sujet original : Sauts et chemins

La grille du jeu donnée dans l'exemple du sujet, la liste des sauts, le bonus en case (1,2) et le chemin optimal sont renseignés dans le fichier Python :

```
# 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)
chemin_optimal = [(0, 0), (0, 1), (0, 2), (1, 1), (1, 2), (2, 2),
                  (3, 2), (3, 3)]
```

1. Tester la fonction `poids(T,chemin)`, étant donné un plateau de jeu T et un chemin `chemin`, renvoie le poids de ce chemin :
2. La fonction est donnée ci-dessous : quelle est sa complexité ?

```
def poids(T, chemin):
    total = 0
    for (i, j) in chemin:
        total += T[i][j]
    return total
```

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.

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.

On dit qu'un ensemble de sauts C est bien formé s'il ne peut pas mener à un cycle. Autrement dit, s'il n'existe pas de suite de sauts construite à partir des sauts de C qui, en partant de la case (0, 0) permettrait d'arriver sur une case (i, j) puis de revenir à cette case. Une condition suffisante est que chaque saut (δ_i, δ_j) de l'ensemble de sauts C soit strictement positif lexicographiquement, condition notée (δ_i, δ_j) $\gg 0$ et définie par :

$$\delta_i > 0 \; ou \; bien \; (\delta_i = 0 \; et \; \delta_j > 0) \quad (*)$$

Une suite de sauts $\vec{\delta}$ est positive lexicographiquement, propriété notée $\vec{\delta} \gg 0$, si chaque saut (δ_i, δ_j) de $\vec{\delta}$ satisfait (*).

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.

Dans le reste du sujet, on suppose que les sauts utilisés satisfont la condition (*).

I.2. Partie II du sujet original : Recherche exhaustive

On cherche maintenant à calculer un chemin optimal. Une première solution est de tester tous les chemins corrects et de retenir le chemin de poids minimum. L'énumération de tous les chemins corrects se fera avec la fonction auxiliaire récursive suivante :

```
trouve_complet_rec(T,sauts,bonus,sauts_max,i,j)
```

Étant donné une grille de jeu T, les sauts par défaut `sauts`, les sauts associés aux cases bonus `bonus`, une valeur entière `sauts_max` et une case (i, j), la fonction ci-dessus calcule un chemin $\vec{p}$ de poids minimum partant de (i, j).

Le chemin $\vec{p}$ n'arrive pas forcément à (N-1, N-1), mais il a au plus `sauts_max`+1 cases. La fonction renvoie le couple `(poids_min, sauts_min)` où `poids_min` est le poids de $\vec{p}$ et `sauts_min` est la liste des sauts pour construire $\vec{p}$.

La valeur `sauts_max` est utilisée pour limiter la complexité de la recherche. Ainsi, la longueur du résultat `sauts_min` sera inférieure ou égale à `sauts_max`. Cette limite est appelée horizon. Si l'horizon est assez grand, la fonction renvoie un chemin optimal partant de (i, j).

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.

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.

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

```
def trouve_complet(T, sauts_defaut, bonus, sauts_max, i=0, j=0):
    # Taille de la grille : NxN
    N = len(T)

    # (i,j) : case de départ
    def trouve_complet_rec(i, j, horizon, sauts_disponibles):
        if (i, j) in bonus:
            for saut_bonus in bonus[(i, j)]:
                if saut_bonus not in sauts_disponibles:
                    sauts_disponibles = sauts_disponibles + [saut_bonus]

        # Cas de base : arrivée atteinte
        if i == N-1 and j == N-1:
            return (T[i][j], [])

        # Cas de base : horizon épuisé
        if horizon == 0:
            return (T[i][j], [])

        # On DOIT continuer si un saut est possible
        meilleur_poids = INFINI
        meilleurs_sauts = []

        for (delta_i, delta_j) in sauts_disponibles:
            i_dest = i + delta_i
            j_dest = j + delta_j

            if 0 <= i_dest < N and 0 <= j_dest < N:
                (poids_dest, sauts_dest) = trouve_complet_rec(
                    i_dest, j_dest, horizon - 1, sauts_disponibles[:])

                poids_total = T[i][j] + poids_dest
                if poids_total < meilleur_poids:
                    meilleur_poids = poids_total
                    meilleurs_sauts = [(delta_i, delta_j)] + sauts_dest

        # Si aucun saut valide, on est bloqué ici
        if meilleur_poids == INFINI:
            return (T[i][j], [])

        return (meilleur_poids, meilleurs_sauts)
    return trouve_complet_rec(0, 0, sauts_max, sauts_defaut[:])
```

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 ?

Dans la suite de ce TD, les fonctions qui sont demandées à être testées n'avaient pas à être écrites par le candidat et étaient supposées disponibles dans le sujet.

II) RECHERCHE GLOUTONNE

On peut réduire la complexité de la recherche exhaustive en limitant, à chaque étape, l'horizon de la recherche, c'est-à-dire le nombre de sauts regardés à partir de la case courante. D'où l'idée de construire un algorithme glouton pour trouver une solution. En partant de la case (0, 0), on utilise la recherche exhaustive avec un « petit horizon » k pour déterminer la meilleure suite locale de k sauts, c'est-à-dire la suite de poids minimal et d'au plus k sauts. On joue les sauts de la meilleure suite locale, puis on recommence sur la case atteinte, jusqu'à la case d'arrivée (N-1, N-1). Si la recherche exhaustive avec l'horizon k ne trouve pas une suite locale, alors l'algorithme abandonne la recherche.

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 ?

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 ?

III) RECHERCHE PAR PROGRAMMATION DYNAMIQUE

On va construire une méthode par programmation dynamique pour trouver un chemin optimal. Comme la solution optimale en partant de (i, j) dépend des cases bonus déjà rencontrées (cases activées), on remplit le tableau `poids_opt[i][j][code_bonus]` qui contient le poids du chemin optimal en partant de la case (i, j) et où la 3ème dimension (`code_bonus`) encode l'ensemble des cases bonus activées.

En parallèle, on remplira un tableau `saut_opt[i][j][code_bonus]` qui contient un saut optimal à jouer en partant de la case (i, j). Ce tableau permettra de retrouver un chemin optimal.

III.1. Encodage des cases bonus activées

Soit n le nombre de cases bonus. On numérote les cases bonus de 0 à n - 1. On représente les cases bonus activées par une liste de booléens (masque binaire) [b_0, …, b_{n-1}] où b_k vaut `True` si et seulement si la case bonus k est activée. L'ensemble des cases bonus activées est encodé par l'entier dont la représentation binaire est $\hat{b}_{n-1} \dots \hat{b}_0$, où `Trûe = 1` et `Fâlse = 0`. Ce code est noté <b_0 … b_{n-1}>. Par exemple, le code associé au masque `[False, False]` est 0, le code associé au masque `[True, False]` est 1, etc.

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.

III.2. Récurrence

On va maintenant donner une récurrence pour le calcul de `poids_opt[i][j][code_bonus]`, pour toute case (i, j) et valeur de `code_bonus`. Cette récurrence sera utilisée dans la section suivante pour construire un algorithme.

Pour simplifier l'explication, ignorons les cases bonus dans un premier temps. Si un chemin partant de la case (i, j) a un poids minimal, alors le chemin obtenu en lui retirant (i, j) (en partant de la case suivante) a lui-même un poids minimal. Une fois `poids_opt` calculé pour tous les successeurs possibles de la case (i, j), il suffit de garder le plus petit et de lui ajouter le poids de la case T[i][j]. On obtient ainsi `poids_opt` pour la case (i, j).

En notant Γ l'ensemble des sauts par défaut, la définition de poids_opt[i][j] est dans ce cas :

$$poids\_opt[i][j] = T[i][j] + \min_{(\delta_i,\delta_j) \in \Gamma} \big(poids\_opt[i+\delta_i][j+\delta_j]\big)$$

Avec les cases bonus, c'est le même principe sauf qu'il faut gérer les sauts supplémentaires des cases bonus activées. Deux cas sont possibles :

- (i, j) **n'est pas une case bonus**. Dans ce cas, il suffit de calculer `poids_opt[i_s][j_s][code_bonus]` pour tous les successeurs possibles (i_s, j_s) de la case (i, j) avec les sauts par défaut et les sauts associés à chaque case bonus activée de code_bonus. Le `poids_opt[i][j][code_bonus]` est alors le minimum de ces `poids_opt[i_s][j_s][code_bonus]` auquel s'ajoute la pénalité T[i][j] de la case (i, j).
- (i, j) **est une case bonus**. Dans ce cas, c'est le même processus, sauf que : 1) parmi les sauts possibles, on ajoute ceux activés par la case bonus et 2) on considère les successeurs (i_s, j_s) avec un nouveau code bonus `code_bonus'` dans lequel la case bonus (i, j) est activée : le minimum doit ainsi être calculé parmi les `poids_opt[i_s][j_s][code_bonus']`.
  Si code_bonus = <b_0 … b_{n-1}>, en notant k le numéro de la case bonus (i, j), on changera b_k à True pour obtenir `code_bonus'` = <b_0 … b'_k … b_{n-1}>, où b'_k = True.

1. Écrire la définition de `poids_opt[i][j][`<b_0 … b_{n-1}>`]`. On notera Γ l'ensemble des sauts par défaut et Δ_k l'ensemble des sauts associé à la k-ième case bonus.

III.3. Algorithme

On évalue itérativement les différentes cases `poids_opt[i][j][code_bonus]` de `poids_opt` à l'aide de trois boucles imbriquées en itérant respectivement sur i, j et le masque de bonus (d'où on tirera code bonus). La difficulté est de trouver un ordre d'évaluation correct. Les `poids_opt[i][j][`<b_0 … b_{n-1}>`]` doivent être évalués après avoir obtenu les poids des successeurs : `poids_opt[i+`δ_i`][j+`δ_j`][`<b_0'… b_{n-1}'>`]`.

- Comme (δ_i, δ_j) >> 0 (condition (*)), alors i + δ_i > i ou bien δ_i = 0 et j + δ_j > j, donc les itérations sur i et j doivent être « à l'envers », de N - 1 à 0.
- Soit `[`b'_0, …,b'_{n-1}`]` est égal à `[`b_0, …,b_{n-1}`]`, soit `[`b'_0, …,b'_{n-1}`]` est obtenu à partir de `[`b_0, …,b_{n-1}`]` en mettant un b_k à `True`. En d'autres termes, le choix de bonus représenté par `[`b_0, …, b_{n-1}`]` est inclus dans le choix de bonus représenté par `[`b'_0, …,b'_{n-1}`]`. La boucle sur les masques de bonus doit donc itérer par choix de bonus décroissant au sens de l'inclusion.

Avec 3 cases bonus, un ordre est le suivant :

`[True,True,True]`, puis
`[False,True,True]`, `[True,False,True]`, `[True,True,False]`, puis
`[False,False,True]`, `[False,True,False]`, `[True,False,False]`, puis
`[False,False,False]`

Noter que les choix rassemblés sur une ligne ne sont pas classables entre eux. Par contre, les choix d'une ligne sont strictement supérieurs au sens de l'inclusion à l'un des choix de la ligne suivante.

Un algorithme pour calculer l'ordre d'évaluation des masques de bonus peut procéder comme suit. On commence par le choix total, dans notre exemple `[True,True,True]`. On construit la ligne suivante en insérant les choix obtenus en basculant une coordonnée `True` à `False`. Par exemple, on insère `[False,True,True]`, `[True,False,True]`, `[True,True,False]`. On itère ensuite sur chaque choix obtenu pour construire la ligne d'après. On s'arrête lorsqu'on atteint le choix vide, dans notre exemple `[False,False,False]`. On pourra utiliser une *file* pour défiler les choix de bonus à traiter et enfiler progressivement les choix de bonus de la ligne suivante.

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.

```
Tester :    >>> combinaisons_bonus(3)
            [[True, True, True], [False, True, True], [True, False,
            True], [True, True, False], [False, False, True], [False,
            True, False], [True, False, False], [False, False,
            False]]
```

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

Le résultat de cette fonction est utilisé comme paramètre pour la question suivante et pour la fonction `ajouter_bonus` décrite après.

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.

```
Tester :  >>> sauts = [(0, 1), (1, -1), (1, 1)]
          >>> bonus = {(1, 2): [(1, 0),(-1,0)], (1,3):[(-1,-1)]}
          >>> bonus_au_rang, rang_du_bonus = ranger_bonus(bonus)
          >>> trouver_sauts_possibles(sauts, bonus_au_rang, [False,False])
          [(0, 1), (1, -1), (1, 1)]
```

```
>>> trouver_sauts_possibles(sauts, bonus_au_rang, [False,True])
[(0, 1), (1, -1), (1, 1), (-1, -1)]
>>> trouver_sauts_possibles(sauts, bonus_au_rang, [True,False])
[(0, 1), (1, -1), (1, 1), (1, 0), (-1, 0)]
>>> trouver_sauts_possibles(sauts, bonus_au_rang, [True,True])
[(0, 1), (1, -1), (1, 1), (1, 0), (-1, 0), (-1, -1)]
```

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`.

Le code Python incomplet de la page suivante (également donné dans le fichier source du TD) implémente la fonction `trouve_dynamique(T,sauts,bonus)` qui, étant donnés une grille de jeu T, la liste des sauts par défaut `sauts` et les sauts associés aux cases bonus `bonus`, calcule le chemin optimal par programmation dynamique en utilisant la récurrence trouvée en question dans la partie III.2. Le résultat de la fonction est le couple `(poids_opt,sauts_opt)` où `poids_opt` est le poids du chemin optimal trouvé et `sauts_opt` est la liste des sauts de ce chemin.

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

6. Quelle est la complexité de cette fonction ?

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.

```
def trouve_dynamique(T,sauts,bonus):
    N = len(T)
    nb_bonus = .......
    nb_code_bonus = 2**nb_bonus
    poids_opt = [[[INFINI for bonus_code in range(nb_code_bonus)]
                          for j in range(N)]
                          for i in range(N)]
    saut_opt = [[[(0,0,0) for bonus_code in range(nb_code_bonus)]
                          for j in range(N)]
                          for i in range(N)]
    (bonus_au_rang ,rang_du_bonus) = ranger_bonus(bonus)
    for bonus_actifs in combinaison_bonus(nb_bonus):
        code_bonus_actifs = code_bonus(bonus_actifs)
        poids_opt[N-1][N-1][code_bonus_actifs] = .....
        sauts_possibles = ........
        for i in range(.......):
            for j in range(......):
                if i == N-1 and j == N-1:
                    continue
                code_bonus_dest = ajouter_bonus(bonus,rang_du_bonus,i,j,
                                                bonus_actifs,code_bonus_actifs)
                if (i,j) in bonus:
                    sauts_possibles_final = sauts_possibles + bonus[(i,j)]
                else:
                    sauts_possibles_final = sauts_possibles
                for (delta_i ,delta_j) in sauts_possibles_final:
                    i_dest = i + delta_i
                    j_dest = j + delta_j
                    if (i_dest in range(N) and j_dest in range(N)):
                      poids_opt_dest = poids_opt[i_dest][j_dest][code_bonus_dest]
                      if (poids_opt[i][j][code_bonus_actifs] > poids_opt_dest):
                             poids_opt[i][j][code_bonus_actifs] = .........
                             saut_opt[i][j][code_bonus_actifs] = ..........
                poids_opt[i][j][code_bonus_actifs] += T[i][j]
    return (poids_opt, saut_opt)
```