ECOLE POLYTECHNIQUE - ESPCI
ECOLES NORMALES SUPERIEURES

CONCOURS D'ADMISSION 2025

JEUDI 17 AVRIL 2025
16h30 - 18h30

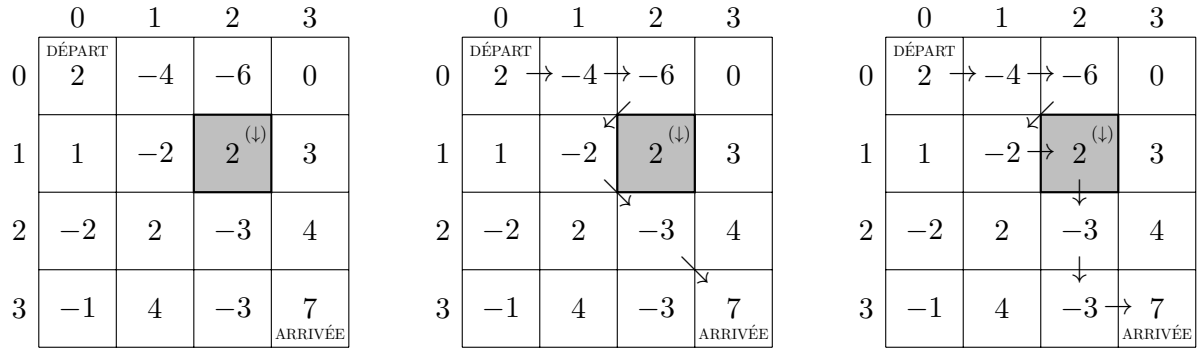
FILIERES MP-PC-PSI
Epreuve n° 8
INFORMATIQUE B (XELSR)

Durée : 2 heures

L'utilisation des calculatrices n'est pas autorisée pour cette épreuve

Le jeu de Röckse

On dispose sur les cases d'une grille $N \times N$ 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 $(i, j) \to (i, j+1)$ ($\to$), $(i, j) \to (i+1, j-1)$ ($\swarrow$) et $(i, j) \to (i+1, j+1)$ ($\searrow$). On suppose par ailleurs une case bonus en $(1, 2)$ (grisée) qui, une fois atteinte, ajoute aux sauts autorisés $(i, j) \to (i+1, j)$ ($\downarrow$) :



(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 $\to, \to, \swarrow, \to$, on arrive sur la case bonus. À partir de cette case, le saut $\downarrow$ est désormais autorisé. On applique ensuite les sauts $\downarrow, \downarrow, \to$ 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 $N \times N$ 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 $(i, j) \to (i + \delta i, j + \delta j)$ est représenté par le couple d'entiers $(\delta i, \delta j)$. 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. Les parties peuvent être traitées indépendamment. La partie I porte sur les fonctions de base sur les chemins et les sauts. Ensuite, la partie II propose de trouver un chemin optimal avec une recherche exhaustive. La partie III utilise les résultats de la partie II pour construire une méthode de recherche gloutonne. Enfin, la partie IV étudie une résolution du problème par programmation dynamique. **On rappelle que le code doit être commenté.**

Partie I : Sauts et chemins

Question 1 *Écrire une fonction* `poids(T,chemin)` *qui, étant donné un plateau de jeu* `T` *et un chemin* `chemin`*, renvoie le poids de ce chemin. Donner la complexité de cette fonction.*

Question 2 *Écrire une 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.*

Question 3 *Écrire une 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 $(\delta i, \delta j)$ de l'ensemble de sauts C soit strictement positif lexicographiquement, condition notée $(\delta i, \delta j) \gg 0$ et définie par :

$$\delta i > 0 \text{ ou bien } (\delta i = 0 \text{ et } \delta j > 0) \qquad \textbf{(*)}$$

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

Question 4 *Écrire une 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 (*).

Partie II : 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 donnés 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) et n'arrivant pas forcement à $(N - 1, N - 1)$, mais ayant 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).

Question 5 *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*.*

Question 6 *Écrire la fonction auxiliaire* `trouve_complet_rec(T,sauts,bonus,sauts_max,i,j)` *et la fonction principale* `trouve_complet(T,sauts,bonus,sauts_max)` *qui renvoie le couple* `(poids,sauts)` *où* `poids` *est le poids du chemin trouvé et* `sauts` *est la liste des sauts du chemin trouvé. Quelle est la complexité de cette fonction ?*

Question 7 *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 ?*

La recherche exhaustive d'un chemin optimal est obtenue en appelant `trouve_complet(T, sauts, bonus, L)` avec `L` la longueur maximale d'un chemin dans `T`.

Partie III : 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.

Question 8 *Sur l'exemple donné en introduction, quel est le chemin trouvé pour $k = 2$? Est-il optimal ? Mêmes questions pour $k = 3$. En augmentant k, obtient-on forcément un chemin de poids plus petit ?*

Question 9 *Écrire une 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. Quand la recherche exhaustive avec l'horizon k ne trouve pas de suite locale, la fonction renvoie* `(INFINI,[])`*.*

Partie IV : Recherche par programmation dynamique

On se propose maintenant de 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 (dites *activées*), on va remplir un tableau `poids_opt[i][j][code_bonus]` qui contient le poids du chemin optimal en partant de la case `(i,j)` et où la 3^e 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.

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*) $[b_0, \ldots, b_{n-1}]$ 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 representation binaire est $\hat{b}_{n-1} \ldots \hat{b}_0$, où $\hat{\texttt{True}} = 1$ et $\hat{\texttt{False}} = 0$. Ce code est noté $\langle b_0 \ldots b_{n-1} \rangle$. Par exemple, le code associé au masque `[False, False]` est 0, le code associé au masque `[True, False]` est 1, etc.

Question 10 *Écrire la fonction* `code_bonus(masque_bonus)` *qui renvoie le code associé au masque binaire* `masque_bonus` *représentant l'ensemble des cases bonus activées.*

2) Récurrence

On va maintenant donner une récurrence pour calculer les valeurs `poids_opt[i][j][code_bonus]` pour tout 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).

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` $= \langle b_0 \dots b_{n-1} \rangle$, en notant k le numéro de la case bonus (i, j), on changera b_k à `True` pour obtenir `code_bonus'` $= \langle b_0 \dots b'_k \dots b_{n-1} \rangle$, où $b'_k =$ `True`.

Question 11 *Écrire la définition de* `poids_opt[i][j][`$\langle b_0 \dots b_{n-1} \rangle$`]`*. On notera* Γ *l'ensemble des sauts par défaut et* Δ_k *l'ensemble des sauts associé à la* k*-ième case bonus.*

3) Algorithme

On évalue itérativement les différentes cases `poids_opt[i][j][code_bonus]` de `poids_opt` à l'aide de trois boucles imbriquées 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. À partir de la récurrence, on voit que `poids_opt[i][j][`$\langle b_0 \dots b_{n-1} \rangle$`]` doit être évalué *après* avoir obtenu les poids des successeurs : `poids_opt[i+`δi`][j+`δj`][`$\langle b'_0 \dots b'_{n-1} \rangle$`]`.

— Comme $(\delta i, \delta j) \gg 0$ (condition **(*)**), alors $i + \delta i > i$ ou bien $\delta i = 0$ et $j + \delta j > j$, donc les itérations sur i et j doivent être « à l'envers », de $N - 1$ à 0.

— Soit $[b'_0, \dots, b'_{n-1}]$ est égal à $[b_0, \dots, b_{n-1}]$, soit $[b'_0, \dots, b'_{n-1}]$ est obtenu à partir de $[b_0, \dots, b_{n-1}]$ en mettant un b_k à `True`. En d'autres termes, le choix de bonus représenté par $[b_0, \dots, b_{n-1}]$ est inclus dans le choix de bonus représenté par $[b'_0, \dots, b'_{n-1}]$. 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.

Question 12 *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.*

On suppose qu'il existe une 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.

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

Question 13 *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.*

On suppose donnée une 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`.

Question 14 *Le code Python incomplet de la page suivante 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 11. 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.*

1. *Donner les sept parties manquantes indiquées par « ... » dans le code.*
2. *Quelle est la complexité de cette fonction ?*

```
def trouve_dynamique(T,sauts,bonus):
  N = len(T)
  nb_bonus = len(bonus)
  nb_code_bonus = ...                                    # A COMPLETER (1)
  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 combinaisons_bonus(nb_bonus):
    code_bonus_actifs = code_bonus(bonus_actifs)
    poids_opt[N-1][N-1][code_bonus_actifs] = ...         # A COMPLETER (2)
    sauts_possibles = ...                                # A COMPLETER (3)
    for i in range(...):                                 # A COMPLETER (4)
      for j in range(...):                               # A COMPLETER (5)
        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] = ... # A COMPLETER (6)
              saut_opt[i][j][code_bonus_actifs] = ...  # A COMPLETER (7)
        poids_opt[i][j][code_bonus_actifs] += T[i][j]

  return (poids_opt,saut_opt)
```

Question 15 *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.*

– | F | I | N | –