Le jeu de Röckse

I Sauts et chemins

Q1 La fonction `poids` somme le poids des cases parcourues sur le chemin et renvoie la somme totale.

```
2  def poids(T, chemin):
3      p = 0 # poids du chemin
4      for (i, j) in chemin:
5          p += T[i][j]
6      return p
```

Complexité de la fonction `poids` :

La complexité de la fonction dépend de n `= len(chemin)`.

- À l'extérieur de la boucle `for`, les instructions sont en $\mathcal{O}(1)$.
- Le nombre de tours de la boucle `for` est égal à n.
  À chaque tour de boucle, les instructions exécutées dans le corps de boucle sont en $\mathcal{O}(1)$. La complexité de la boucle `for` est donc en $\mathcal{O}(n)$.

Par somme, la complexité de la fonction est en $\mathcal{O}(n)$ **où** n **`= len(chemin)`**.[1]

Q2 La fonction `appliquer_sauts` somme les variations d'abscisse et d'ordonnée à partir de la case de coordonnées (i, j) et renvoie les coordonnées de la case d'arrivée, supposée être dans la grille.

```
15 def appliquer_sauts(i, j, sauts):
16     x, y = i, j
17     for (dx, dy) in sauts:
18         x, y = x + dx, y + dy
19     return x, y
```

Q3 La fonction `sauts_corrects` vérifie si un chemin donné (qui reste dans la grille) est cohérent avec les sauts autorisés donnés par `saut` et `bonus`.

On suppose $N \geq 1$ et $n \geq 1$ où n `= len(chemin)`.

On parcourt les cases du chemin et on vérifie que le passage de l'une à l'autre est conforme aux sauts autorisés.

Lorsqu'un bonus est activé (i.e. il s'agit d'une case du chemin), on ajoute les sauts activés à ceux de départ.

On suppose dans tout le corrigé, comme le suggère l'énoncé, qu'il s'agit de nouveaux sauts possibles, sans redondance avec les sauts déjà autorisés.

1. Notons qu'avec l'exemple de la liste `sauts` de l'énoncé, à chaque étape du chemin, soit i, soit j augmente de 1. Comme on autorise un retour-arrière sur les colonnes, le chemin de longueur maximal est obtenu en parcourant un "Z" sur la grille. Ainsi, $n \leq 3N$ et la complexité est en $\mathcal{O}(N)$ où N est la taille de la grille. Par contre avec `sauts = [(0,1), (0,-1), (1,0)]`, on peut parcourir en serpentant chaque ligne de la grille et la complexité est en $\mathcal{O}(N^2)$.

```
28 def sauts_corrects(sauts, bonus, chemin):
29     n = len(chemin) # on suppose n >= 1
30     sauts_autorises = sauts[:]
31 
32     for k in range(n-1): # on entre dans la boucle ssi n >= 2
33         i_depart, j_depart = chemin[k]
34         i_arrivee, j_arrivee = chemin[k+1]
35         di, dj = i_arrivee - i_depart, j_arrivee - j_depart
36 
37         if (i_depart, j_depart) in bonus:
38             sauts_autorises += bonus[(i_depart, j_depart)]
39 
40         if (di, dj) not in sauts_autorises:
41             return False
42 
43     return True
```

Q4 La fonction `sauts_bien_formes` vérifie une condition suffisante de non existence de cycle dans l'ensemble des sauts.

```
108 def saut_est_positif(di, dj):
109     return di > 0 or (di == 0 and dj > 0)
110 
111 def sauts_bien_formes(sauts, bonus):
112     if sauts == []:
113         return False
114     for (di, dj) in sauts:
115         if not saut_est_positif(di, dj):
116             return False
117     for case in bonus:
118         for (di, dj) in bonus[case]:
119             if not saut_est_positif(di, dj):
120                 return False
121     return True
```

II Recherche exhaustive

Q5 Parmi tous les ensembles de sauts satisfaisants (*), la longueur maximale L d'un chemin de $(0,0)$ à $(N-1,N-1)$ est atteinte si les seuls sauts autorisés sont $(1,0)$ et $(0,1)$.
En effet, le cumul des longueurs des arêtes selon l'axe des abscisses est alors de $N-1$, ainsi que le cumul des longueurs des arêtes selon l'axe des ordonnées.
On obtient :

$$L = 2(N-1)$$

Par exemple, avec par défaut **`sauts = [(1,0), (0,1)]`**, un chemin « **en zigzag** » convient :

```
chemin = {(0, 0), ..., (i, i-1), (i, i), ..., (N-1, N-2), (N-1, N-1)}
```

où i varie de 1 à $N-2$ inclus.

Q6

```
157 INFINI = 10_000
158
159 def trouve_complet_rec(T, sauts, bonus, sauts_max, i, j):
160     N = len(T)
161
162     # Arrêt quand dépassement d'horizon ou arrivée au coin en bas à
    droite
163     if sauts_max == 0 or (i, j) == (N-1, N-1):
164         return poids(T, [(i,j)]), []
165
166     else:
167         poids_min = INFINI
168         sauts_min = []
169
170         # Activation immédiate des sauts bonus si on est sur une case
    bonus
171         sauts_autorises = sauts[:]
172         if (i, j) in bonus:
173             sauts_autorises += bonus[(i, j)]
174
175         # Détermination du poids minimal et du saut minimal pour tous
    les sauts possibles à partir de (i,j)
176         for (di, dj) in sauts_autorises: # boucle non vide
177             if 0 <= i + di < N and 0 <= j + dj < N:
178                 poids_suivant, saut_suivant = trouve_complet_rec(T,
    sauts_autorises, bonus, sauts_max - 1, i + di, j +  dj)
179
180                 # Évaluation du poids du chemin par rapport au poids
    minimal en cours
181                 if T[i][j] + poids_suivant < poids_min:
182                     poids_min = T[i][j] + poids_suivant
183                     sauts_min = [(di, dj)] + saut_suivant
184
185         return poids_min, sauts_min
186
187 def trouve_complet(T, sauts, bonus, sauts_max):
188     return trouve_complet_rec(T, sauts, bonus, sauts_max, 0, 0)
```

Complexité de la fonction `trouve_complet` :

La complexité de la fonction dépend de `sauts_max`, de N, et de s, le nombre total de sauts autorisés aussi bien dans `sauts` que dans `bonus`.

À chaque appel de `trouve_complet_rec`, la fonction exécute au plus s tours de boucle où elle appelle récursivement `trouve_complet_rec`, les autres instructions étant en $\mathcal{O}(1)$.
Le nombre d'appels successifs permettant de calculer un poids minimal est au plus égal à min(`sauts_max`,L) (i.e. `saut_max` pour de petites valeurs, puis L quand `saut_max` le dépasse).

Ainsi,

- Quand `saut_max` $< L$, la complexité de `trouve_complet` est en $\mathcal{O}(s^{sauts\_max})$, nombre fixe qui dépend du choix des sauts incluant ceux des bonus et de l'horizon.
- Quand $L \leq$ `saut_max`, la complexité de `trouve_complet` est en $\mathcal{O}(s^L)$.
  Or L est maximal quand δi et δj valent 1 et vaut $L = 2N - 2$.
  Alors, la complexité de `trouve_complet` est en $\mathcal{O}(s^{2N-2})$.

Comme l'indique l'énoncé, la recherche exhaustive est obtenue en appelant `trouve_complet` avec `sauts_max` égal à L. La complexité de cette fonction est alors **exponentielle**.

Q7 Cette question suppose que l'on peut tout simplement mémoïser les résultats calculés dans un dictionnaire `memo` avec pour clé un triplet $(i, j, \texttt{sauts\_max})$. Or, ce n'est pas aussi simple, car le résultat dépend des bonus activés tout au long du chemin.

Pour s'en sortir (sans coder conformément à la partie IV), on peut prévoir :

— associer à chaque triplet $(i, j, \texttt{sauts\_max})$ une liste de tuples `(bonus_actives, min_poids, min_sauts)`,

— modifier la signature de la fonction `trouver_rec` en intégrant les bonus activés : `trouve_complet_rec(T, sauts,bonus,bonus_actives, sauts_max, i, j, memo)`,

— s'il existe déjà lors de l'appel de la fonction, renvoyer directement le résultat par parcours de `memo[(i,j,sauts_max)]` en prenant le résultat qui correspond au dictionnaire `bonus_actives`,

— une fois le résultat calculé, le sauvegarder dans la liste associée à `memo[(i,j,sauts_max)]`.

On fournit ci-joint un code possible, ce qui n'est pas demandé par l'énoncé :

```
200 INFINI = 10_000
201
202 def trouve_complet_rec(T, sauts, bonus, bonus_actives, sauts_max, i, j,
    memo):
203     N = len(T)
204
205     if sauts_max == 0 or (i, j) == (N-1, N-1):
206         return (T[i][j], [])
207
208     cle = (i, j, sauts_max)
209     # On parcourt les versions mémorisées pour cette clé
210     if cle in memo:
211         for bonus_mem, poids_mem, sauts_mem in memo[cle]:
212             if bonus_mem == bonus_actives:
213                 return poids_mem, sauts_mem
214
215     # Sauts disponibles avec les bonus déjà activés
216     sauts_possibles = sauts[:]
217     for (di, dj) in bonus_actives:
218         if (di, dj) in bonus:
219             sauts_possibles += bonus[(di, dj)]
220
221     # Activation immédiate des sauts bonus si on est sur une case bonus
222     nouveaux_bonus_actives = {case:valeur for (case, valeur) in
    bonus_actives.items()}
223     if (i, j) in bonus:
224         sauts_possibles += bonus[(i, j)]
225         nouveaux_bonus_actives[(i, j)] = True
226
227     min_poids = INFINI
228     min_sauts = []
229
230     # Détermination du poids minimal et du saut minimal pour tous les
    sauts possibles à partir de (i,j)
231     for (di, dj) in sauts_possibles:
232         if 0 <= i + di < N and 0 <= j + dj < N:
233             poids_suivant, sauts_suivant = trouve_complet_rec(T, sauts,
    bonus, nouveaux_bonus_actives, sauts_max - 1, i + di, j + dj, memo)
234
235             if T[i][j] + poids_suivant < min_poids:
236                 min_poids = T[i][j] + poids_suivant
237                 min_sauts = [(di, dj)] + sauts_suivant
238
```

```
    # Mémoïsation avec le bon état de bonus utilisés
    if cle not in memo:
        memo[cle] = []
    memo[cle].append((bonus_actives, min_poids, min_sauts))

    return min_poids, min_sauts

def trouve_complet(T, sauts, bonus, sauts_max):
    memo, bonus_actives = {}, {}
    return trouve_complet_rec(T, sauts, bonus, bonus_actives, sauts_max,
    0, 0, memo)
```

Complexité de la fonction `trouve_complet` avec mémoïsation :

La fonction alimente pour chacune de N^2 valeurs (i, j), pour au plus chacune des valeurs de $[\![0, \texttt{sauts\_max}]\!]$, pour chacun des 2^b sous-ensembles de bonus activés, un résultat.

Le nombre d'appels récursifs **distincts** (pour déterminer chacun des résultats) de la fonction `trouve_complet_rec` est donc en $\mathcal{O}(2^b \times \texttt{sauts\_max} \times N^2)$.

Pour chacun de ces appels distincts, la fonction `trouve_complet_rec` parcourt l'un des s sauts possibles de `sauts_possibles` et se déroule avec une complexité en $\mathcal{O}(s)$.

Finalement, la complexité de la fonction `trouve_complet` est en $\mathcal{O}(2^b \times s \times \texttt{sauts\_max} \times N^2)$.

III Recherche gloutonne

Q8 On prend l'exemple de l'introduction.

Chemin pour l'horizon $k = 2$:

— Étape 1 :
On part de $(0, 0)$. Il y a a priori 6 chemins dont les poids apparaissent dans le tableau suivant :

Chemin	Poids
$[(0,0),(0,1),(0,2)]$	-8
$[(0,0),(0,1),(1,0)]$	-1
$[(0,0),(0,1),(1,2)]$	0
$[(0,0),(1,1),(1,2)]$	2
$[(0,0),(1,1),(2,0)]$	-2
$[(0,0),(1,1),(2,2)]$	-3

La meilleure suite locale de 2 sauts à partir de $(0, 0)$ est donc $[(0,0),(0,1),(0,2)]$, de poids -8.

— Étape 2 :
On part de $(0, 2)$. Il y a a priori 5 chemins dont les poids apparaissent dans le tableau suivant :

Chemin	Poids
$[(0,2),(0,3),(1,2)]$	2
$[(0,2),(1,1),(1,2)]$	4
$[(0,2),(1,1),(2,0)]$	-4
$[(0,2),(1,1),(2,2)]$	-5
$[(0,2),(1,3),(2,2)]$	0

La meilleure suite locale de 2 sauts à partir de $(0, 2)$ est donc $[(0,2),(1,1),(2,2)]$, de poids -5.

— Étape 3 :
On part de $(2,2)$.
Il y a a priori 3 chemins dont les poids apparaissent dans le tableau suivant :

Chemin	Poids
$[(2,2),(2,3),(3,2)]$	1
$[(2,2),(3,1),(3,2)]$	1
$[(2,2),(3,3)]$	7

Les meilleures suites locales de 2 sauts à partir de $(2,2)$ sont donc $[(2,2),(2,3),(3,2)]$ et $[(2,2),(3,1),(3,2)]$, de poids 1.

— Étape 4 :
On part de $(3,2)$.
Il y a a priori 1 seul chemin d'au plus 2 étapes, à savoir $[(3,2),(3,3)]$, de poids 7.

Finalement pour $k=2$, on obtient deux chemins possibles de même poids $-8-5+1+7=-5$, à savoir $[(0,0),(0,1),(0,2),(1,1),(2,2),(2,3),(3,2),(3,3)]$ ou $[(0,0),(0,1),(0,2),(1,1),(2,2),(3,1),(3,2),(3,3)]$.
Aucun des 2 chemins trouvés n'est optimal.

Chemin pour l'horizon $k=3$:

— Étape 1 :
On part de $(0,0)$.
Il y a a priori 18 chemins dont les poids apparaissent dans le tableau suivant :

Chemin	Poids
$[(0,0),(0,1),(0,2),(0,3)]$	-8
$[(0,0),(0,1),(0,2),(1,1)]$	-10
$[(0,0),(0,1),(0,2),(1,3)]$	-5
$[(0,0),(0,1),(1,0),(1,1)]$	-3
$[(0,0),(0,1),(1,0),(2,1)]$	1
$[(0,0),(0,1),(1,2),(1,3)]$	3
$[(0,0),(0,1),(1,2),(2,1)]$	2
$[(0,0),(0,1),(1,2),(2,3)]$	4
$[(0,0),(0,1),(1,2),(2,2)]$	-3
$[(0,0),(1,1),(1,2),(1,3)]$	5
$[(0,0),(1,1),(1,2),(2,1)]$	4
$[(0,0),(1,1),(1,2),(2,3)]$	6
$[(0,0),(1,1),(1,2),(2,2)]$	-1
$[(0,0),(1,1),(2,0),(2,1)]$	0
$[(0,0),(1,1),(2,0),(3,1)]$	2
$[(0,0),(1,1),(2,2),(2,3)]$	1
$[(0,0),(1,1),(2,2),(3,1)]$	1
$[(0,0),(1,1),(2,2),(3,3)]$	4

La meilleure suite locale de 3 sauts à partir de $(0,0)$ est donc $[(0,0),(0,1),(0,2),(1,1)]$, de poids -10.

— Étape 2 :
On part de $(1,1)$.
Il y a a priori 16 chemins dont les poids apparaissent dans le tableau suivant :

Chemin	Poids
$[(1,1),(1,2),(1,3),(2,2)]$	2
$[(1,1),(1,2),(2,1),(2,2)]$	1
$[(1,1),(1,2),(2,1),(3,0)]$	3
$[(1,1),(1,2),(2,1),(3,2)]$	2
$[(1,1),(1,2),(2,2),(2,3)]$	3
$[(1,1),(1,2),(2,2),(3,1)]$	3
$[(1,1),(1,2),(2,2),(3,3)]$	6
$[(1,1),(1,2),(2,2),(3,2)]$	-4
$[(1,1),(1,2),(2,3),(3,2)]$	3
$[(1,1),(2,0),(2,1),(2,2)]$	-3
$[(1,1),(2,0),(2,1),(3,0)]$	-1
$[(1,1),(2,0),(2,1),(3,2)]$	-3
$[(1,1),(2,0),(3,1),(3,2)]$	-1
$[(1,1),(2,2),(2,3),(3,2)]$	-2
$[(1,1),(2,2),(3,1),(3,2)]$	-2
$[(1,1),(2,2),(3,3)]$	4

La meilleure suite locale de 3 sauts à partir de $(1,1)$ est donc $[(1,1),(1,2),(2,2),(3,2)]$, de poids -4.

— Étape 3 :
On part de $(3,2)$.
Il y a a priori 1 seul chemin d'au plus 3 étapes, à savoir $[(3,2),(3,3)]$, de poids 7.

Finalement pour $k=3$, on obtient le chemin optimal de poids $-10-4+7=-7$, à savoir $[(0,0),(0,1),(0,2),(1,1),(1,2),(2,2),(3,2),(3,3)]$.

Chemin lorsque l'on augmente l'horizon k :
A priori, en augmentant k dans un algorithme glouton, le poids du chemin diminue, mais pas forcément strictement (typiquement si le minimum de poids est atteint, comme ici avec $k=3$).

Q9 Comme indiqué dans l'énoncé en page 2, on réutilise la fonction de la partie II.

```
INFINI = 10_000

def trouve_glouton(T, sauts, bonus, k):
    N = len(T)

    i, j = 0, 0
    poids_total = T[i][j]
    sauts_total = []

    while (i, j) != (N-1, N-1):
        poids_min, sauts_min = trouve_complet_rec(T, sauts, bonus, k, i, j)

        if poids_min == INFINI or sauts_min == []:
            return INFINI, []
        else:
            poids_total += poids_min - T[i][j]
            i, j = appliquer_sauts(i, j, sauts_min)
            sauts_total += sauts_min

    return poids_total, sauts_total
```

IV Recherche par programmation dynamique

Q10 La fonction proposée est **récursive**.
Le code exploite le fait que **les booléens sont un sous-type des entiers**.

```
368 def code_bonus(masque_bonus):
369     if len(masque_bonus) == 1:
370         return int(masque_bonus[0])
371     else:
372         return int(masque_bonus[0] + 2 * code_bonus(masque_bonus[1:]))
```

Q11 Soit Γ l'ensemble des sauts par défaut et Δ_k l'ensemble des sauts associé à la k^e case bonus.

Si (i,j) n'est **pas** une case bonus. Alors,

$$\texttt{poids\_opt[i][j}[\langle b_0 \dots b_{n-1}\rangle\texttt{]} = \texttt{T[i][j]} + \min_{(di,dj)\in S} (\texttt{poids\_opt[i+di][j+dj}[\langle b_0 \dots b_{n-1}\rangle\texttt{]})$$

avec $S = \Gamma \cup \bigcup_{r\in[\![0,n-1]\!],\ b_r=True} \Delta_r$

Si (i,j) est une case bonus numérotée k. Alors,

$$\texttt{poids\_opt[i][j}[\langle b_0 \dots b_{n-1}\rangle\texttt{]} = \texttt{T[i][j]} + \min_{(di,dj)\in S'} (\texttt{poids\_opt[i+di][j+dj}[\langle b_0 \dots b_k = 1 \dots b_{n-1}\rangle\texttt{]})$$

avec $S' = \Gamma \cup \bigcup_{r\in[\![0,n-1]\!],\ b_r=True} \Delta_r \cup \Delta_k$

Q12 La fonction proposée utilise une file implémentée par une "double-ended queue" (i.e deque) afin d'optimiser l'insertion « à gauche ».
On définit dans un premier temps les primitives de cette file.

```
396 from collections import deque
397
398 def creer_file():
399     return deque()
400
401 def enfiler(e, f):
402     f.appendleft(e)
403
404 def est_vide(f):
405     return len(f) == 0
406
407 def defiler(f):
408     if not est_vide(f):
409         return f.pop()
```

La sous-fonction `creer_masques` crée une ligne de nouveaux choix à partir d'un masque donné. Ces nouveaux choix s'obtiennent en faisant basculer une coordonnée `True` à `False`.
La fonction `combinaison_bonus` suit l'algorithme de l'énoncé. Elle vérifie au passage qu'un masque est inséré dans la file uniquement s'il n'y existe pas déjà.

```
412 def creer_masques(nb_bonus, masque):
413     '''création de la liste des masques inclus dans masque'''
414     liste_masques = []
415     for i in range(nb_bonus):
416         copie_masque = masque[:]
417         if masque[i]: # masque[i] vaut True
418             copie_masque[i] = False
419             liste_masques.append(copie_masque)
420     return liste_masques
```

```
421
422 def combinaisons_bonus(nb_bonus):
423     liste_masques_ordonnes = []
424     f = creer_file()
425     masque_choix_total = [True]*nb_bonus
426     enfiler(masque_choix_total, f)
427     while not est_vide(f):
428         masque = defiler(f)
429         liste_masques_ordonnes.append(masque)
430         liste_masques = creer_masques(nb_bonus, masque)
431         for masque in liste_masques:
432             if masque not in f:
433                 enfiler(masque, f)
434     return liste_masques_ordonnes
```

Q13 La fonction `trouver_sauts_possibles` sélectionne les indices de `masque_bonus` égaux à `True`. Pour chacun de ces indices, la fonction récupère la liste des sauts activés de `bonus_au_rang`. Les listes des sauts activés sont concaténées avec la liste des sauts par défaut.

```
456 def trouver_sauts_possibles(sauts, bonus_au_rang, masque_bonus):
457     nb_bonus = len(masque_bonus)
458     sauts_possibles = sauts[:]
459     for k in range(nb_bonus):
460         if masque_bonus[k]:
461             sauts_possibles += bonus_au_rang[k]
462     return sauts_possibles
```

Q14

```
516 def trouve_dynamique(T, sauts, bonus):
517     N = len(T)
518     nb_bonus = len(bonus)
519
520     nb_code_bonus = 1 + code_bonus([True]*nb_bonus) # complété (1)
521     poids_opt = [[[INFINI for bonus_code in range(nb_code_bonus)] for j
    in range(N)] for i in range(N)]
522     saut_opt = [[[(0, 0, 0) for bonus_code in range(nb_code_bonus)] for
    j in range(N)] for i in range(N)]
523     bonus_au_rang, rang_du_bonus = ranger_bonus(bonus)
524
525     for bonus_actifs in combinaisons_bonus(nb_bonus):
526         code_bonus_actifs = code_bonus(bonus_actifs)
527         poids_opt[N-1][N-1][code_bonus_actifs] = 0 # complété (2)
528         sauts_possibles = trouver_sauts_possibles(sauts, bonus_au_rang,
    bonus_actifs) # complété (3)
529         for i in range(N-1, -1, -1): # complété (4)
530             for j in range(N-1, -1, -1): # complété (5)
531                 code_bonus_dest = ajouter_bonus(bonus, rang_du_bonus, i,
     j, bonus_actifs, code_bonus_actifs)
532                 if (i, j) in bonus:
533                     sauts_possibles_final = sauts_possibles + bonus[(i,
    j)]
534                 else:
535                     sauts_possibles_final = sauts_possibles
536                 for (delta_i, delta_j) in sauts_possibles_final:
537                     i_dest = i + delta_i
538                     j_dest = j + delta_j
539                     if (i_dest in range(N) and j_dest in range(N)):
540                         poids_opt_dest = poids_opt[i_dest][j_dest][
    code_bonus_dest]
```

```
541                         if poids_opt[i][j][code_bonus_actifs] >
    poids_opt_dest:
542                             poids_opt[i][j][code_bonus_actifs] =
    poids_opt_dest # complété (6)
543                             saut_opt[i][j][code_bonus_actifs] =  (
    delta_i, delta_j, code_bonus_dest) # complété (7)
544                 poids_opt[i][j][code_bonus_actifs] += T[i][j]
545
546     return poids_opt, saut_opt
```

Complexité de la fonction `trouve_dynamique` :

La complexité de la fonction dépend de N, de s (le nombre total de sauts autorisés aussi bien dans `sauts` que dans `bonus`), et de `nb_code_bonus`. Or, si b est le nombre de cases de bonus, alors le nombre de code bonus est égal au nombre de parties d'un ensemble à b éléments, càd. 2^b. Ainsi, `nb_code_bonus` vaut 2^b.

— Prenons chaque instruction à l'extérieur de la boucle `for` :

- `N = len(T); nb_bonus = len(bonus)` et `return poids_opt, saut_opt` s'exécutent en $\mathcal{O}(1)$,
- `nb_code_bonus = 1 + code_bonus ([True]* nb_bonus)` se déroule en $\mathcal{O}(b)$,
- les créations de `poids_opt` et `saut_opt` se déroulent respectivement en $\mathcal{O}(\texttt{nb\_code\_bonus} \times N^2) = \mathcal{O}(2^b \times N^2)$,
- `bonus_au_rang, rang_du_bonus = ranger_bonus(bonus)` est en $\mathcal{O}(b)$, par simple parcours du dictionnaire `bonus`.

Par somme, les instructions à l'extérieur de la boucle se déroulent en $\mathcal{O}(2^b \times N^2)$.

— Le nombre de tours de la boucle
`for bonus_actifs in combinaisons_bonus(nb_bonus)` est égal à 2^b, nombre de parties à b éléments.
Prenons chaque instruction dans le corps de la boucle `for` :

- `code_bonus_actifs = code_bonus(bonus_actifs)` se déroule en $\mathcal{O}(b)$,
- `poids_opt[N-1][N-1][code_bonus_actifs] = 0` s'exécute en $\mathcal{O}(1)$,
- `sauts_possibles = trouver_sauts_possibles(...)` se déroule en $\mathcal{O}(b)$, par simple parcours du dictionnaire `bonus`,
- Ensuite se déroule une boucle imbriquée exécutant exactement N^2 tours de boucle. Prenons chaque instruction dans le corps de la boucle `for` :
  - `code_bonus_dest = ajouter_bonus(...)` s'exécute en $\mathcal{O}(1)$, par simple mise à jour du bonus actif,
  - `if (i, j) in bonus` s'exécute en $\mathcal{O}(1)$, car `bonus` est un dictionnaire. Dans le cas où la condition est vérifiée, elle se déroule en $\mathcal{O}(s)$. Dans le cas contraire, en $\mathcal{O}(1)$ (définition d'un alias).
  - Ensuite se déroule une boucle `for` exécutant au plus s tours de boucle. Les instructions du corps de la boucle `for` sont en $\mathcal{O}(1)$. Cette dernière boucle `for` a donc une complexité en $\mathcal{O}(s)$.

  Par somme, la boucle imbriquée a une complexité en $\mathcal{O}(s \times N^2)$.

Comme $b \leq s$, la complexité de la boucle `for` générale est donc en $\mathcal{O}(2^b \times s \times N^2)$.

Par somme, **la complexité de la fonction `trouve_dynamique` est en** $\mathcal{O}(2^b \times s \times N^2)$.

Q15

```
571 def solution_dynamique(saut_opt, N):
572     i, j = 0, 0
573     chemin = [(i, j)]
574     code_bonus = 0  # Aucun bonus activé au départ
575 
576     while (i, j) != (N-1, N-1):
577         di, dj, nouveau_code_bonus = saut_opt[i][j][code_bonus]
578 
579         # Saut invalide : aucun chemin trouvé
580         if (di, dj) == (0, 0) and (i, j) != (N-1, N-1):
581             return []
582 
583         i += di
584         j += dj
585         code_bonus = nouveau_code_bonus
586         chemin.append((i, j))
587 
588     return chemin
```