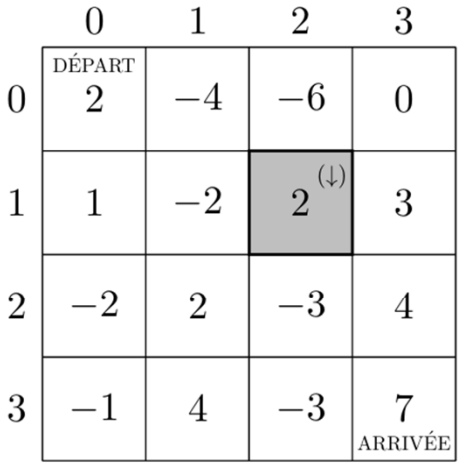
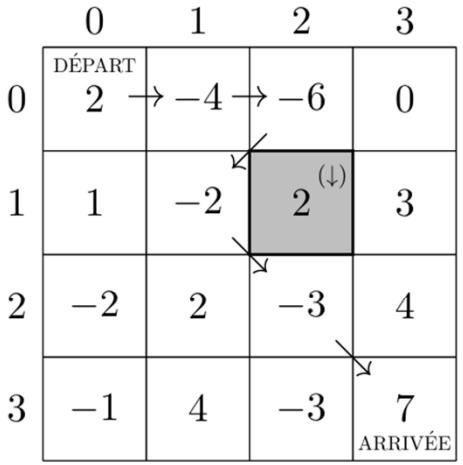
MPII
ITC

Corrigé DS1
Jeu de Röckse

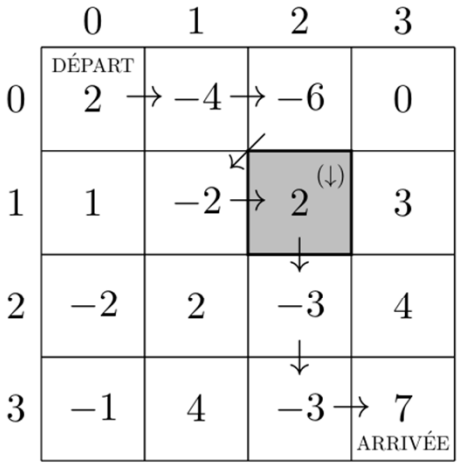
Jeu de Röckse



(a) Grille de jeu



(b) Chemin non-optimal
poids : −6

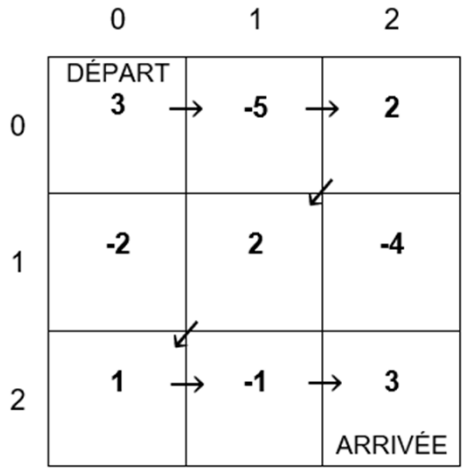


(c) Chemin optimal
poids : −7

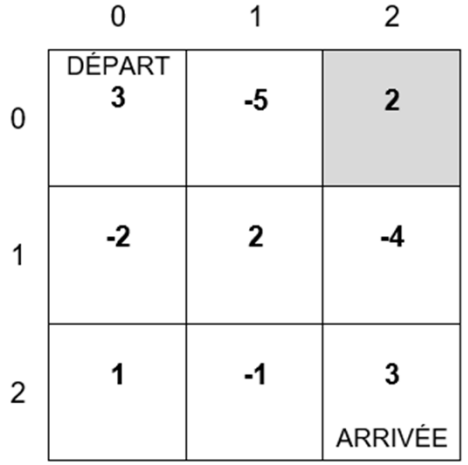
Grille exemple

```
T_ex = [[2,-4,-6,0],[1,-2,2,3],[-2,2,-3,4],[-1,4,-3,7]]
sauts_ex = [(0,1),(1,-1),(1,1)]
bonus_ex = { (1,2): [(1,0)] }
chemin_ex = [(0,0),(0,1),(0,2),(1,1),(1,2),(2,2),(3,2),(3,3)]
```

PARTIE 1. MODELISATION, SAUTS ET CHEMINS, VERIFICATION DE GRILLE



(a) Chemin 1 sur grille de jeu sans case bonus



(b) Grille de jeu avec case bonus

Grille Jeu

Q1- `T_jeu = [[3,-5,2],[-2,2,-4],[1,-1,3]]`

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

→ ↙

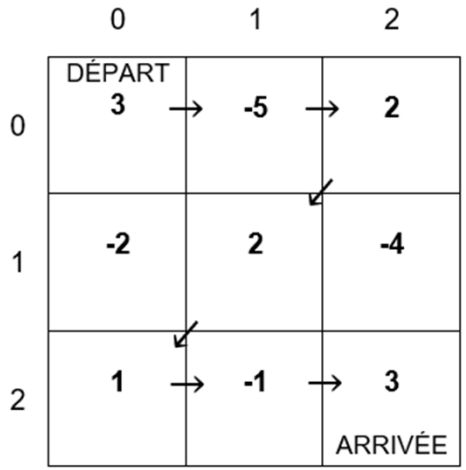
`chemin_jeu = [(0,0), (0,1), (0,2), (1,1), (2,0), (2,1), (2,2)]`

Q2- `bonus_ex = {(0,2): [(1,0)]}` ↓

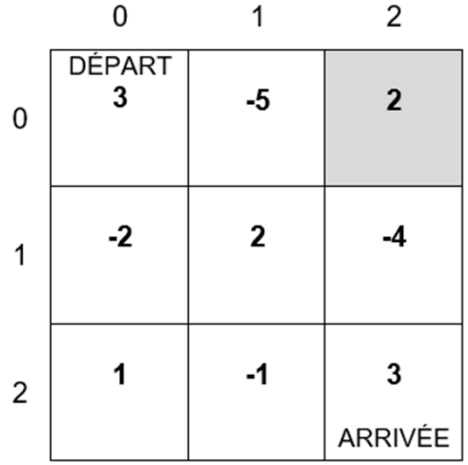
`chemin_jeu_b = [(0,0), (0,1), (0,2), (1,2), (2,2)]`

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

PARTIE 1. MODELISATION, SAUTS ET CHEMINS, VERIFICATION DE GRILLE



(a) Chemin 1 sur grille de jeu sans case bonus



(b) Grille de jeu avec case bonus

Grille Jeu

Q3-
```
chemin_jeu = [(0,0), (0,1), (0,2), (1,1), (2,0), (2,1), (2,2)]
 # poids = 3 – 5 + 2 + 2 + 1 – 1 + 3 = 5
chemin_jeu_b = [(0,0), (0,1), (0,2), (1,2), (2,2)]
 # poids = 3 – 5 + 2 – 4 + 3 = -1
chemin_jeu_opt = [(0,0), (0,1), (1,0), (1,1), (1,2), (2,1), (2,2)]
 # poids = 3 – 5 - 2 + 2 – 4 -1 + 3 = -4
chemin_jeu__b_opt = [(0,0), (0,1), (0,2), (1,2), (2,1), (2,2)]
 # poids_b_opt = 3 – 5 + 2 – 4 -1 + 3 = -2
```

PARTIE 1. MODELISATION, SAUTS ET CHEMINS, VERIFICATION DE GRILLE

- **Sauts et chemin**

Q4-

```
def poids(T, chemin):
    pds = 0      # initialisation
    for case in chemin:
        pds += T[case[0]][case[1]]
    return pds
```

Complexité $\mathcal{C} = \mathcal{O}(len(chemin))$

```
def poids(T, chemin):
    pds = 0     # initialisation
    for x,y in chemin:
        pds += T[x][y]
    return pds
```

PARTIE 1. MODELISATION, SAUTS ET CHEMINS, VERIFICATION DE GRILLE

- **Sauts et chemin**

Q5-

Complexité $\mathcal{C} = \mathcal{O}(len(ordre\_sauts))$

```
def appliquer_sauts( i, j, ordre_sauts ):
    for k in range(len(ordre_sauts)): # boucle sur indices
        i += ordre_sauts[k][0] # nouvelle position de i
        j += ordre_sauts[k][1] # nouvelle position de j
    return (i,j)

def appliquer_sauts(i,j,ordre_sauts):
    new_i, new_j = i, j # initialisation de la position finale
    for saut in ordre_sauts: # boucle sur éléments
        dx, dy = saut[0], saut[1] # décalage imposer par le saut étudié
        new_i += dx # nouvelle position de i
        new_j += dy # nouvelle position de j
    return (new_i, new_j)
```

PARTIE 1. MODELISATION, SAUTS ET CHEMINS, VERIFICATION DE GRILLE

- **Sauts et chemin**

Q6-

```
def sauts_corrects (sauts, chemin):
    nc = len(chemin)
    for k in range(nc-1):                          # boucle les éléments du chemin (positions)
        ecart_x, ecart_y = chemin[k+1][0] - chemin[k][0] , chemin[k+1][1] - chemin[k][1]
                                                    # calcul du vecteur déplacement
        for saut in sauts:                          # test sur chaque saut pour vérifier si le déplacement est possible
            if (ecart_x, ecart_y) not in sauts:  # test sur déplacement
                return False
    return True
```

nc = len(chemin), ns = len(sauts)

dans le pire des cas : $\mathcal{C}(nc, ns) = \mathcal{C}0 + (nc - 1)(\mathcal{C}1 + ns.\mathcal{C}test) = \boldsymbol{\mathcal{O}(nc \times ns)}$

PARTIE 1. MODELISATION, SAUTS ET CHEMINS, VERIFICATION DE GRILLE

- **Sauts et chemin**

Q7-

```
def sauts_corrects_b( sauts, bonus, chemin ):
    nc = len(chemin)
    for k in range(nc-1):
        if chemin[k] in bonus.keys():          # Prise en compte des cases bonus quand on y passe
            sauts += bonus[chemin[k]]
        ecart_x, ecart_y= chemin[k+1][0]-chemin[k][0] , chemin[k+1][1]-chemin[k][1]
                                               # calcul du vecteur déplacement
        for saut in sauts:                     # plus de sauts car cases bonus…
            if (ecart_x, ecart_y) not in sauts:
                return False
    return True
```

nc = len(chemin), ns = len(sauts), nb = len(bonus) = nbre de sauts en plus maximum
dans le pire des cas : $\mathcal{C}(nc, ns, nb) = \mathcal{C}0 + (nc-1)(\mathcal{C}bonus + \mathcal{C}1 + (ns+nb).\mathcal{C}test) = \boldsymbol{\mathcal{O}(nc \times (ns+nb))}$

PARTIE 1. MODELISATION, SAUTS ET CHEMINS, VERIFICATION DE GRILLE

- **Sauts et chemin**

Q8- Condition saut bien formé : $\delta i > 0$ **ou bien** $\delta i = 0$ **et** $\delta j > 0$ **(*)**

```
def sauts_bien_formes(sauts, bonus) :
    ensemble_sauts = sauts # initialisation
    for position in bonus.keys():                    # ajout des sauts bonus dans liste des sauts
        ensemble_sauts += bonus[position]
    for dx, dy in ensemble_sauts:                    # boucle sur tous les sauts
            if not (dx > 0 or (dx == 0 and dy >0)):  # Test validité
                return False
    return True
```

PARTIE 2. PREMIERES RECHERCHES

- **Recherche naïve**

Q9- Démarche Gloutonne

1. Au vu de la grille et du nombre de cases, il y a de **nombreuses possibilités** de chemin menant du départ à l'arrivée.

2. La démarche initialisée au départ consistant à choisir la case la moins pénalisante: **choix localement optimal** reste identique durant l'ensemble de la démarche.

3. On peut **quantifier la qualité de la solution** en calculant son poids, **le poids doit être minimisé**.

PARTIE 2. PREMIERES RECHERCHES

- **Recherche naïve**

Q10-

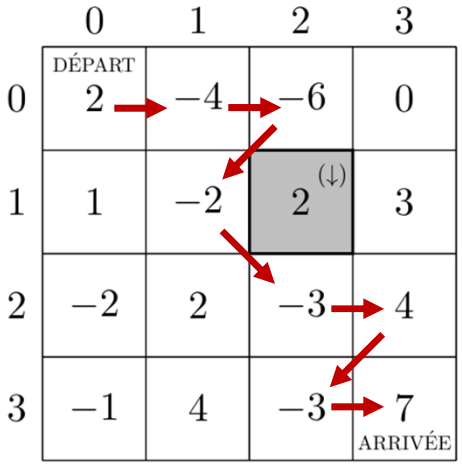
```
1  def recherche_glout(T,sauts):
2      pds_chemin = T[0][0]                                          # INIT poids du chemin
3      sauts_chemin = []                                             # INIT liste des sauts
4      pos_x, pos_y = 0, 0                                  à compléter  # position de départ
5      N = len(T)
6      while not (pos_x == N-1 and pos_y == N-1):                    # condition d'arrivée
7          pds_min = float(' inf ')                         à compléter  # INIT poids min
8          for saut in sauts:                                        # pour chaque saut possible on cherche la
9                                                                    position à poids min
10             if (pos_x + saut[0] < N) and (0 <= pos_y + saut[1] < N):  # à compléter
11                                                                   # test pour rester dans la grille de jeu
12                 if T[pos_x + saut[0]][pos_y + saut[1]] < pds_min:  # test sur le poids de la position courante
13
14                                                       à compléter  # MAJ nouvelle valeur de pds_min, nouvelle
15                     pds_min = T [pos_x + saut[0]] [pos_y + saut[1]]  position, le saut testé est retenu comme
16                     new_pos_x = pos_x + saut[0]                   saut_min
17                     new_pos_y = pos_y + saut[1]
18                     saut_min = saut
19         pos_x = new_pos_x                                         # nouvelle position choisie
20         pos_y = new_pos_y
21         pds_chemin = pds_chemin + pds_min                         # MAJ du poids du chemin
22         sauts_chemin  = sauts_chemin + [saut_min]                 # MAJ de la liste des sauts
23     return (pds_chemin,sauts_chemin)
```

Dans le pire des cas, la boucle **while** a lieu N fois, et la boucle **for** ns = len(sauts) fois, soit $\mathcal{C}(N, ns) = \boldsymbol{\mathcal{O}(N \times ns)}$

PARTIE 2. PREMIERES RECHERCHES

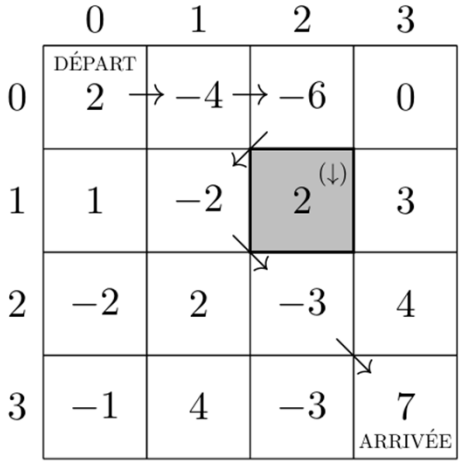
- **Recherche naïve**

Q11- recherche_glout(T, sauts) renvoie (poids_chemin, saut_chemin)




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

→ ↙ ↘



(b) Chemin
poids : −6

recherche_glout(T_ex, sauts_ex) renvoie (-5, [(0,1), (0,1), (1,-1), (1,1), (0,1), (1,-1), (0,1)])

Le chemin n'offre pas un résultat optimal

PARTIE 2. PREMIERES RECHERCHES

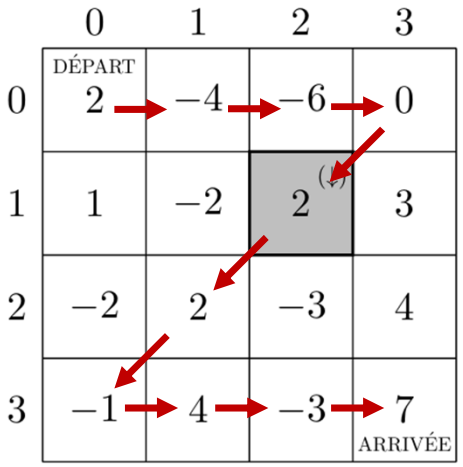
- **Recherche exhaustive**

Q12-

Longueur max obtenue en s'éloignant au maximum de l'arrivée à chaque fois pour allonger le chemin:

1. Parcourir toute la première ligne (N-1 sauts de type (0,1))
2. Parcourir la diagonale emmenant en bas à gauche (N-1 sauts de type (1, -1))
3. Parcourir toute la dernière ligne (N-1 sauts de type (0,1))

soit **L = 3.(N-1)**

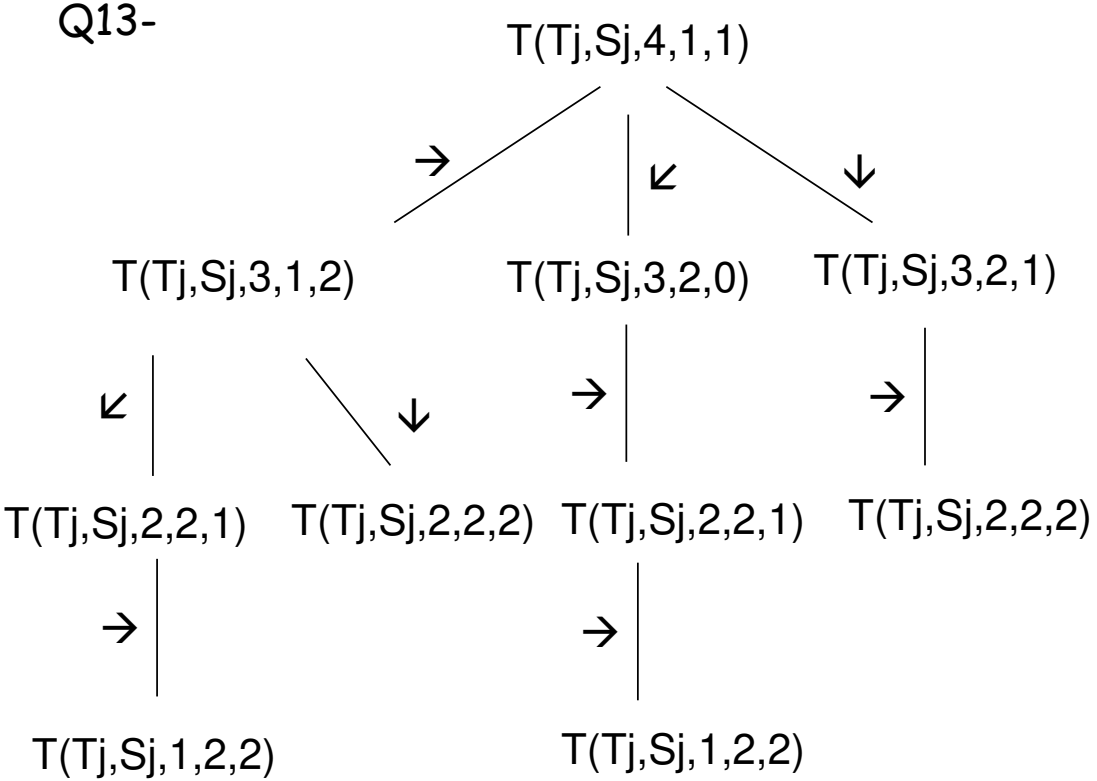


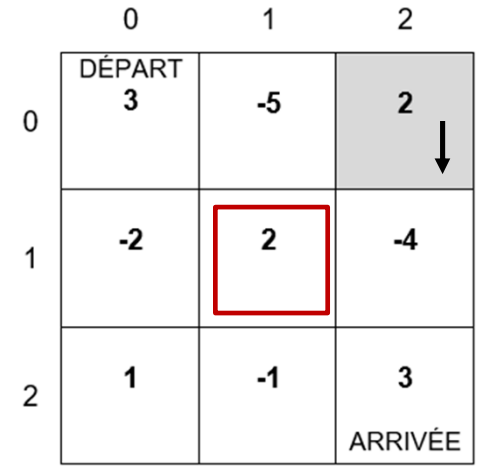
Ensemble de sauts pour N=4 :
[(0,1), (0,1), (0,1), (1, -1), (1, -1), (1, -1), (0,1), (0,1), (0,1)]

PARTIE 2. PREMIERES RECHERCHES

- **Recherche exhaustive**

Q13-





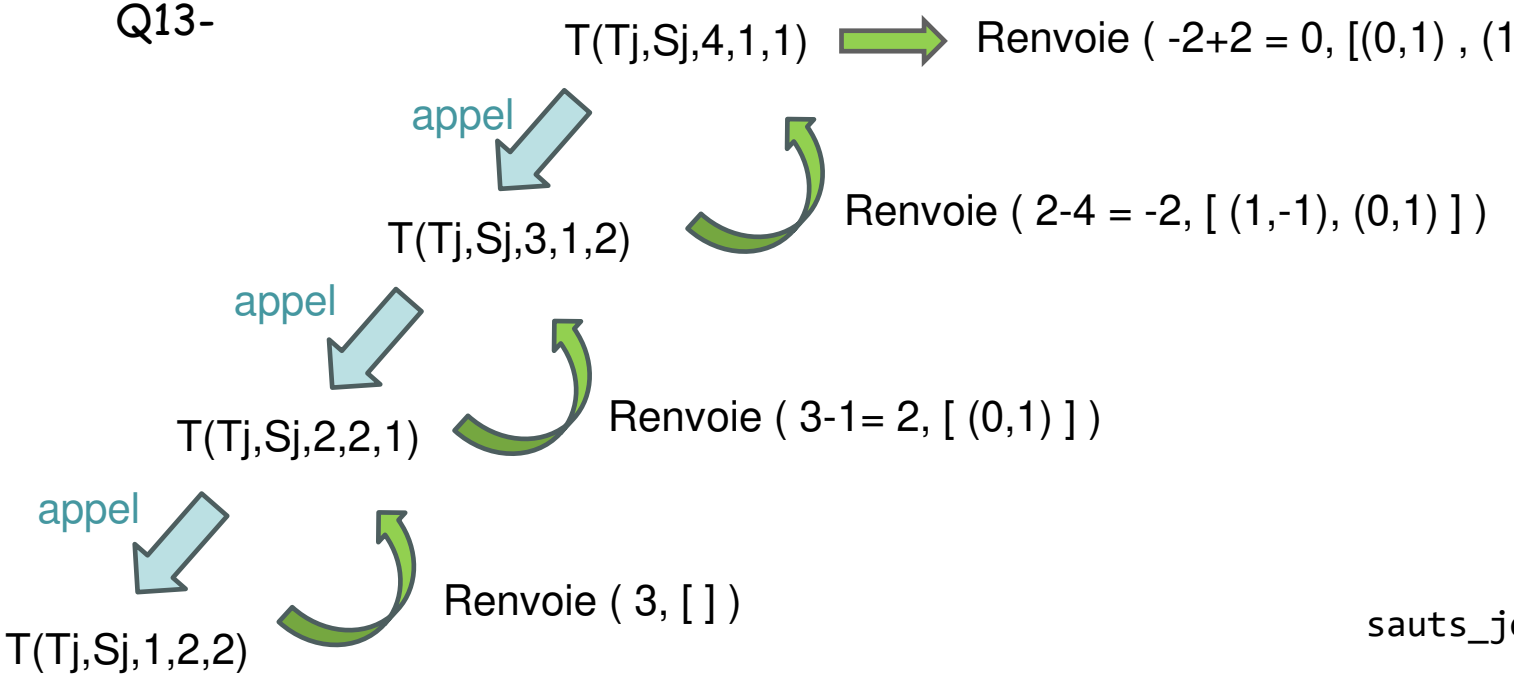
(b) Grille de jeu avec case bonus

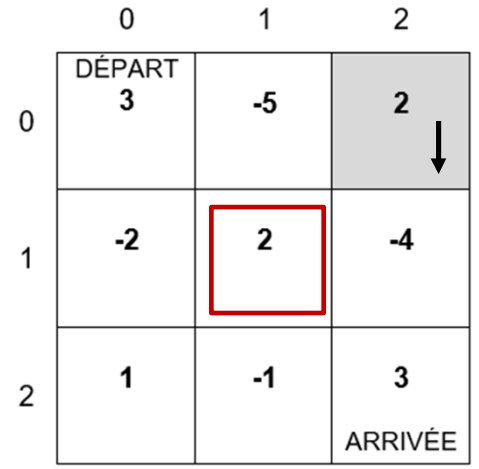
```
sauts_jeu_b = [(0,1),(1,-1),(1,0)]
                  →     ↙     ↓
```

PARTIE 2. PREMIERES RECHERCHES

- **Recherche exhaustive**

Q13-





(b) Grille de jeu avec case bonus

```
sauts_jeu_b = [(0,1),(1,-1),(1,0)]
                 →      ↙      ↓
```

PARTIE 2. PREMIERES RECHERCHES

- **Recherche exhaustive**

Q14-
```
def trouve_complet_rec( T, sauts, nb_sauts_max, i, j ):
    n = len(T)
    if nb_sauts_max == 0 or (i == n-1 and j == n-1):   # condition d'arrêt nb_sauts=0 ou case ARRIVEE atteinte
        return (T[ i ][ j ], [ ])
    pds_min = float('inf')                              # initialisation poids min
    sauts_min = [ ]                                     # initialisation d'une liste de sauts
    for saut in sauts:                                  # Test sur chaque saut
        new_pos_x, new_pos_y = i + saut[0], j + saut[1]
        if new_pos_x < n and 0 <= new_pos_y < n :       # vérification que l'on reste dans la grille
            new_pds, new_sauts = trouve_complet_rec(T, sauts, nb_sauts_max-1, new_pos_x, new_pos_y)
            if new_pds + T[ i ][ j ] < pds_min:         # test pour savoir si meilleure solution
                pds_min = new_pds +T[ i ][ j ]
                sauts_min = [saut] + new_sauts
    if pds_min == float('inf'):                         # cas où l'on sort de la grille
       return (T[ i ][ j ],[])
    return (pds_min, sauts_min)
```

n = len(T), m = nb_sauts_max $\mathcal{C}(m,n) = \mathcal{C}arret + m.(\mathcal{C}newPos + \mathcal{C}testRec) + \mathcal{C}testExt$

$\mathcal{C}(m,n) = \mathcal{C}arret + m.(\mathcal{C}newPos + \mathcal{C}t1 + \mathcal{C}(m,n-1) + \mathcal{C}t2 + \mathcal{C}newPos) + \mathcal{C}testExt$

$\mathcal{C}(m,n) = \mathcal{O}(1) + m \times \big(\mathcal{O}(1) + \mathcal{C}(m,n-1)\big) \Rightarrow \mathcal{C}(m,n) = m \times \mathcal{C}(m,n-1)$

suite géométrique de valeur initiale $\mathcal{C}(m,0) = \mathcal{O}(1)$ (condition d'arrêt) d'où $\boldsymbol{\mathcal{C}(m,n) = \mathcal{O}(m^n)}$ très mauvaise complexité

PARTIE 2. PREMIERES RECHERCHES

- **Recherche exhaustive**

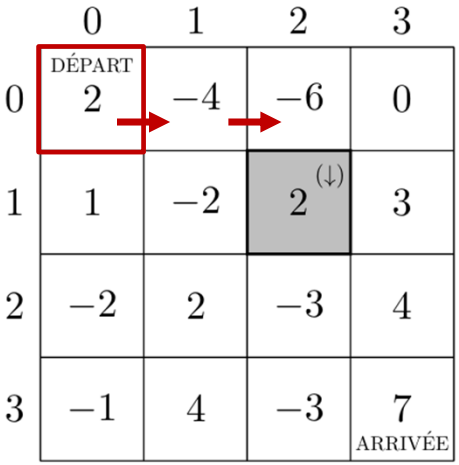
Q15-

```
def trouve_complet(T, sauts, nb_sauts_max):
    return trouve_complet_rec(T, sauts, nb_sauts_max, 0, 0)
```

n = len(T), m = nb_sauts_max
même complexité $\mathcal{C}(\boldsymbol{m}, \boldsymbol{n}) = \mathcal{O}(\boldsymbol{m^n})$

PARTIE 3. RECHERCHE AMELIOREE

Q16-



(a) Grille de jeu

Etape 1: 2 → -4 → −6 pds =-8

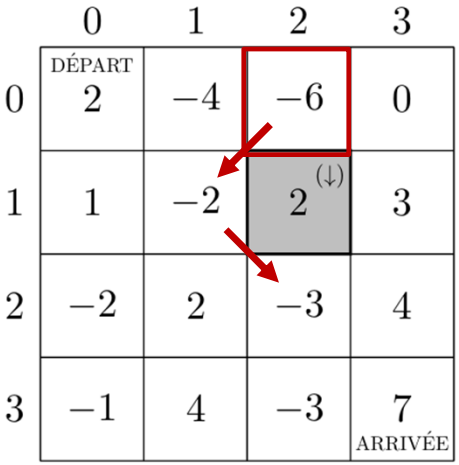
```
saut_min = [(0,1),(0,1)]
```

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

→ ↙ ↘

PARTIE 3. RECHERCHE AMELIOREE

Q16-



(a) Grille de jeu

Etape 1: 2 → -4 → -6 pds = -8

```
saut_min = [(0,1),(0,1)]
```

Etape 2: (-6) → -2 → -3 pds += -5

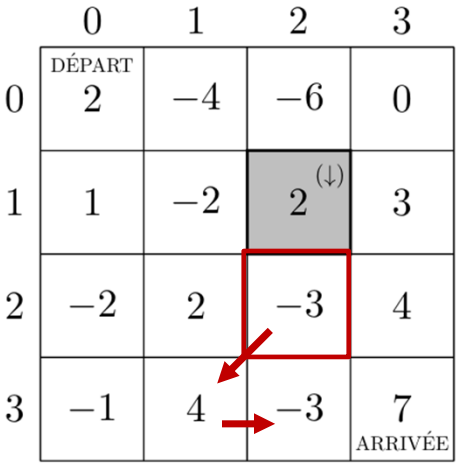
```
saut_min += [(1,1),(1,-1)]
```

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

→ ↙ ↘

PARTIE 3. RECHERCHE AMELIOREE

Q16-



(a) Grille de jeu

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

→ ↙ ↘

Etape 1: 2 → -4 → -6 pds = -8

```
saut_min = [(0,1),(0,1)]
```

Etape 2: (-6) → -2 → -3 pds += -5

```
saut_min += [(1,1),(1,-1)]
```

Etape 3: (-3) → 4 → -3 pds += 1

```
saut_min += [(1,-1),(0,1)]
```

PARTIE 3. RECHERCHE AMELIOREE

Q16-

	0	1	2	3
0	DÉPART 2	−4	−6	0
1	1	−2	2 (↓)	3
2	−2	2	−3	4
3	−1	4	−3 →	7 ARRIVÉE

(a) Grille de jeu

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

→ ↙ ↘

Etape 1: 2 → -4 → -6 pds = -8

```
saut_min = [(0,1),(0,1)]
```

Etape 2: (-6) → -2 → -3 pds += -5

```
saut_min += [(1,1),(1,-1)]
```

Etape 3: (-3) → 4 → -3 pds += 1

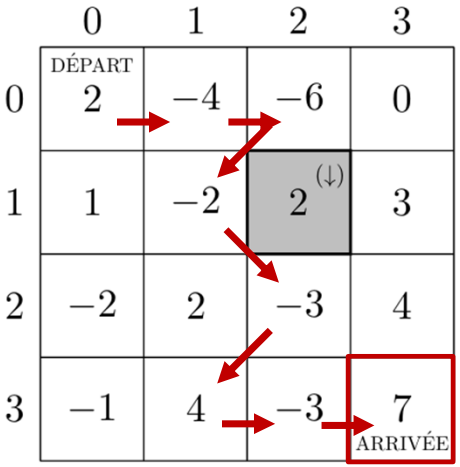
```
saut_min += [(1,-1),(0,1)]
```

Etape 4: (-3) → 7 pds += 7

```
saut_min += [(0,1)]
```

PARTIE 3. RECHERCHE AMELIOREE

Q16-



(a) Grille de jeu

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

→ ↙ ↘

Etape 1: 2 → -4 → -6 pds = -8

`saut_min = [(0,1),(0,1)]`

Etape 2: (-6) → -2 → -3 pds += -5

`saut_min += [(1,1),(1,-1)]`

Etape 3: (-3) → 4 → -3 pds += 1

`saut_min += [(1,-1),(0,1)]`

Etape 4: (-3) → 7 pds += 7

`saut_min += [(0,1)]`

Pour k=2

`chemin = [(0,0), (0,1), (0,2), (1,1), (2,2), (3,1), (3,2), (3,3)] poids = -5 non optimal`

Si on augmente k, on obtient la recherche exhaustive donc le chemin optimal de poids le plus petit.

PARTIE 3. RECHERCHE AMELIOREE

Q17-

```
def trouve_glouton(T, sauts, k):
    pos_x, pos_y = 0, 0
    n = len(T)
    pds_opt, sauts_opt = T[pos_x][pos_y], [ ]
    while (pos_x,pos_y) != (N-1,N-1):              #pos_x != N-1 and pos_y != N-1:
        pds_loc, sauts_loc = trouve_complet_rec(T, sauts, k, pos_x, pos_y)
        if len(sauts_loc) == 0:
            return (float('inf'),[ ])
        pds_opt += pds_loc
        sauts_opt += sauts_loc
        pos_x, pos_y = appliquer_sauts(pos_x, pos_y, sauts_loc)
    pds_opt += T[pos_x][pos_y]
    return (pds_opt, sauts_opt)
```