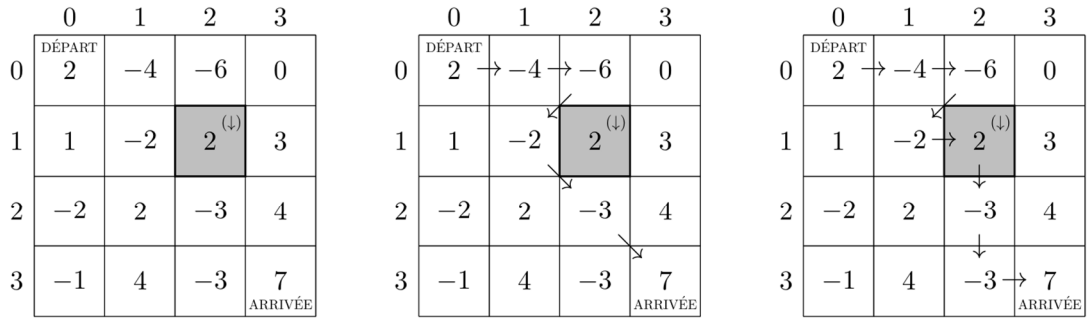
Jeu de Röckse - Tri



(a) Grille de jeu

(b) Chemin non-optimal
poids : -6

(c) Chemin optimal
poids : -7

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

Vérification de validité de grille de jeu

Une grille est déclarée valide suivant sa répartition des *pénalités* (points positifs) et des gains (points négatifs). Une façon d'étudier la répartition de ceux-ci consiste à les lister, les trier puis compter les fréquences des uns et des autres.
La grille est déclarée valide si les cases *pénalités* représentent au maximum les deux tiers des cases comptabilisées.
Le cas du zéro : Il n'est comptabilisé ni dans les gains, ni dans les pénalités.

Question 1. **Écrire** une fonction `trier_points(T)` qui, étant donnée une grille de jeu `T` :

- crée une liste de l'ensemble des *pénalités* et des gains, puis
- les trie dans l'ordre croissant en utilisant le principe du tri par insertion.
  Calculer la complexité de cette fonction dans le pire puis dans le meilleur des cas et conclure.

Question 2. **Écrire** une fonction `repart_correcte(T)` qui, étant donnée une grille de jeu `T` vérifie la validité de la grille. **Donner** la complexité de cette fonction.

Element de correction

Question 1.

```
def trier_points(T):
    # Mise en place de la liste de points
    liste_points = []  # C0=1
    for element in T :
        for point in element :
            liste_points.append(point)  # C1=1
    # mise en place du tri par insertion
    q = len(liste_points)  # C2=1
    for i in range(1,q):
        point = liste_points[i]  # C3=1
        j = i - 1  # C4=2 (soustraction+affectation)
        while j >= 0 and liste_points[j] >= point :
            liste_points[j+1] = liste_points[j]  #décalage
            j = j-1
        liste_points[j+1] = point #insertion C5=2 (j+1 et
affectation)
    return liste_points

# test
T_ex = [[2,-4,-6,0],[1,-2,2,3],[-2,2,-3,4],[-1,4,-3,7]]
Liste_points_ex = trier_points(T_ex)
```

soit N la taille de `liste_points` finale donc le nombre de points dans T :

$$C(N) = (C0 + N * C1) + (C2 + (N-1) * (C3 + C4 + C5) + N(N-1)Cdecalage/2)$$

- dans le pire des cas (liste triée dans le sens inverse) on a :

$$C(N) = (1 + N) + (1 + (N-1) * (1 + 2 + 2) + N(N-1)(2+2)/2)$$
$$= 2 - 5 + N + 5N - 2N + 2N^2$$
$$C(N) = -3 + 4N + 2N^2$$

Accès à l'elément j + affectation, calcul j-1 + affectation

- dans le meilleur des cas :

$$C(N) = (1 + N) + (1 + (N-1) * (1 + 2 + 2) + N(N-1)/2)$$
$$= 2 - 5 + N + 5N - N/2 + N^2/2$$
$$C(N) = -3 + 11N/2 + N^2/2$$

le meilleur des cas est plus rapide…

Question 2.

```
def repart_correcte(T):
    points_tries = trier_points(T)  # liste triée par ordre
croissant
    taille = len(points_tries)
    penalites, gain = 0,0  # initialisation du décompte du nombre de
case
    for point in points_tries:
        if point > 0:
            penalites += 1  # décompte cases pénalités
        elif point < 0 :
            gain += 1
        else:
            taille -= 1  # prise en compte du zéro
    if penalites <= (2/3)*taille:
        return True
    return False

# test
T_correct = repart_correcte(T_ex)
```

```
def repart_correcte1(T):
    points_tries = trier_points(T)  # liste triée par ordre
croissant
    taille = len(points_tries)
    penalites= 0 # initialisation du décompte du nombre de case
    i = 0
    while points_tries[i] < 0:  # décompte cases pénalités
        penalites += 1
        i += 1
    while points_tries[i] == 0:  # prise en compte du zéro
        taille -= 1
        i += 1
    if penalites <= (2/3)*taille:
        return True
    return False

# test
T_correct = repart_correcte1(T_ex)
```

soit N le nombre de points dans T :

$$C(N) = \underbrace{O(N^2)}_{tri} + \underbrace{O(N)}_{boucle} = O(N^2)$$