Rectangles are Lava.

I Algortihmes de cours

Question 1. *Fibonacci*

La suite de Fibonacci est définie par récurrence double :

$$F_0 = 0,\ F_1 = 1,\ F_{n+2} = F_{n+1} + F_n$$

Écrire une fonction `fibonacci` avec la spécification suivante :
Entrées : un entier $n \geq 0$
Sortie : F_n

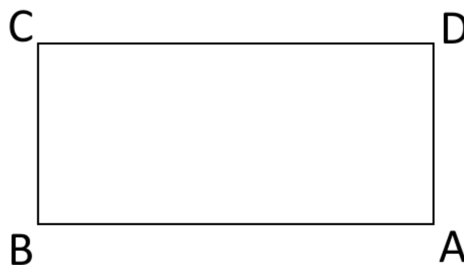
Question 2. *Exponentiation Rapide*

Écrire une fonction récursive `exponentiation_rapide` avec la spécification suivante :
Entrées : a et b deux entiers
Sortie : a^b calculé par exponentiation rapide.

II Traversée de Rectangles

Les notions présentées ici et les fonctions de cette section seront utilisées dans la dernière section pour calculer le graphe d'obstacles.

Un point sera représenté par un couple de flottants `p = (x,y)`, `x` étant l'abscisse et `y` l'ordonnée. Un segment sera noté [PQ] et sera représenté par un couple de points *différents*, `s = (p,q)`. Un rectangle sera noté [ABCD] et sera représenté par un quadruplé de points *différents deux à deux*, `r = (a,b,c,d)`, `a` étant le sommet inférieur droit du rectangle, `b` le sommet inférieur gauche, `c` le sommet supérieur gauche et `d` le sommet supérieur droit. Étant donnés deux points différents P et Q on notera (PQ) la droite passant par P et Q.



rectangle

Une liste d'obstacles est une liste de rectangles. On résume les types des structures utilisées :

- `point` est un couple de flottant (`tuple[float,float]`)
- `segment` est un couple de points (`tuple[point,point]`)
- `rectangle` est un quadruplé de points (`tuple[point,point,point,point]`)
- `obstacles` est une liste de rectangles (`list[rectangle]`)

Question 3. *Orientation*

On rend disponible une fonction ayant la spécification suivante :

```
def cote(s, p) :
    """
    Entrées : s = (A,B) un segment [AB]
    Sortie : -1 si p est à gauche de la droite (AB),
             1 si p est à droite de la droite (AB),
             0 si p est sur la droite (AB)
    """
```

1. Quelle est la signature de la fonction `cote` ?
2. Quelles sont les préconditions de la fonction ?
3. Quelles sont les postcondition de la fonction ?

Question 4. *Croisement de segment*

On dit que deux segments [AB] et [CD] se croisent si et seulement si

— A et B ne sont pas sur la droite (CD), et ne sont pas du même coté de la droite (CD), et
— C et D ne sont pas sur la droite (AB), et ne sont pas du même coté de la droite (AB)

Implémentez une fonction `interS(s1, s2)` qui prend en entrée deux segments et qui renvoie un booléen vrai si et seulement si `s1` croise `s2`.

Question 5. *Croisement d'obstacle*

On admet qu'un segment croise un rectangle si et seulement s'il croise un de ses côtés.

Implémentez une fonction `interO(s, o)` qui prend en entrée un segment `s` et des obstacles `o` et qui renvoie un booléen vrai si et seulement si `s` croise au moins un obstacle (un rectangle de la liste).

III Priorité par listes de couples.

On souhaite implémenter une file de priorité à l'aide d'une liste de couples (priorité, clé), où la priorité est un flottant positif.

On dispose déjà des deux premières opérations.

```
def creer_fp_liste() :
    return []

def est_vide_fp_liste(fp) :
    return len(fp) == 0
```

Question 6. *Suppression dans une liste*

Pour supprimer l'élément à l'indice `i` d'une liste de longueur `n`, sans préserver l'ordre des éléments, on peut échanger l'élément d'indice `i` et celui d'indice `n-1` puis supprimer le dernier élément de la liste.

Implémenter une fonction `enlever(l,i)` qui supprime l'élément d'indice `i` de `l` et le renvoi, sans nécessairement préserver l'ordre des éléments de `l`.

Question 7. *Défiler*

Implémenter une fonction `defiler_fp_liste(fp)` qui prend en entrée une FP sous forme de liste de couple et défile la clé prioritaire de la FP. *c.f.* introduction pour les détails de l'opération *defiler*.

Par exemple, `defiler_fp_liste([(3, 'a'), (1, 'b'), (5, 'c')])` renvoie `(1,'b')`.

Question 8. *Enfiler*

Implémenter une fonction `enfiler_fp_liste(fp, x, p)` qui prend en entrée une FP sous forme de liste de couple et enfile la clé `x` dans `fp` avec priorité `p`. *c.f.* introduction pour les détails de l'opération *enfiler*.

Par exemple, pour `fp = [(3, 'a'), (1, 'b'), (5, 'c')]`, après l'exécution de :

— `b = enfiler_fp_liste(fp, 'd', 10)`,
  on a `fp = [(3, 'a'), (1, 'b'), (5, 'c'), (10, 'd')]` et `b = True`.
— `b = enfiler_fp_liste(fp, 'a', 12)`,
  on a `fp = [(3, 'a'), (1, 'b'), (5, 'c')]` et `b = False`
— `b = enfiler_fp_liste(fp, 'b', 1)`,
  on a `fp = [(3, 'a'), (1, 'b'), (5, 'c')]` et `b = False`
— `b = enfiler_fp_liste(fp, 'c', 2)`,
  on a `fp = [(3, 'a'), (1, 'b'), (2, 'c')]` et `b = True`

Question 9. *Complexités*

Faire le bilan des complexités des opérations de FP représentées par liste de couples `creer_fp_liste`, `est_vide_fp_liste`, `defiler_fp_liste` et `enfiler_fp_liste` en fonction de la longueur de la FP.

IV Priorité par listes de couples *triées*.

On essaye dans cette section de trier la liste de couples de la section précédente. On implémente donc une file de priorité à l'aide d'une liste de couples (priorité, clé) *triée* par ordre de priorité *décroissante*, où la priorité est un flottant positif.

On dispose déjà des trois premières opérations.

```python
def creer_fp_tri() :
    return []

def est_vide_fp_tri(fp) :
    return len(fp) == 0

def defiler_fp_tri(fp) :
    return fp.pop()
```

On suppose dans cette partie uniquement qu'on dispose des opérations `l.insert(i,x)` et `x = l.pop(i)` qui permettent respectivement d'insérer l'élément `x` à l'indice `i` dans `l` et de supprimer et renvoyer l'élément d'indice `i` de `l`. Toutes les deux ont une complexité en O(`len(l)`).

Question 10. *Enfiler - Insertion*

Pour insérer une clé `x` à la priorité `p` dans une FP sous forme de liste de couple triée, il faut trouver l'indice `i` où insérer le couple `(x,p)` tout en préservant l'ordre.

On peut trouver cette position à l'aide d'une recherche dichotomique. On compare `p` à la priorité `pm` de la clé à l'indice `m`, milieu de la liste, et :

- si `p = pm`, on doit insérer la clé `x` à l'indice `m`;
- si `p > pm`, on doit insérer la clé `x` à un indice `i ⩽ m`;
- si `p < pm`, on doit insérer la clé `x` à un indice `i > m`;

Implémenter une fonction `inserer_fp_tri(fp, x, p)` qui prend en entrée une FP sous forme de liste de couple triée et insère l'élément `(p,x)` dans `fp` de telle manière à ce que la liste reste triée.

Par exemple, pour `fp = [(7, 'z'), (4, 'c'), (2, 't')]` après l'exécution de `inserer_fp_tri(fp, 'd', 6)`, on a `fp = [(7, 'z'), (6,'d'), (4, 'c'), (2, 't')]`

Question 11. *Enfiler - Analyse*

Quelle est la complexité de la fonction `inserer_fp_tri` en fonction de `n` la longueur de la FP représentée en liste de couple triée ?

Pour enfiler dans une FP, on peut utiliser la fonction `inserer_fp_tri` pour insérer la clé lorsqu'elle n'est pas dans la FP.

Si l'élément est dans la FP, il faut trouver son indice dans la liste pour le modifier. Pour enfiler `x` de priorité `p`, peut-on utiliser une recherche dichotomique pour chercher l'indice de `x` et modifier sa priorité ?

Expliquer (sans l'implémenter) comment écrire `enfiler_fp_tri(fp,x,p)`. Quelle complexité obtient-on ?

V Priorité par tas-min.

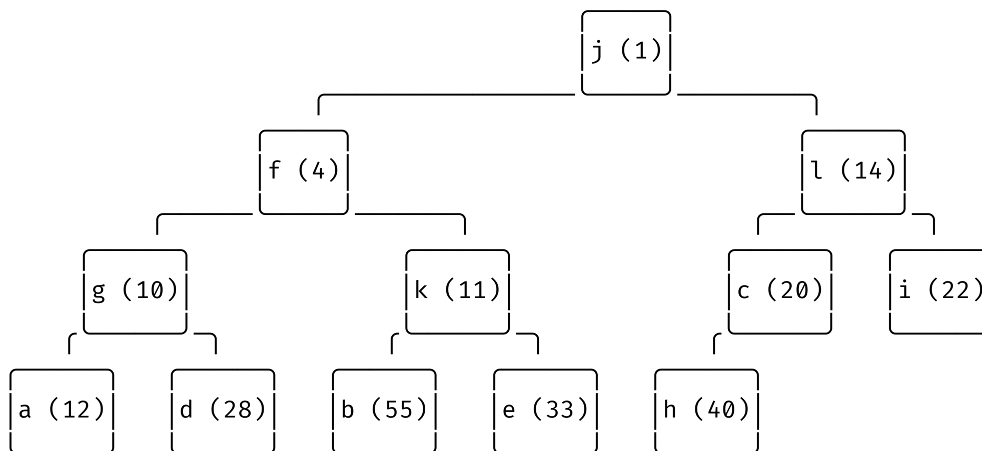
Les listes triées n'apportent pas d'amélioration en complexité temporelle puisque l'insertion et la suppression restent coûteuses.

Une manière efficace de régler ce problème et d'utiliser un *tas-min*. Un tas-min est une liste qui peut être vue comme une structure d'arbre binaire complet où chaque élément est plus petit que son parent direct.

Représentation Liste

```
[(1, 'j'), (4, 'f'), (14, 'l'), (10, 'g'), (11, 'k'), (20, 'c'),
 (22, 'i'), (12, 'a'), (28, 'd'), (55, 'b'), (33, 'e'), (40, 'h')]
```

Représentation Arborescente



Étant donné une liste de longueur `n` on peut organiser les éléments par niveau et les hiérarchiser. On introduit pour se faire quelques définitions :

- *Numéro.* le numéro de l'élément à l'indice i est $i+1$. Les éléments du tas-min sont ainsi numéroté de `1` à `n` (inclus).
- *Racine.* l'élément prioritaire est l'élément numéroté 1, qu'on nomme la *racine*.
- *Parent.* l'élément numéro $i > 1$ (*i.e* autre que la racine) possède un parent. Il s'agit de l'élément numéroté $\lfloor \frac{i}{2} \rfloor$ (`i // 2` en Python).
- *Enfants.* chaque élément numéroté i possède au plus deux enfants. Il s'agit des éléments numérotés $2i$ et $2i+1$ si ils existent (*i.e.* si $2i \leq n$ et/ou $2i+1 \leq n$)

Un tas-min est alors une liste de couple de longueur `n`, tel que

- le premier élément du couple est un flottant appelé *priorité*.
- le second élément du couple est appelé *clé*.
- ($*$) la priorité d'un élément autre que la racine est *toujours supérieure* à celle de son parent.

Question 12. *Manipulation*

On considère le tas-min `T` suivant :

```
[(4, 'h'), (30, 'b'), (10, 'y'), (34, 'n'), (50, 'm'), (13, 'r'), (12, 'p'),
 (36, 'z'), (67, 'c'), (78, 't')]
```

1. Donner la clé de la racine du tas.
2. Donner le numéro de l'élément de clé `'c'`.
3. Donner la clé du parent de l'élément de clé `'m'`.
4. Donner l'ensemble des clés des enfants de l'élément de clé `'y'`.
5. Donner l'ensemble des clés des enfants de l'élément de clé `'m'`.
6. Donner l'ensemble des clés des enfants de l'élément de clé `'p'`.

Question 13. *Codage binaire*

Dans le tas `T` de la question précédente, donner l'écriture binaire du numéro de l'élément de clé `'n'`, de son parent et de ses enfants. Expliquer comment obtenir le numéro du parent et des enfants d'un élément à partir par des opérations simples (aucune opération arithmétique) sur le code binaire de son numéro.

Question 14. *Parents et enfants efficaces*

Certaines opérations sur le code binaire (avec le bit de poids fort en premier) peuvent être spécifiée directement en Python, car elles sont très efficaces. Pour `x` un entier :

— `x << 1` ajoute un 0 à la fin du code binaire de `x`. Si `x = 13`, `x` s'écrit `1101` en binaire et `y = x << 1` sera l'entier codé par `11010`, donc `y` sera 26.
— `x >> 1` enlève le dernier bit du code binaire de `x`. Si `x = 13`, `x` s'écrit `1101` en binaire et `y = x >> 1` sera l'entier codé par `110`, donc `y` sera 6.
— `x | 0b1` remplace le dernier bit du code binaire de `x` par 1. Si `x = 13`, `x` s'écrit `1101` en binaire et `x | 0b1` sera donc inchangé, toujours égal à 13. Si `x = 18`, `x` s'écrit `10010` en binaire et `y = x | 0b1` sera `10011`, donc `y` sera `19`.

On suppose que `x` est le numéro d'un élément de `T`. Implémenter deux fonctions `parent(T,x)` et `enfants(T,x)` qui prennent un tas-min, `T`, et un entier `1 ⩽ x ⩽ len(T)` en entrée et renvoie respectivement le numéro du parent et la liste des numéros des enfants de l'élément numéroté `x`. `parent(T,1)` renvoie `None` puisque la racine n'a pas de parent. La liste renvoyée par `enfants(T,x)` peut être de longueur 0, 1 ou 2. On utilisera les opérations efficaces sur le code binaire décrites ci-dessus.

Question 15. *Percolation haute*

Un quasi-tas vérifie les deux premières propriétés des tas-min, mais pas nécessairement (*), *i.e.* c'est une liste de couple de priorité et clé.

L'opération de *percolation haute* dans un quasi-tas de couples est l'opération consistant à échanger un élément avec son prédécesseur si sa priorité est plus faible.

Implémenter une fonction `percolation_haute(T,x)` qui prend un quasi-tas, `T` et un entier `1 ⩽ x ⩽ len(T)` en entrées, et échange l'élément numéroté `x`, `(p,v)`, avec son parent, `(q,w)`, si `p < q` et si `x > 1`. La fonction renvoie `False` si `x = 1` ou si l'échange n'a pas eu lieu, `True` sinon.

Question 16. *Diminution de priorité*

Soit `T` un tas-min et `x` le numéro d'un élément de `T` de priorité `p`. Pour diminuer la priorité de l'élément de `p` à `q < p` il ne suffit pas de modifier le couple. La propriété de tas-min (∗) n'étant plus nécessairement respectée on obtient un quasi-tas.

Pour que la propriété soit de nouveau respectée après modification du couple, il faut *percoler* l'élément tant que (∗) n'est pas vérifiée, *i.e.* tant que `T` n'est pas redevenu un tas.

Voici un exemple des percolations successives à exécuter pour le tas `T` suivant :

```
[(4, 'h'), (30, 'b'), (10, 'y'), (34, 'n'), (50, 'm'), (13, 'r'), (12, 'p'),
 (36, 'z'), (67, 'c'), (78, 't')]
# == Modification du numéro 8 (36,'z') -> (6,'z')
[(4, 'h'), (30, 'b'), (10, 'y'), (34, 'n'), (50, 'm'), (13, 'r'), (12, 'p'),
 (6, 'z'), (67, 'c'), (78, 't')]
# ==== Percolation du numéro 8
[(4, 'h'), (30, 'b'), (10, 'y'), (6, 'z'), (50, 'm'), (13, 'r'), (12, 'p'),
 (34, 'n'), (67, 'c'), (78, 't')]
# ==== Percolation du numéro 4
[(4, 'h'), (6, 'z'), (10, 'y'), (30, 'b'), (50, 'm'), (13, 'r'), (12, 'p'),
 (34, 'n'), (67, 'c'), (78, 't')]
# == Fin
```

L'algorithme de diminution de priorité s'écrit donc en pseudo-code comme ceci :

entrées : T un tas-min, x le numéro d'un élément (p, v) de T, $q < p$ un entier.
sorties/effets : T est toujours un tas-min où (p, v) est transformé en (q, v).

Soit (p, v) l'élément numéro x de T.
l'élément numéro x de T *devient* (q, v)
$y = x$
/ Boucle (1) */*
tant que la percolation haute du numéro y dans T a réalisé un échange, **faire**
  y = le numéro du parent de l'élément numéroté y dans T
fin tant que

Implémenter l'algorithme ci-dessus dans une fonction `diminuer_prio(T,x,q)`.

Question 17. *Analyse de* `diminuer_prio`.

On analyse l'algorithme précédent sur la base de son pseudo-code.

1. On supposera acquis l'invariant « y est un numéro valide de T » pour la boucle (1). Montrer que y est un variant de la boucle (1). En conclure que l'algorithme termine.
2. Montrer que « I : pour tout élément e numéroté z de T, les enfants de z ont une priorité plus grande que le parent de e s'il existe et si $z \neq y$ la priorité de e est plus grande que celle de son parent s'il existe. » est un invariant de boucle. En déduire qu'à la fin de l'algorithme T est bien un tas-min.
3. Montrer que la complexité de l'algorithme est en $O(\log_2(n))$ avec n le nombre d'éléments du tas-min. On admettra qu'une percolation peut se faire en $O(1)$.

Question 18. *File de priorité*

Pour implémenter une file de priorité, on peut utiliser la combinaison d'un tas-min, pour gérer les priorités efficacement, et d'un dictionnaire qui sauvegarde efficacement le numéro de chaque élément dans le tas.

Une file de priorité sera donc un couple `fp = (tas,num)` tel que, `tas` est un tas-min et `num` est un dictionnaire tel que :

- `c` est une clé de `num` si et seulement si c'est une clé d'un élément de `tas`.
- pour chaque élément `(p,c)` de `tas` numéroté `x`, `num[c] = x`.

On nommera clé de `fp` les clés de `tas` ou de `num` indifféremment, et priorité de la clé `c` de `fp`, l'unique entier `p` tel que `(p,c)` est l'élément numéroté `num[c]` de `tas`.

Voici un exemple de file de priorité :

```
tas = [(4, 'h'), (30, 'b'), (10, 'y'), (34, 'n'), (50, 'm'),
       (13, 'r'), (12, 'p'), (36, 'z'), (67, 'c'), (78, 't')]
num = { 'h' : 1, 'b' : 2, 'y' : 3, 'n' : 4, 'm' : 5,
        'r' : 6, 'p' : 7, 'z' : 8, 'c' : 9, 't' : 10 }
fp = (tas,num)
```

Expliquer, sans écrire le code, comment modifier les fonctions `percolation_haute` et `diminuer_prio` pour obtenir les fonctions

- `percolation_haute_fp(fp,x)` qui prend en entrée une file de priorité `fp`, un entier `x` et effectue une percolation sur l'élément numéroté `x` dans le tas de la FP.
- `diminuer_prio_fp(fp,c,q)` qui prend en entrée une file de priorité `fp`, une clé `c` d'un élément `(p,c)` de `fp`, `q < p` un entier et diminue à `q` la priorité de l'élément `(p,c)`.

On supposera la fonction `diminuer_prio_fp(fp,c,q)` disponible dans la suite et ayant une complexité en $O(\log_2(n))$, avec n le nombre de clés de la FP.

Question 19. *Enfiler*

Pour enfiler une clé `c` avec priorité `p` dans une FP `fp = (tas,num)`, il faut d'abord vérifier si `c` est une clé de `fp`. Si ce n'est pas le cas, on ajoute à la fin du tas la clé `c` avec une priorité $+\infty$ (`float("inf")` en Python), pour être sur de vérifier la propriété $(*)$ de tas-min. On n'oublie pas de mettre à jour `num` pour associer à `c` son numéro dans le tas.

Ensuite, `c` est nécessairement dans le tas avec une priorité `q`, éventuellement infinie. Donc :

- si `p < q` il suffit de diminuer sa priorité à `p` à l'aide de l'algorithme `diminuer_prio_fp`. La fonction renvoie alors `True`.
- sinon on ne fait rien et la fonction renvoie `False`.

Implémenter une fonction `enfiler_fp_tas(fp,c,p)` qui enfile dans `fp` la clé `c` avec priorité `p`.

Question 20. *Défiler*

On supposera la fonction `augmenter_prio_fp(fp)` analogue à `diminuer_prio_fp`, qui permet d'augmenter la priorité d'une clé `c` à `q` et ayant une complexité en $O(\log_2(n))$, avec n le nombre de clés de la FP.

On supposera aussi disponible la fonction `d.pop(c)` qui supprimer l'association du dictionnaire `d` à la clé `c`, en $O(1)$.

Pour défiler la clé prioritaire d'une FP `fp = (tas,num)` :

- on sauvegarde l'élément `e` numéro 1 du `tas`. On supprime la clé de `e` de `num`.
- on supprime et sauvegarde le dernier élément `(q,k)` du tas.
- on change la clé du numéro 1 en `k` dans le `tas` et dans `num`.
- on augmente la priorité de `k` dans `fp` à `q`.
- on renvoie `e`.

Implémenter une fonction `defiler_fp_tas(fp)` qui défile dans `fp` la clé prioritaire.

Question 21. *Complexités*

On ajoute à ces deux opérations les deux suivantes :

```
def creer_fp_tas() :
    return ([],{})

def est_vide_fp_tas(fp) :
    return len(fp[0]) == 0
```

Quelles sont les complexités des quatre opérations de file de priorité implémentées à l'aide de tas-min ?

Comparer à l'implémentation par liste.

VI Recherche de chemin

On dispose d'une file de priorité efficace pour implémenter Dijkstra. Il est temps de récupérer les informations du terrain pour exécuter l'algorithme et trouver le chemin qui sauvera Nadine.

Question 22. *Graphe de rectangles*

On suppose dans cette question que les obstacles sont donnés comme une liste de rectangle comme décrit en introduction.

On admet que chemin optimal passe forcément par des segments qui relient le depart, la sortie et les sommets des rectangles entre eux sans traverser d'obstacles.

Formellement, soit O un ensemble de rectangle, Σ l'ensemble des sommets des rectangles de O, d et s deux points. On considère le graphe non orienté $G = (S, A)$ défini par :

- $S = \Sigma \cup \{s, d\}$
- $\{A, B\} \in A$ si et seulement si le segment $[AB]$ ne croise aucun rectangle de O.

On pondère ensuite chaque arête par la longueur du segment.

Implémenter une fonction `graphe_rec(O, s, d)` qui prend en entrée des obstacles `O` (liste de rectangles comme définie en introduction), deux points `s` et `d` et renvoie le graphe G représenté par dictionnaire d'adjacence ci-dessus.

Question 23. *Complexité de l'algorithme de Dijkstra*

Il suffit d'appliquer l'algorithme de Dijkstra sur le graphe ci-dessus en partant de `s` et en s'arrêtant en `d` pour trouver le chemin optimal de `s` à `d`.

Voici une implémentation possible de l'algorithme de Dijkstra.

```
def sauver_nadine(g, depart, sortie) :
    """
    Entrées : g, le dictionnaire d'adjacence d'un graphe pondéré
                (poids uniquement positifs).
              depart, un sommet de g
              sortie, un sommet de g
    Sortie : c, une liste de sommets correspondant au chemin optimal
                allant de depart à sortie dans g.
                None si sortie est inaccessible.
    """
    # Structures
    predecesseurs = {}
    marquage = {}
    fp = creer_fp()
    # Initialisation
    enfiler_fp(fp, depart, 0)
    # Algorithme
    while not est_vide_fp(fp) :
        # On récupère le sommet non visité à distance minimale
        d, s = defiler_fp(fp)
        marquage[s] = True
        if s == sortie :
            return chemin(predecesseurs, sortie)
        # On relache tous les voisins
        for poids, voisin in g[s] :
            distance_par_s = d + poids
            if ( voisin not in marquage
                 and enfiler_fp(fp,voisin,distance_par_s) ) :
                predecesseurs[voisin] = s
    return predecesseurs
```

On suppose qu'on dispose d'une fonction `chemin(parent,sortie)` qui prend un dictionnaire de prédécesseurs, un sommet et renvoi le chemin optimal allant du départ jusqu'à la sortie, et cela en O(`len(parent)`).

On suppose que `creer_fp` et `est_vide_fp` ont une complexité en O(1) et on note $C_e(n,m)$ et $C_d(n,m)$ les complexité respectives des fonctions `defiler_fp` et `enfiler_fp` en fonction du nombre de sommets (n) et du nombre d'arêtes (m) du graphe non orienté g.

1. Exprimer la complexité de l'algorithme de Dijkstra en fonction de n, m, $C_e(n,m)$ et $C_d(n,m)$.
2. Donner la complexité de l'algorithme si on implémente la file de priorité par une liste de couples.
3. Donner la complexité de l'algorithme si on implémente la file de priorité par un tas-min.

Question 24. *Calcul du graphe - Image*

On cherche une autre manière de calculer un graphe qui prendrait un terrain plus complexe en compte.

On suppose dans cette question que le plan 2D est discrétisé et que les informations du terrain sont sauvegardées dans une matrice $M = n \times m$.

Pour toute coordonnée `(i,j)`, `M[i][j]` correspond au coût de déplacement pour *entrer* dans la case. `M[i][j]` est un flottant positif, éventuellement $+\infty$ (`float('inf')`) si la case est inaccessible.

Le graphe du terrain est donc le graphe orienté pondéré $G = (S, A)$ tel que $S = \{(i,j), i,j \in \{0, \cdots, n-1\},\}$ et il y a un arc d'un sommet `(i,j)` à un sommet `(u,v)` de poids `p` si et seulement si :

— `p = M[u][v]`

— `|u-i| = 1` et `v = j`, ou `u = i` et `|j-v| = 1`.

Visuellement les successeurs d'un sommet sont les sommets situés au-dessus, en-dessous, à gauche et à droite.

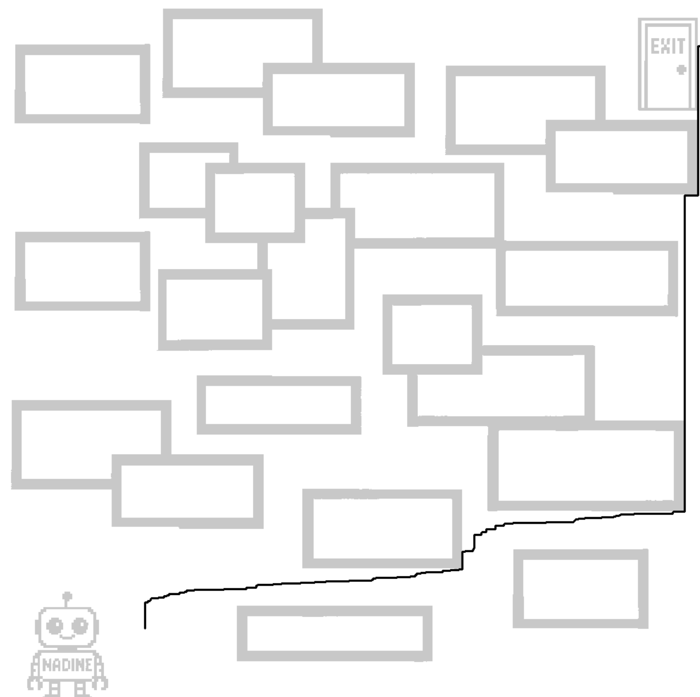
Implémenter une fonction `graphe_terrain(M)` qui prend une matrice contenant les informations du terrain et renvoie le graphe du même terrain.

Question 25. *Heuristique* A^*

Dans l'optique d'améliorer l'efficacité en utilisant l'algorithme A^* on propose comme heuristique la *distance de Manhattan* définie par :

$$\texttt{Man}((i1, j1), (i2, j2)) = |i1 - i2| + |j1 - j2|$$

La distance est-elle admissible (est-ce que `Man` peut renvoyer une distance plus grande que la distance réelle) ? Le chemin renvoyé par A^* est-il optimal ?



Sauvetage assuré. GG WP.