Corrigé — Rectangles are Lava.

I Algortihmes de cours

Question 1. *Fibonacci*

coeff
1

c.f. cours.

Question 2. *Exponentiation Rapide*

coeff
1

c.f. cours.

II Traversée de Rectangles

Question 3. *Orientation*

coeff	signature	préconditions	postcondition
1	3	3	3

La signature donne le nom de la fonction, le type des entrées et de la sortie :

- Nom : `cote`
- Entrées : `s` un couple de couples de flottants, `p` un couple de flottant
- Sortie : un entier.

Les préconditions sont les conditions sur les entrées. Il y a une condition sur `s`. En tant que *segment* les deux points de `s` doivent être différents.

La postcondition est la condition sur la sortie. C'est exactement sa derscription :

- -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)

On donne la fonction en bonus. L'orientation peut-être obtenue par le signe de la troisième coordonnée du produit vectoriel des deux vecteurs $\overrightarrow{AB}$ et $\overrightarrow{AP}$.

```
def cote(s : segment, p : point) -> int :
    a,b = s
    xa,ya = a
    xb, yb = b
    xp, yp = p
    zvect = (xb-xa)*(yp-ya) - (yb-ya)*(xp-xa)
```

```
    if zvect < 0 :
        return 1
    elif zvect > 0 :
        return -1
    return 0
```

Question 4. *Croisement de segment*

coeff	Cas 1 et 3	Cas 2	Cas 4
1	3	3	3

```
def interS(s1 : segment, s2 : segment) -> bool :
    a,b = s1
    c,d = s2
    return ( cote(s2,a) * cote(s2,b) < 0 and cote(s1,c) * cote(s1, d) < 0 )
```

Question 5. *Croisement d'obstacle*

coeff	parcours	test	retour
2	3	3	3

On défini une fonction auxiliaire pour savoir si un segment croise un rectangle.

```
def interR(s : segment, r : rectangle) -> bool :
    a,b,c,d = r
    return ( interS(s,(A,B))
             or interS(s,(B,C))
             or interS(s,(C,D))
             or interS(s,(D,A)) )
```

On cherche un obstacle qui croise `s` dans `o` par recherche linéaire.

```
def interO(s : segment, o : obstacles) -> bool :
    for r in o :
        if interR(s,r) :
            return True
    return False
```

III Priorité par listes de couples.

```
def creer_fp_liste() :
    return []

def est_vide_fp_liste(fp) :
```

```
    return len(fp) == 0
```

Question 6. *Suppression dans une liste*

coeff	echange	suppression
1	6	3

```
def enlever(l, i) :
    n = len(l)
    l[n-1], l[i] = l[i], l[n-1]
    return l.pop()
```

Question 7. *Défiler*

coeff	parcours	maj minimum	enlever
2	3	4	2

C'est l'algorithme de recherche d'indice du minimum, puis on utilise `enlever`.

```
def defiler_fp_liste(fp) :
    i_min = None
    p_min = float("inf")
    for i in range(len(fp)) :
        p, e = fp[i]
        if p < p_min :
            i_min = i
            p_min = p
    e_min = enlever(fp,i_min)
    return e_min
```

Question 8. *Enfiler*

coeff	recherche	modification	ajout
2	4	3	2

On fait une recherche linéaire du couple avec `x` comme élément. Si on le trouve, on modifie sa priorité si elle strictement est plus petite (attention, un tuple est immuable). Si on ne le trouve pas on ajoute le couple `(p,x)` à la fin.

```
def enfiler_fp_liste(fp, x, p) :
    for i in range(len(fp)) :
        pe, e = fp[i]
        if e == x :
            if pe <= p :
                return False
```

```
            fp[i] = (p, x)
            return True
    fp.append((p,x))
    return True
```

Question 9. *Complexités*

coeff	creer	est_vide	defiler	enfiler
2	1	1	4	3

Pour une liste de longueur n.

- `creer_fp_liste` en O(1);
- `est_vide_fp_liste` en O(1);
- `defiler_fp_liste`
  - la complexité de `enlever` est en O(1)
  - chaque itération de la boucle de `defiler_fp_liste` est en O(1) donc la boucle est en O(n) (recherche de minimum).
  - la fonction est au final en O(n);
- `enfiler_fp_liste` : chaque itération de la boucle de `enfiler_fp_liste` est en O(1) donc la boucle est en O(n) (recherche linéaire).

```
def creer_fp_tri() :
    return []

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

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

Question 10. *Enfiler - Insertion*

coeff	division correcte de l'intervale	arrêt correct	insertion
3	4	3	2

La propriété invariante est la suivante : l'élément doit être inséré à un indice `d` $\leqslant$ `i` $\leqslant$ `f`. Quand `d = f`, la boucle s'arrête et on insère l'élément à l'indice `d` ou `f`, qui sont égaux.

```
def inserer_fp_tri(fp, x, p) :
    d, f = 0, len(fp)
    while d < f :
        m = (d +f) // 2
        pm = fp[m][0]
        if pm == p :
```

```
            fp.insert(m,(p,x))
            return
        elif pm > p :
            d = m + 1
        else :
            f = m
    fp.insert(d,(p,x))
```

Question 11. *Enfiler - Analyse*

coeff	complexité inserer_fp_tri	description	complexité finale
2	3	3	3

La complexité est celle de la recherche dichotomique, en O(log(n)), à laquelle doit s'ajouter la complexité de l'insertion en O(n). En effet l'insertion peut-être faîte au pire des cas à l'indice 0 et la fonction `inserer_fp_tri(l,i,e)` a une complexité O(n) avec `n` la longueur de `l`.

Donc la complexité finale est en O(n).

On ne peut pas chercher l'élément avec une recherche dichotomique car les éléments ne sont pas triés.

On doit donc chercher l'élément avec une recherche linaire, puis le supprimer avec `enlever_tri` et le réinsérer avec `inserer_fp_tri`. Au final la complexité serait en O(n).

IV Priorité par tas-min.

Question 12. *Manipulation*

coeff	numéro	prédecesseur	succésseurs
1	2	2	5

1. `'h'`
2. 9
3. `'b'`
4. `'r'` et `'p'`
5. `'t'`
6. ∅

Question 13. *Codage binaire*

coeff	numéro	parent	enfants
2			

`'n'` est la clé de l'élément numéro 4, `100` en binaire. Son parent est le numéro 2, `10` en binaire. Ses enfants sont les numéros 8 et 9, `1000` et `1001` en binaire.

On remarque que le parent d'écrit avec un bit de moins à droite, puisque c'est le quotient par 2. Et que les enfants s'écrivent avec un bit de plus à droite, `0` ou `1`, puisque le numéro est le quotient par 2 du numéro de ses enfants.

Question 14. *Parents et enfants efficaces*

coeff	parent	enfants
2	3	6

```
def parent(T : Tas, x : int) -> int :
    if x == 1 :
        return None
    return x >> 1

def enfants(T : Tas, x : int) -> list[int] :
    e = []
    e1 = x << 1
    if e1 <= len(T) :
        e.append(e1)
        e2 = e1 | 0b1
        if e2 <= len(T) :
            e.append(e2)
    return e
```

Question 15. *Percolation haute*

coeff	echange	accès	retour
2	3	3	3

```
def percolation_haute(T : Tas,x : int) -> bool :
    if x == 1 :
        return False
    px = parent(T,x)
    p = T[x-1][0]
    q = T[px-1][0]
    if p < q :
        T[x-1], T[px-1] = T[px-1], T[x-1]
        return True
    return False
```

Question 16. *Diminution de priorité*

coeff	initialisation	condition	mise à jour
2	2	4	3

```
def diminuer_prio(T : Tas, x : int, q : float) :
    p, v = T[x-1]
    T[x-1] = (q, v)
    y = x
    while percolation_haute(T, y) :
        y = parent(T,y)
```

Question 17. *Analyse de* `diminuer_prio`.

coeff	terminaison	correction	complexité
6	3	3	3

1. *Initialisation.* y est initialement égal à x qui est un numéro donc un entier plus grand ou égal à 1. y est bien un entier naturel.

   Hérédité. Supposons $y \in \mathbb{N}$ avant la boucle (1) et qu'on réalise une itération complète. On a supposé acquis que y est bien un numéro. Les préconditions de la fonction `percolation_haute` sont donc vérifiées et la fonction renvoie vrai puisqu'on réalise l'itération. Ceci indique que $y > 1$. Puis y', la nouvelle valeur de y est égale à celle du parent de y, c'est à dire $y' = \lfloor \frac{y}{2} \rfloor <= \frac{y}{2} < y$. y est resté un entier naturel (et un numéro d'ailleurs) et est strictement plus petit qu'avant la boucle.

   Conclusion. y est un variant de la boucle (1) qui est la seule boucle **`while`** de l'algorithme. La boucle termine et l'algorithme aussi.
2. *Initialisation.* T est un tas avant la modification de l'élément numéroté x, donc la priorité $(*)$ est vérifiée. Après modification, le seul élément à éventuellement ne pas avoir une priorité supérieure à celle de son parent est l'élément e numéroté x. Mais les enfants de e gardent une priorité plus grande que le parent de e par transitivité. Or $y = x$ donc I est vrai avant la première itération de la boucle (1).

   Hérédité. Suppose I vrai avant la boucle (1) et qu'on réalise une itération complète. On note e l'élément numéroté y avant l'itération et e' son parent. Un échange a été réalisé puisque la percolation haute a eu lieu. e' est maintenant numéroté y et e est numéroté y' la nouvelle valeur de y. e' avait une priorité inférieure au enfants de e avant l'itération, qui sont maintenant les enfants de e' et qui ont bien une priorité inférieur à leur parent par hypothèse de récurrence. La priorité de e est inférieure à celle de e' vu que l'échange à eu lieu et si e' avant un second enfant, sa priorité est supérieure à celle de e' donc à celle de e. Ainsi la priorité des nouveaux enfants de e est bien supérieure à celle de e. Le parent de e s'il existe est l'ancien parent de e' et sa priorité est inférieure à celle de e et de l'éventuelle second enfant de e' par hypothèse de récurrence. Donc I est reste vrai après l'échange.

   Conclusion. I est vrai avant chaque itération de la boucle (1) et en particulier avant la fin de la boucle. Or à la fin de la boucle, la percolation n'a pas lieu, ce qui signifie que e est la racine ou que le parent de e a une priorité inférieur à e. Mais alors d'après I, $(*)$ est respectée. T est un tas-min.
3. Toutes les opérations de l'algorithme sont constantes. La complexité est majorée par le nombre d'itérations de la boucle (1). y est un variant minoré par 1 et majoré par **`len`**`(T)`. Or y est divisé par 2 à chaque itération, donc comme pour la recherche dichotomique, le nombre d'itération

est majoré par $\log_2(\texttt{len(T)})$.

Question 18. *File de priorité*

coeff	percolation	diminuer
1	6	3

Pour la percolation haute, il faut mettre à jour les numéros des clés à chaque échange. Ainsi, si l'élément de clé c est échangé avec son parent de clé c', il faut rajouter `num[c], num[c']␣=␣num[c'], num[c]` avec `num = fp[1]`.

Pour `diminuer_prio_fp` l'algorithme n'est pas réellement impacté. Il faut simplement récupérer le numéro de l'élément de clé `c` à l'aide de `num[c]`.

Bonus, les fonctions :

```
def percolation_haute_fp(fp : FP,x : int) -> bool :
    if x == 1 :
        return False
    T, N = fp
    px = parent(T,x)
    p,c = T[x-1]
    q,k = T[px-1]
    if p < q :
        T[x-1], T[px-1] = T[px-1], T[x-1]
        N[c], N[k] = px, x
        return True
    return False
```

```
def diminuer_prio_fp(fp : FP, c : cle, q : float) :
    T, N = fp
    x = N[c]
    p, v = T[x-1]
    T[x-1] = (q, v)
    y = x
    while percolation_haute_fp(fp, y) :
        y = parent(T,y)
```

Question 19. *Enfiler*

coeff	ajout	test et appel
2	5	4

```
def enfiler_fp_tas(fp : FP, c : cle, p : float) -> bool :
    tas, num = fp
    if not c in num :
        tas.append((float("inf"),c))
```

```
        num[c] = len(tas)
    x = num[c]
    q = tas[x-1][0]
    if p >= q :
        return False
    diminuer_prio_fp(fp, c, p)
    return True
```

Question 20. *Défiler*

coeff	renvoi c	echange fin	augmentation
2	3	3	3

```
def defiler_fp_tas(fp : FP) -> cle :
    tas, num = fp
    p, c = tas[0]
    num.pop(c)
    q,k = tas.pop()
    if len(tas) > 0 :
        tas[0] = (p,k)
        num[k] = 1
        augmentation_prio_fp(fp, k, q)
    return (p,c)
```

Bonus :

```
def percolation_basse_fp(fp : FP,x : int) -> int :
    T, N = fp
    exs = enfants(T,x)
    if len(exs) == 0 :
        return 0
    ex = exs[0]
    if len(exs) > 1 and T[ex-1][0] > T[exs[1]-1][0] :
        ex = exs[1]
    q,k = T[ex-1]
    p,c = T[x-1]
    if p > q :
        T[x-1], T[ex-1] = T[ex-1], T[x-1]
        N[c], N[k] = ex, x
        return ex
    return 0
```

```
def augmentation_prio_fp(fp : FP, c : cle, q : float) :
    T, N = fp
    x = N[c]
    p, v = T[x-1]
    T[x-1] = (q, v)
```

```
    y = percolation_basse_fp(fp, x)
    while y != 0 :
        y = percolation_basse_fp(fp, y)
```

Question 21. *Complexités*

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

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

coeff	defiler	enfiler	comparaison
1	3	3	3

`defiler_fp_tas` et `enfiler_fp_tas` on les mêmes complexités que `augmentation_prio_fp` et `diminution_prio_fp` qui sont en $O(\log_2(n))$ avec n le nombre de clés de la file de priorité. Les opérations de création et de test à vide sont en $O(1)$.

L'implémentation par liste permet d'enfiler plus rapidement mais défile en $O(n)$ donc plus lentement. La fonction logarithmique progressant très lentement, si on doit défiler autant de fois qu'on doit enfiler, on préférera augmenter le coup de l'opération *enfiler* pour diminuer celle de *defiler*.

V Recherche de chemin

Question 22. *Graphe de rectangles*

coeff	sommets	arêtes
3	3	6

On récupère tous les sommets des rectangles de O, puis on vérifie pour chaque paire si elle constitue un segment qui ne croise aucun obstacle.

```
def graphe_rec(O : obstacles, s : point, d : point) -> graphe :
    g = { s : [], d : []}
    for A,B,C,D in O :
        g[A], g[B], g[C], g[D] = [], [], [], []
    for v in g.keys() :
        for w in g.keys() :
            if v != w and not interO((v,w),O) :
                d = ((v[0] - w[0]) ** 2 + (v[1] - w[1]) ** 2) ** 0.5
                g[v].append((d, w))
    return g
```

Question 23. *Complexité 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
```

coeff	1)	2)	3)
2	5	2	2

1. La boucle **while** tourne une fois pour chaque sommet du graphe. Pour chaque sommet, on test le vide en O(1), on défile en $C_d(n,m)$, puis pour chaque voisin on enfile le sommet en $C_e(n,m)$. Cette dernière opération se fait au final pour chaque arc. La complexité finale est donc en $O(nC_d(n,m) + mC_e(n,m))$
2. Pour une implémentation par liste on obtient donc un $O(n^2 + m) = O(n^2)$.
3. Pour une implémentation par tas on obtient donc un $O((n+m)\log_2(n))$ ce qui est mieux. Mais pas le mieux qu'on puisse faire !

Bonus :

```
def chemin(P,s) :
    c = [s]
    while s in P :
```

```
        c.append(P[s])
        s = P[s]
    return c
```

Question 24. *Calcul du graphe - Image*

coeff	parcours	tests	dictionnaire
3	3	3	3

```
def graphe_terrain(M) :
    n, m = len(M), len(M[0])
    g = {}
    for i in range(n) :
        for j in range(m) :
            g[(i,j)] = []
            if 0 < i and M[i-1][j] < float("inf"):
                g[(i,j)].append((M[i-1][j],(i-1,j)))
            if i < n-1 and M[i+1][j] < float("inf"):
                g[(i,j)].append((M[i+1][j],(i+1,j)))
            if 0 < j and M[i][j-1] < float("inf"):
                g[(i,j)].append((M[i][j-1],(i,j-1)))
            if j < n-1 and M[i][j+1] < float("inf"):
                g[(i,j)].append((M[i][j+1],(i,j+1)))
    return g
```

Question 25. *Heuristique* A*

coeff
1

Oui car la distance la plus petite dans un damier pour aller d'un point à un autre est la distance de Manhattan. Du coup le chemin est optimal.